Recursion

Sarah Nadi
nadi@ualberta.ca
Department of Computing Science
University of Alberta

CMPUT 201 - Practical Programming Methodology

[With material/slides from Guohui Lin, Davood Rafei, and Michael Buro. Most examples taken from K.N. King's book]

What is Recursion?

- At its core, recursion is the idea of solving a problem by dividing it into subproblems, that each can be solved on their own

High-level View of Recursion

- If a given instance of a problem is small or simple enough, solve it

- Otherwise, reduce the problem to one or more simpler **instances of the same problem**

Recursive Functions

- A recursive function is one that calls itself

```
int factorial (int n){
  if (n <= 1)
     return 1;
  else
    return n * factorial(n-1);
}
```

```
int factorial (int n){
  return n <= 1 ? 1 : n *factorial(n-1);
}
```

- Recursive functions must have a termination condition
- We must ensure that the termination condition will eventually be met

Recursion Example: Fibonacci Numbers

`0, 1, 1, 2, 3, 5, 7, 13, 21, 34, 55, 89, 144, …`

Basically each number is the sum of the previous two numbers:

$$F_n = F_{n-1} + F_{n-2}$$
$$F_0 = 0$$
$$F_1 = 1$$

Let's write a recursive function to calculate F_n, given any n

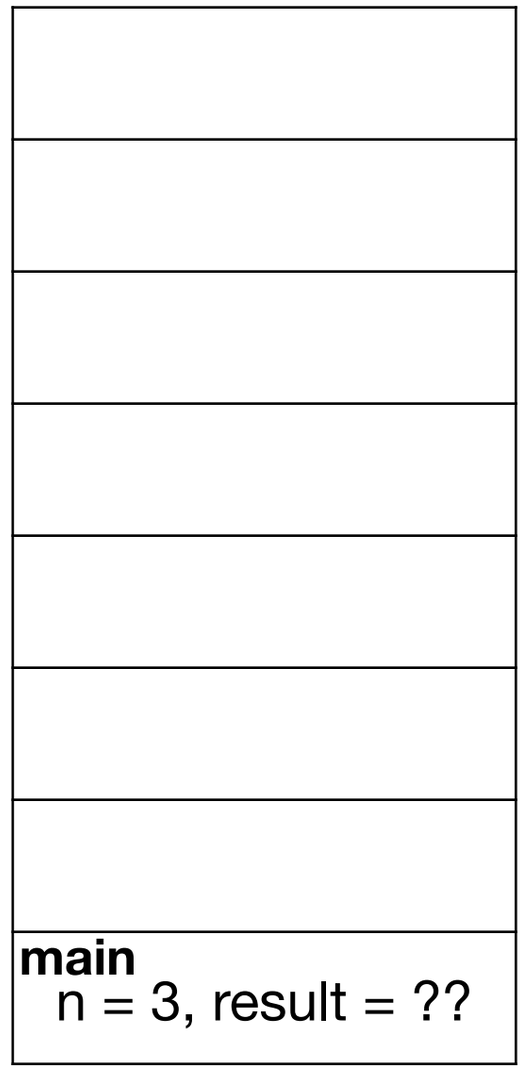
Fibonacci Numbers Solution

```
int fib(int n)
{
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}

int main ()
{
  int n = 9;
  printf("%d", fib(n));
  return 0;
}
```

Tracing through recursive function

```
int fib(int n)
{
    if (n <= 1)
        return n;
    int result = fib(n-1) + fib(n-2);
    return result;
}

int main ()
{
   int n = 3;
    int result = fib(n);
   printf("%d", result);
   return 0;
}
```

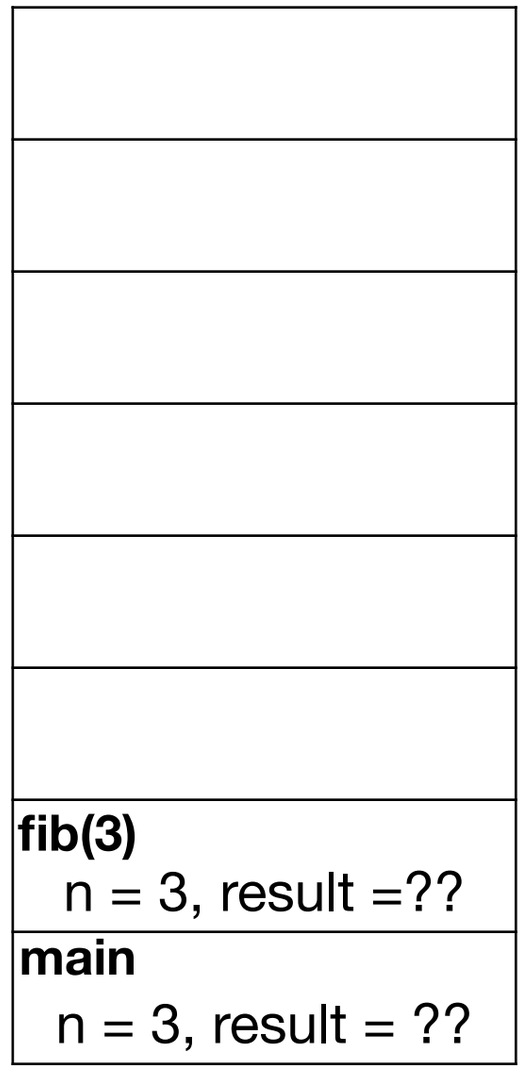
Target output: 2 (remember 0, 1, 1, 2)



Tracing through recursive function

```
int fib(int n)
{
    if (n <= 1)
        return n;
    int result = fib(n-1) + fib(n-2);
    return result;
}

int main ()
{
   int n = 3;
    int result = fib(n);
   printf("%d", result);
   return 0;
}
```

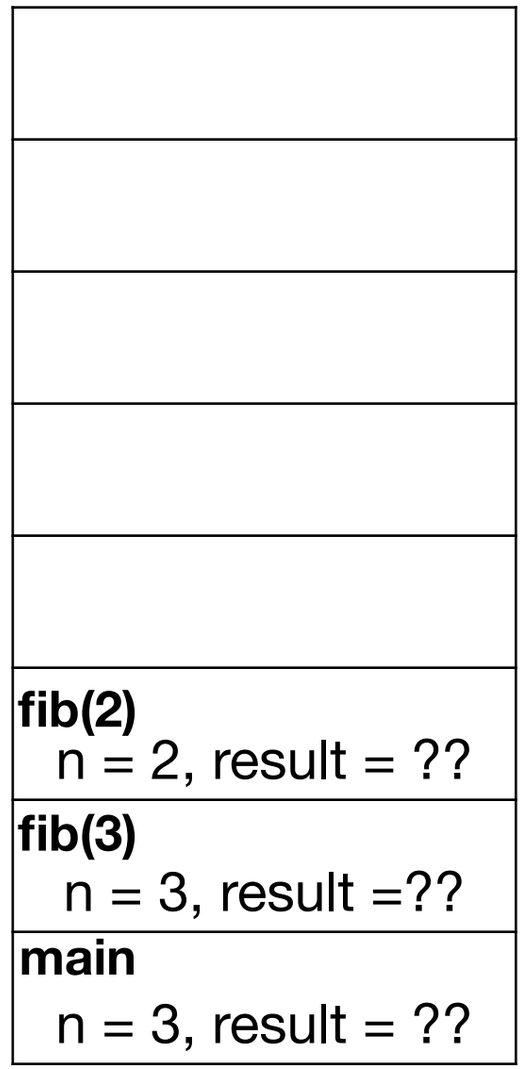
Target output: 2 (remember 0, 1, 1, 2)



Tracing through recursive function

```
int fib(int n)
{
    if (n <= 1)
        return n;
    int result = fib(n-1) + fib(n-2);
    return result;
}

int main ()
{
   int n = 3;
    int result = fib(n);
   printf("%d", result);
   return 0;
}
```

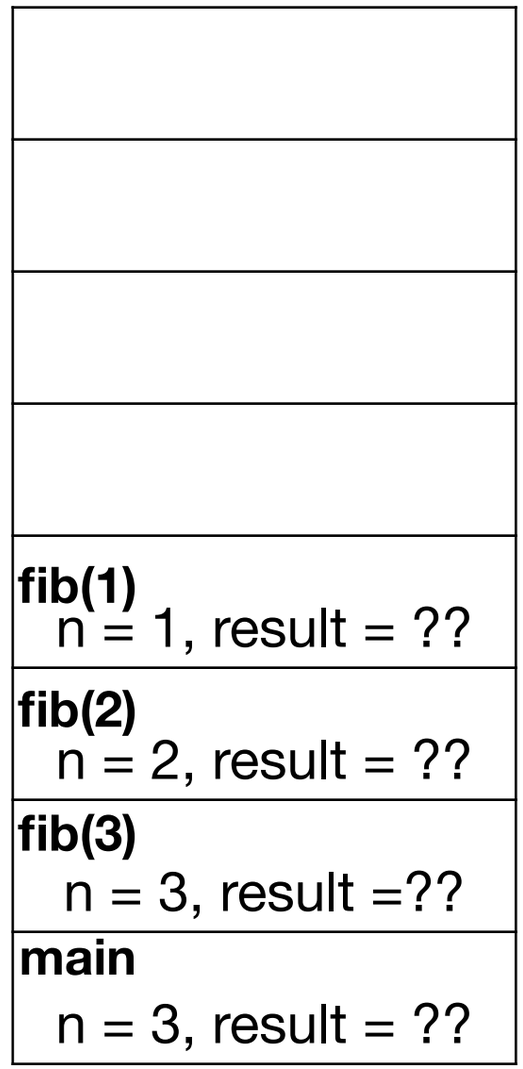
Target output: 2 (remember 0, 1, 1, 2)



Tracing through recursive function

```
int fib(int n)
{
    if (n <= 1)
        return n;
    int result = fib(n-1) + fib(n-2);
    return result;
}

int main ()
{
   int n = 3;
    int result = fib(n);
   printf("%d", result);
   return 0;
}
```

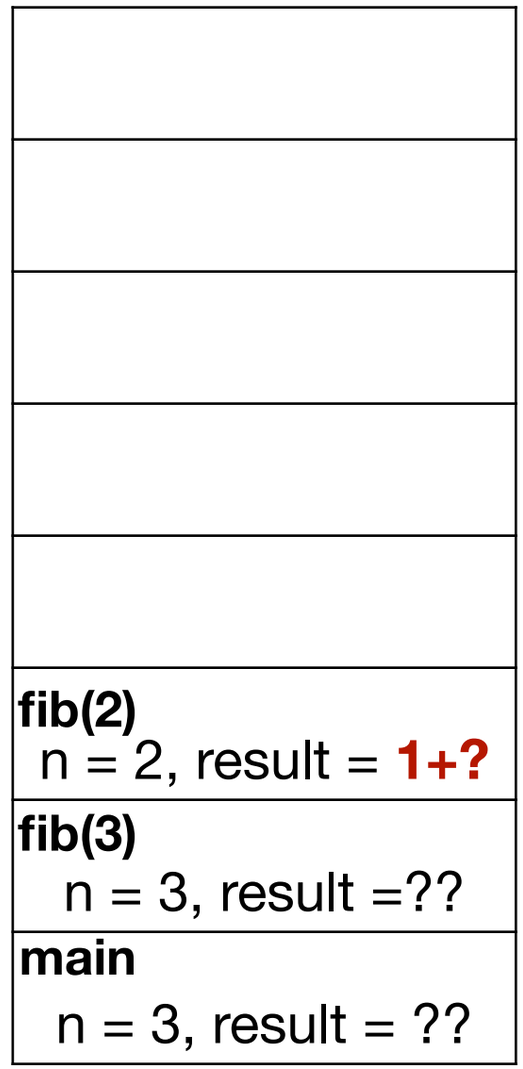
Target output: 2 (remember 0, 1, 1, 2)



Tracing through recursive function

```
int fib(int n)
{
    if (n <= 1)
        return n;
    int result = fib(n-1) + fib(n-2);
    return result;
}

int main ()
{
   int n = 3;
    int result = fib(n);
   printf("%d", result);
   return 0;
}
```

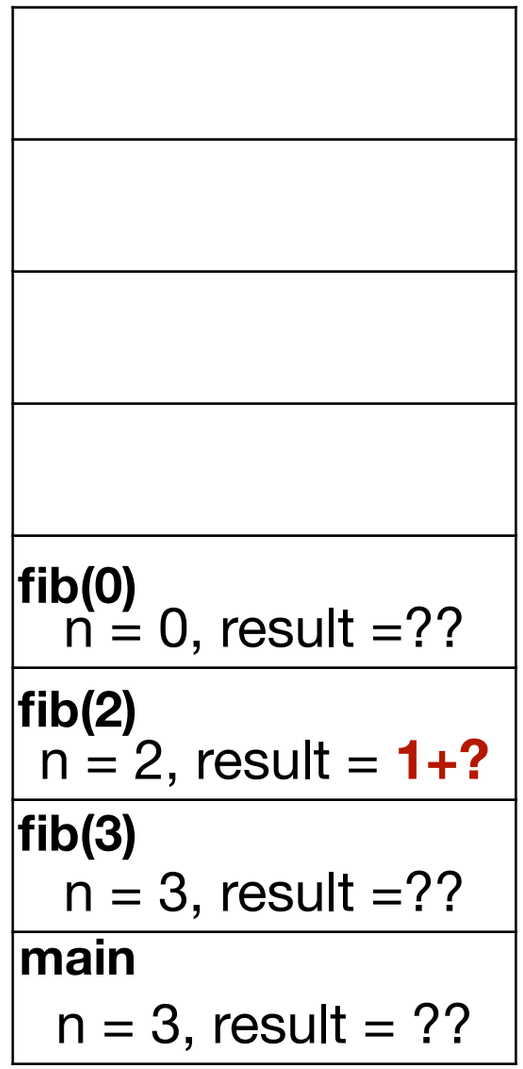
Target output: 2 (remember 0, 1, 1, 2)



Tracing through recursive function

```
int fib(int n)
{
    if (n <= 1)
        return n;
    int result = fib(n-1) + fib(n-2);
    return result;
}

int main ()
{
   int n = 3;
    int result = fib(n);
   printf("%d", result);
   return 0;
}
```

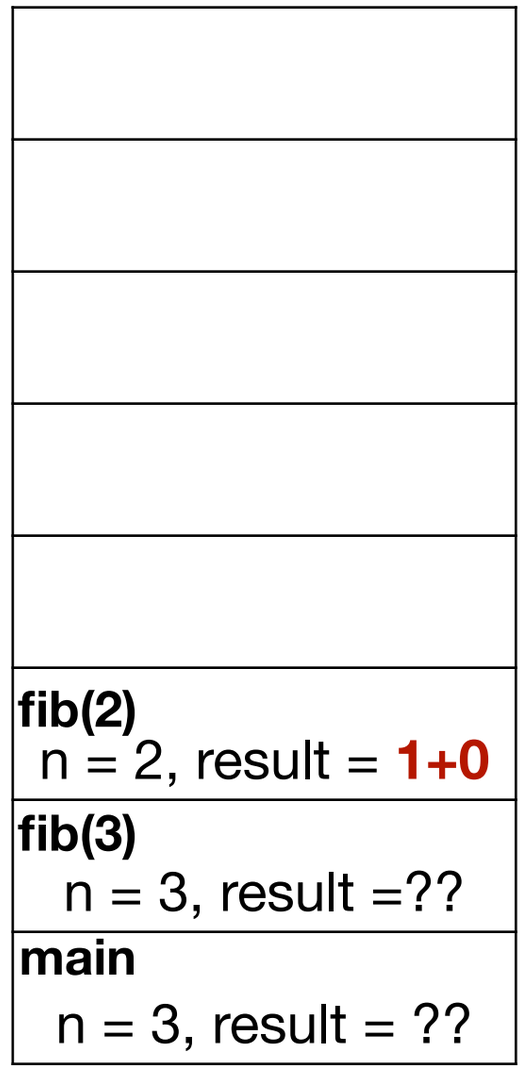
Target output: 2 (remember 0, 1, 1, 2)



Tracing through recursive function

```
int fib(int n)
{
    if (n <= 1)
        return n;
    int result = fib(n-1) + fib(n-2);
    return result;
}

int main ()
{
   int n = 3;
    int result = fib(n);
   printf("%d", result);
   return 0;
}
```

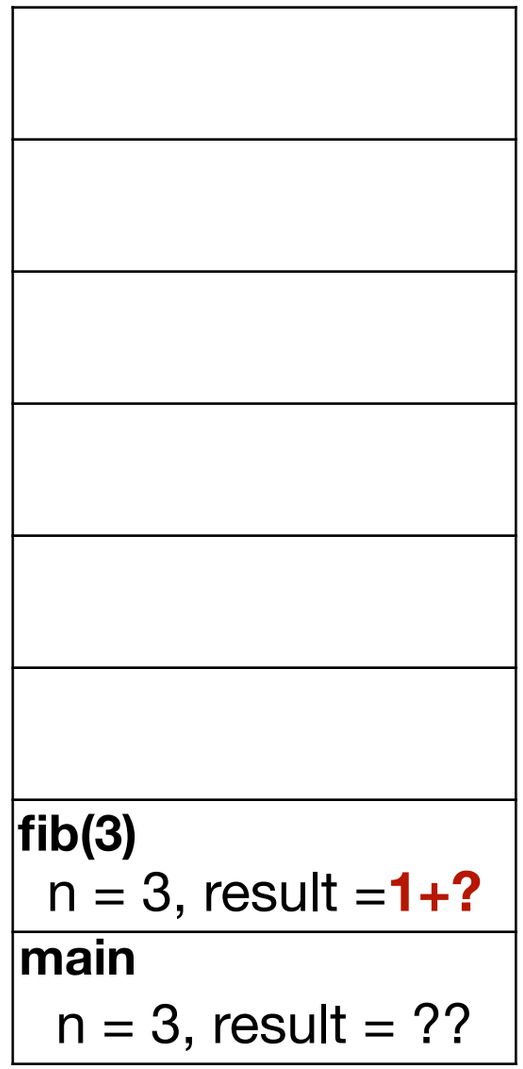
Target output: 2 (remember 0, 1, 1, 2)



Tracing through recursive function

```
int fib(int n)
{
    if (n <= 1)
        return n;
    int result = fib(n-1) + fib(n-2);
    return result;
}

int main ()
{
   int n = 3;
    int result = fib(n);
   printf("%d", result);
   return 0;
}
```

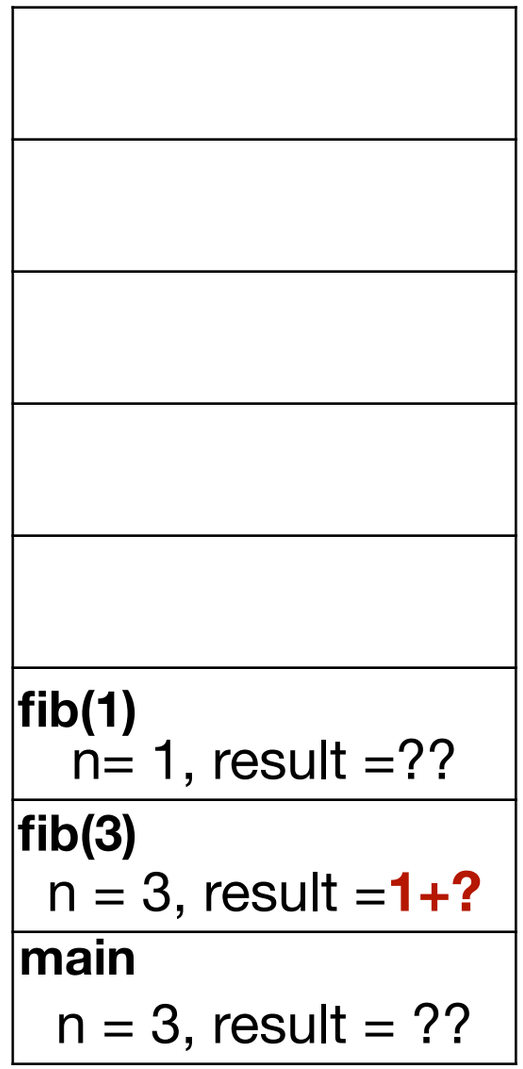
Target output: 2 (remember 0, 1, 1, 2)



Tracing through recursive function

```
int fib(int n)
{
    if (n <= 1)
        return n;
    int result = fib(n-1) + fib(n-2);
    return result;
}

int main ()
{
   int n = 3;
    int result = fib(n);
   printf("%d", result);
   return 0;
}
```

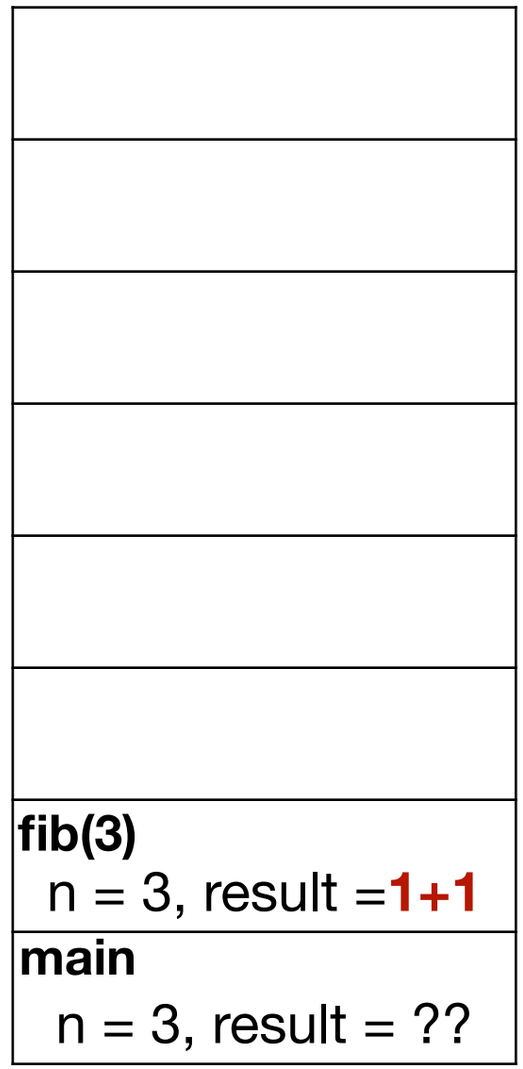
Target output: 2 (remember 0, 1, 1, 2)



Tracing through recursive function

```
int fib(int n)
{
    if (n <= 1)
        return n;
    int result = fib(n-1) + fib(n-2);
    return result;
}

int main ()
{
   int n = 3;
    int result = fib(n);
   printf("%d", result);
   return 0;
}
```

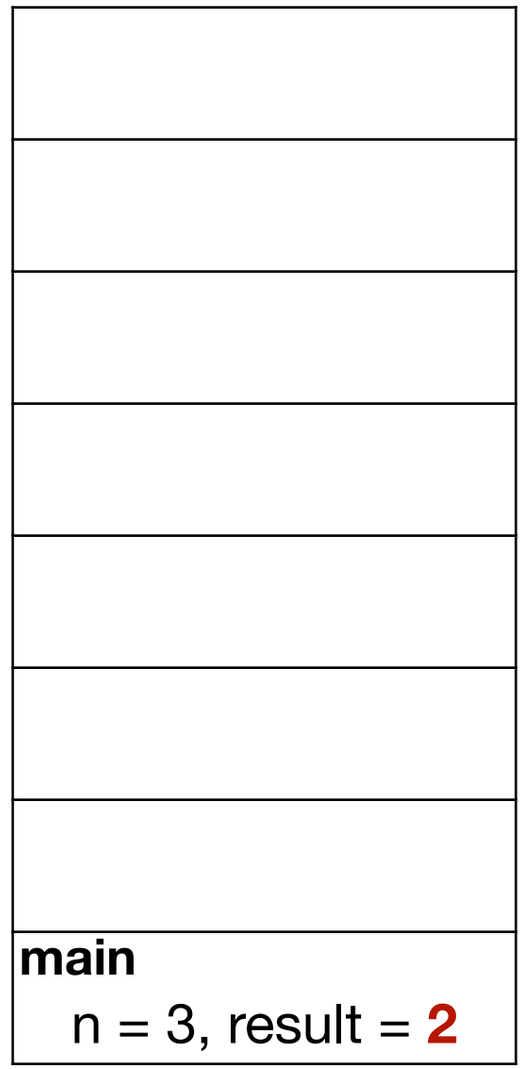
Target output: 2 (remember 0, 1, 1, 2)



Tracing through recursive function

```
int fib(int n)
{
    if (n <= 1)
        return n;
    int result = fib(n-1) + fib(n-2);
    return result;
}

int main ()
{
   int n = 3;
    int result = fib(n);
   printf("%d", result);
   return 0;
}
```

Target output: 2 (remember 0, 1, 1, 2)



Recursion Example: QuickSort

- A comparison-based sorting algorithm, whose goal is to sort an array of integers into a non-decreasing order

- Good example of a divide and conquer approach

- The mathematical algorithm:

  - Choose an array element e (“the partitioning element”), then rearrange the array so that elements 1,…, i-1 are less than or equal to e, element i contains e, and elements i+1,…, n are greater than or equal to e.

  - Sort elements 1,…i-1 using Quicksort recursively

  - Soft elements i+1,…, n using Quicksort recursively

Partition in Quicksort

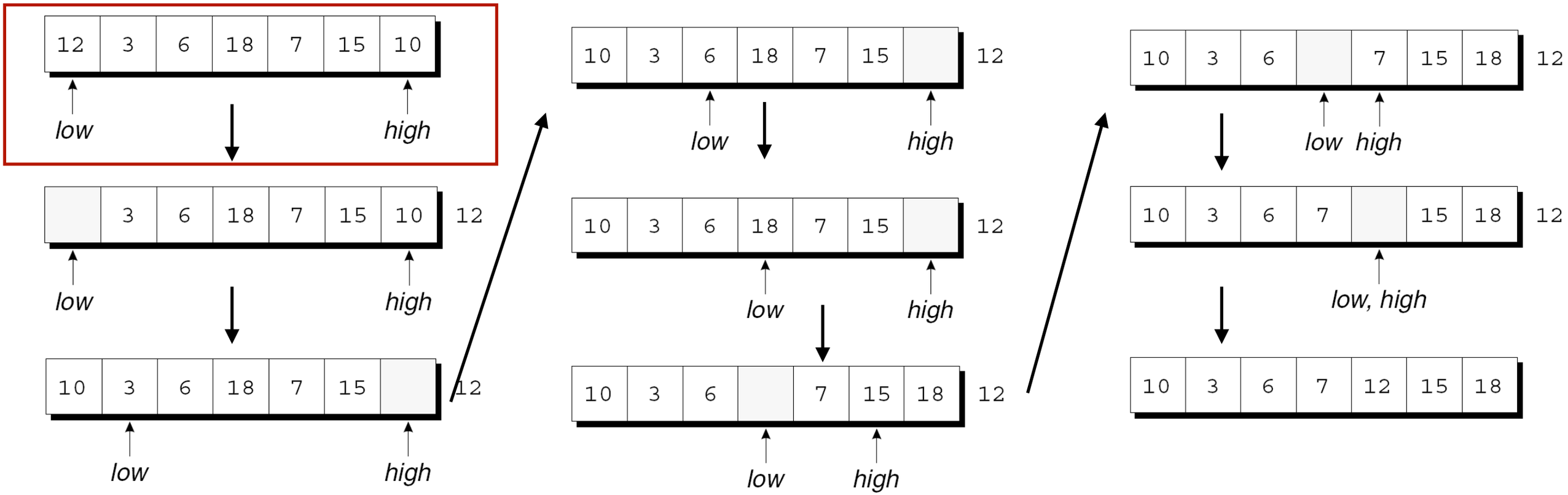
- Step 1 in the algorithm is critical since it affects the rest of the algorithm.

- Let's create a partition algorithm that relies on two "markers" low and high, which keep track of positions in the array

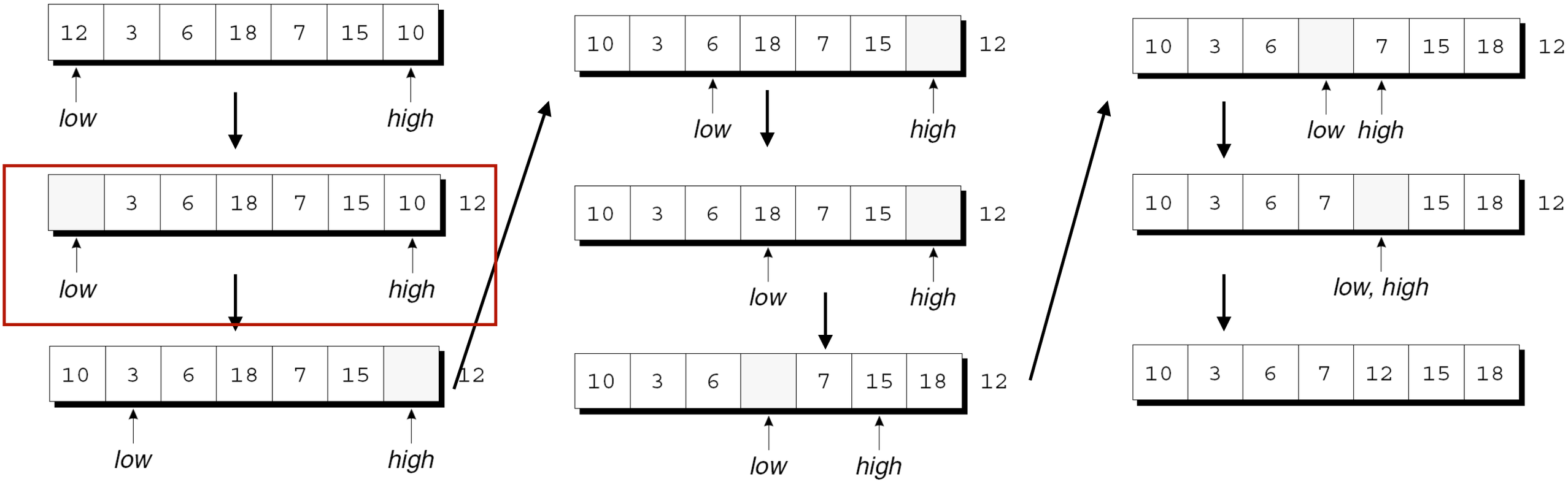
Quicksort: Partitioning the Array

- Initially, low points to the first element; high points to the last.
- We copy the first element (the partitioning element) into a temporary location, leaving a “hole” in the array.
- Next, we move high across the array from right to left until it points to an element that’s smaller than the partitioning element.
- We then copy the element into the hole that low points to, which creates a new hole (pointed to by high).
- We now move low from left to right, looking for an element that’s larger than the partitioning element. When we find one, we copy it into the hole that high points to.
- The process repeats until low and high meet at a hole.
- Finally, we copy the partitioning element into the hole

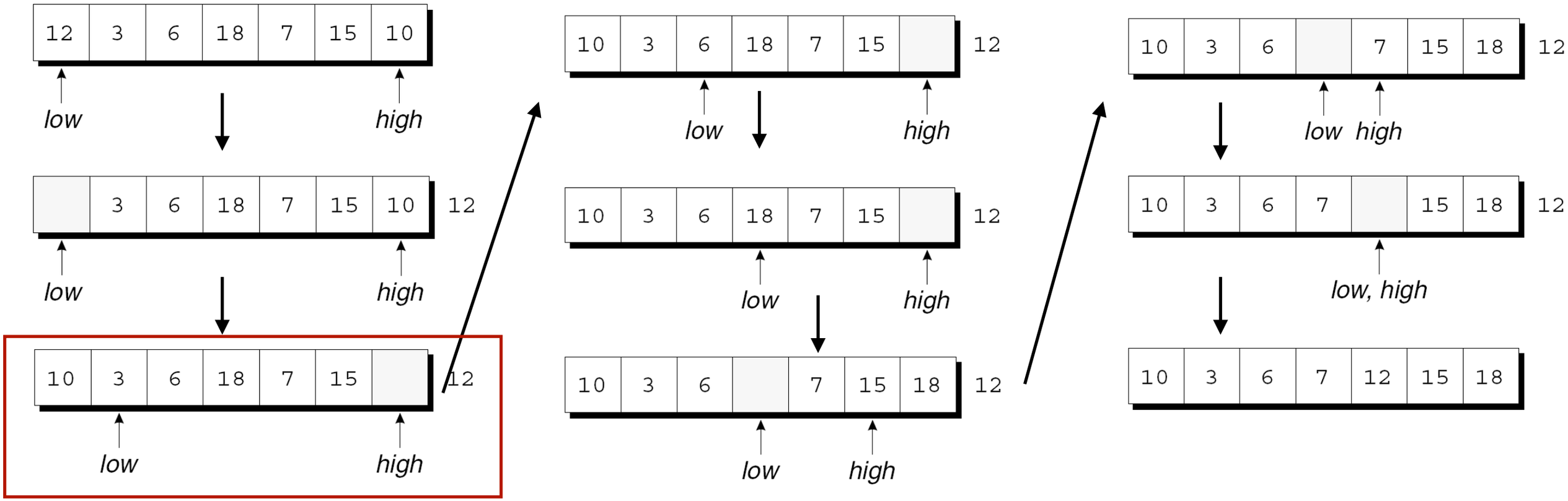
Quicksort: Partitioning the Array



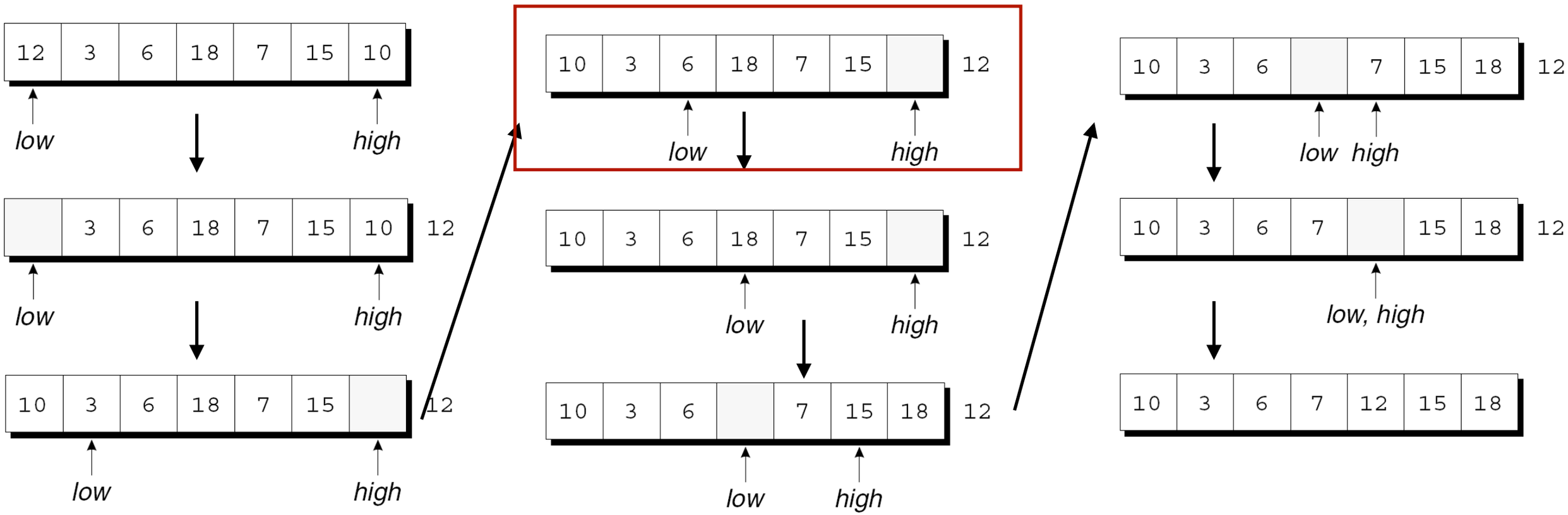
Quicksort: Partitioning the Array



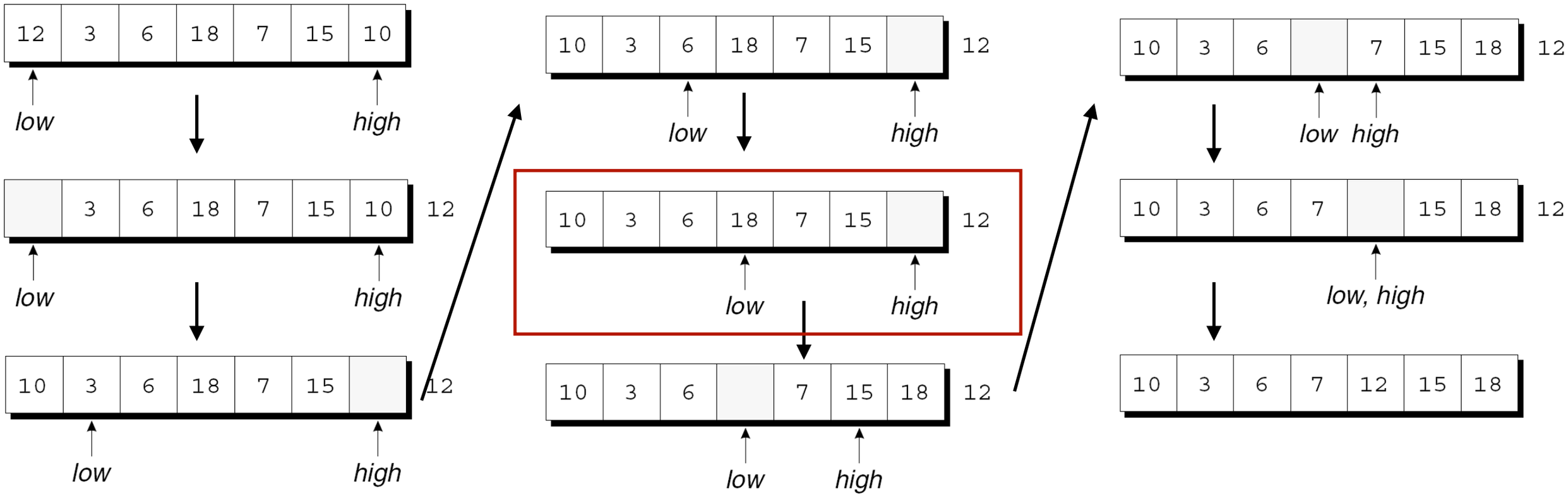
Quicksort: Partitioning the Array



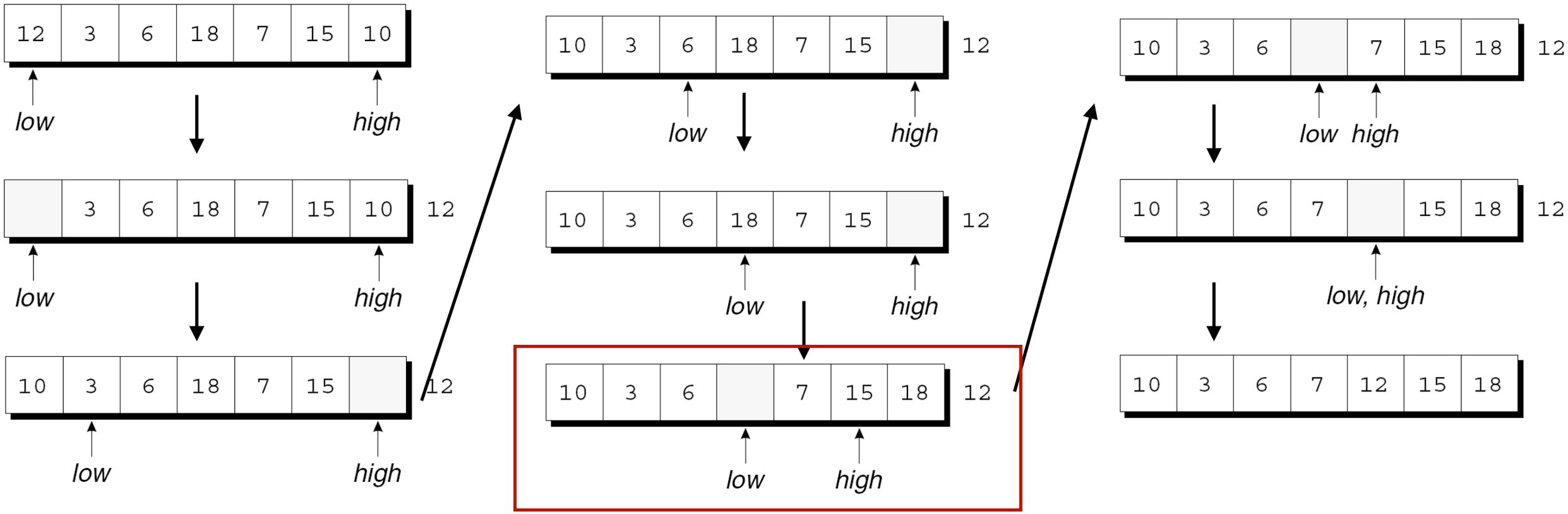
Quicksort: Partitioning the Array



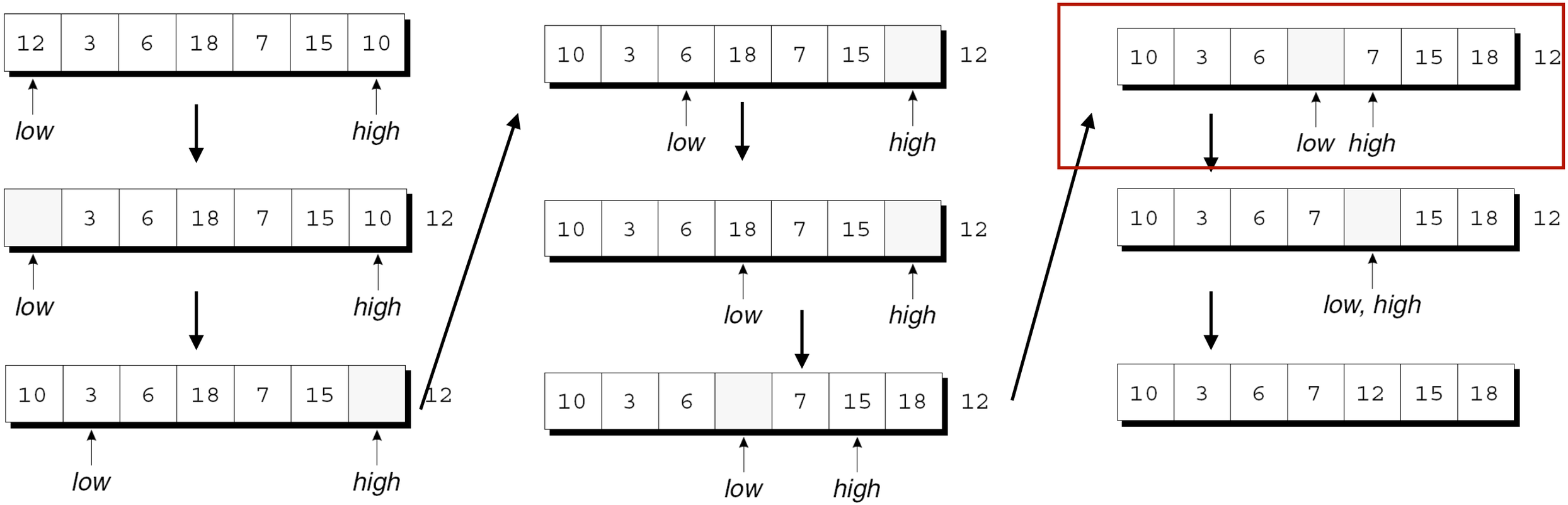
Quicksort: Partitioning the Array



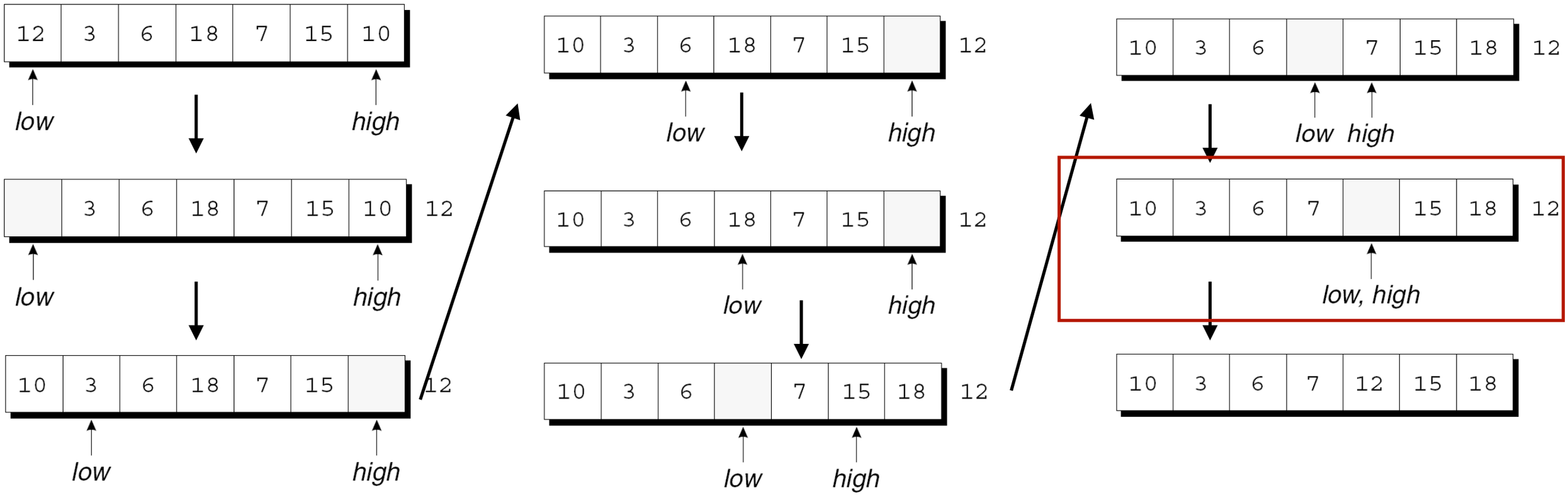
Quicksort: Partitioning the Array



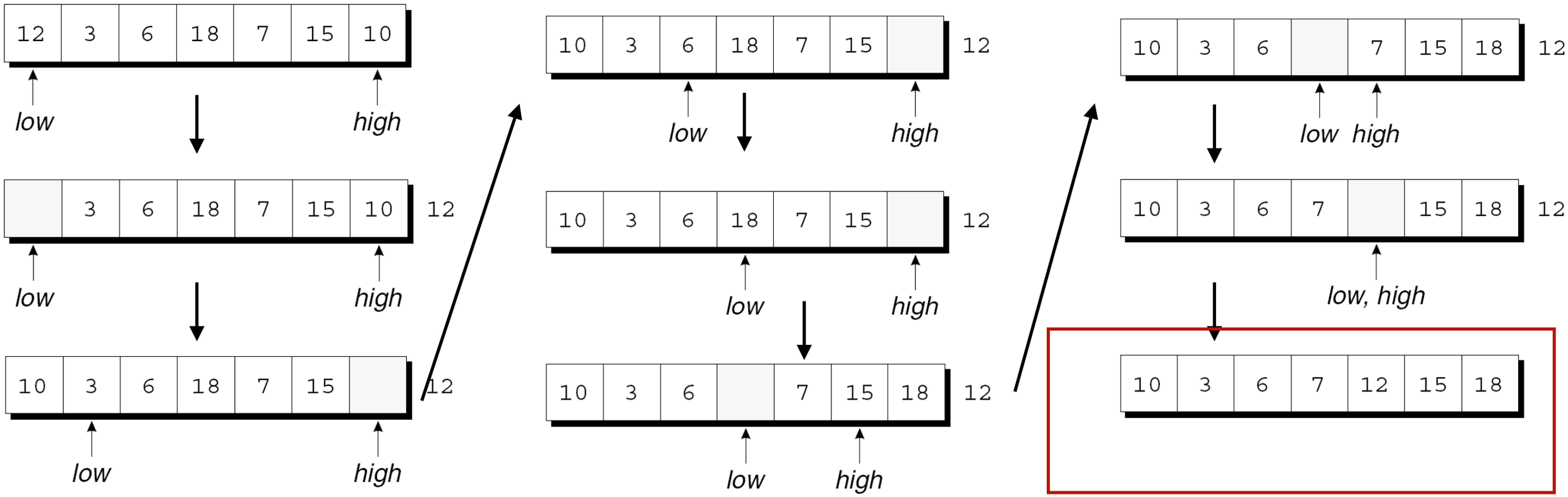
Quicksort: Partitioning the Array



Quicksort: Partitioning the Array



Quicksort: Partitioning the Array



Now, all elements less than or equal to 12 are on the left and all elements greater than or equal to 12 are on the right. Now, we can repeat the process for the first four elements of the array and the last two elements of the array