CIS 606 Analysis of Algorithms

Divide-and-Conquer

RATIONALE

- A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solution to the sub-problems are then combined to get a solution to the original problem — Wikipedia
- A divide-and-conquer algorithm is implemented as a recursive function and solving a recurrence computes the running time.

OBJECTIVES

- Understand the divide-and-conquer framework
- Learn to use the divide-and-conquer to solve the largest number problem

PRIOR KNOWLEDGE

- Time complexity
- Asymptotic notation

DIVIDE-AND-CONQUER

- A Divide-and-Conquer algorithm contains three steps:
- Divide: divide the problem into a number of subproblems of smaller size
- Conquer: Solve the subproblem recursively
  - Base case: if the size of the subproblem is small enough, solve it in a straightforward way.
- Combine: Combine the solutions to the subproblems to obtain a solution of the original problem.

THE LARGEST NUMBER PROBLEM

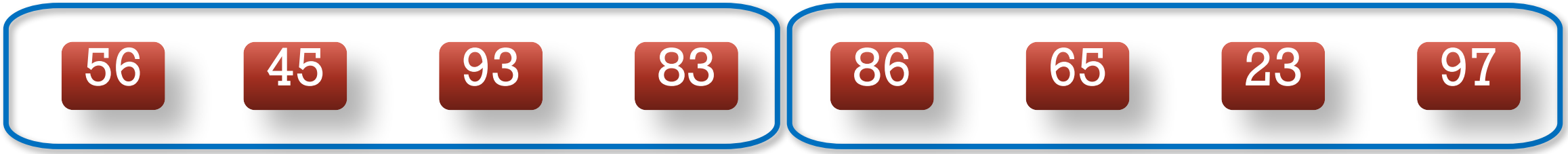
- Input: an array A[1… n]
- Output: the largest number of A
- The straightforward way:
- max(A) // T(n) = 2n+2 in worst case m === O(n) or $\Theta(n)$
- {
- m = A[1] // 1
- For i = 1: n // n * 2 worst case
- if m< A[i] //2 worst case 1 best case
- m = A[i] //executed in worst case not in best case
- Return m // 1
- }

THE LARGEST NUMBER PROBLEM

- Input: an array A[1… n]
- Output: the largest number of A
- 1. Divide:
  - Divide A[1… n] two subarrays A[1… n/2] and A[n/2+1, …, n]
- 2. Conquer: solving each subproblem — solutions for subproblems
  - Compute the largest number max1 of A[1… n/2] recursively
  - Compute the largest number max2 of A[n/2+1, …, n] recursively
- 3. Combine: solution from solutions of subproblems.
  - Return the larger one of max1 and max2;

EXAMPLE

- Divide:
  - Divide A[1… n] two subarrays A[1… n/2] and A[n/2+1, …, n]

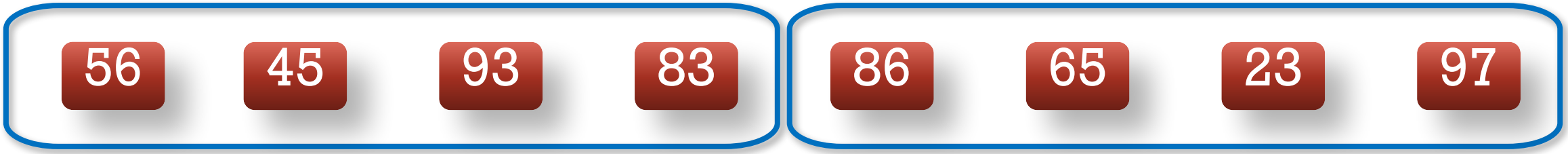



```
MAX(A, L, R)    //compute the greatest number of A[L, …, R]
{
     m = ⌊(L+R)/2⌋;

}
```

EXAMPLE (CONT)

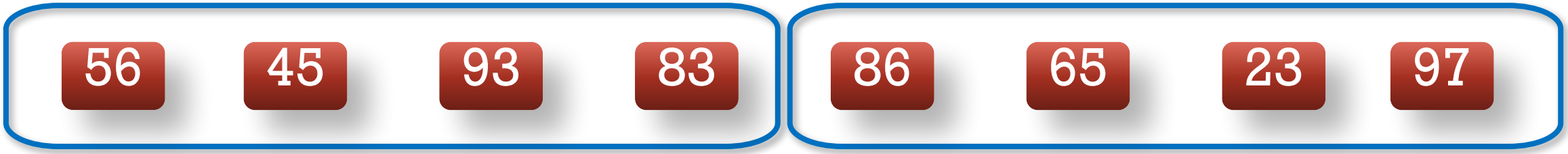
- Conquer:
  - Compute the largest number max1 of A[1… n/2]
  - Compute the largest number max2 of A[n/2+1, …, n]




```
MAX(A, L, R)
{
    m = ⌊(L+R)/2⌋;
    max1=MAX(A, L, m);
    max2=MAX(A, m+1, R);
```

EXAMPLE (CONT)

- Combine:
  - Return the larger one of max1 and max2;




```
MAX(A, L, R)
{
    m = ⌊(L+R)/2⌋;
    max1=MAX(A, L, m);
    max2=MAX(A, m+1, R);
    return max(max1, max2);
```

THE DIVIDE-AND-CONQUER ALGOR

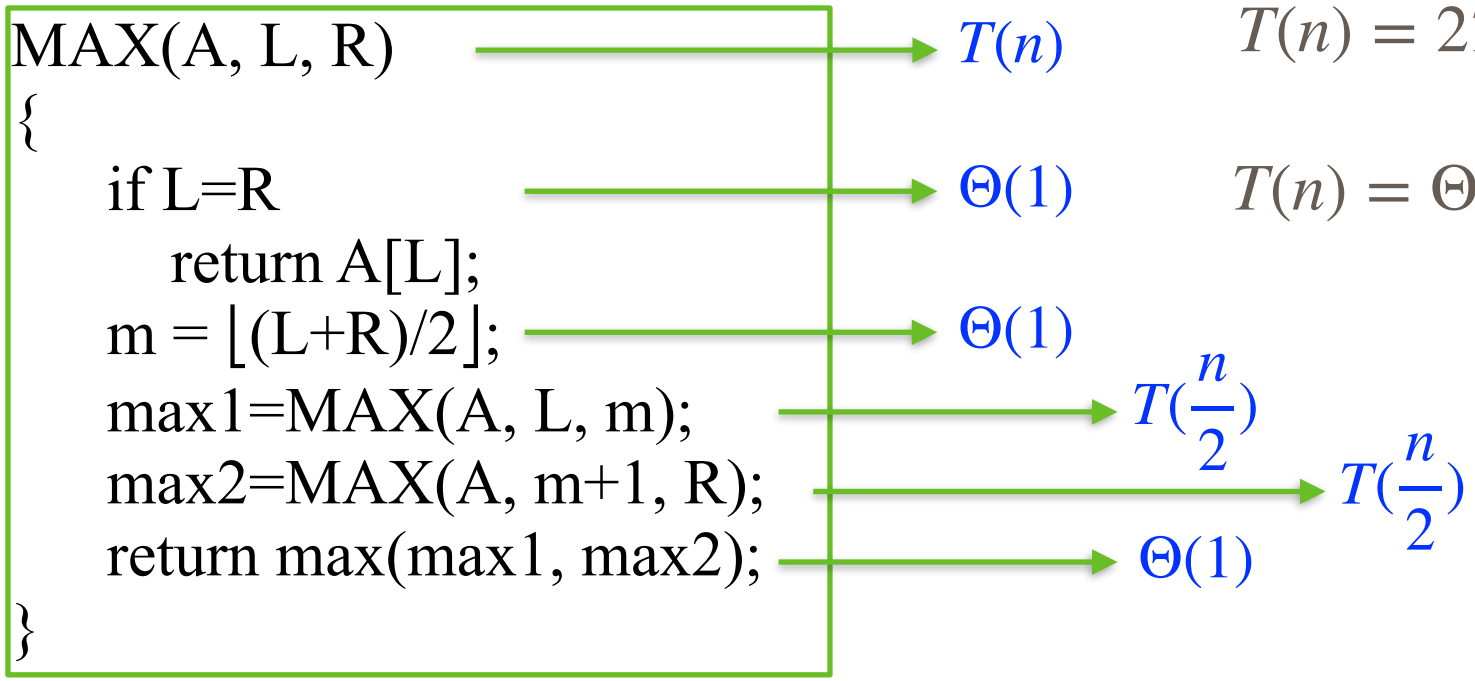
- Input: an array A[1… n]
- Output: the largest number of A
- The divide-and-conquer way:

```
Main(A)
{
    If |A| =0

    n = A.size;
    MAX(A, 1, n)
```

```
MAX(A, L, R)
{
    if L=R          return A[L];
    m = ⌊(L+R)/2⌋;  //divide
    max1=MAX(A, L, m); //conquer
    max2=MAX(A, m+1, R);
    return max(max1, max2); //combine
}
```

THE RECURRENCE



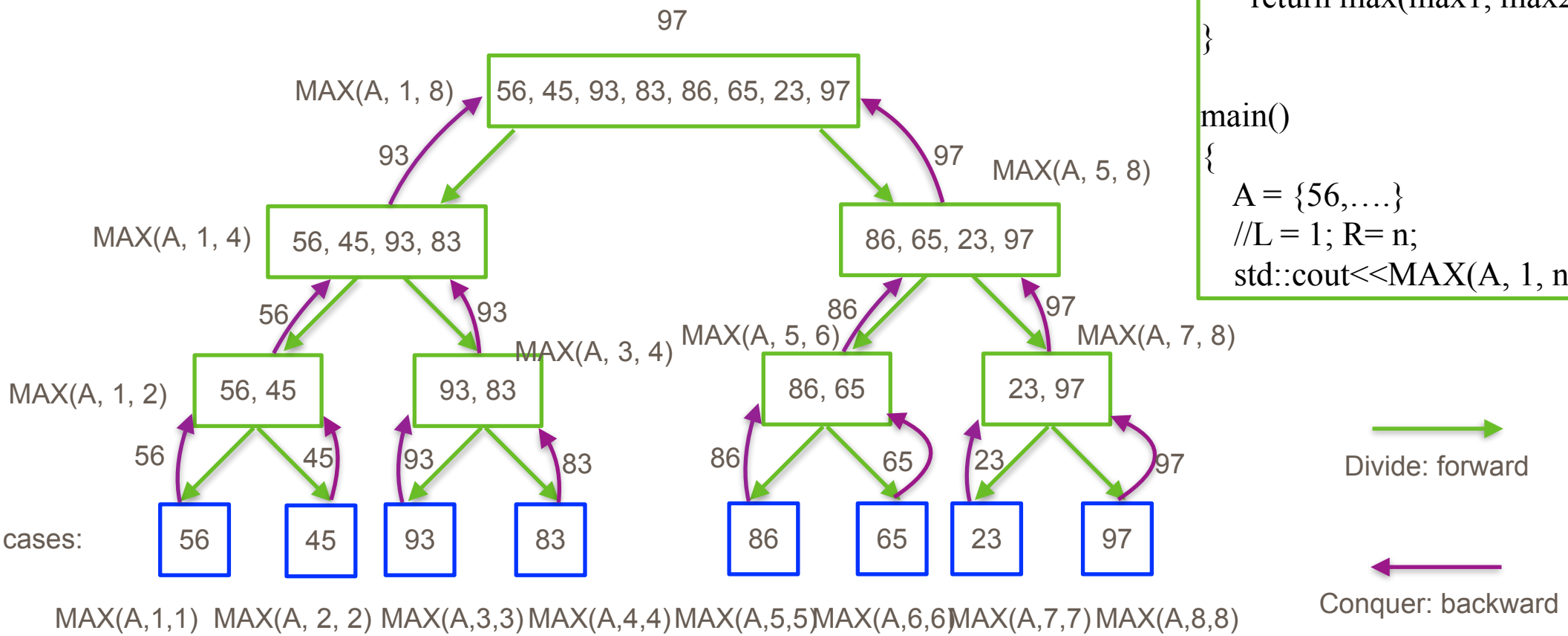
$$T(n) = 2T(\frac{n}{2}) + \Theta(1) \qquad \text{if } n > 1$$

$$T(n) = \Theta(1) \qquad \text{if } n = 1$$

EXECUTION PROCEDURE

```
MAX(A, L, R)
{
    if L=R
        return A[L];
    m = ⌊(L+R)/2⌋;
    max1=MAX(A, L, m);
    max2=MAX(A, m+1, R);
    return max(max1, max2);
}

main()
{
   A = {56,….}
   //L = 1; R= n;
   std::cout<<MAX(A, 1, n);
```



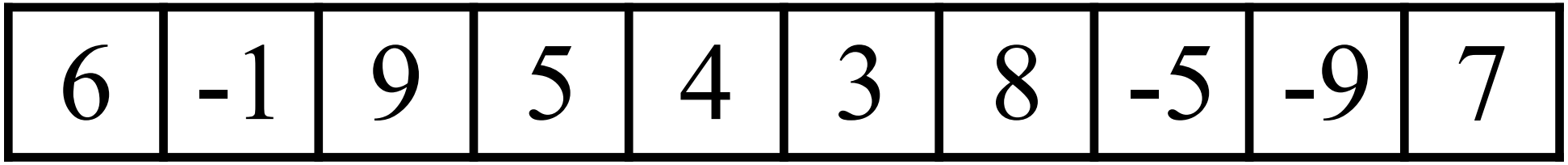
Divide-and-Conquer

Merge-Sort

MERGE SORT

- Input: an array A[1… n] of values in random order
- Output: the ascending order of A
- 1. Divide:
  - Divide A[1… n] two subarrays A[1… n/2] and A[n/2+1, …, n]
  - //m=(L+R)/2 = L+(R-L)/2
- 2. Conquer:
  - Sort A[1… n/2] recursively // MergeSort(A[1… n/2]);
  - Sort A[n/2+1, …, n] recursively //MergeSort(A[n/2+1, …, n]);
- 3. Combine: //A[1… n/2] sorted, A[n/2+1, …, n] sorted.
  - Merge sorted A[1… n/2] and A[n/2+1, …, n] to a whole sorted list;

EXAMPLE



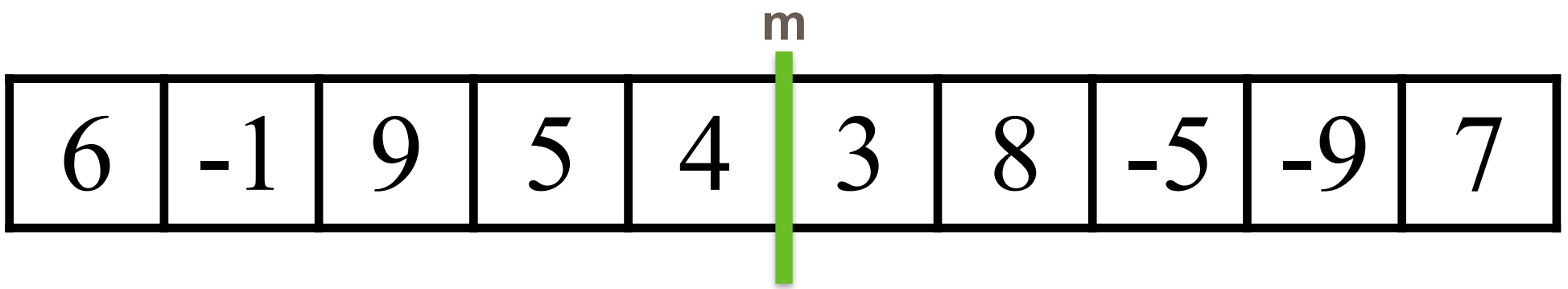

```
MergeSort(A, L, R)    //sort A[L, …, R] and initially MergeSort(A, 1, n)
{



}
```

EXAMPLE (CONT)

- Divide:
  - Divide A[L, …, R] into two subarrays A[L, …, m] and A[m+1, …, R]

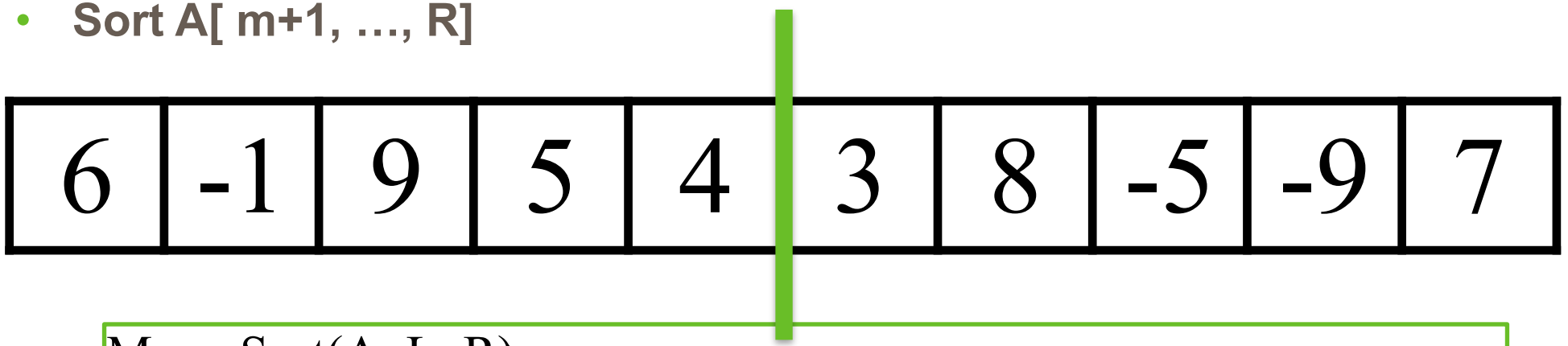



```
MergeSort(A, L, R)    //R-L+1 is the size of A
{
     m = ⌊(L+R)/2⌋;

}
```

EXAMPLE (CONT)

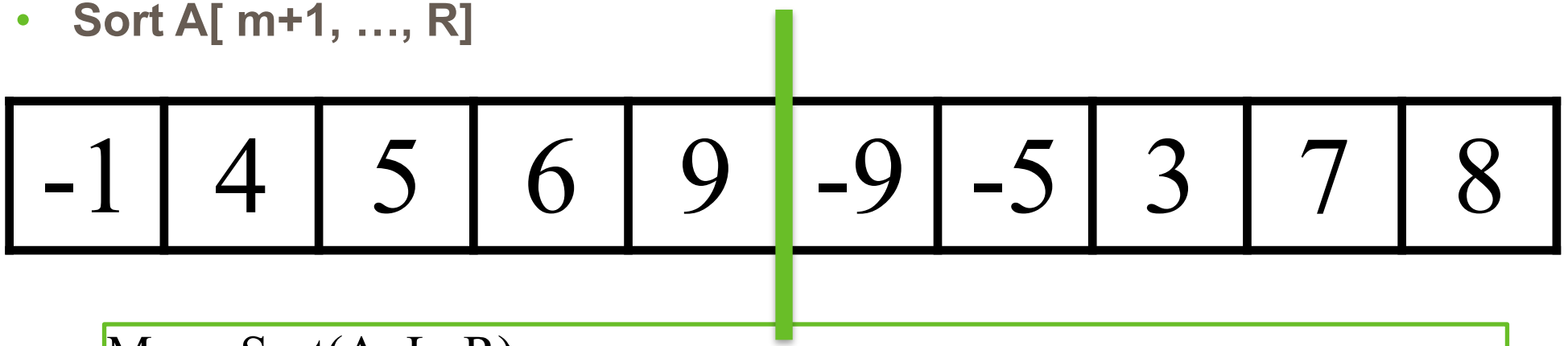
- Conquer:
  - Sort A[L, …, m]
  - Sort A[m+1, …, R]




```
MergeSort(A, L, R)
{    m = ⌊(L+R)/2⌋;
     MergeSort(A, L, m);
     MergeSort(A, m+1, R);
```

EXAMPLE (CONT)

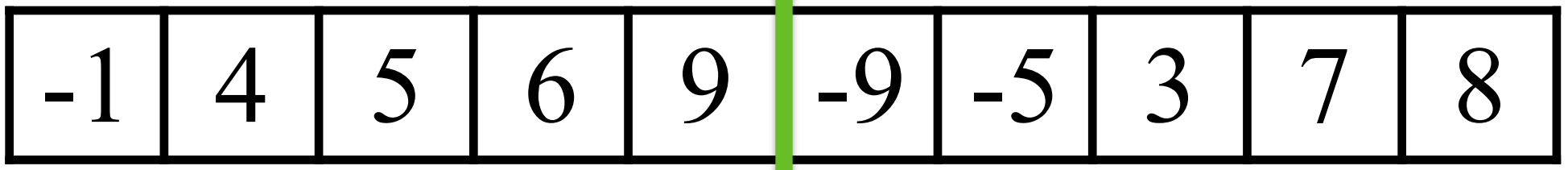
- Conquer:
  - Sort A[L, …, m]
  - Sort A[m+1, …, R]




```
MergeSort(A, L, R)
{    m = ⌊(L+R)/2⌋;
     MergeSort(A, L, m);
     MergeSort(A, m+1, R);
```

EXAMPLE (CONT)

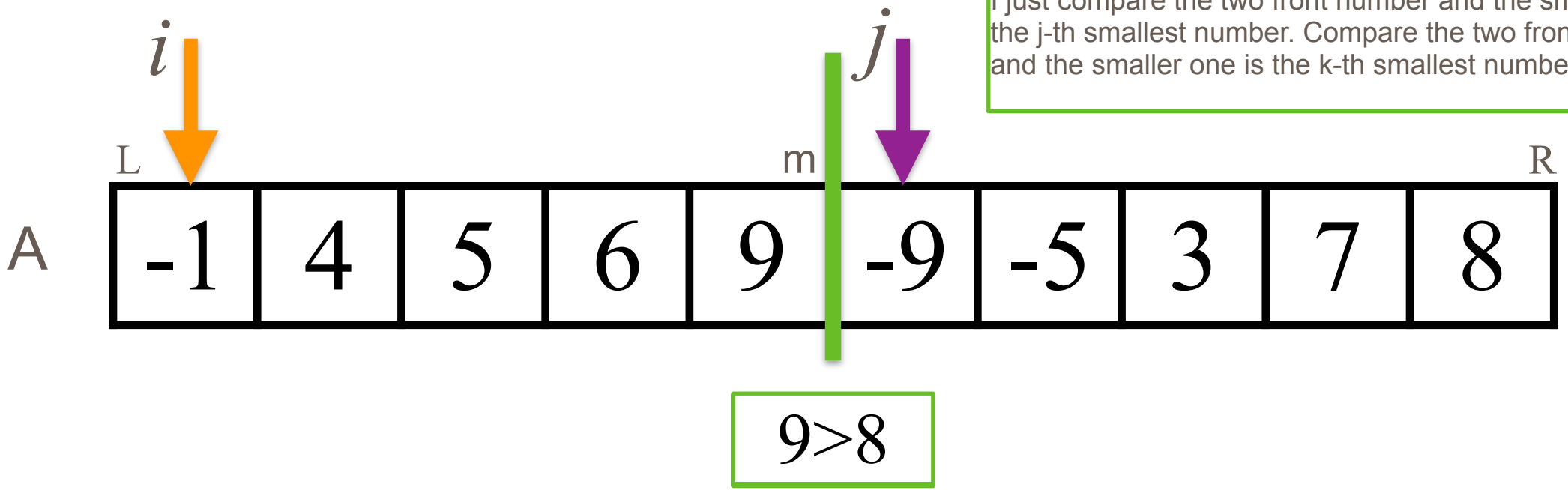
- Combine:
  - Merge sorted A[L, …, m] and A[m+1, …, R] to a whole sorted list;




```
MergeSort(A, L, R)
{    m = ⌊(L+R)/2⌋
     MergeSort(A, L, m)
     MergeSort(A, m+1, R)
   //Merge sorted A[L, …, m] and A[m+1, …, R] into a whole sorted list
     Merge(A, L, m, R)
}
```

Correctness:
For B[k], I need to compute the k-th smallest number of A and put it at B[k]. I do this for every position of B. After all, I can get the whole sorted list of all numbers in A.

//To compute the k-th smallest number, I need to find the smallest number of all remaining numbers in A. To do this, I just compare the two front number and the smaller one is the j-th smallest number. Compare the two front values and the smaller one is the k-th smallest number.



MERGE STEP

```
Merge(A, L, m, R)    f(n) = 5.5n + 5
{
      create an array B[1, …, (R-L+1)] ;  // n time
      i=L, j=m+1, k=1;                                  // 3
      for(k=1; k<=R-L+1 & i<=m & j<=R; k++). // 3n
      {                                                 //
            if A[i]<=A[j]
                  B[k] = A[i], i++
            else
                  B[k] = A[j], j++
```

What is the running time?

```
// n/2 + 2
{
 if i<=m. // 1
     Copy A[i, …, m] to the end of B; //n/2
if j<=R    // 1
     Copy A[j, …, R] to the end of B; //n/2
 }
 Copy B to A[L, …, R];      // n
}
```

RUNNING TIME T(n)

```
MergeSort(A, L, R)
{
    //base case
    if L=R
       return;
    m = ⌊(L+R)/2⌋
    MergeSort(A, L, m)
    MergeSort(A, m+1, R)
    Merge(A, L, m, R)
}
```

$T(n)$

$\Theta(1)$

1

$T(m - L + 1) = T(\frac{n}{2})$

$T(R - m) = T(\frac{n}{2})$

$5.5n + 5 = \Theta(n)$

Recurrence:

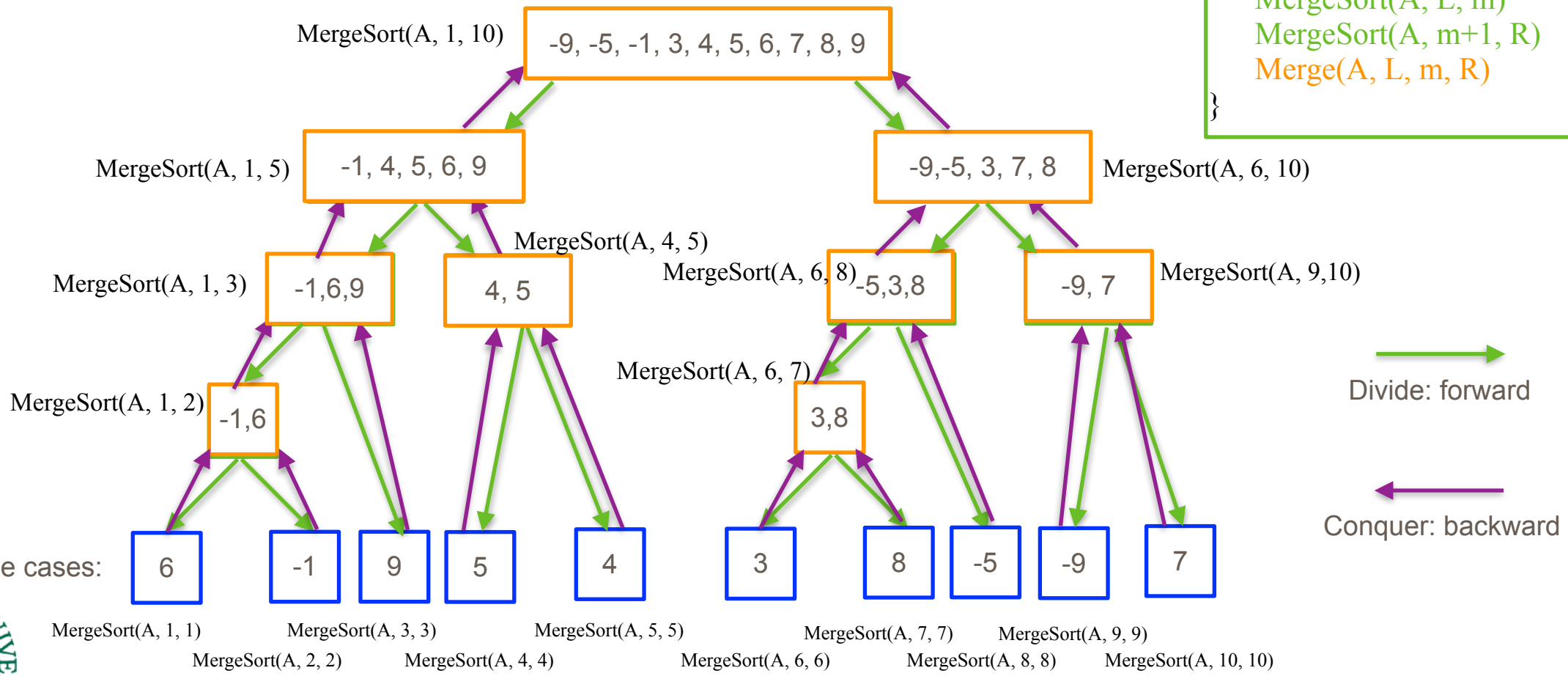
$$T(n) = 2T(\frac{n}{2}) + O(n) \qquad \text{if } n > 1$$

$$T(n) = \Theta(1) \qquad \text{if } n <= 1$$

Solving recurrence to compute the running time T(n)

Execution Procedure

```
MergeSort(A, L, R)
{
    //base case
    if L=R
      return;
    m = ⌊(L+R)/2⌋
    MergeSort(A, L, m)
    MergeSort(A, m+1, R)
    Merge(A, L, m, R)
}
```



SOLVING RECURRENCE

- Expanding the Recurrence
- Recursion Tree
- Guess-and-Verification
- Master Theorem

1. EXPANDING THE RECURRENCE

$$T(n) = 2T(\frac{n}{2}) + n = 2[2T(\frac{n}{2^2}) + \frac{n}{2}] + n$$

$$= 2^2 \cdot T(\frac{n}{2^2}) + 2n$$

$$= 2^2 \cdot [2T(\frac{n}{2^3}) + \frac{n}{2^2}] + 2n$$

$$= 2^3 \cdot T(\frac{n}{2^3}) + 3n \;\; = 2^4 \cdot T(\frac{n}{2^4}) + 4n = \cdots = 2^k \cdot T(\frac{n}{2^k}) + kn = 2^k \cdot T(1) + kn$$

$$\frac{n}{2^k} = 1 \qquad k = \log n$$

$$T(n) = 2T(\frac{n}{2}) + \Theta(n) \qquad \text{if } n > 1$$

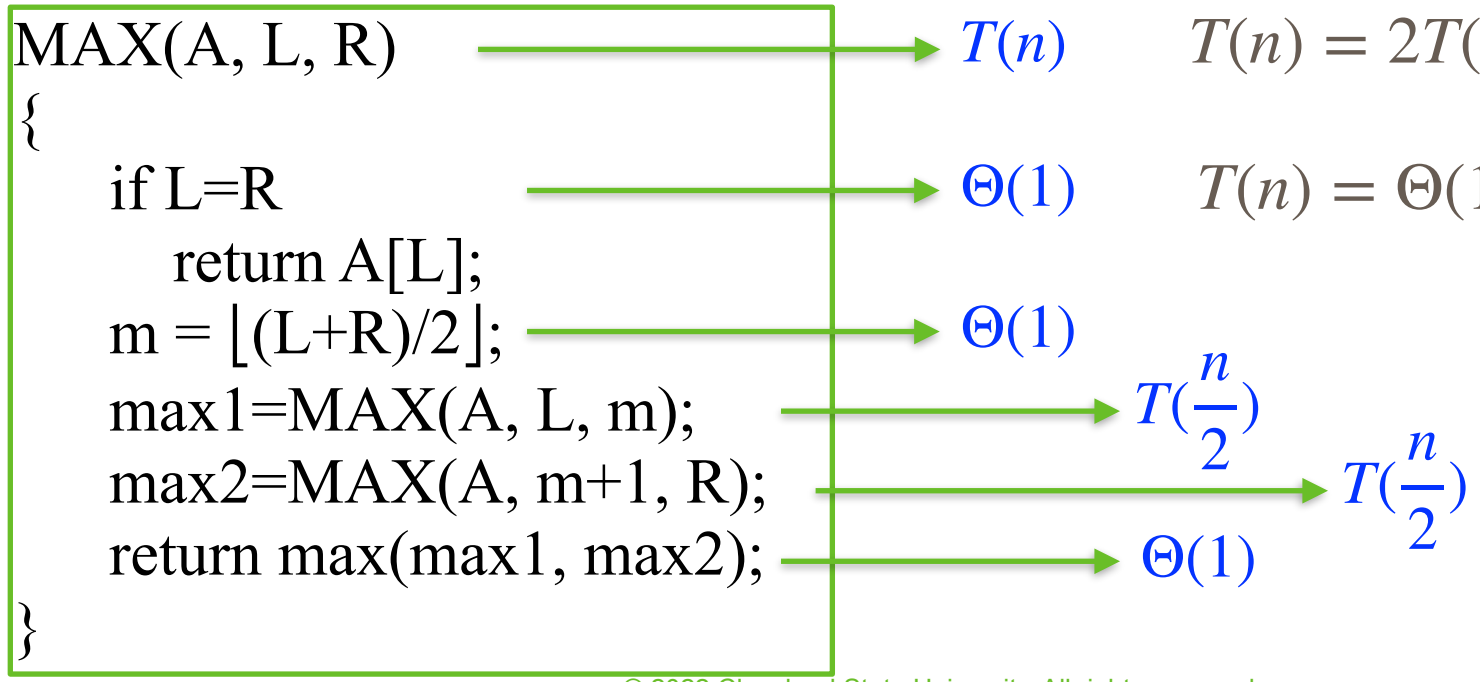
$$T(n) = \Theta(1) \qquad \text{if } n <= 1$$

$$= 2^k \cdot T(1) + kn$$

$$= n \cdot 1 + n \log n = O(n \log n)$$

2. RECURSION TREE

- Input: an array A[1… n]
- Output: the largest number of A
- The divide-and-conquer way:



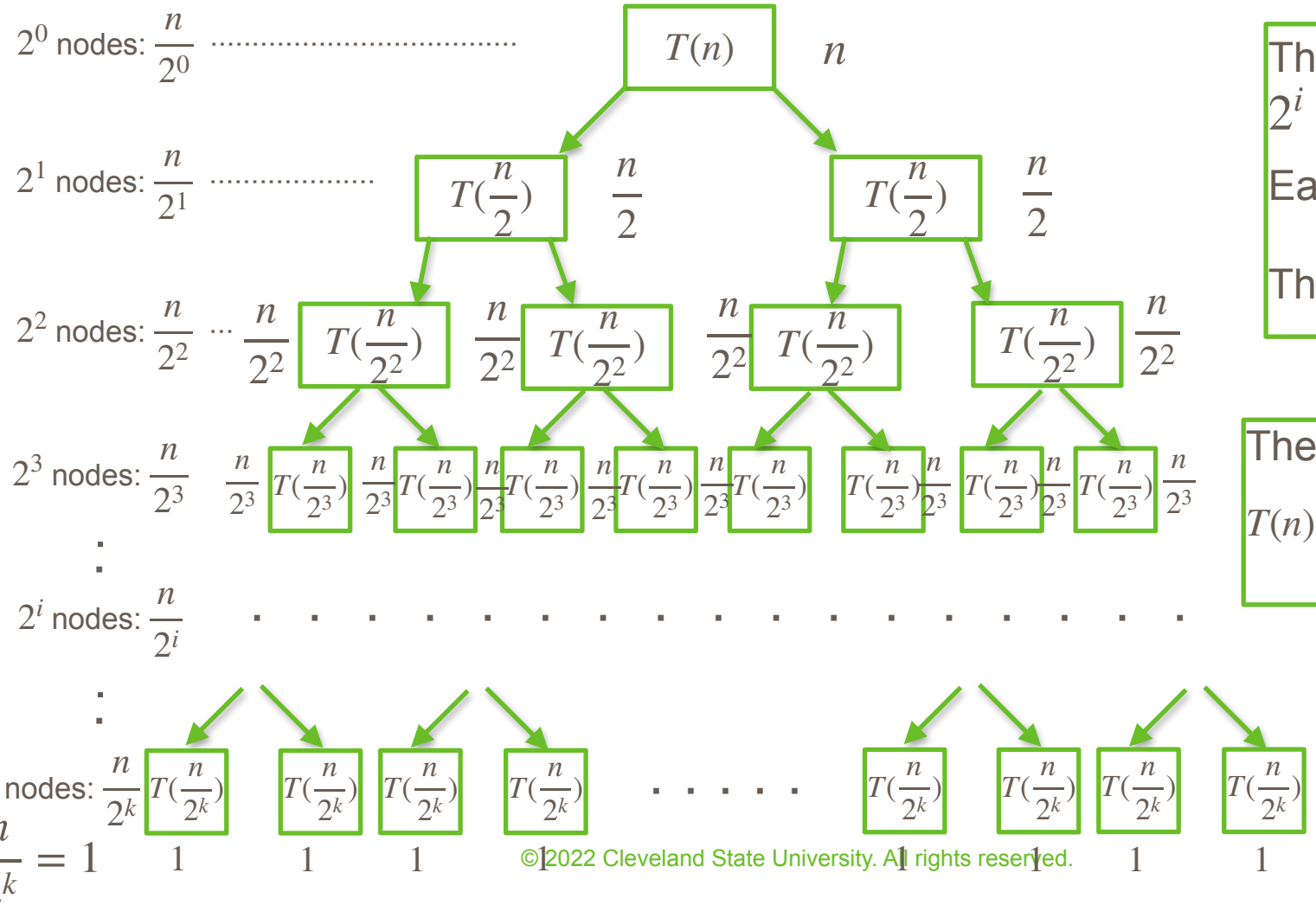
$$T(n) = 2T(\frac{n}{2}) + \Theta(1) \quad \text{if } n > 1$$

$$T(n) = \Theta(1) \quad \text{if } n = 1$$

2. RECURSION TREE

$$T(n) = 2T(\frac{n}{2}) + \Theta(n) \qquad \text{if } n > 1$$

$$T(n) = \Theta(1) \qquad \text{if } n = 1$$



The i-th level for $0 \le i \le k$:
2^i nodes (recursive calls);
Each node takes time $\frac{n}{2^i}$;
The total is $2^i \times \frac{n}{2^i} = n.$

The total 0, 1, …, k levels take

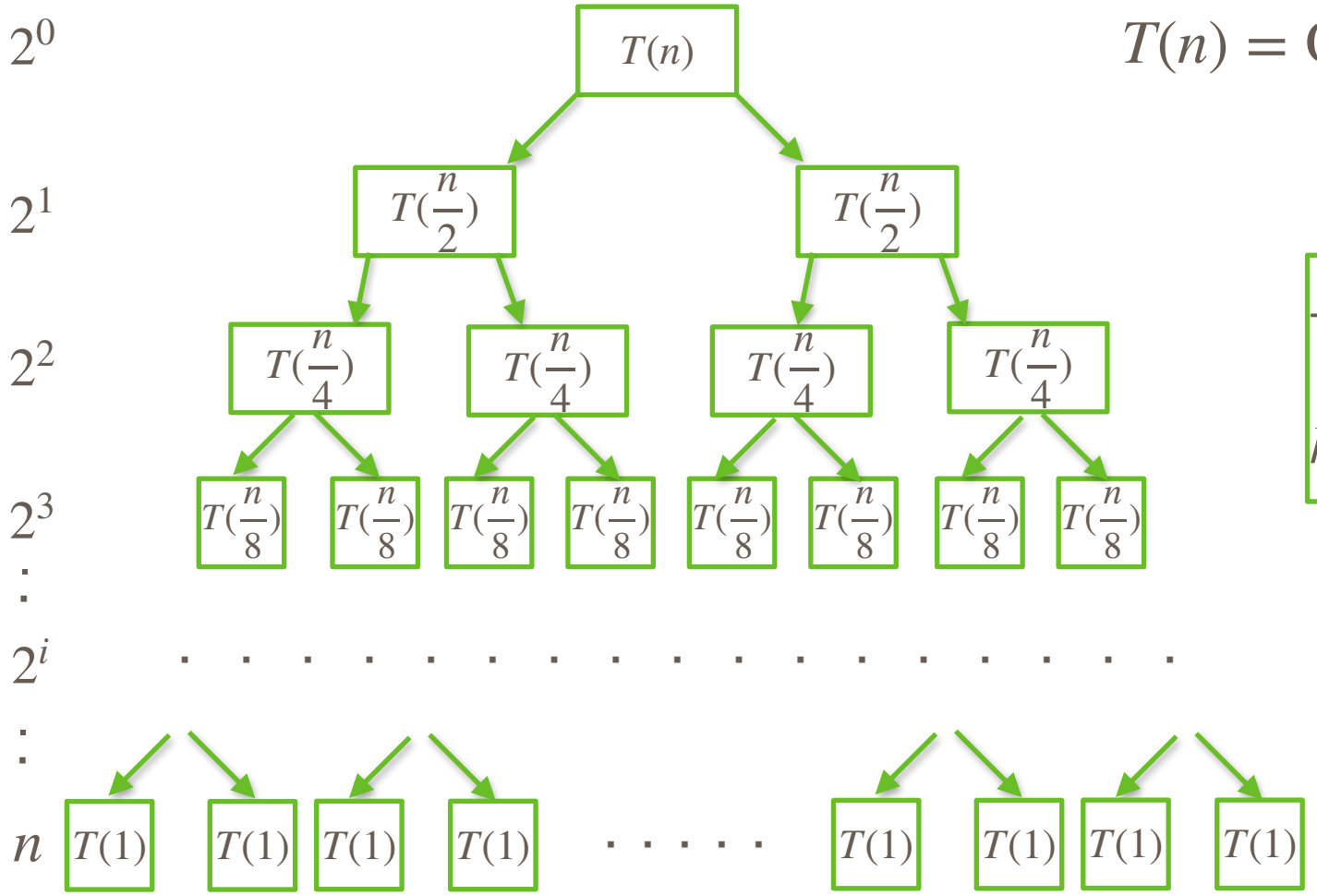
$$T(n) = \sum_{i=0}^{k} n = (k+1)n = O(n \log n)$$

$$k = \log n$$

2. RECURSION TREE(CONT)

$$T(n) = 2T(\frac{n}{2}) + \Theta(1) \quad \text{if } n > 1$$

$$T(n) = \Theta(1) \quad \text{if } n = 1$$



$$T(n) = \sum_{i=0}^{k} 2^i = n$$

$$k = \log n$$

3. GUESS-AND-VERIFICATION

- This approach essentially uses the definition of the big-O notation and follows the mathematical induction.
  - In the guess step, we make a guess (upper bound) for the running time.
  - In the verification step, we assume that the guess is true for smaller values of n, and then we use the recurrence to prove that the guess is also true for any general n.

3. GUESS-AND-VERIFICATION (CONT)

- **Example**

$$\begin{cases} T(n) = 2T(\frac{n}{2}) + \Theta(n) & \text{if } n > 1 \\ T(n) = \Theta(1) & \text{if } n \leq 1 \end{cases}$$

- **The guess step:**
  - **We guess** $T(n) = O(n \log n)$.
  - **According to the definition of the big-O notation, we want to find a constant** $c > 0$ **and an integer** $n_0 > 1$ **such that** $T(n) < cn \log n$ **for all** $n > n_0$

3. GUESS-AND-VERIFICATION (CONT)

- **The verification step:**
  - **We assume the guess is true for** $\frac{n}{2}, T(\frac{n}{2}) = O(\frac{n}{2}\log\frac{n}{2})$**, i.e.,** $T(\frac{n}{2}) < c \cdot \frac{n}{2} \cdot \log(\frac{n}{2})$ **where a constant** $c > 0$ **and** n **is larger than an integer** $n_0 > 1$.
  - **Then we have the following:** $T(n) = O(n \log n)$

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + n \\
T(n) &\leq 2 \cdot (c \cdot \frac{n}{2} \cdot \log(\frac{n}{2})) + n \\
&= c \cdot n \log(\frac{n}{2}) + n \\
&= c \cdot n \log n - cn + n \\
T(n) &\leq c \cdot n \log n + (1 - c)n
\end{aligned}
$$

3. GUESS-AND-VERIFICATION (CONT)

- **The verification step:**
  - **Then we have the following:** $T(n) \leq c \cdot n \log n + (1 - c)n$
  - **To prove** $T(n) \leq c \cdot n \log n$**, it is sufficient to prove**

$$T(n) \leq c \cdot n \log n + (1 - c)n \leq c \cdot n \log n$$

$$\downarrow$$

$$(1 - c)n \leq 0 \qquad \text{Any c>1 suffices}$$

The above proves that when $c = 1$ (or $c = 2, 3, \cdots$) and $n_0 = 1$ (n_0 can actually be any positive integer), $T(n) \leq cn \log n$ holds for all $n > n_0$.

Conclusion: $T(n) = O(n \log n)$

4. MASTER THEOREM

$$\begin{cases} T(n) = aT(\frac{n}{b}) + f(n) & \textbf{if } n > n_0 \\ T(n) = 1 & \textbf{if } n \leq n_0 \end{cases}$$

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 the following two conditions hold:

a) $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$, and

b) $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n,

then $T(n) = \Theta\left(f(n)\right)$.

4. MASTER THEOREM (CONT)

$$\begin{cases} T(n) = aT(\frac{n}{b}) + f(n) & \textbf{if } n > n_0 \\ T(n) = 1 & \textbf{if } n \leq n_0 \end{cases}$$

If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

$$T(n) = 2T(\frac{n}{2}) + \sqrt{n}$$

$$a = 2,\ b = 2,\ f(n) = \sqrt{n} = n^{0.5}$$

$$n^{\log_b a - \epsilon} = n^{\log_2 2 - \epsilon} = n^{1-\epsilon}$$

let $\epsilon = 0.1$

$$n^{\log_b a - \epsilon} = n^{1-\epsilon} = n^{0.9}$$

$$Obviously,\ f(n) = n^{0.5} = O(n^{1-\epsilon}) = O(n^{0.9})$$

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n)$$

4. MASTER THEOREM (CONT)

$$T(n) = T(\frac{n}{\frac{3}{2}}) + 1$$

$$a = 1,\ b = \frac{3}{2},\ f(n) = 1$$

$$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$$

$$Obviously,\ f(n) = 1 = \Theta(1)$$

$$T(n) = \Theta(\log n)$$

If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \log n\right)$.

4. MASTER THEOREM (CONT)

$T(n) = 2T(\frac{n}{2}) + n^2$

$a = 2, b = 2, f(n) = n^2 \quad n^{\log_b a} = n^{\log_2 2} = n$

$n^{\log_b a + \epsilon} = n^{\log_2 2 + \epsilon} = n^{(1+\epsilon)} \cdot n^2 = \Omega(n^{1+\epsilon})$

$let\ \epsilon = 0.1 \quad n^{\log_b a + \epsilon} = n^{1.1}$

$Obviously,\ f(n) = n^2 = \Omega(n^{1.1})$

$af(\frac{n}{b}) = 2f(\frac{n}{2}) = 2(\frac{n}{2})^2 = \frac{n^2}{2} = 0.5 \cdot n^2 \leq c \cdot n^2$ **when** $c = 0.6 \quad T(n) = \Theta(n^2)$

If the following two conditions hold:

a) $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and

b) $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n,

then $T(n) = \Theta(f(n))$.

NOTES

- To solve a recurrence, you may want to try Master theorem first because it can “directly”give you the solution if it works. If Master theorem does not apply, then try expanding or recursion tree approaches.
- For the guess-and-verification approach, the key is to make a “good” guess, which is based on your experience. However, if you have a clear target to prove, i.e., you want to particularly prove $T(n) = O(n^2)$, then the substitution approach might be the easiest approach.
- Not every recurrence can be solved by Master theorem. The other three approaches, however, should be able to solve all recurrences in general (although might be not easy).

Divide-and-Conquer

Maximum Subarray Problem

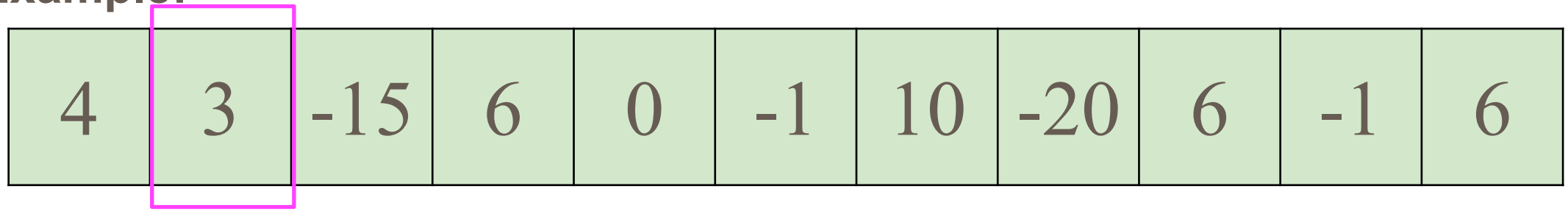
MAXIMUM SUBARRAY PROBLEM

```
greatestsum=0;
for i from 1 to n
    sum = 0;
    for j from i to n
        //consider A[i, …, j]
        sum = sum+ A[j];
        if greatestsum<sum
            greatestsum = sum ;
```

Input: an array A[1⋯ n]

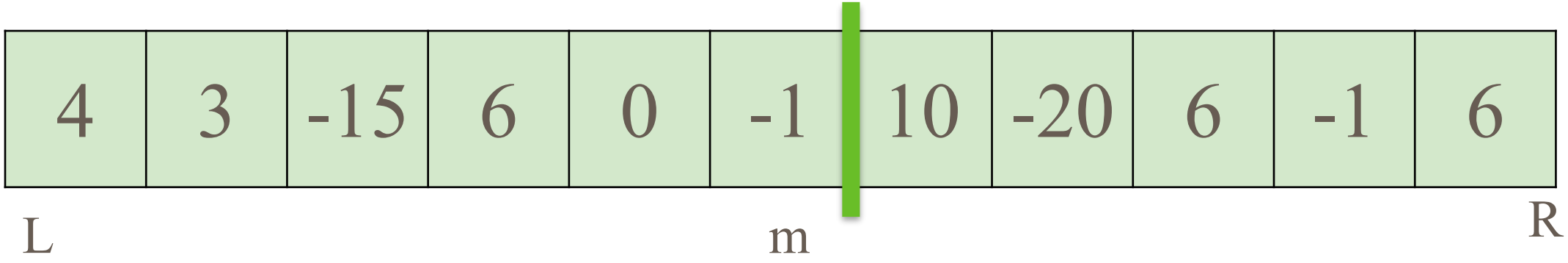
Output: the sum of the maximum subarray which is a subarray A[i, ⋯, j], with 1<=i<=j<=n, of maximum total sum.

Example:



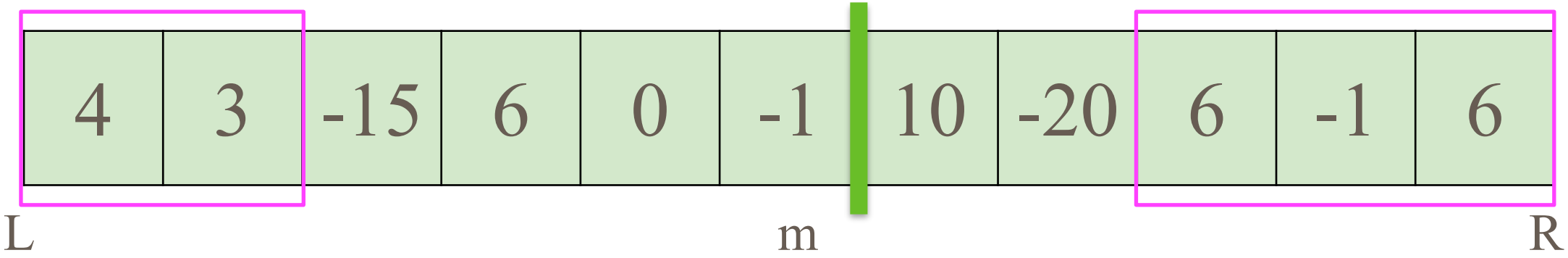
Algorithm 1: Try all i and j, brute force $O(n^2)$

MAXIMUM SUBARRAY PROBLEM (CONT)



Divide: Divide A[L, ···, R] into two subarrays A[L, ···, m] and A[m+1, ···, R]

MAXIMUM SUBARRAY PROBLEM (CONT)

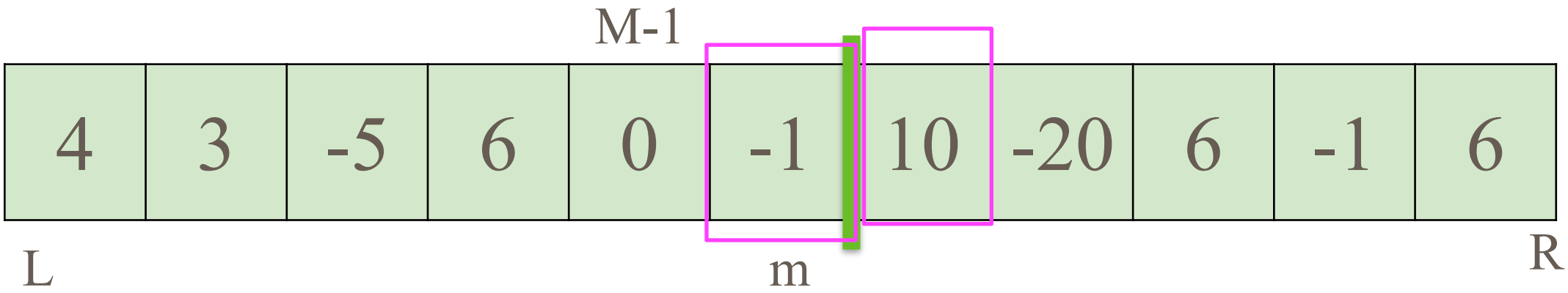


Conquer:

Compute the maximum subarray sum max1 of A[L, ···, m] recursively

Compute the maximum subarray sum max2 of A[m+1, ···, R] recursively

MAXIMUM SUBARRAY PROBLEM (CONT)



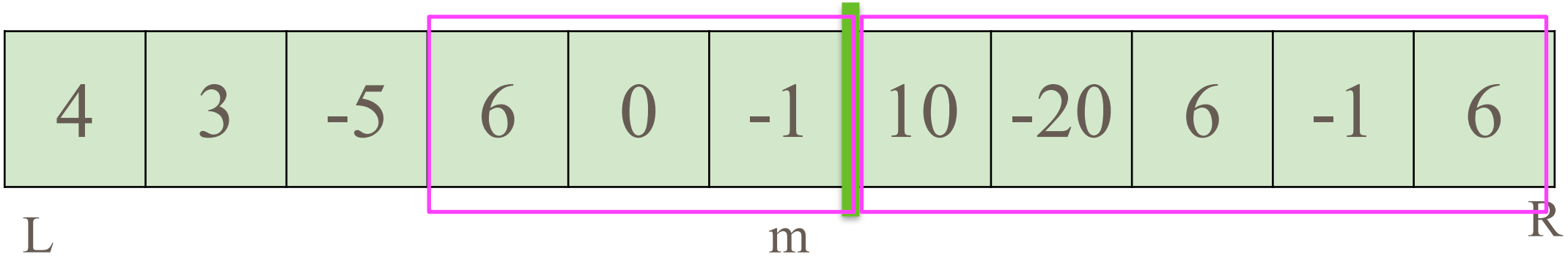
Sum of A suffix subarray of A[1…m] + Sum of A prefix subarray of A[m+1… n]
Greatest sum of the crossing contiguous subarray: =
Sum of A suffix subarray of A[1…m] = a find the suffix subarray of A[1…m] with the greatest sum of all.
Sum of A prefix subarray of A[m+1… n] = a find the prefix subarray of A[1…m] with the greatest sum of all.

a= the greatest sum of {all suffix subarrays of A[1…m]}
b = the greatest sum of {all prefix subarrays of A[m…n]}

ANALYSIS

- The maximum subarray of A could
  - Case 1: entirely lies A[L, ⋯, m], i.e., max1
  - Case 2: entirely lies A[m+1, ⋯, R], i.e., max2
  - Case 3: cross A[m] with the form A[i,⋯, m,⋯j], denoted by max3
- Observation:
  - Assume max1, max2 and max3 are known, the maximum subarray sum of A is the largest of max1, max2, and max3.
- Combining step:
  - Computing a maximum array crossing A[m], i.e., max3.
  - Then return max(max1, max2, max3).

MAXIMUM SUBARRAY PROBLEM



Combine:

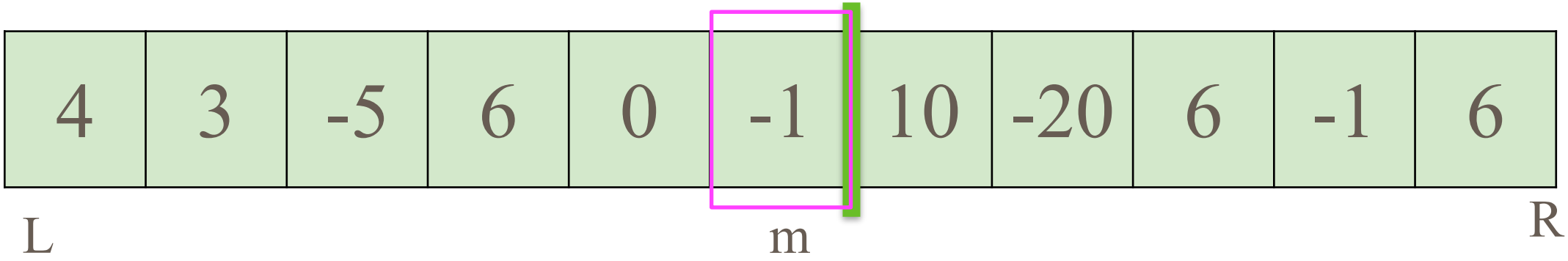
1. Compute the maximum subarray max3 crossing A[m] by calling the function Max-Cross-Subarray(A, L, m, R)
2. Return the maximum of max1, max2, and max3

PSEUDO-CODE

```
Max-Subarray(A, L, R).        T(n) = 2T(n/2) + O(n) =O(n\log n)
{
    if L=R        return A[L]   //base case where only one element is left
    m = ⌊(L+R)/2⌋          //divide.  1

    max1= Max-Subarray(A, L, m)  T(n/2)
    max2= Max-Subarray(A, m+1, R). T(n/2)
    max3= Max-Cross-Subarray(A, L, m, R) //O(n)
    return max(max1, max2, max3) //O(1)
}
```

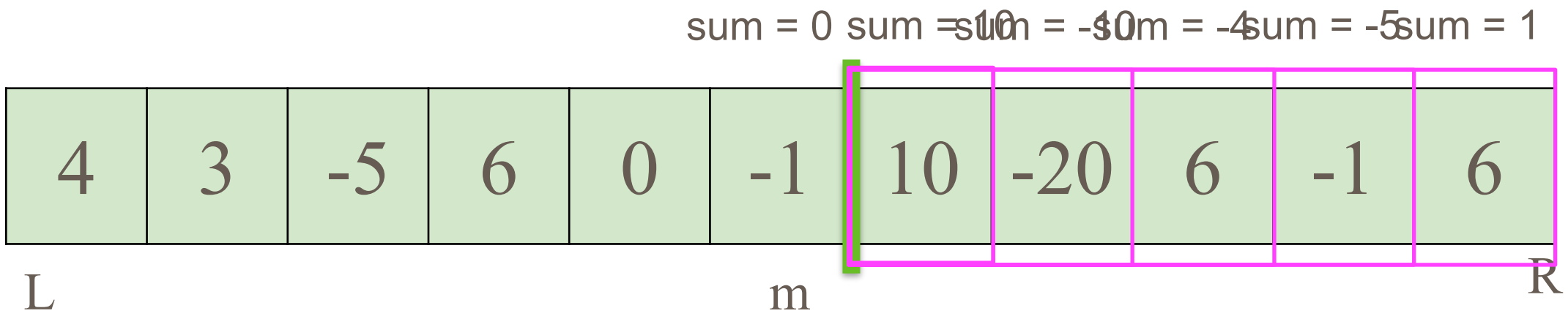
Analysis: How to compute the maximum subarray A[i,⋯,m,⋯j] crossing A[m]?



The maximum subarray A[i, ⋯m, ⋯, j] of A crossing A[m] has two parts: A[i,⋯, m] and A[m+1,⋯, j]:

1. Computing the maximum subarray A[i, ⋯, m]
2. Computing the maximum subarray A[m+1,⋯, j]
3. Combine them to get the maximum subarray A[i,⋯, m, ⋯j]

COMPUTING THE RIGHT MAXIMUM SUM OF A[M, …]




```
Sum = 0

For i= m+1 to n // n * 1
    Sum = Sum+ A[i]
     Add sum to a list Q
Done with calculating the sum for every prefix
subarray
Scan Q to find the greatest sum a

For i= m down to 1 // n * 1
    Sum = Sum+ A[i]
```

Right_maxsum = $-\infty$
Right_maxsum = 10

COMPUTING MAXIMUM SUBARRAY CROSSING A[M]

```
Max-Cross-Subarray(A, L, m, R)
{
    Right_maxsum = −∞;
    sum = 0
    for i = m+1 to R
    {     sum = sum +A[i];
          if Right_maxsum < sum
                Right_maxsum = sum;
     }
    Left_maxsum = −∞;
    sum = 0
    for i = m down to L
    {     sum = sum +A[i];
          if Left_maxsum < sum
                Left_maxsum = sum
     }
     Return Right_maxsum+Left_maxsum
}
```

What is the running time?

PSEUDO-CODE

```
Max-Subarray(A, L, R)                                   → T(n)
{
    //base case where only one element is left
    if L=R        return A[i];                          → O(1)
    m = ⌊(L+R)/2⌋ ;                                     → O(1)

    max1= Max-Subarray(A, L, m);                        → T(n/2)
    max2= Max-Subarray(A, m+1, R);                      → T(n/2)
    max3= Max-Cross-Subarray(A, L, mid, R);             → O(n)
    return max(max1, max2, max3);                       → O(1)
```

$$T(n) = 2T(\frac{n}{2}) + O(n) \quad \text{if } n > 1$$

$$T(n) = O(1) \quad \text{if } n = 1$$

MAXIMUM SUBARRAY— RUNNING TIME

$$T(n) = 2T(\frac{n}{2}) + O(n) \qquad \text{if } n > 1$$

$$T(n) = O(1) \qquad \text{if } n <= 1$$

$$T(n) = 2T(\frac{n}{2}) + n = 2[2T(\frac{n}{2^2}) + \frac{n}{2}] + n$$

$$= 2^2 \cdot T(\frac{n}{2^2}) + 2n$$

$$= 2^2 \cdot [2T(\frac{n}{2^3}) + \frac{n}{2^2}] + 2n$$

$$= 2^3 \cdot T(\frac{n}{2^3}) + 3n \; = 2^4 \cdot T(\frac{n}{2^4}) + 4n = \cdots = 2^k \cdot T(\frac{n}{2^k}) + kn = 2^k \cdot T(1) + kn$$

$$\frac{n}{2^k} = 1 \qquad k = \log n$$

$$= 2^k \cdot T(1) + kn$$

$$= n \cdot 1 + n \log n = O(n \log n)$$