RECURRENCE ANALYSIS

[CLRS SEC. 4.5]

METHODS

- ✓**Substitution Method:** Guessing the closed form using substitution and then using induction to prove.
- ✓**Recursion Tree:** Guessing the closed form using the recurrence tree and then using induction to prove.
- ➢ **Master Theorem:** Finding asymptotic bounds of some recurrences using predefined rules.

MASTER THEOREM

For some classes of recurrences which follow a **certain structure,** Master Theorem provides us with the solution.

MASTER THEOREM

Consider $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, $a \geq 1, b > 1$, $\frac{n}{b}$ can be $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$.

1. If $f(n) = O\left(n^{(\log_b a)-\epsilon}\right)$ for some constant $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \log n\right)$.
3. If $f(n) = \Omega\left(n^{(\log_b a)+\epsilon}\right)$ for some constant $\epsilon > 0$ and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$ and $n > n_0$, then $T(n) = \Theta\left(f(n)\right)$.

Note: A more general form of case 2 is $f(n) = \Theta\left(n^{\log_b a}(\log n)^k\right)$, for integer $k \geq 0$, then $T(n) = \Theta\left(n^{\log_b a}(\log n)^{k+1}\right)$.

EXAMPLES

Example 1: $T(n) = 2T\left(\frac{n}{2}\right) + n.$

- Here, $a = 2, b = 2, f(n) = n$. Hence, $n^{\log_b a} = n.$
- Therefore, case 2 of Master Theorem applies, and $T(n) = \Theta(n \log n).$

EXAMPLES

Example 2: $T(n) = 3T\left(\frac{n}{4}\right) + n\log n.$

- Here, $a = 3, b = 4, f(n) = n\log(n).$
- Notice that $n^{\log_b a} = n^{\log_4 3} \approx n^{0.792}$, which means $f(n) = n\log(n) = \Omega\left(n^{(\log_4 3)+\epsilon}\right)$ for $\epsilon \approx 0.1$ since $\lim_{n\to\infty} \frac{n\log n}{n^{(\log_4 3)+\epsilon}} = \infty.$
- Thus, we are in case 3 and check if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$ and $n > n_0$.

$af\left(\frac{n}{b}\right) = 3\frac{n}{4}\log\left(\frac{n}{4}\right) = \frac{3}{4}n(\log n - \log 4) = \frac{3}{4}n\log n - \frac{3}{2}n \leq cf(n)$ for $c = \frac{3}{4}$ and any $n \geq 1$.

- Therefore, $T(n) = \Theta(n\log n).$

EXAMPLES

Example 3: $T(n) = 2T\left(\frac{n}{2}\right) + n\log n.$

- Here, $a = 2, b = 2, f(n) = n\log n.$
- Since $n^{\log_b a} = n^{\log_2 2} = n.$
- It might seem we are in case 3.
- However, case 3 can't be used (no matter how small we choose ϵ to be, it doesn't work since $\lim_{n\to\infty} \frac{n^{1+\epsilon}}{n\log n} = \lim_{n\to\infty} \frac{n^{\epsilon}}{\log n} = \infty$).
- Instead, we can use the general form of case 2 with $k = 1$ and hence, $T(n) = \Theta(n(\log n)^2).$

PROOF OF MASTER THEOREM?

We'll not cover the proof of Master Theorem in this class, but if you're interested, please read it in Section 4.6 of CLRS.

The recursion tree and summation of sequences we've covered so far should enable you to follow the proof.

THREE METHODS FOR RECURRENCE ANALYSIS

- ✓Substitution Method
- ✓Recursion Tree
- ✓Master Theorem

SEARCH PROBLEMS

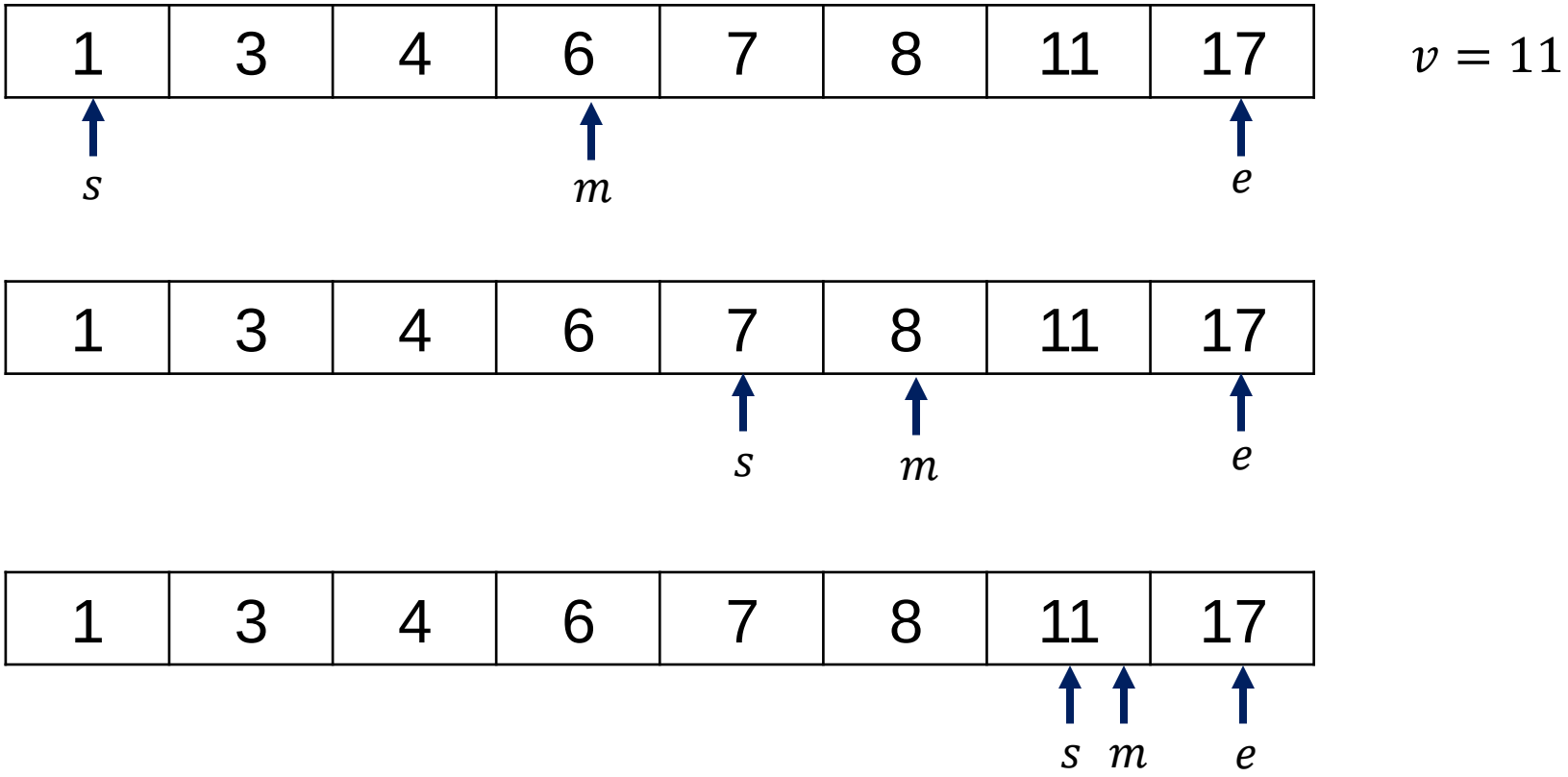
BINARY SEARCH

Input: An array of integers A sorted in ascending order and a target integer v.

Output: The index of v if in A, and *null* otherwise.

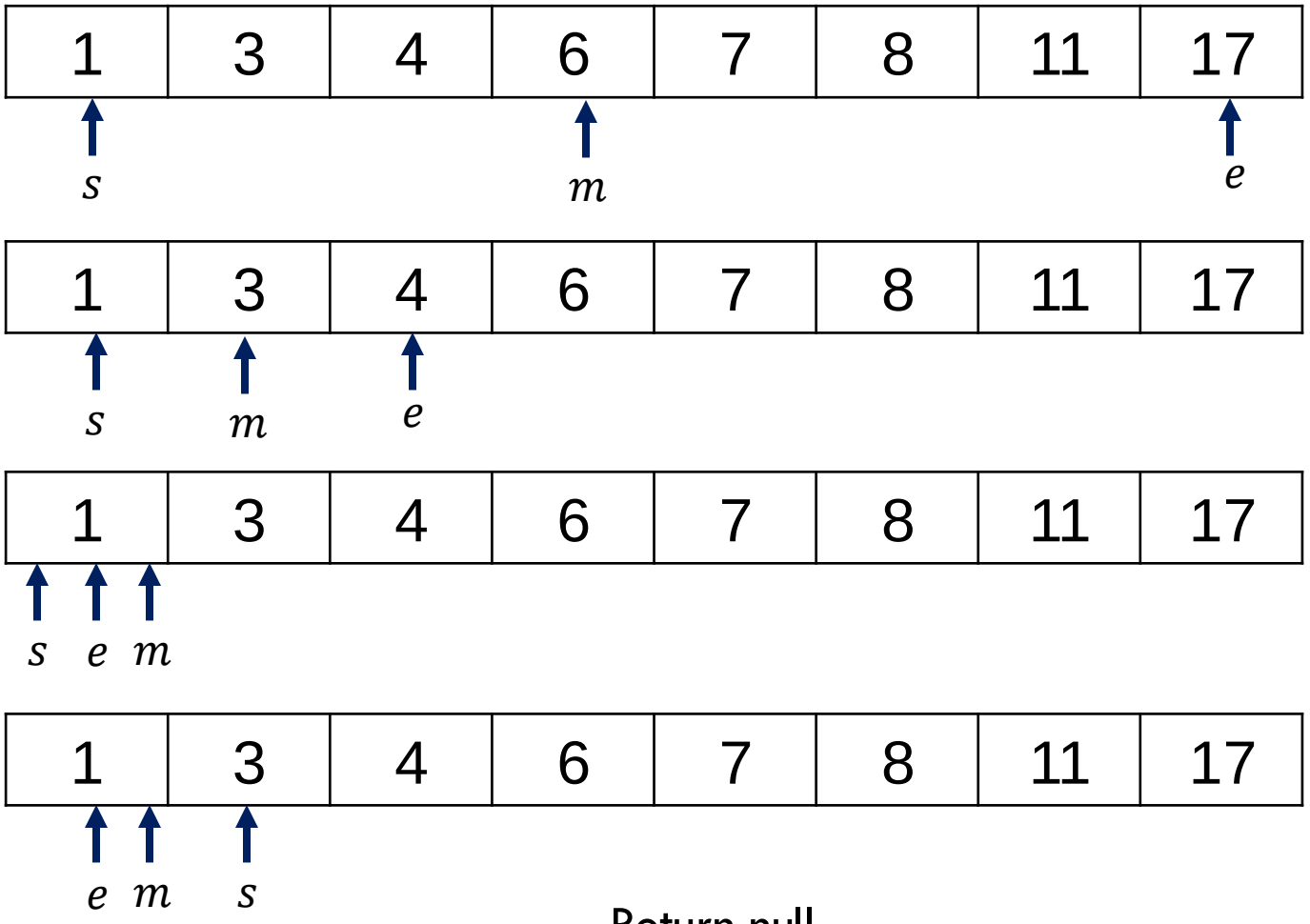
```
BinarySearch(A,v)
s=1
e=n
while s <= e
    m = ⌊(s+e)/2⌋
    if A[m] == v then
        return m
    else if A[m]<v then
        s=m+1
    else
        e=m-1
return null
```

BINARY SEARCH



Return 7

BINARY SEARCH



$v = 2$

Return null

BINARY SEARCH

Statement. Let $T(n)$ be the run time of Binary Search (in the worst case). Then, $T(n) = O(\log n)$.

You will prove this in your weekly exercises.

MINIMUM IN ROTATED SORTED ARRAY

Input: Consider an array of length n with unique elements and sorted in ascending order is **rotated** between 1 and n times.

For example, the array $A = [1,2,3,4,5,6]$ might become:

- $[3,4,5,6,1,2]$ if it was rotated 4 times.
- $[1,2,3,4,5,6]$ if it was rotated 6 times.

Output: The minimum element in the array.

BINARY SEARCH APPROACH

5	6	7	1	2	3	4

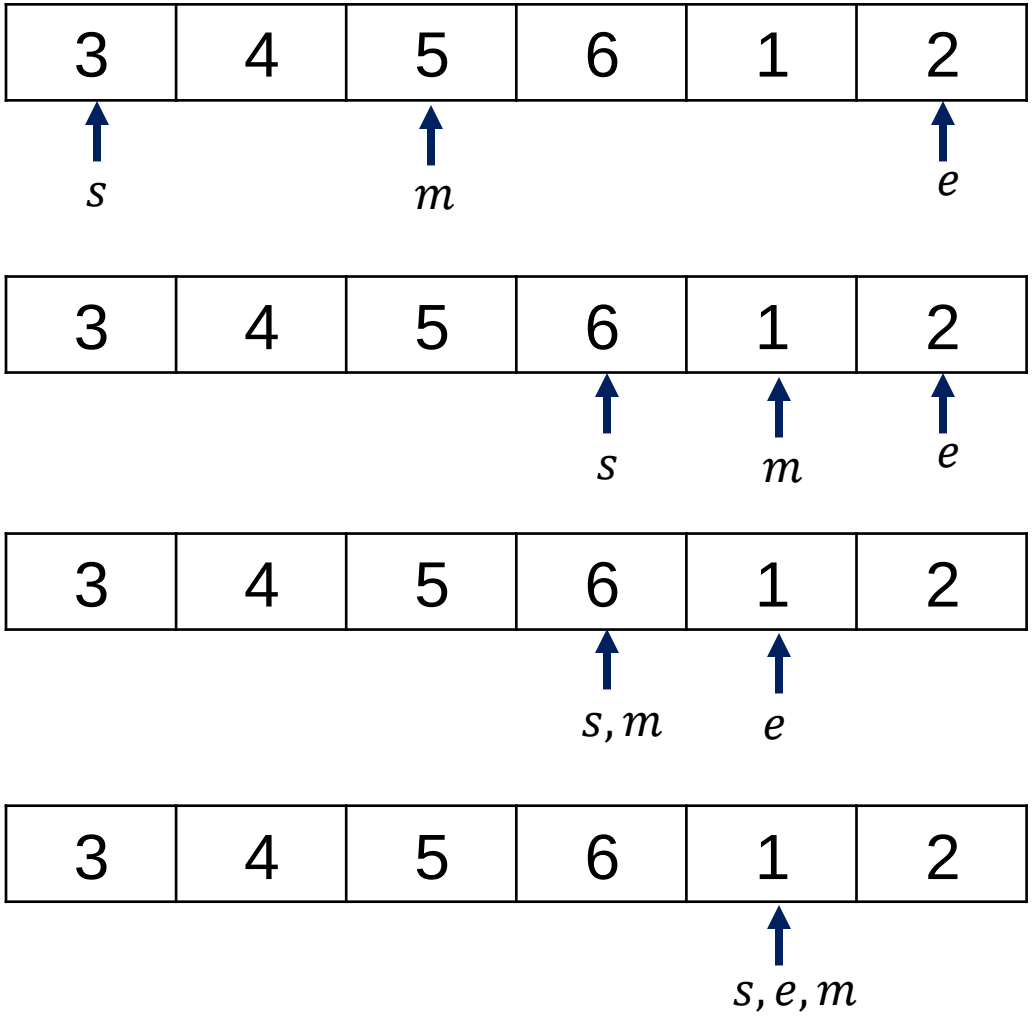
- The idea is to use some form of binary search to (almost) half the search space each time.
- The key point is if we compare the middle element with the rightmost (or leftmost element), we can decide in which half we should continue looking.
- If the middle element is larger than the rightmost element, we look into the right half. Otherwise, we look into the left half.

Why does this work?

ALGORITHM

```
FindMin(A)
s=1, e=n
while s<e
    m=⌊(s+e)/2⌋
    if A[m]>A[e]
        s=m+1
    else
        e=m
return A[s]
```

EXAMPLE



RUN TIME

What is the run time?

Similar to the binary search, it is easy to prove that the run time is in $O(\log n)$.

TWO SUM PROBLEM

Input: An array of integers A sorted in ascending order and a target integer v.

Output: Yes, if there exist a and b in A such that $a + b = v$.

Example:

1	3	4

$v = 5$: Yes, $a = 1$ and $b = 4$.

$v = 3$, No.

$v = 2$, Yes, $a = b = 1$.

TWO SUM PROBLEM: BRUTE FORCE

Brute Force Approach: Check all possible choices of a and b. That is, loop through the array twice and check every possible choice of a and b.

```
for i = 1 to n
    for j = i to n
        if A[i]+A[j] = v
            return Yes
return No
```

It is easy to prove that $T(n) = \Theta(n^2)$.

IMPORTANT NOTE

We do not provide formal proof for the run-time analysis of the algorithms in today's lecture. However, you should be able to do that using materials from the last two weeks.

TWO SUM PROBLEM: BINARY SEARCH

Binary Search Approach: Fix a and use Binary Search to check if $v - a$ is in A.

```
for i=1 to n
    if BinarySearch(A,v-A[i]) not null
        return Yes
return No
```

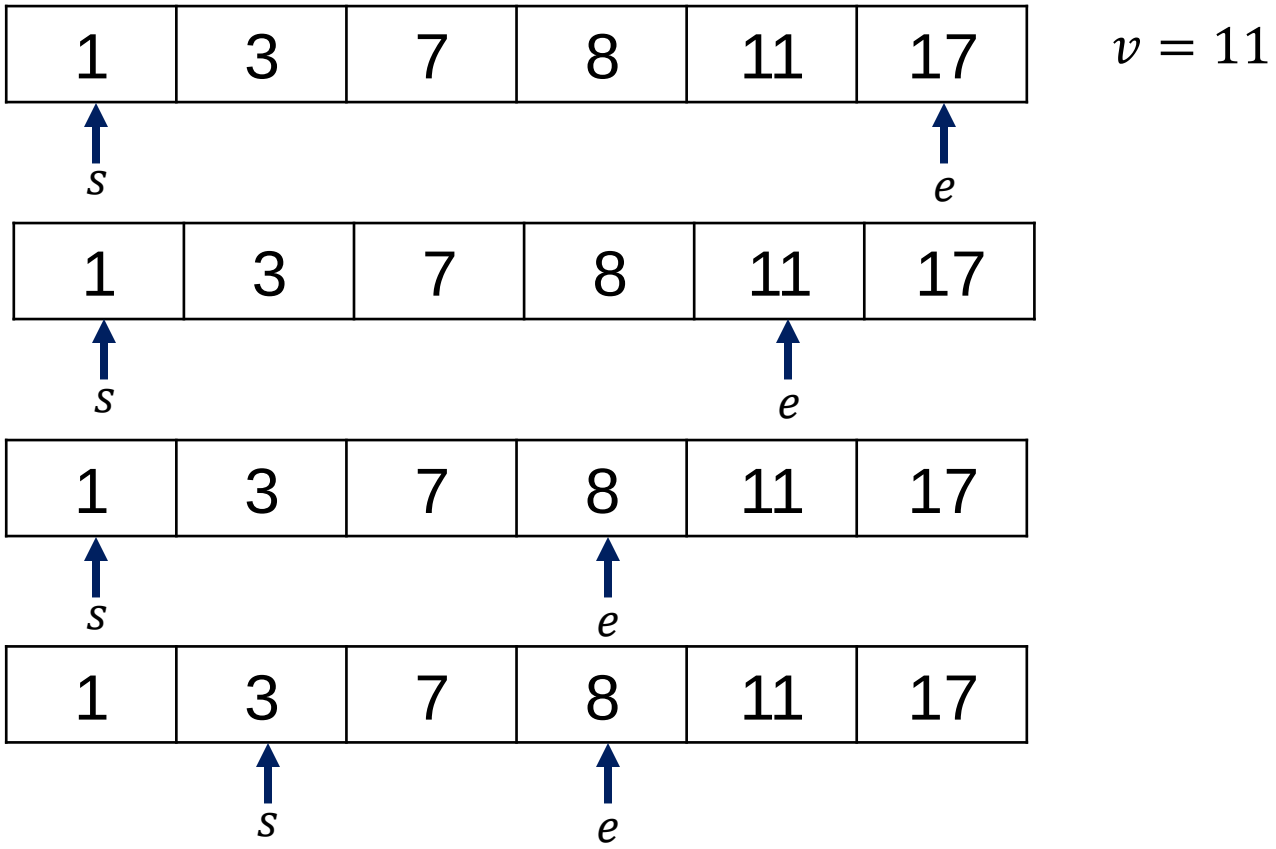
$T(n) = O(n \log n)$.

Faster, but can we do better?

TWO SUM PROBLEM: LINEAR ALGORITHM

```
int s = 1
int e = n
while s <= e
    sum= A[s]+A[e]
    if sum = v
        return Yes
    else if sum > v
        e=e-1
    else
        s=s+1
return No
```

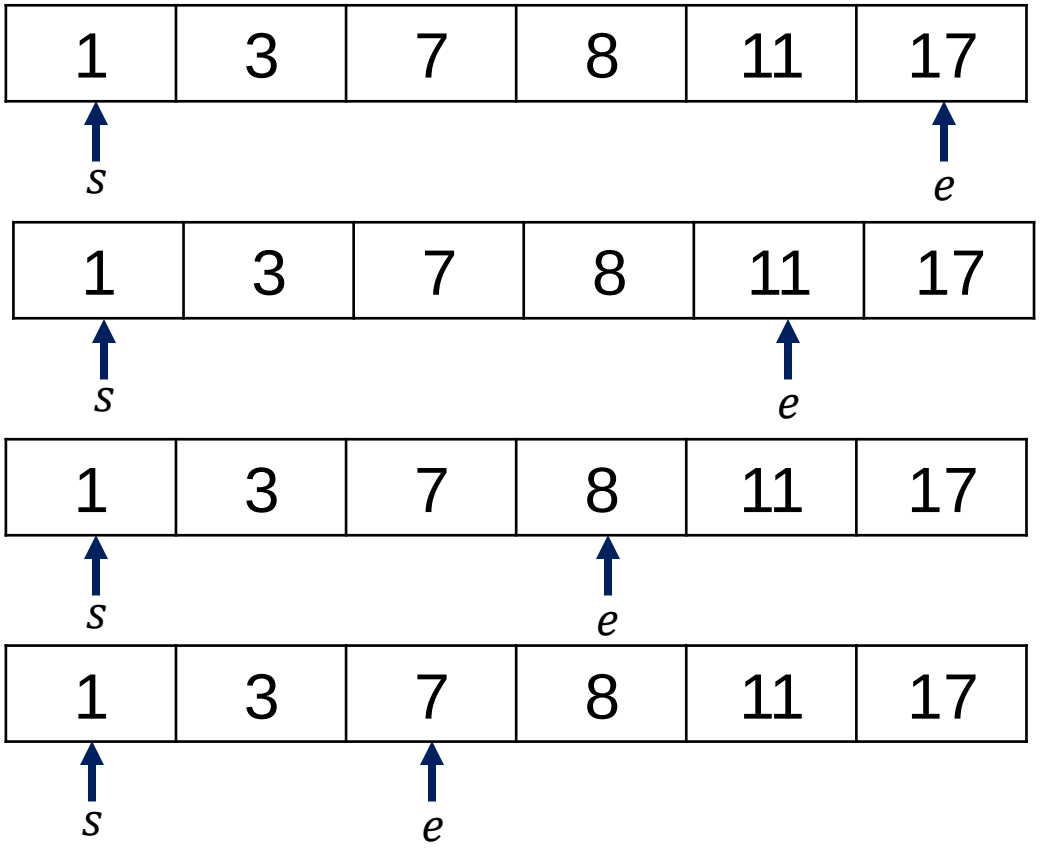
TWO SUM PROBLEM: LINEAR ALGORITHM



$v = 11$

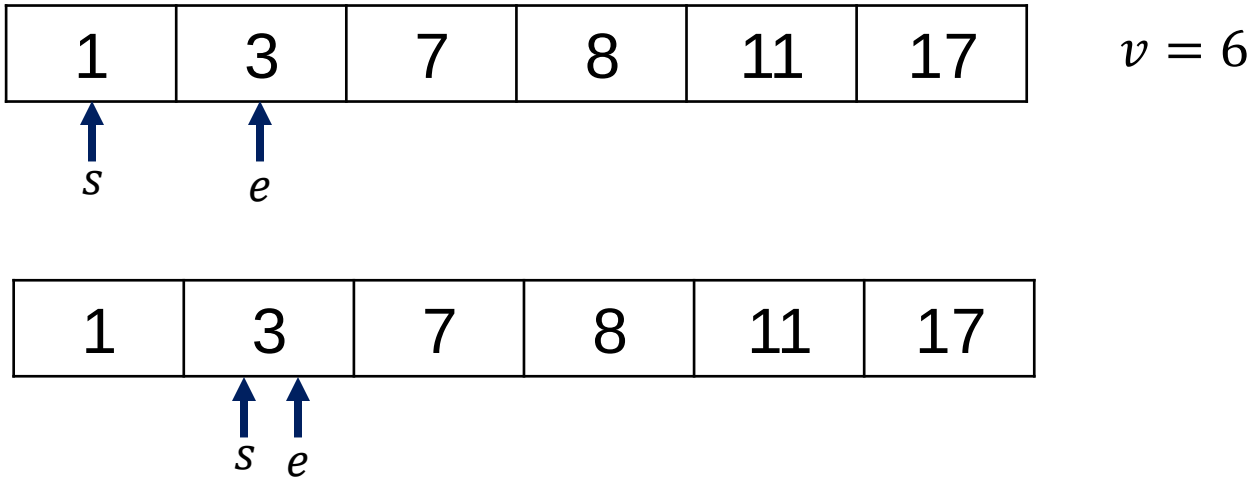
Return Yes

TWO SUM PROBLEM: LINEAR ALGORITHM



$v = 6$

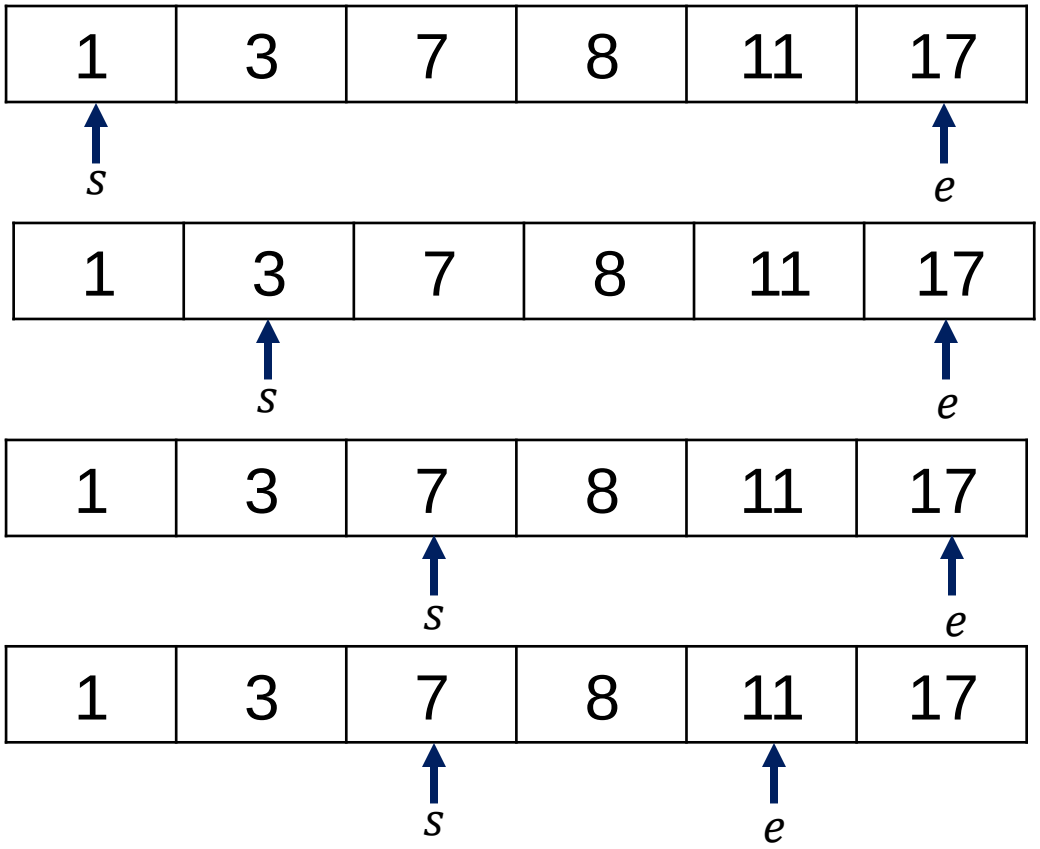
TWO SUM PROBLEM: LINEAR ALGORITHM



$v = 6$

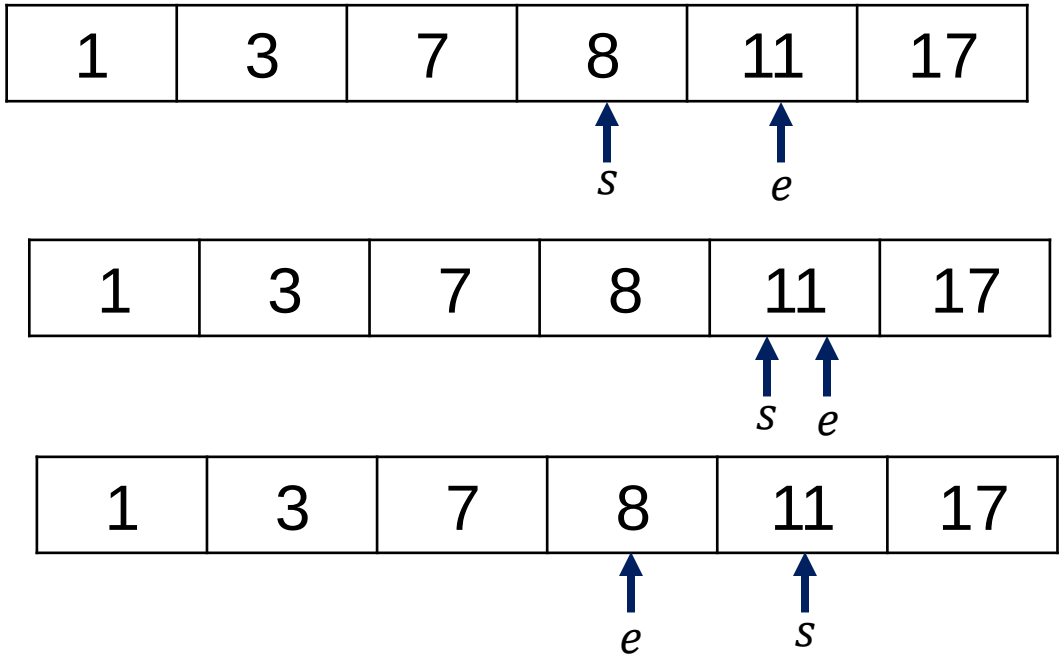
Return Yes

TWO SUM PROBLEM: LINEAR ALGORITHM



$v = 21$

TWO SUM PROBLEM: LINEAR ALGORITHM



$v = 21$

Return No

TWO SUM PROBLEM: LINEAR ALGORITHM

In each iteration of the *while* loop, we spend a constant time.

We **check** the loop at most $n + 1$ times. Thus, we can show that:

$$T(n) = O(n)$$

Not a formal proof!

```
int s = 1
int e = n
while s <= e
    sum= A[s]+A[e]
    if sum = v
        return Yes
    else if sum > v
        e=e-1
    else
        s=s+1
return No
```

TWO SUM PROBLEM: LINEAR ALGORITHM

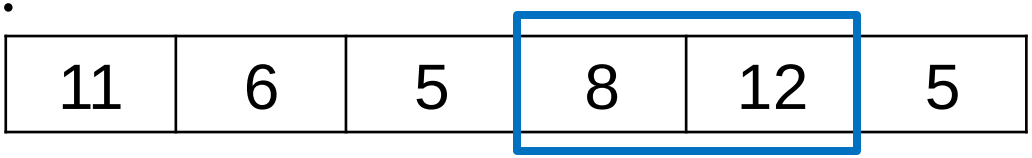
- It is easy to show that the algorithm is correct using an inductive argument.
- In the initial case, we check $A[s] + A[e]$ and return Yes if the sum is equal to v.
- In the following steps, we only increment s, if $A[s] + A[e] < v$.
- This is OK to do since if $A[s] + A[e] < v$, then $A[s] + A[e'] < v$ for $e' < e$.
- A similar argument applies to when we decrement e.

MINIMUM SIZE SUBARRAY SUM

Input: An array of *positive* integers A and a target integer v.

Output: The *minimal length* of a subarray whose sum is *larger than* v. (Return 0 if there is no such subarray.)

Example:



For $v = 18$, the solution is 2 since the window including $8, 12$ gives $8 + 12 > 18$ and there is no window of length 1 which gives a value larger than 18.

BRUTE FORCE APPROACH

- Check all possible start-points s and end-points e for the window.
- There are $\binom{n}{2} + n = \frac{n(n+1)}{2}$ possibilities.
- Thus, the run time is in $\Omega(n^2)$.
- This is too slow!

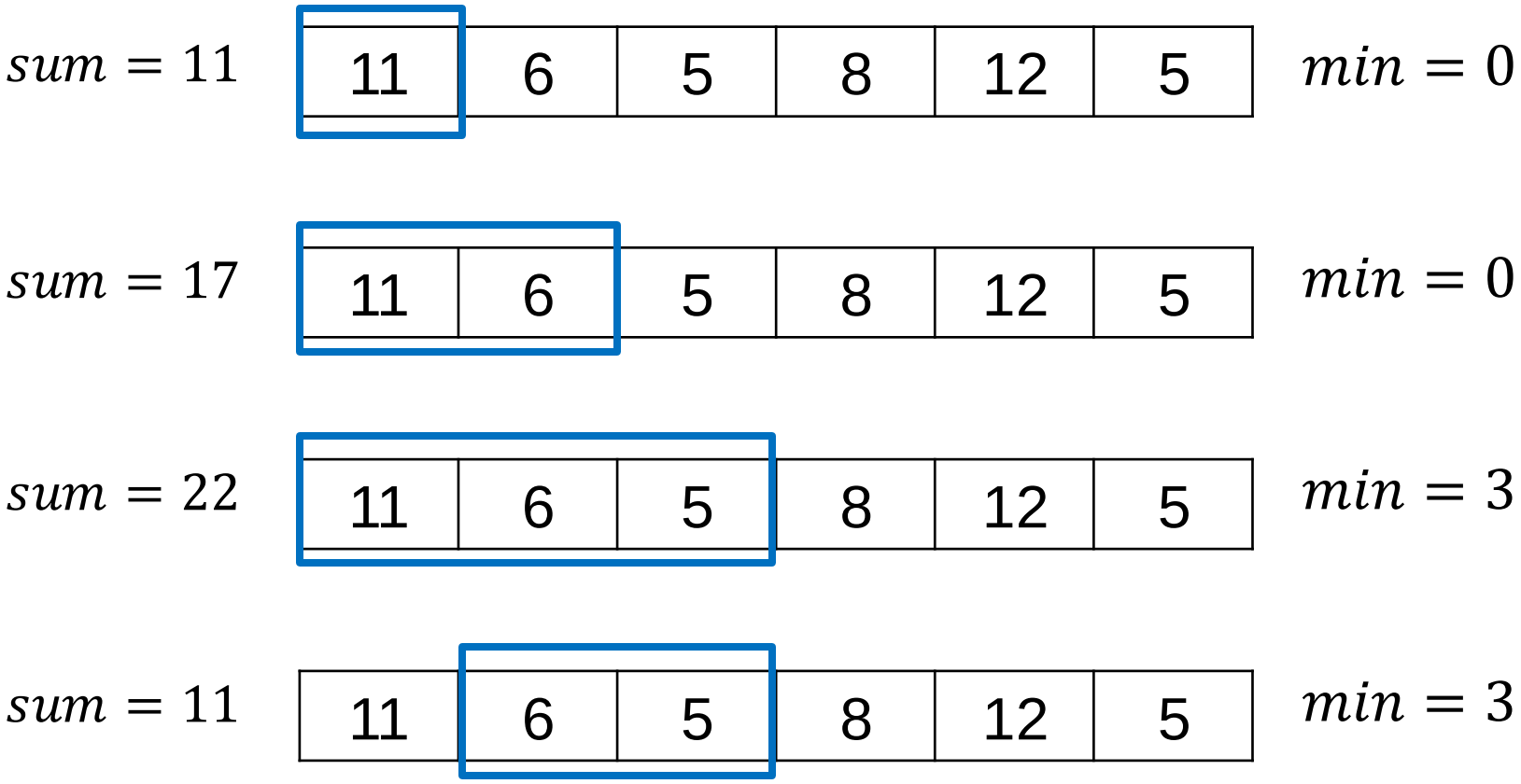
SLIDING WINDOW

The idea is to have a window that we slide from the start to the end of the array.

We first explain the algorithm through an example.

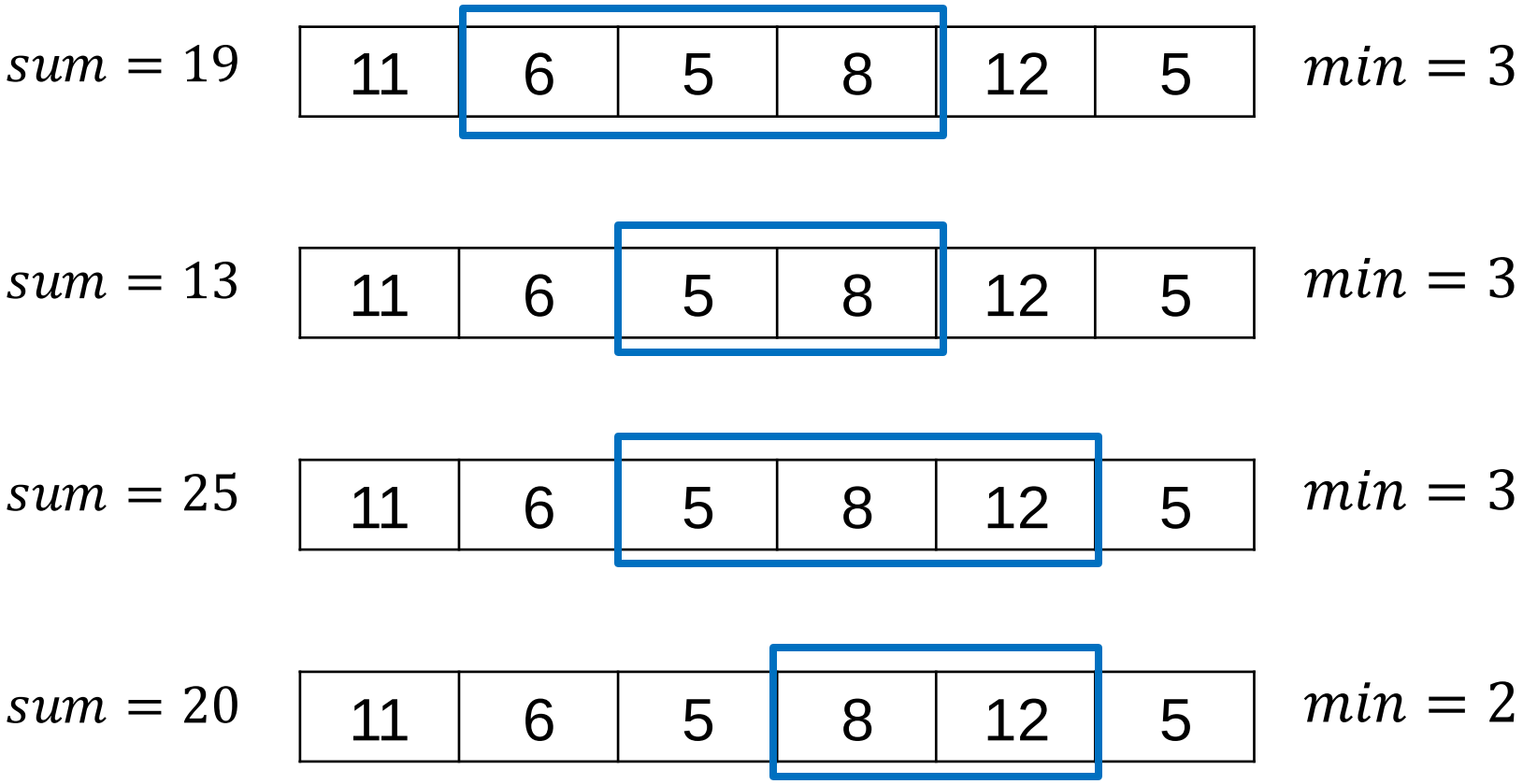
SLIDING WINDOW

$v = 18$



SLIDING WINDOW

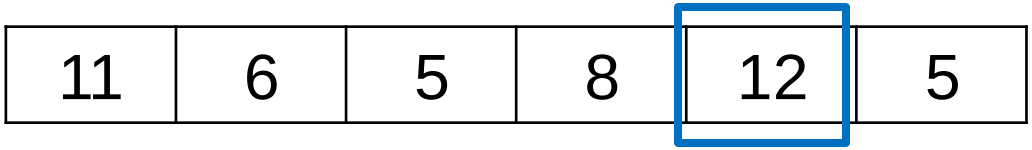
$v = 18$



SLIDING WINDOW

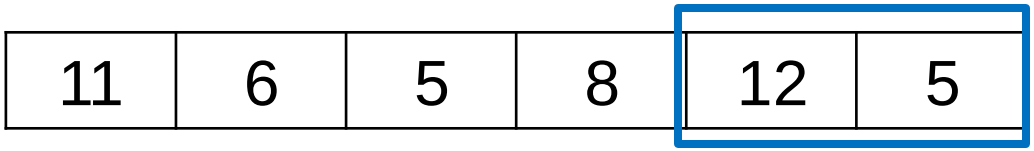
$v = 18$

$sum = 12$



$min = 2$

$sum = 17$



$min = 2$

SLIDING WINDOW

```
MinSubArrayLen(A,v)
s=1, e=1, sum=0, min=0;
while(e<=n)
    sum+=A[e]
    while(sum>v)
        min=minimum(e-s+1,min)
        sum=sum-A[s]
        s++
    e++
return min
```

RUN TIME

- The first and last lines clearly take a constant amount of time.
- In the **while** loops, we always do at most a constant amount of work before increasing e or s.
- Since s and e are 1 initially and can reach at most n, this takes $O(n)$ time.
- Thus, the overall run time is in $O(n)$.

CORRECTNESS

- We claim that when an element is left out of the window, it is OK.
- We can use an inductive argument.
- Suppose that the window is in position s to e and all elements from 1 to $s-1$ are left out correctly.
- When $sum > v$:
  - We update min if necessary
  - Set $s = s + 1$.
- It is OK to increase s since we don't need to consider the windows starting at s and ending at $e + 1$ or larger. Why?
- A similar argument applies to the case of $sum \leq v$ and incrementing e.

EGG DROPPING PROBLEM

Input: There is a building with n floors.

Output: You want to find the **highest** floor that if you drop an egg from it doesn't break.

Assumptions:

- If the egg breaks when dropped from floor k, then it would also have broken from any floor above that.
- If the egg survives a fall, then it will survive any fall below that.

***Objective**: Minimize the number of required drop tests (trials).*

SCENARIO 1

What strategy would you use if only one egg is given?

- Since we only have one egg, the only option is to start from floor 1, then floor 2, and so on, until it breaks.
- This can take n trials in the worst case.

SCENARIO 2

What if you are given an unlimited number of identical eggs?

- We can simply do a binary search.
- We start at floor $\lceil \frac{n}{2} \rceil$.
- If the egg breaks, we repeat for the lower half.
- Otherwise, we repeat for the upper half.
- In the worst case, this takes $O(\log n)$ trials.

SCENARIO 3

What if you are given 2 identical eggs?

Binary Search Approach

- Drop the egg from floor$\left\lceil \frac{n}{2} \right\rceil$.
- If it breaks, then we have to apply the strategy with 1 egg for floors $1, \cdots, \left\lceil \frac{n}{2} \right\rceil - 1$.
- This takes $\Omega(n)$ in the worst case.

Can we do better?