Advanced Algorithmic Techniques (COMP523)

Recursion and Divide and Conquer Techniques

Recap and plan

Recap and plan

- **Last lecture:**
  - Examples of algorithms (searching and sorting in linear time).
  - Analysis of correctness, running time and memory.
  - Asymptotic notation and asymptotic complexity.

Recap and plan

- **Last lecture:**
  - Examples of algorithms (searching and sorting in linear time).
  - Analysis of correctness, running time and memory.
  - Asymptotic notation and asymptotic complexity.
- **This lecture:**
  - Asymptotic complexity (cont.)
  - Searching in logarithmic time.
  - Finding *majority* in an array.

Asymptotic Complexity

Asymptotic Notation

$\mathbf{O(g(n)) = f(n)}$: there exist positive constants c and n_0 such that

$$0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0.$$

$\mathbf{\Omega(g(n)) = f(n)}$: there exist positive constants c and n_0 such that

$$0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0.$$

$\mathbf{\Theta(g(n)) = f(n)}$: there exist positive constants c_1, c_2 and n_0 such that

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0.$$

$\mathbf{o(g(n)) = f(n)}$: for any constant $c > 0,$ there exists a constant

$$n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0.$$

$\omega\mathbf{(g(n)) = f(n)}$: for any constant $c > 0,$ there exists a constant

$$n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0$$

Comparing functions

- Asymptotic comparisons satisfy several relational properties.
  - Transitivity
  - Reflexivity
  - Symmetry
  - Transpose Symmetry
  - Sum and maximum

Transitivity

- If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$.
- If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.
- If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, then $f(n) = \Omega(h(n))$.
- If $f(n) = o(g(n))$ and $g(n) = o(h(n))$, then $f(n) = o(h(n))$.
- If $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$, then $f(n) = \omega(h(n))$.

Reflexivity

- $f(n) = \Theta(f(n))$
- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$

- Is it true that $f(n)$ is $o(f(n))$ and $\omega(f(n))$?

Symmetric Relations

- Symmetry:
  - $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
- Transpose Symmetry:
  - $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
  - $f(n) = o(g(n))$ if any only if $g(n) = \omega(f(n))$.

Sum and maximum

Sum and maximum

- $f_1(n) + f_2(n) + \ldots + f_k(n) = \Theta(\max(f_1(n), f_2(n), \ldots, f_k(n))$
    - for any constant positive integer k.

Sum and maximum

- $f_1(n) + f_2(n) + \ldots + f_k(n) = \Theta(\max(f_1(n), f_2(n), \ldots, f_k(n))$
  - for any constant positive integer k.
- If k is not constant, this is not true!
  - Let $f_j(n) = j$.
  - Let k = n
  - $f_1(n) + f_2(n) + \ldots + f_k(n) = n(n+1)/2 = \Theta(n^2)$.

Searching in logarithmic time

Example: Running Time of LinearSearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

17	1	2	4	6	10	14	17	19	21	24

Example: Running Time of LinearSearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

17

1	2	4	6	10	14	17	19	21	24

- We read through the array until we find the number.

Example: Running Time of LinearSearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

17

1	2	4	6	10	14	17	19	21	24

- We read through the array until we find the number.
- It requires at least n steps in the worst case.

Example: Running Time of LinearSearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

17

1	2	4	6	10	14	17	19	21	24

- We read through the array until we find the number.
- It requires at least n steps in the worst case.
- Are we using all the information we have?

Example: Running Time of LinearSearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

17	**1**	**2**	**4**	**6**	**10**	**14**	**17**	**19**	**21**	**24**

- We read through the array until we find the number.
- It requires at least n steps in the worst case.
- Are we using all the information we have?
    - We never used the fact that the array is sorted!

Searching faster with **BinarySearch**

- Find if a number **x** exists in an **array** of **sorted numbers**.

17	1	2	4	6	10	14	17	19	21	24

Searching faster with BinarySearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

17

1	2	4	6	10	14	17	19	21	24

Searching faster with BinarySearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

17

1	2	4	6	10	14	17	19	21	24

compare with element n/2

Searching faster with **BinarySearch**

- Find if a number **x** exists in an **array** of **sorted numbers**.

17

1	**2**	**4**	**6**	**10**	**14**	**17**	**19**	**21**	**24**

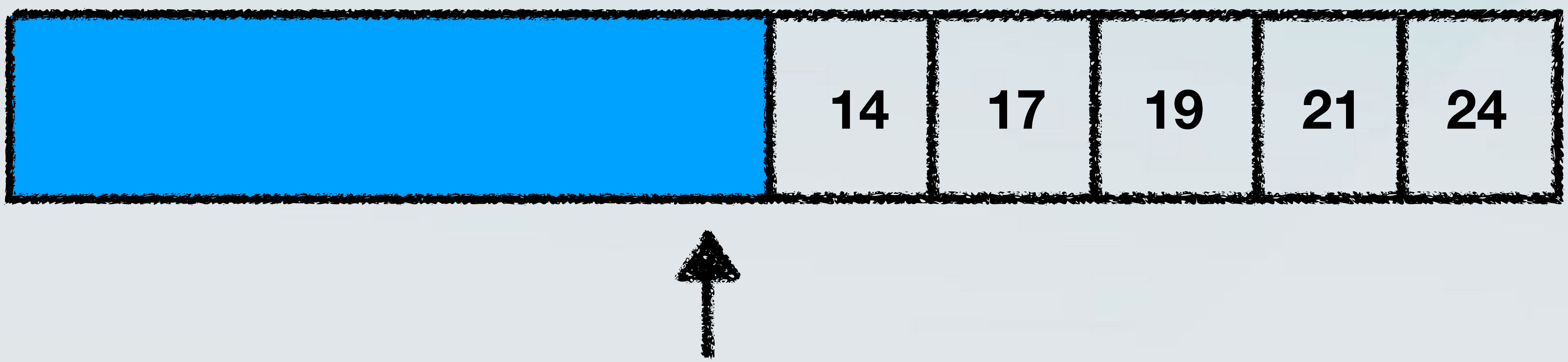
compare with element n/2

is 17 > 10?

Searching faster with BinarySearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

17



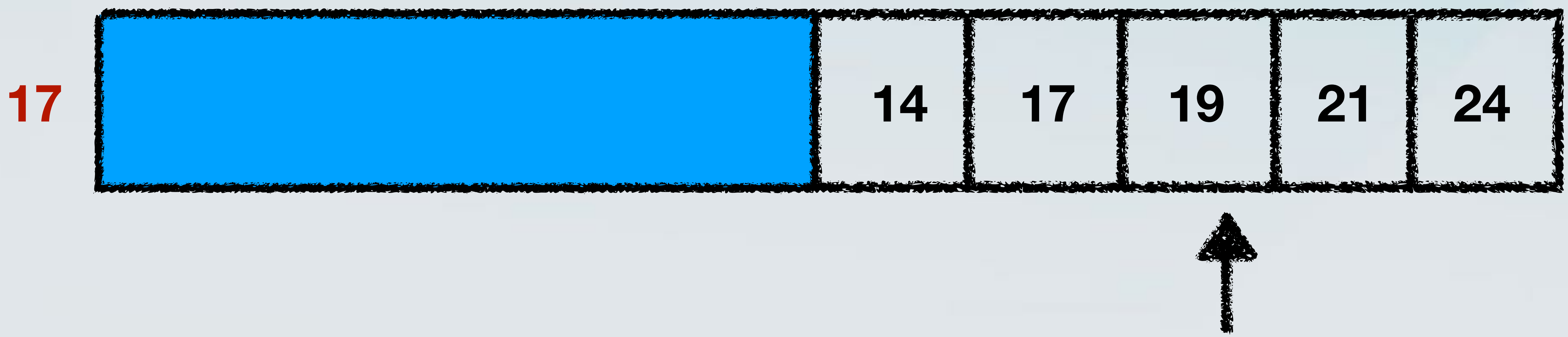
compare with element n/2

is 17 > 10?

We never have to search the blue region again.

Searching faster with **BinarySearch**

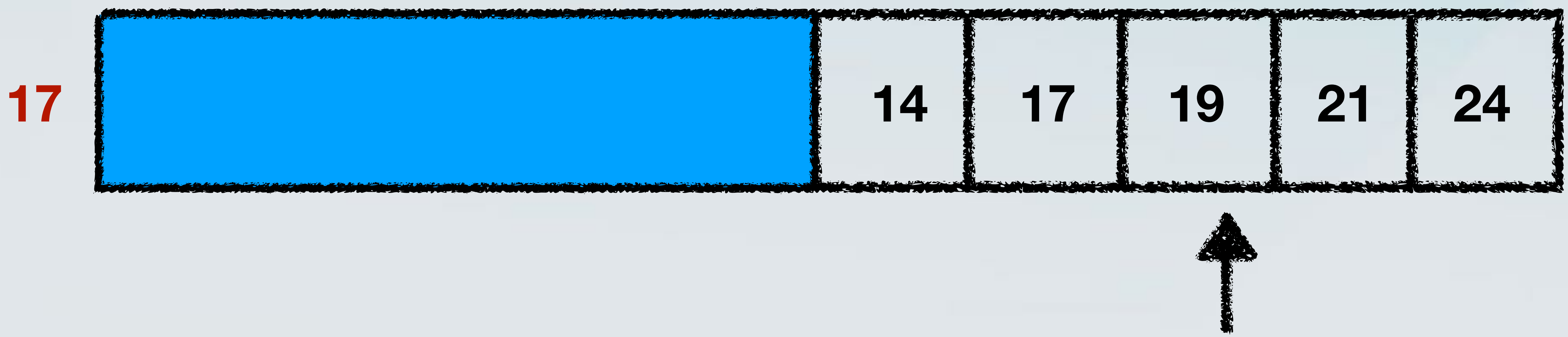
- Find if a number **x** exists in an **array** of **sorted numbers**.



We never have to search the blue region again.

Searching faster with **BinarySearch**

- Find if a number **x** exists in an **array** of **sorted numbers**.

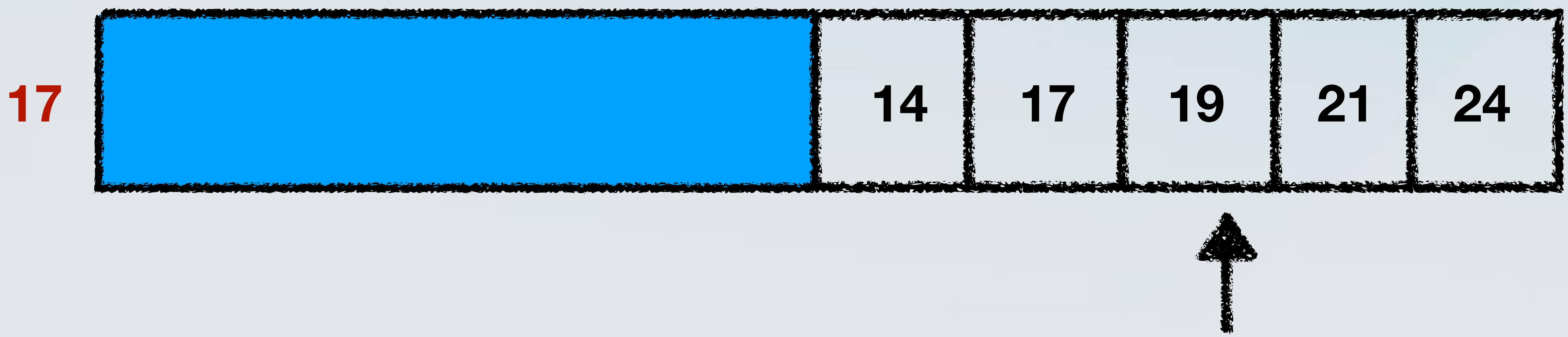


compare with element $n_1/2$

We never have to search the blue region again.

Searching faster with BinarySearch

- Find if a number **x** exists in an **array** of **sorted numbers**.



compare with element $n_1/2$

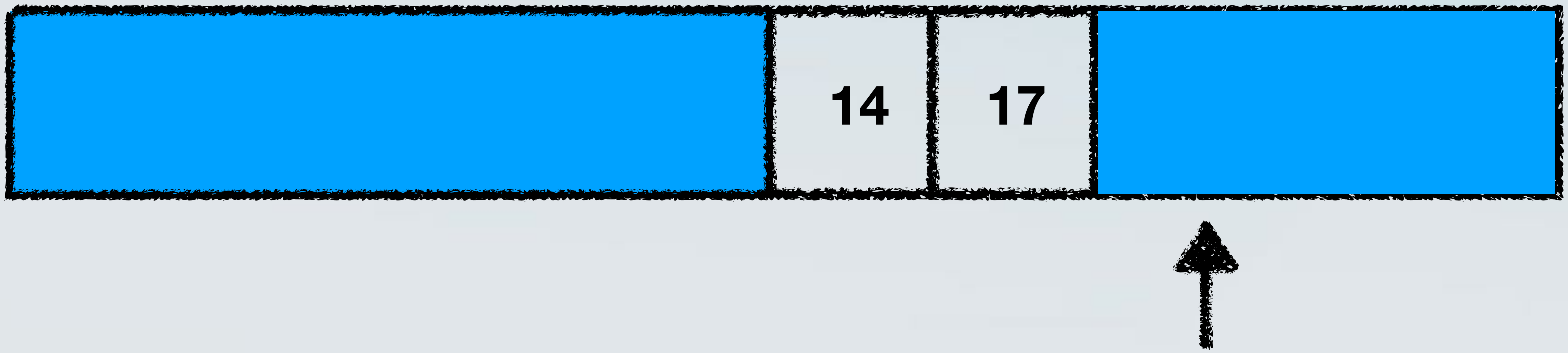
is 14 > 19?

We never have to search the blue region again.

Searching faster with **BinarySearch**

- Find if a number **x** exists in an **array** of **sorted numbers**.

17



compare with element $n_1/2$

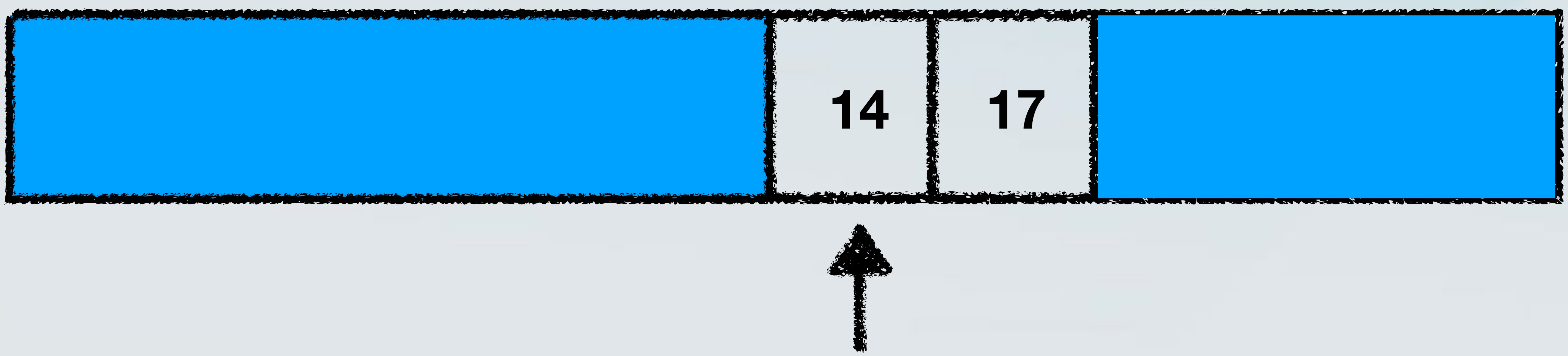
is 14 > 19?

We never have to search the blue region again.

Searching faster with
BinarySearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

17



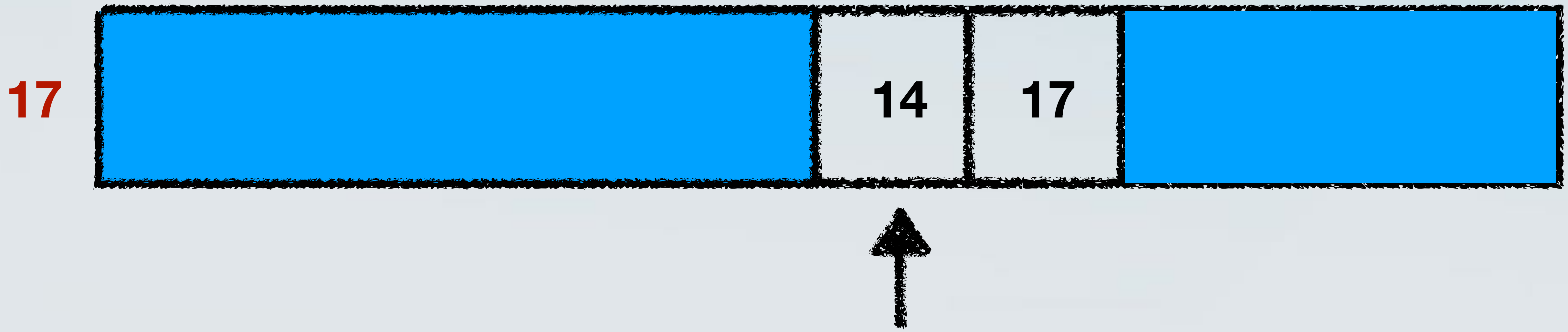
compare with element $n_1/2$

is 14 > 19?

We never have to search the blue region again.

Searching faster with BinarySearch

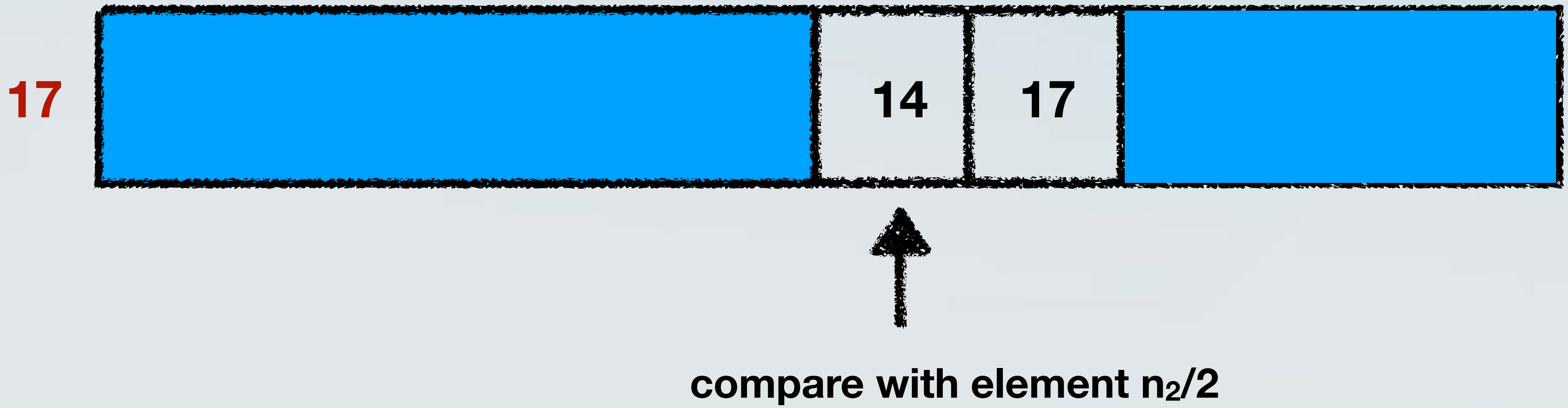
- Find if a number **x** exists in an **array** of **sorted numbers**.



We never have to search the blue region again.

Searching faster with BinarySearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

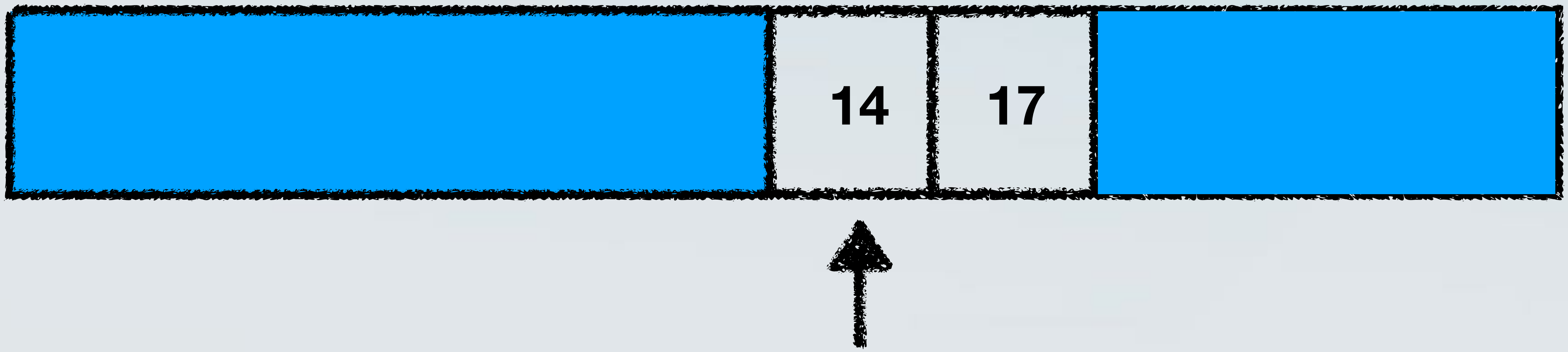


We never have to search the blue region again.

Searching faster with **BinarySearch**

- Find if a number **x** exists in an **array** of **sorted numbers**.

17



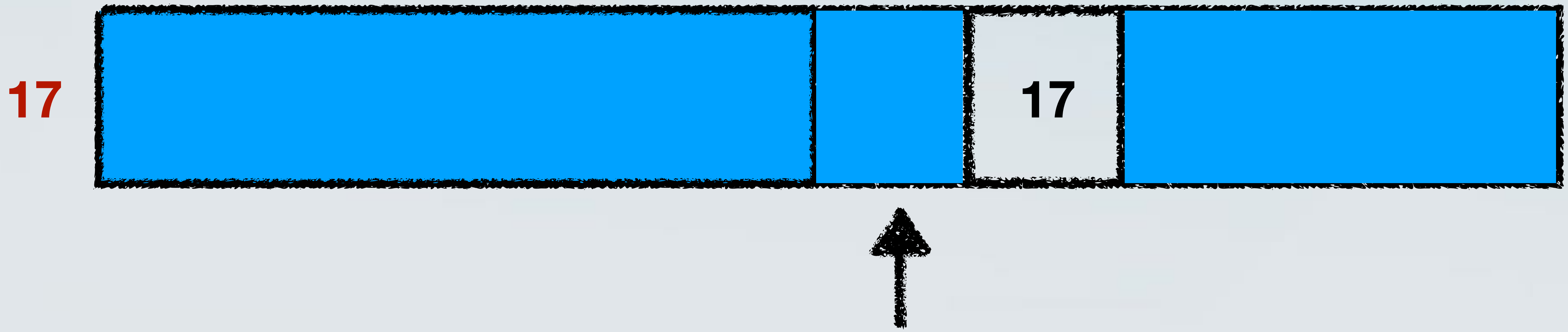
compare with element $n_2/2$

is 14 > 17?

We never have to search the blue region again.

Searching faster with **BinarySearch**

- Find if a number **x** exists in an **array** of **sorted numbers**.



compare with element $n_2/2$

is 14 > 17?

We never have to search the blue region again.

Searching faster with BinarySearch

- Find if a number **x** exists in an **array** of **sorted numbers**.

We never have to search the blue region again.

Searching faster with **BinarySearch**

- Find if a number **x** exists in an **array** of **sorted numbers**.

We never have to search the blue region again.

Searching faster with **BinarySearch**

Searching faster with **BinarySearch**

- How to implement **BinarySearch**?

Searching faster with **BinarySearch**

- How to implement **BinarySearch**?
  - We compare with the middle element, which tells us in which half we might find x.

Searching faster with **BinarySearch**

- How to implement **BinarySearch**?
  - We compare with the middle element, which tells us in which half we might find x.
  - If only we had an algorithm for solving the problem on that half.

Searching faster with **BinarySearch**

- How to implement **BinarySearch**?
  - We compare with the middle element, which tells us in which half we might find x.
  - If only we had an algorithm for solving the problem on that half.
  - Do we know of any such good algorithms?

Searching faster with **BinarySearch**

- How to implement **BinarySearch**?
  - We compare with the middle element, which tells us in which half we might find x.
  - If only we had an algorithm for solving the problem on that half.
  - Do we know of any such good algorithms?
    - **BinarySearch** is such an algorithm! Just run it on half of the array.

Searching faster with **BinarySearch**

- How to implement **BinarySearch**?
  - We compare with the middle element, which tells us in which half we might find x.
  - If only we had an algorithm for solving the problem on that half.
  - Do we know of any such good algorithms?
    - **BinarySearch** is such an algorithm! Just run it on half of the array.
    - We stop running when we reach an array of length 1, which we can trivially check for x.

BinarySearch pseudocode

- **Procedure** BinarySearch(x, *i*, *j*):
    - If $i=j$ then
        - If x = A[*i*], return yes
        - If x ≠ A[*i*], return no
    - Else
        - If x = A[$(i+j)/2$], return yes
        - If x < A[$(i+j)/2$], return BinarySearch(x, *i*, $(i+j)/2 - 1$)
        - If x > A[$(i+j)/2$], return BinarySearch(x, $(i+j)/2$, *j*)

BinarySearch pseudocode

- **Procedure** BinarySearch(x, i, j):
  - If $i=j$ then
    - If x = A[i], return yes
    - If x ≠ A[i], return no
  - Else
    - If x = A[$(i+j)/2$], return yes
    - If x < A[$(i+j)/2$], return BinarySearch(x, i, $(i+j)/2 -1$)
    - If x > A[$(i+j)/2$], return BinarySearch(x, $(i+j)/2$, j)

Then run BinarySearch(x, 1, n)

Design principle

- **Recursion:** A procedure that calls itself one or multiple times, on different inputs.

Running time of BinarySearch

Running time of BinarySearch

- All operations take constant time and there is only a constant number of non-comparison operations.

Running time of BinarySearch

- All operations take constant time and there is only a constant number of non-comparison operations.
- We will measure the number of comparisons.

Running time of BinarySearch

- All operations take constant time and there is only a constant number of non-comparison operations.
- We will measure the number of comparisons.

Running time of BinarySearch

- All operations take constant time and there is only a constant number of non-comparison operations.
- We will measure the number of comparisons.
- Every call of the procedure performs at most 4 comparisons.

BinarySearch pseudocode

- **Procedure** BinarySearch(x, *i*, *j*):
  - If *i*=*j* then
    - If x = A[*i*], return yes
    - If x ≠ A[*i*], return no
  - Else
    - If x = A[*(i+j)/2*], return yes
    - If x < A[*(i+j)/2*], return BinarySearch(x, *i*, *(i+j)/2 -1*)
    - If x > A[*(i+j)/2*], return BinarySearch(x, *(i+j)/2*, *j*)

Running time

Running time

- The number of comparisons performed by **BinarySearch** is

$$T(n) \le T(n/2) + 4$$

Running time

- The number of comparisons performed by **BinarySearch** is

$$T(n) \leq T(n/2) + 4$$

- Let's try to calculate this:

$$\begin{aligned}
T(n) &\leq T(n/2) + 4 \\
&\leq [T(n/4) + 4] + 4 = T(n/4) + 8 \\
&\leq T(n/8] + 12 \\
&\ldots \\
&\leq T(n/2^j) + 4j \\
&\ldots \\
&\leq T(n/2^{\log n -1}) + 4(\log n - 1) \\
&= T[n/(n/2)] + 4(\log n - 1) = T(2) + 4(\log n - 1) \\
&\leq 4 + 4(\log n -1) = 4 \log n
\end{aligned}$$

How to do this formally

- By (strong) **induction:**
  - **Base case:** Show that it holds for input size $n=1$ or $n=2$.
  - **Induction step:** Assume that it holds for all inputs of size at most $n-1$ **(induction hypothesis).**

    Prove that it holds for input size n.

Running time

Running time

- The number of comparisons performed by **BinarySearch** is

$$T(n) \leq T(n/2) + 4$$

Running time

- The number of comparisons performed by **BinarySearch** is

$$T(n) \leq T(n/2) + 4$$

- Let's try to prove that $T(n) \leq 4 \log n$

Running time

- The number of comparisons performed by **BinarySearch** is

$$T(n) \leq T(n/2) + 4$$

- Let's try to prove that $T(n) \leq 4 \log n$
    - **Base Case:** $n=2$, straightforwardly $T(2) \leq 4 \leq 4 \log 2$

Running time

- The number of comparisons performed by **BinarySearch** is

$$T(n) \leq T(n/2) + 4$$

- Let's try to prove that $T(n) \leq 4 \log n$
  - **Base Case:** $n=2$, straightforwardly $T(2) \leq 4 \leq 4 \log 2$
  - **Inductive step:** Assume $T(n/2) \leq 4 \log (n/2)$

Running time

- The number of comparisons performed by **BinarySearch** is

$$T(n) \leq T(n/2) + 4$$

- Let's try to prove that $T(n) \leq 4 \log n$
  - **Base Case:** $n=2$, straightforwardly $T(2) \leq 4 \leq 4 \log 2$
  - **Inductive step:** Assume $T(n/2) \leq 4 \log (n/2)$
    - It holds that $T(n) \leq T(n/2) + 4 \leq 4\log(n/2) + 4$
      $\leq 4\log n - 4\log 2 + 4 \leq 4\log n$

Divide-and-Conquer

Divide-and-Conquer

- Split the input into smaller sub-instances.

Divide-and-Conquer

- Split the input into smaller sub-instances.
- Solve each sub-instance separately.

Divide-and-Conquer

- Split the input into smaller sub-instances.
- Solve each sub-instance separately.
- Combine the solutions of the sub-instances into a solution for the problem.

Divide-and-Conquer

- Split the input into smaller sub-instances.
- Solve each sub-instance separately.
- Combine the solutions of the sub-instances into a solution for the problem.
- **Often:** For each sub-instance, the algorithm calls itself to solve it **(recursion).**

  The instances become so small that they can be solved via a **brute force** algorithm.

Question

- Could we have stopped the **BinarySearch** procedure earlier and used brute-force on the remaining sequence without changing its asymptotic running time?

  How much earlier?

Memory requirements of BinarySearch

- Memory used as part of the input:
  n (to store the array) + *1* (to store the number x).

- Auxiliary memory:

  - The algorithm calls itself within its execution.

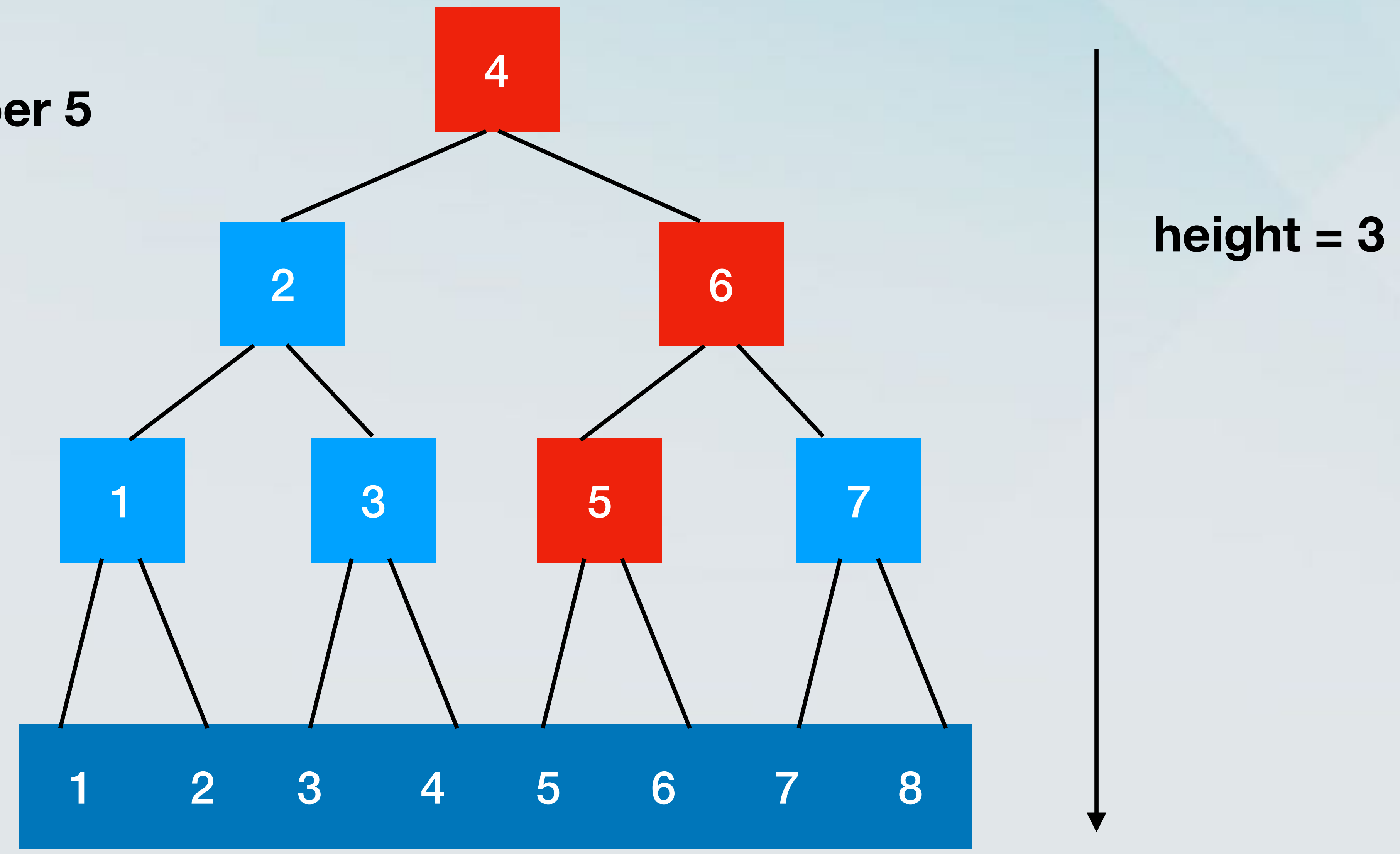
  - Needs to maintain these executions “active” in memory.

  - How many executions do we have?

    - O(log n).

Tree structure

Search for number 5



8 leaves

We have to store the path from the root to the leaf.

BinarySearch vs LinearSearch

<table><tr><td>

<u>BinarySearch</u>

Running time: O(log n)

Memory: O(log n)

</td><td>

<u>LinearSearch</u>

Running time: O(n)

Memory: O(1)

</td></tr></table>

Which one we choose depends on the application.

Finding majority in an array

- Given an array of n numbers, a majority element is one that appears more than $n/2$ times in the array.
    - (Ignoring rounding issues, otherwise ceil($n/2$) times).
- **Question:** Given such an array, find a majority element if it exists, or return that it doesn't.

Majority pseudocode

- Algorithm **Majority**(**A**[*1,…,n*])
  - If |**A**| = 0 output no, if |**A**| = 1 output **A**[i].
  - (Assume n = |A| is even).
  - Initialise array **B** of size |A|/2.
  - Set *j*=0
  - For i = 1 to n/2, do
    - if **A**[*2i-1*] = **A**[*2i*] then
      - *j*=*j*+1
      - **B**[*j*] = **A**[*2i*]

- **Majority**(**B**[*1,…,j*])
- If **B**[*1,…,j*] returns a value x
  - Iterate through the array **A** and count the number of occurrences of x.
  - if these are more at least n/2, output x.
  - else, output no.

Majority Example

1	10	10	7	10	10	6	6	10	10

Majority Example

1	10	10	7	10	10	6	6	10	10

- Algorithm Majority(**A**[$1,\ldots,n$])
  - If |**A**| = 0 output no, if |**A**| = 1 output **A**[i].
  - (Assume n = |A| is even).

Majority Example

1	10	10	7	10	10	6	6	10	10

- Algorithm Majority(**A**[*1*,…,*n*])
  - If |**A**| = 0 output no, if |**A**| = 1 output **A**[i].
  - (Assume n = |A| is even).

Majority Example

1	10	10	7	10	10	6	6	10	10

- Algorithm Majority(**A**[$1,\ldots,n$])
  - If |**A**| = 0 output no, if |**A**| = 1 output **A**[i].
  - (Assume n = |A| is even).

Majority Example

1	10	10	7	10	10	6	6	10	10

Majority Example

1	10	10	7	10	10	6	6	10	10

- Initialise array **B** of size |A|/2.
- Set *j*=0

Majority Example

1	10	10	7	10	10	6	6	10	10

- Initialise array **B** of size |A|/2.
- Set j=0

Majority Example

1	10	10	7	10	10	6	6	10	10

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[*2i-1*] = **A**[*2i*] then
    - *j*=*j*+1
    - **B**[*j*] = **A**[*2i*]

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[*2i-1*] = **A**[*2i*] then
    - *j*=*j*+1
    - **B**[*j*] = **A**[*2i*]

i=1
A[1]=1
A[2]=10

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[*2i-1*] = **A**[*2i*] then
    - *j*=*j*+1
    - **B**[*j*] = **A**[*2i*]

i=1
A[1]=1
A[2]=10

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[*2i-1*] = **A**[*2i*] then
    - *j*=*j*+1
    - **B**[*j*] = **A**[*2i*]

i=1
A[1]=1
A[2]=10

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[*2i-1*] = **A**[*2i*] then
    - *j*=*j*+1
    - **B**[*j*] = **A**[*2i*]

i=1
A[1]=1
A[2]=10

i=2
A[3]=10
A[4]=7

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[*2i-1*] = **A**[*2i*] then
    - *j*=*j*+1
    - **B**[*j*] = **A**[*2i*]

i=1
A[1]=1
A[2]=10

i=2
A[3]=10
A[4]=7

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[$2i$-1] = **A**[$2i$] then
    - j=j+1
    - **B**[j] = **A**[$2i$]

i=1	i=2
A[1]=1	A[3]=10
A[2]=10	A[4]=7

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[$2i-1$] = **A**[$2i$] then
    - j=j+1
    - **B**[j] = **A**[$2i$]

i=1
A[1]=1
A[2]=10

i=2
A[3]=10
A[4]=7

i=3
A[5]=10
A[6]=10

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[*2i-1*] = **A**[*2i*] then
    - *j*=*j*+1
    - **B**[*j*] = **A**[*2i*]

i=1
A[1]=1
A[2]=10

i=2
A[3]=10
A[4]=7

i=3
A[5]=10
A[6]=10

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[*2i-1*] = **A**[*2i*] then
    - *j*=*j*+1
    - **B**[*j*] = **A**[*2i*]

i=1
A[1]=1
A[2]=10

i=2
A[3]=10
A[4]=7

i=3
A[5]=10
A[6]=10

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
    - if **A**[2*i*-1] = **A**[2*i*] then
        - *j*=*j*+1
        - **B**[*j*] = **A**[2*i*]

i=1
A[1]=1
A[2]=10

i=2
A[3]=10
A[4]=7

i=3
A[5]=10
A[6]=10

10				

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[*2i-1*] = **A**[*2i*] then
    - *j*=*j*+1
    - **B**[*j*] = **A**[*2i*]

i=1
A[1]=1
A[2]=10

i=2
A[3]=10
A[4]=7

i=3
A[5]=10
A[6]=10

i=4
A[7]=6
A[8]=6

10				

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[*2i-1*] = **A**[*2i*] then
    - *j*=*j*+1
    - **B**[*j*] = **A**[*2i*]

i=1
A[1]=1
A[2]=10

i=2
A[3]=10
A[4]=7

i=3
A[5]=10
A[6]=10

i=4
A[7]=6
A[8]=6

10				

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[*2i-1*] = **A**[*2i*] then
    - *j*=*j*+1
    - **B**[*j*] = **A**[*2i*]

i=1
A[1]=1
A[2]=10

i=2
A[3]=10
A[4]=7

i=3
A[5]=10
A[6]=10

i=4
A[7]=6
A[8]=6

10				

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[*2i-1*] = **A**[*2i*] then
    - *j*=*j*+1
    - **B**[*j*] = **A**[*2i*]

10	6			

i=1
A[1]=1
A[2]=10

i=2
A[3]=10
A[4]=7

i=3
A[5]=10
A[6]=10

i=4
A[7]=6
A[8]=6

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[$2i-1$] = **A**[$2i$] then
    - $j=j+1$
    - **B**[j] = **A**[$2i$]

10	6			

i=1
A[1]=1
A[2]=10

i=2
A[3]=10
A[4]=7

i=3
A[5]=10
A[6]=10

i=4
A[7]=6
A[8]=6

i=5
A[9]=10
A[10]=10

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[*2i-1*] = **A**[*2i*] then
    - *j*=*j*+1
    - **B**[*j*] = **A**[*2i*]

10	6			

i=1
A[1]=1
A[2]=10

i=2
A[3]=10
A[4]=7

i=3
A[5]=10
A[6]=10

i=4
A[7]=6
A[8]=6

i=5
A[9]=10
A[10]=10

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[*2i-1*] = **A**[*2i*] then
    - *j*=*j*+1
    - **B**[*j*] = **A**[*2i*]

10	6			

i=1
A[1]=1
A[2]=10

i=2
A[3]=10
A[4]=7

i=3
A[5]=10
A[6]=10

i=4
A[7]=6
A[8]=6

i=5
A[9]=10
A[10]=10

Majority Example

1	10	10	7	10	10	6	6	10	10

- For i = 1 to n/2, do
  - if **A**[2*i*-*1*] = **A**[2*i*] then
    - *j*=*j*+1
    - **B**[*j*] = **A**[2*i*]

10	6	10		

i=1
A[1]=1
A[2]=10

i=2
A[3]=10
A[4]=7

i=3
A[5]=10
A[6]=10

i=4
A[7]=6
A[8]=6

i=5
A[9]=10
A[10]=10

Majority Example

1	10	10	7	10	10	6	6	10	10

10	6	10		

Majority Example

1	10	10	7	10	10	6	6	10	10

- Majority(**B**[*1,...,j*])

10	6	10		

Majority Example

- Majority(**B**[*1,...,j*])

10	6	10

Majority Example

- Majority(**B**[$1,\ldots,j$])

10	6	10

- Algorithm Majority(**A**[$1,\ldots,n$])
  - If |**A**| = 0 output no, if |**A**| = 1 output **A**[i].
  - (Assume n = |A| is even).

Majority Example

- Majority(**B**[$1,\ldots,j$])

10	6	10

- Algorithm Majority(**A**[$1,\ldots,n$])
  - If |**A**| = 0 output no, if |**A**| = 1 output **A**[i].
  - (Assume n = |A| is even).

Majority Example

- Majority(**B**[*1,...,j*])

10	6	10

- Algorithm Majority(**A**[*1,...,n*])
  - If |**A**| = 0 output no, if |**A**| = 1 output **A**[i].
  - (Assume n = |A| is even).

Majority pseudocode

- Algorithm **Majority**(**A**[$1,\ldots,n$])
  - If |**A**| = 0 output no, if |**A**| = 1 output **A**[i].
  - **Check if A[n] is a majority**
    - **Count the number of occurrences.**
    - **Discard it if it is not.**
  - Initialise array **B** of size |A|/2.
  - Set j=0
  - For i = 1 to n/2, do
    - if **A**[$2i-1$] = **A**[$2i$] then
      - j=j+1
      - **B**[j] = **A**[$2i$]

- **Majority**(**B**[$1,\ldots,j$])
- If **B**[$1,\ldots,j$] returns a value x
  - Iterate through the array **A** and count the number of occurrences of x.
  - if these are more at least n/2, output x.
  - else, output no.

Majority Example

- Majority(**B**[$1,\ldots,j$])

10	6	10

- Algorithm Majority(**A**[$1,\ldots,n$])
  - If |**A**| = 0 output no, if |**A**| = 1 output **A**[i].
  - (Assume n = |A| is even).

Majority Example

- Majority(**B**[*1,...,j*])

10	6	10

Majority Example

- Majority(**B**[*1*,...,*j*])

10	6	10

- Check if A[*n*] is a majority
  - Count the number of occurrences.
  - Discard it if it is not.

Majority Example

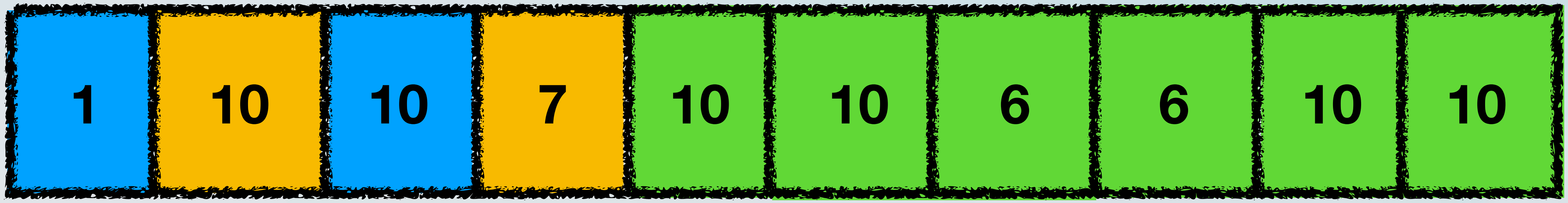
- Majority(**B**[$1,\ldots,j$])

The function returns **10.**

10	6	10

- **Check if A[n] is a majority**
  - **Count the number of occurrences.**
  - **Discard it if it is not.**

Majority Example



- Majority(**B**[*1,...,j*])

The function returns **10.**

Majority Example

1	10	10	7	10	10	6	6	10	10

- Majority(**B**[*1,…,j*])

The function returns **10.**

- If **B**[*1,…,j*] returns a value x
  - Iterate through the array **A** and count the number of occurrences of x.
  - if these are more at least n/2, output x.
  - else, output no.

Majority Example

1	10	10	7	10	10	6	6	10	10

- Majority(**B**[*1,...,j*])

The function returns **10.**

- If **B**[*1,...,j*] returns a value x
  - Iterate through the array **A** and count the number of occurrences of x.
  - if these are more at least n/2, output x.
  - else, output no.

10 appears 6 times in the array.
Majority element!

Correctness

Correctness

Lemma: If **x** is a majority element in **A**, then **x** is a majority element in **B**.

Correctness

Lemma: If **x** is a majority element in **A**, then **x** is a majority element in **B**.

Proof by **induction:**

Correctness

Lemma: If **x** is a majority element in **A**, then **x** is a majority element in **B**.

Proof by **induction:**

Base case: Majority(**B**) works correctly for array **B** of size 1.

Correctness

Lemma: If **x** is a majority element in **A**, then **x** is a majority element in **B**.

Proof by **induction:**

Base case: **Majority**(**B**) works correctly for array **B** of size 1.

Inductive step: Assume that **Majority**(**B**) works correctly for array **B** of size smaller than |**A**| **(inductive hypothesis).**

Correctness

Lemma: If **x** is a majority element in **A**, then **x** is a majority element in **B**.

Proof by **induction:**

Base case: Majority(**B**) works correctly for array **B** of size 1.

Inductive step: Assume that Majority(**B**) works correctly for array **B** of size smaller than |**A**| **(inductive hypothesis).**

Case 1 (There is a majority element in A): Then by the Lemma, it is also a majority element in **B**. Majority(**B**) will output it, by the inductive hypothesis and the last step of Majority(**A**) will output it.

Correctness

Lemma: If **x** is a majority element in **A**, then **x** is a majority element in **B**.

Proof by **induction:**

Base case: Majority(**B**) works correctly for array **B** of size 1.

Inductive step: Assume that Majority(**B**) works correctly for array **B** of size smaller than |**A**| **(inductive hypothesis).**

Case 1 (There is a majority element in A): Then by the Lemma, it is also a majority element in **B**. Majority(**B**) will output it, by the inductive hypothesis and the last step of Majority(**A**) will output it.

Case 2 (There is not a majority element in A): Then the last step of Majority(**A**) will reject any candidate majority elements returned from Majority(**B**).

Proof by contradiction

Proof by contradiction

- We want to prove that statement **S** is true.
- We assume that the statement is not true.
- We reach a conclusion which cannot possibly be true.

Proof of the lemma

Proof of the lemma

- **Assumption**: Suppose to the ***contrary***, that **x** is a majority element in **A** but ***not*** a majority element in **B**.

Proof of the lemma

- **Assumption**: Suppose to the ***contrary***, that **x** is a majority element in **A** but ***not*** a majority element in **B**.
  - Let **m** be the number of occurrences of **x** in **A**.

Proof of the lemma

- **Assumption**: Suppose to the ***contrary***, that **x** is a majority element in **A** but ***not*** a majority element in **B**.
  - Let **m** be the number of occurrences of **x** in **A**.
  - Let **k** be the number of occurrences of **x** in **B**.

Proof of the lemma

- **Assumption**: Suppose to the ***contrary***, that **x** is a majority element in **A** but ***not*** a majority element in **B**.
  - Let **m** be the number of occurrences of **x** in **A**.
  - Let **k** be the number of occurrences of **x** in **B**.
- By the **assumption**, it follows that other values appear at least **k** times in **B**.

Proof of the lemma

- **Assumption**: Suppose to the ***contrary***, that **x** is a majority element in **A** but ***not*** a majority element in **B**.
  - Let **m** be the number of occurrences of **x** in **A**.
  - Let **k** be the number of occurrences of **x** in **B**.
- By the **assumption**, it follows that other values appear at least **k** times in **B**.
- This means that other values appear in **A**:

Proof of the lemma

- **Assumption**: Suppose to the ***contrary***, that **x** is a majority element in **A** but ***not*** a majority element in **B**.
  - Let **m** be the number of occurrences of **x** in **A**.
  - Let **k** be the number of occurrences of **x** in **B**.
- By the **assumption**, it follows that other values appear at least **k** times in **B**.
- This means that other values appear in **A**:
  - at least **2k** times from the pairs that are represented in **B** by a value different than **x** *plus*

Proof of the lemma

- **Assumption**: Suppose to the ***contrary***, that **x** is a majority element in **A** but ***not*** a majority element in **B**.
  - Let **m** be the number of occurrences of **x** in **A**.
  - Let **k** be the number of occurrences of **x** in **B**.
- By the **assumption**, it follows that other values appear at least **k** times in **B**.
- This means that other values appear in **A**:
  - at least **2k** times from the pairs that are represented in **B** by a value different than **x** *plus*
  - **m-2k** times, since each occurrence of **x** in **A** that is not paired with another **x** is paired with some other value (since there are **2k** pairs **xx**, there are **m-2k** other occurrences of **x** in **A**).

Proof of the lemma

- **Assumption**: Suppose to the ***contrary***, that **x** is a majority element in **A** but ***not*** a majority element in **B**.
  - Let **m** be the number of occurrences of **x** in **A**.
  - Let **k** be the number of occurrences of **x** in **B**.
- By the **assumption**, it follows that other values appear at least **k** times in **B**.
- This means that other values appear in **A**:
  - at least **2k** times from the pairs that are represented in **B** by a value different than **x** *plus*
  - **m-2k** times, since each occurrence of **x** in **A** that is not paired with another **x** is paired with some other value (since there are **2k** pairs **xx**, there are **m-2k** other occurrences of **x** in **A**).
- In total, this gives **2k+(m-k) = m** occurrences, which contradicts the fact that **x** is a majority in **A**. **Contradiction!**

Running time of Majority

Majority pseudocode

- Algorithm Majority(**A**[*1,…,n*])
  - If |**A**| = 0 output no, if |**A**| = 1 output **A**[i].
  - Check if **A**[*n*] is a majority
    - Count the number of occurrences.
    - Discard it if it is not.
  - Initialise array **B** of size |A|/2.
  - Set *j*=0
  - For i = 1 to n/2, do
    - if **A**[*2i-1*] = **A**[*2i*] then
      - *j*=*j*+1
      - **B**[*j*] = **A**[*2i*]

- Majority(**B**[*1,…,j*])
- If **B**[*1,…,j*] returns a value x
  - Iterate through the array **A** and count the number of occurrences of x.
  - if these are more at least n/2, output x.
  - else, output no.

Running time of Majority

Running time of Majority

- Recursive formula for the running time:

Running time of Majority

- Recursive formula for the running time:
    - $T(n) \le T(n/2) + cn$ for some constant c

Running time of Majority

- Recursive formula for the running time:
  - $T(n) \leq T(n/2) + cn$ for some constant c
  - We will prove that $T(n) \leq 2cn$

Running time of Majority

- Recursive formula for the running time:
  - $T(n) \leq T(n/2) + cn$ for some constant c
  - We will prove that $T(n) \leq 2cn$
- By **induction:**

Running time of Majority

- Recursive formula for the running time:
    - $T(n) \le T(n/2) + cn$ for some constant c
    - We will prove that $T(n) \le 2cn$
- By **induction:**
    - **Base case:** $T(1) \le c$

Running time of Majority

- Recursive formula for the running time:
  - $T(n) \le T(n/2) + cn$ for some constant c
  - We will prove that $T(n) \le 2cn$
- By **induction:**
  - **Base case:** $T(1) \le c$
  - **Induction step:** Assume that $T(n/2) \le 2c(n/2)$ **(induction hypothesis).**

    We have that $T(n) \le T(n/2) + cn \le 2c(n/2) + cn = 2cn$