Sorting Algorithms

Instructor: Jeeho Ryoo

Announcements

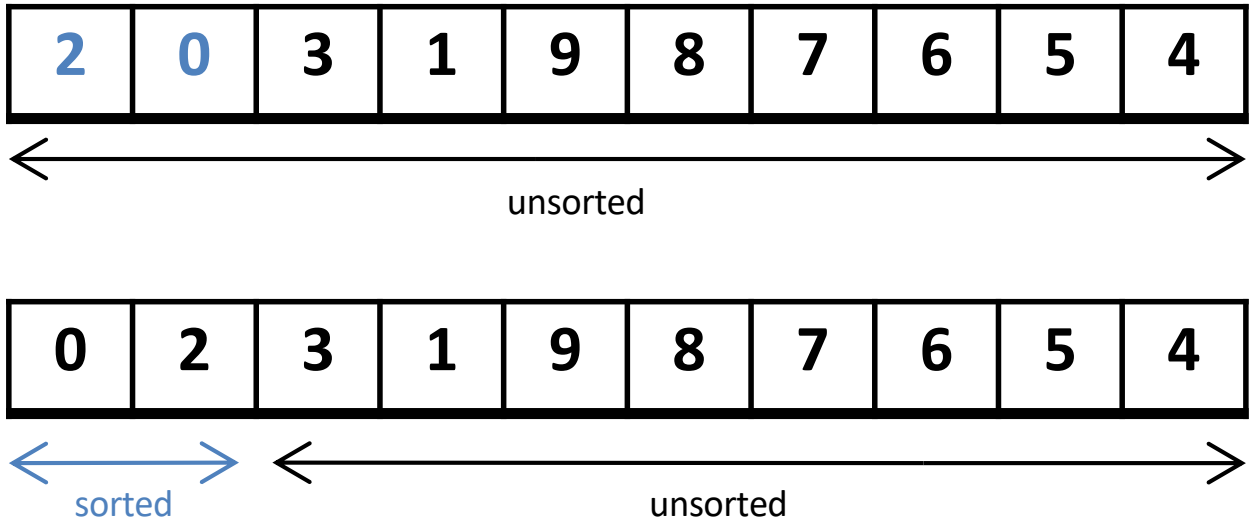
- Quiz in the last 35 minutes
- Comments quiz grading
- Assignment 2
    - Names
    - #include…
    - Compiler flags
- Test distribution

Sorting Algorithms

- Sorting algorithms are some of the first algorithms to be developed and are widely used today
- Today we are going to look at a couple of popular sorting algorithms
    - Insertion sort
    - Selection sort
    - Merge sort
    - Quick sort
- They will help us sort an array in ascending order, but each has its own approach

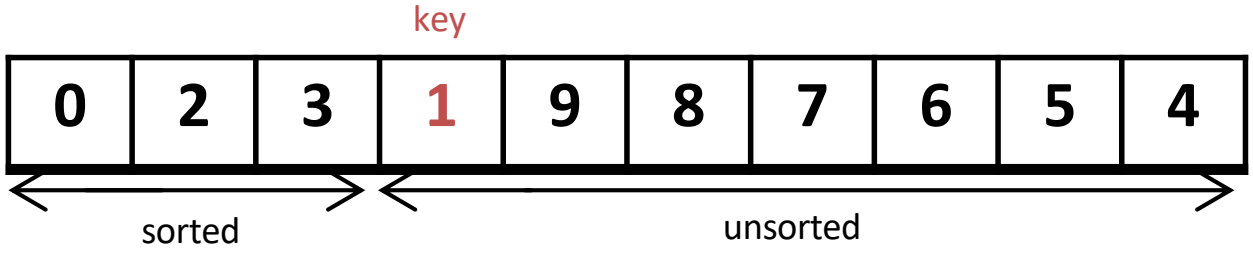
Insertion Sort

The array is split into two parts, a sorted part and an unsorted part
Values from the unsorted part are picked and sorted properly into the sorted part

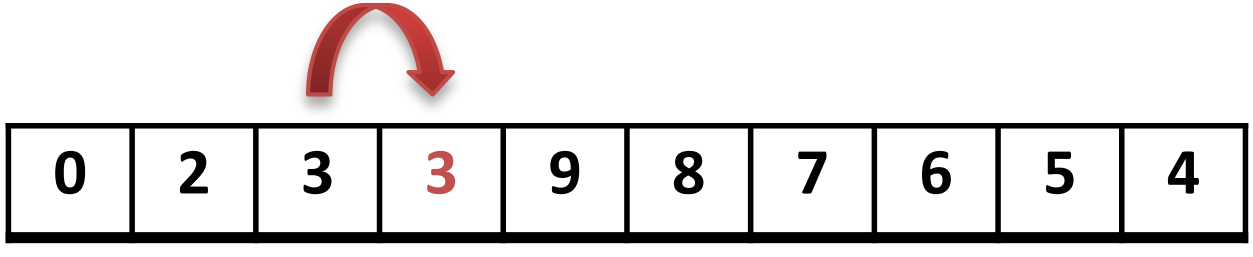


Insertion Sort (cont.)

Let's jump to the third loop iteration, the element in orange, known as the key compares itself with its left most element

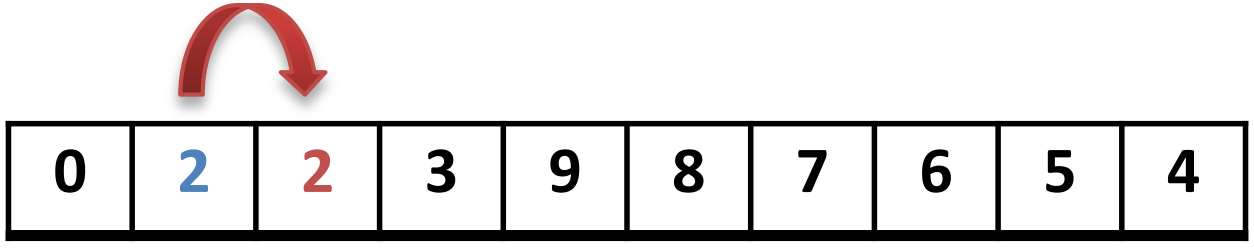


If the left most element is greater then the key's value, then we update key's index to the greater value



Insertion Sort (cont.)

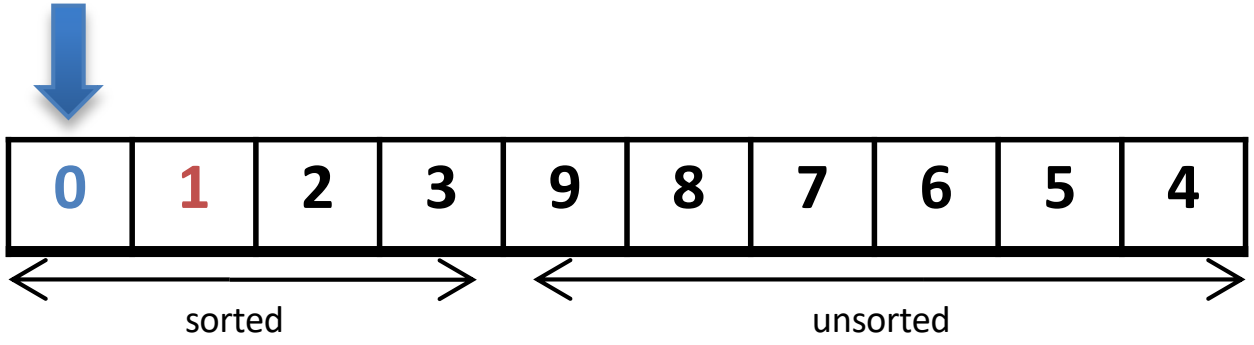
We now check the next left most element and compare it to key's value which is still 1



Since next left most element is still greater than key's value, we update the element right of it

Insertion Sort (cont.)

Again, we check the next left most element and compare it to key's value



Since next left most element is less than key's value, we update the element right of it to keys value

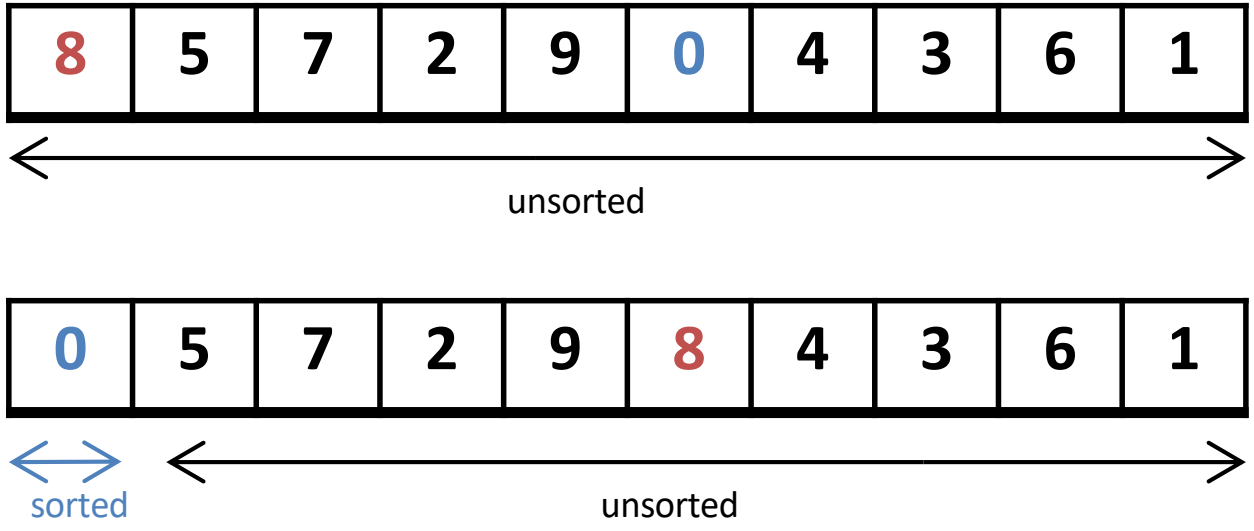
Insertion Sort (cont.)

```
int key, y;
int n = 10;
int arr[] = { 2, 0, 3, 1, 9, 8, 7, 6, 5, 4 };

for (int x = 1; x < n; x++)
{
        key = arr[x];
        y = x - 1;

       while (y >= 0 && arr[y] > key)
       {
              arr[y + 1] = arr[y];
              y = y - 1;
        }

        arr[y+1] = key;
}
```

Selection Sort

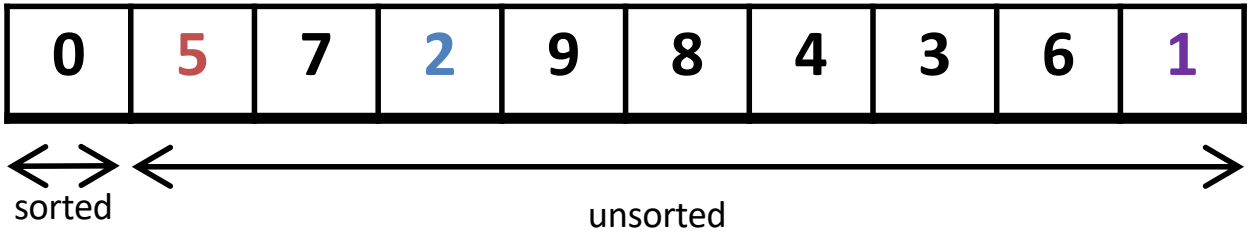
Like Insertion sort the array is split into the sorted and unsorted parts

Each iteration the lowest element from the unsorted part gets put in front of the unsorted part swapping values with the iteration index



Selection Sort (cont.)

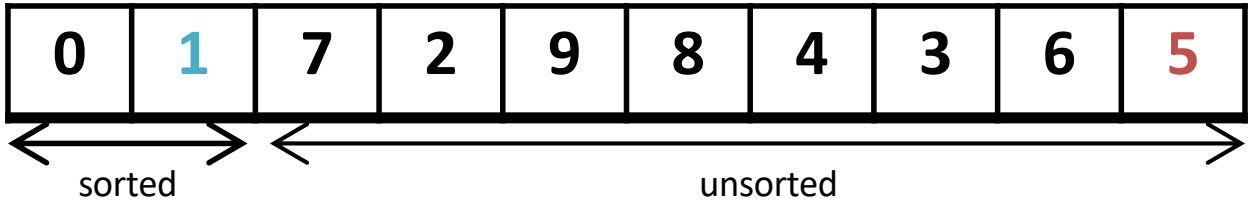
We start each iteration with the current index being set as the 'minimum' (*min*)
In a nested loop it iterates through the unsorted part replacing the *min* value with any smaller



Here we start by setting minimum to index 1
Then since 2 is smaller then 5, minimum gets set to index 3
Then since 1 is smaller then 2, minimum gets set to index 9

Selection Sort (cont.)

Index 9 is then swapped with the parent loop index, which is 1



And the pattern continues with index 2

Selection sort (cont.)

- Walkthrough the code step by step and use the visuals to help

```
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
```

```
int y, min;
int n = 10;
int arr[] = { 8, 5, 7, 2, 9, 0, 4, 3, 6, 1};

for (int x = 0; x < n-1; x++)
{
        min = x;
        for (y = x+1; y < n; y++)
        {
            if (arr[y] < arr[min])
            {
                min = y;
            }
        }
        swap(&arr[min], &arr[x]);
}
```

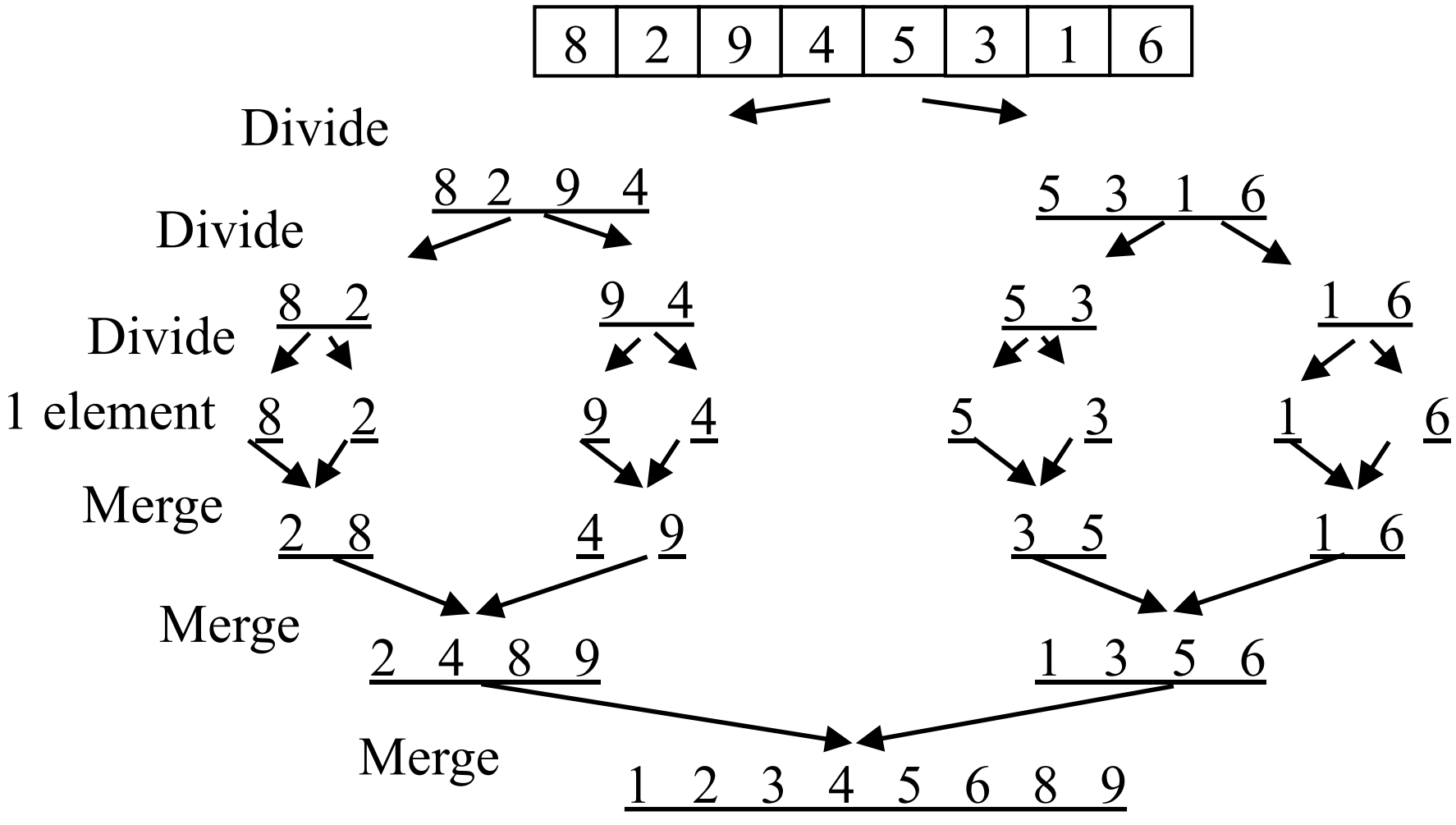
“Divide and Conquer”

- Very important strategy in computer science:
  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine these solutions to get overall solution
- Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves → Mergesort

Mergesort

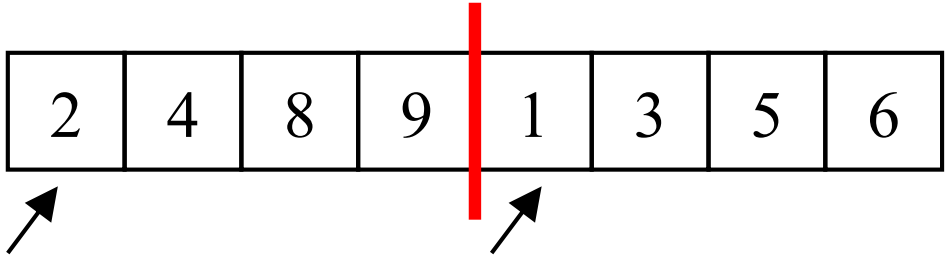
- Divide it in two at the midpoint
- Conquer each side in turn (by recursive sorting)
- Merge two halves together

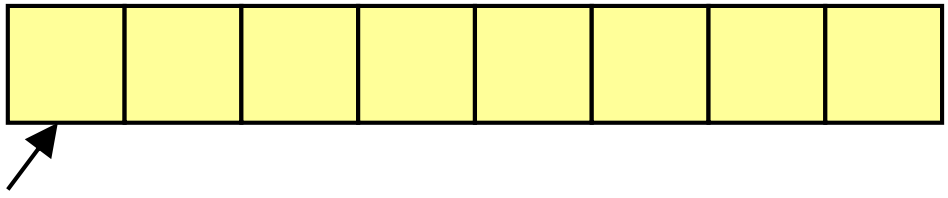
Mergesort Example



Auxiliary Array

The merging requires an auxiliary array.

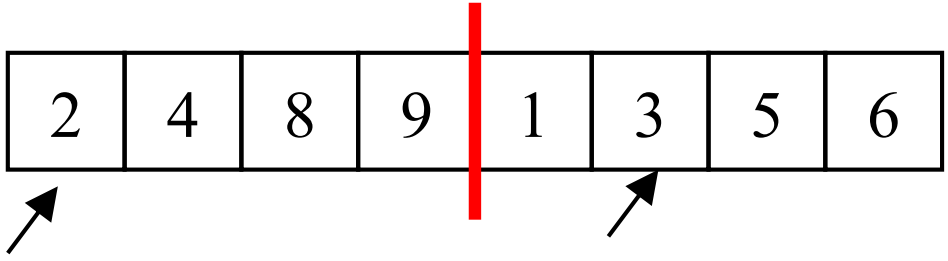


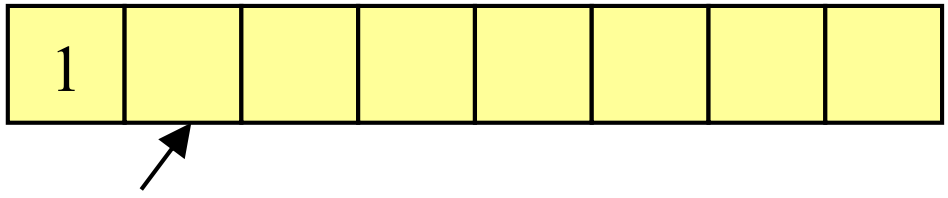


Auxiliary array

Auxiliary Array

The merging requires an auxiliary array.

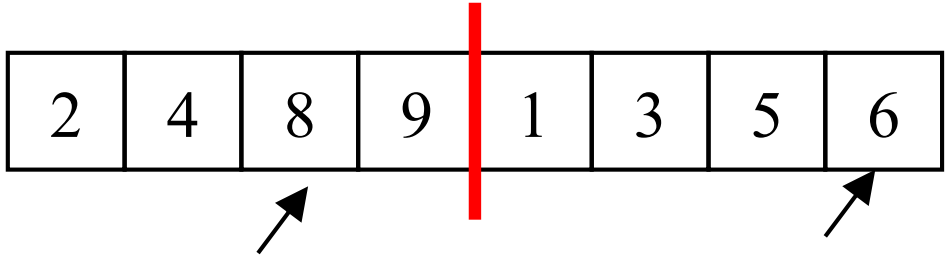


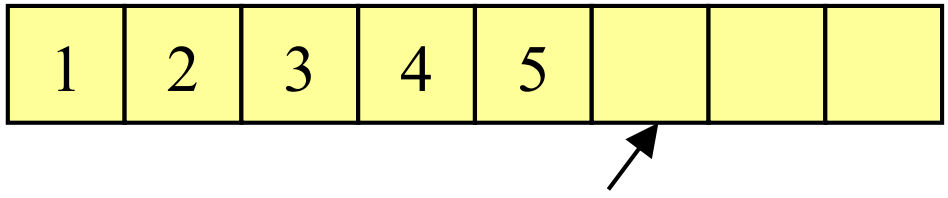


Auxiliary array

Auxiliary Array

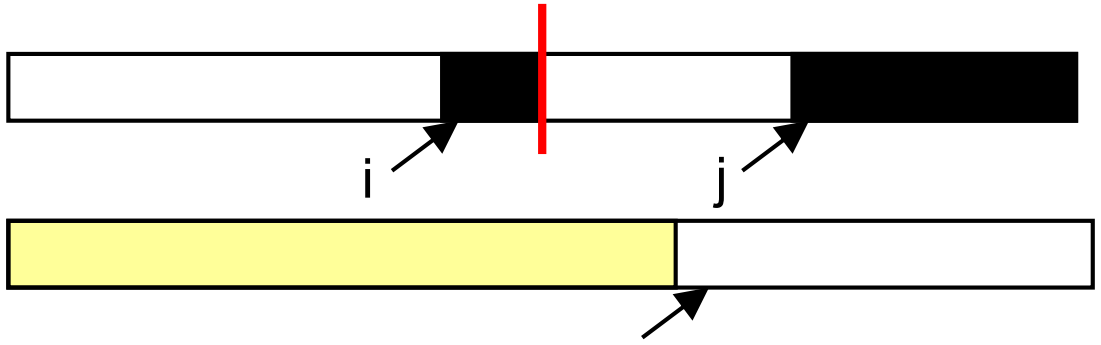
The merging requires an auxiliary array.



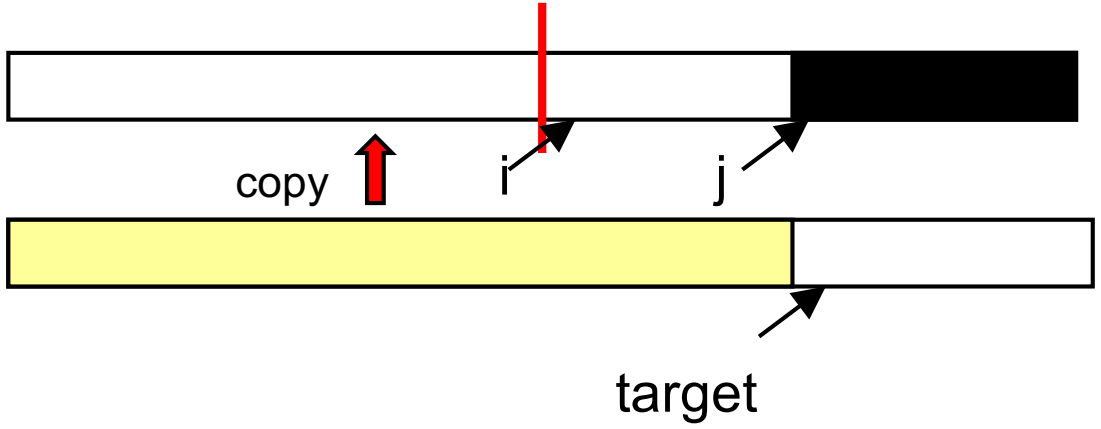


Auxiliary array

Merging

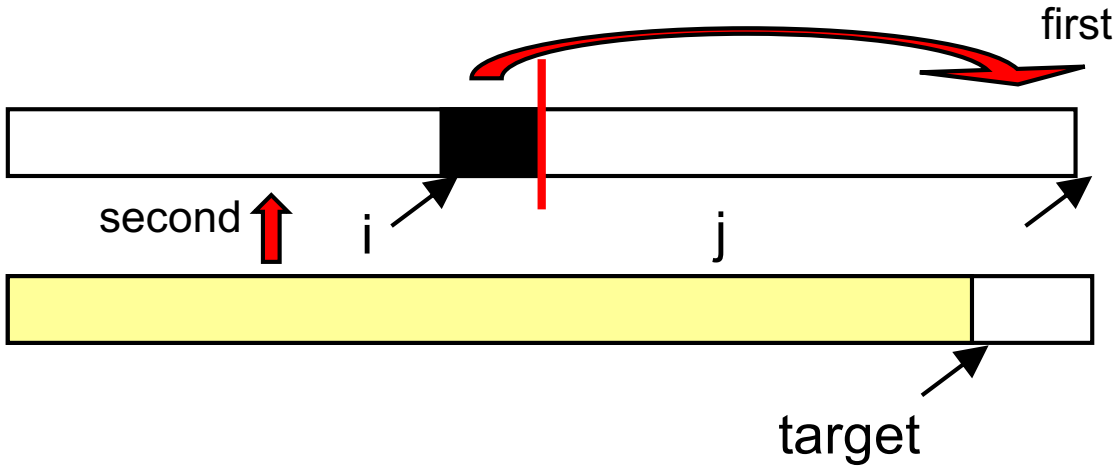


normal



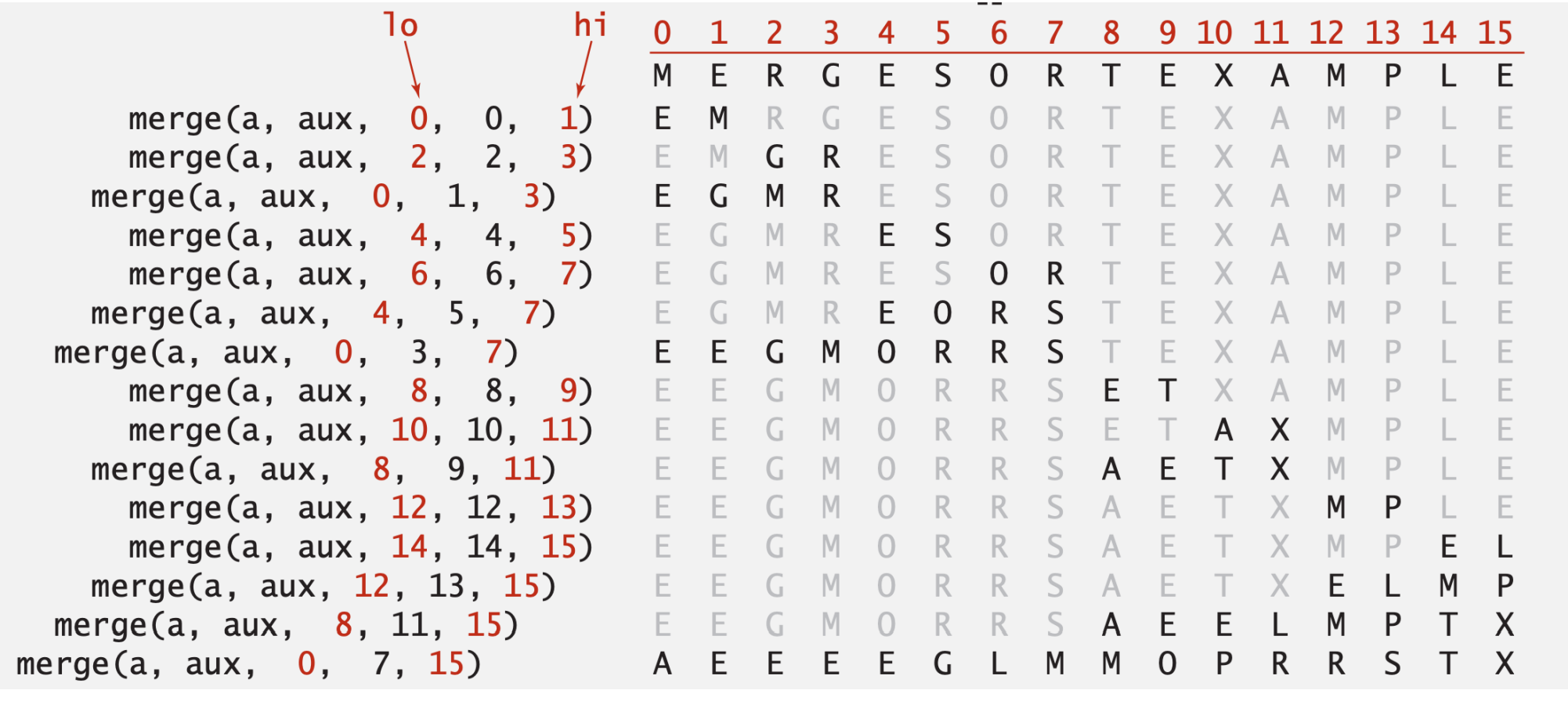
Left completed first

Merging



Right completed first

Visual Diagram of Mergesort



Mergesort

```
void mergeSort(int* arr, int n) {
    if (n > 0){
        int left[n/2], right[n/2];
        for (int i = 0; i < n/2; i++) {
            left[i] = arr[i];
        }
        for (int i = n/2; i < n; i++) {
            right[i - n/2] = arr[i];
        } // recursively sort the two halves
        mergeSort(left);
        mergeSort(right);
        merge(arr, left, right);
    }
}
```

Mergesort

```
void merge(int* arr, int* left, int* right, int m,
int n){
    int i1 = 0; // index into left side
    int i2 = 0; // index into right side
    for (int i = 0; i < m + n; i++) {
        if (i2 >= n || (i1 < m && left[i1] <=
right[i2])) {
            // take from left
            result[i] = left[i1];
            i1++;
        } else {
            // take from right
            result[i] = right[i2];
            i2++;
        }
    }
}
```

Quicksort

- Quicksort is a sorting algorithm that is often faster than most other types of sorts.

- However, although it has an average **O(n log n)** time complexity, it also has a worst-case **O(n^2)** time complexity, though this rarely occurs.

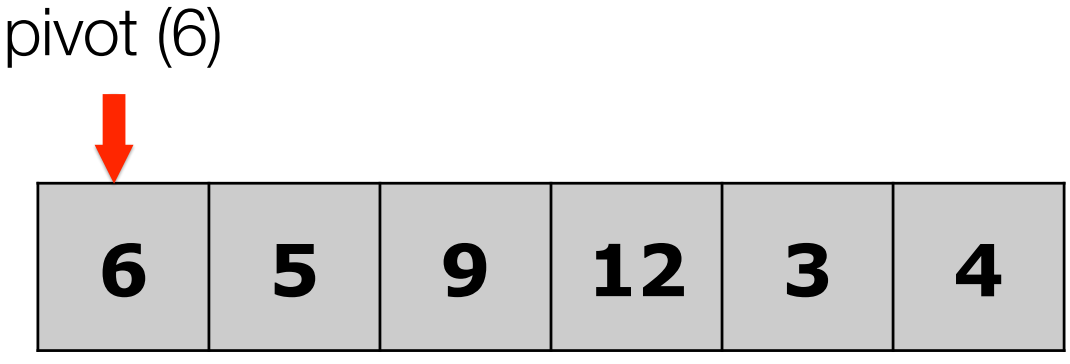
Quicksort

- Quicksort is another divide-and-conquer algorithm.
- The basic idea is to **divide** a list into two smaller sub-lists: **the low elements and the high elements**. Then, the algorithm can recursively sort the sub-lists.

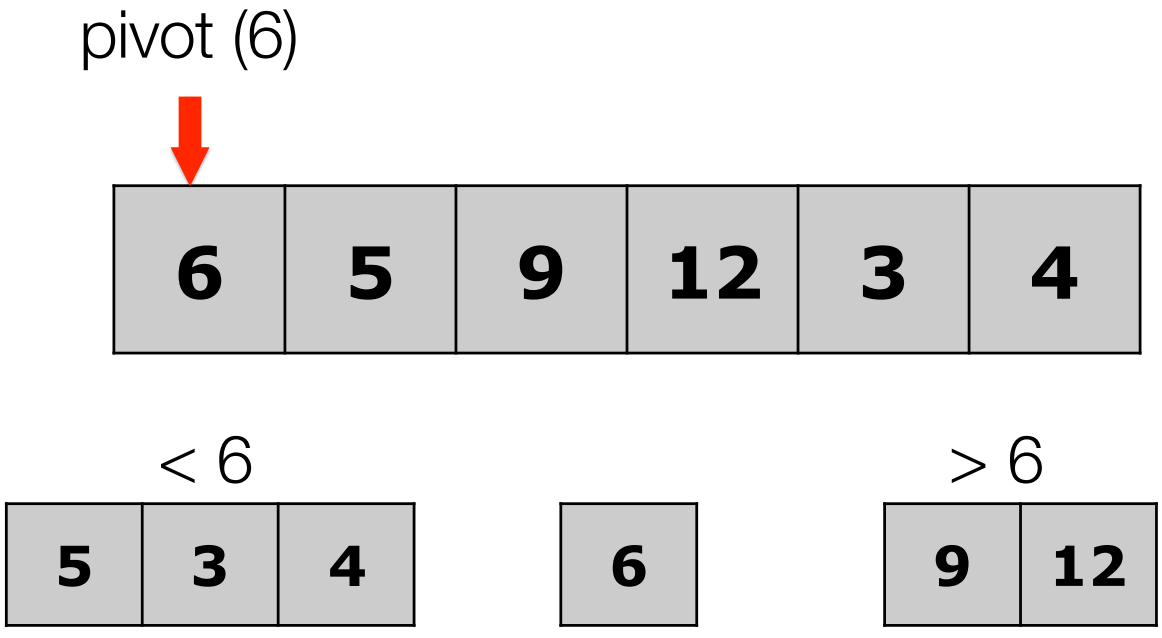
Quicksort

- **Pick an element**, called a **pivot**, from the list
- **Reorder** the list so that all elements with **values less than the pivot come before the pivot**, while all elements with values **greater than the pivot come after it**. After this partitioning, the pivot is in its final position. This is called the partition operation.
- **Recursively apply the above steps to the sub-list of elements** with smaller values and separately to the sub-list of elements with greater values.
- The **base case** of the recursion is for **lists of 0 or 1** elements, which do not need to be sorted.

Quicksort



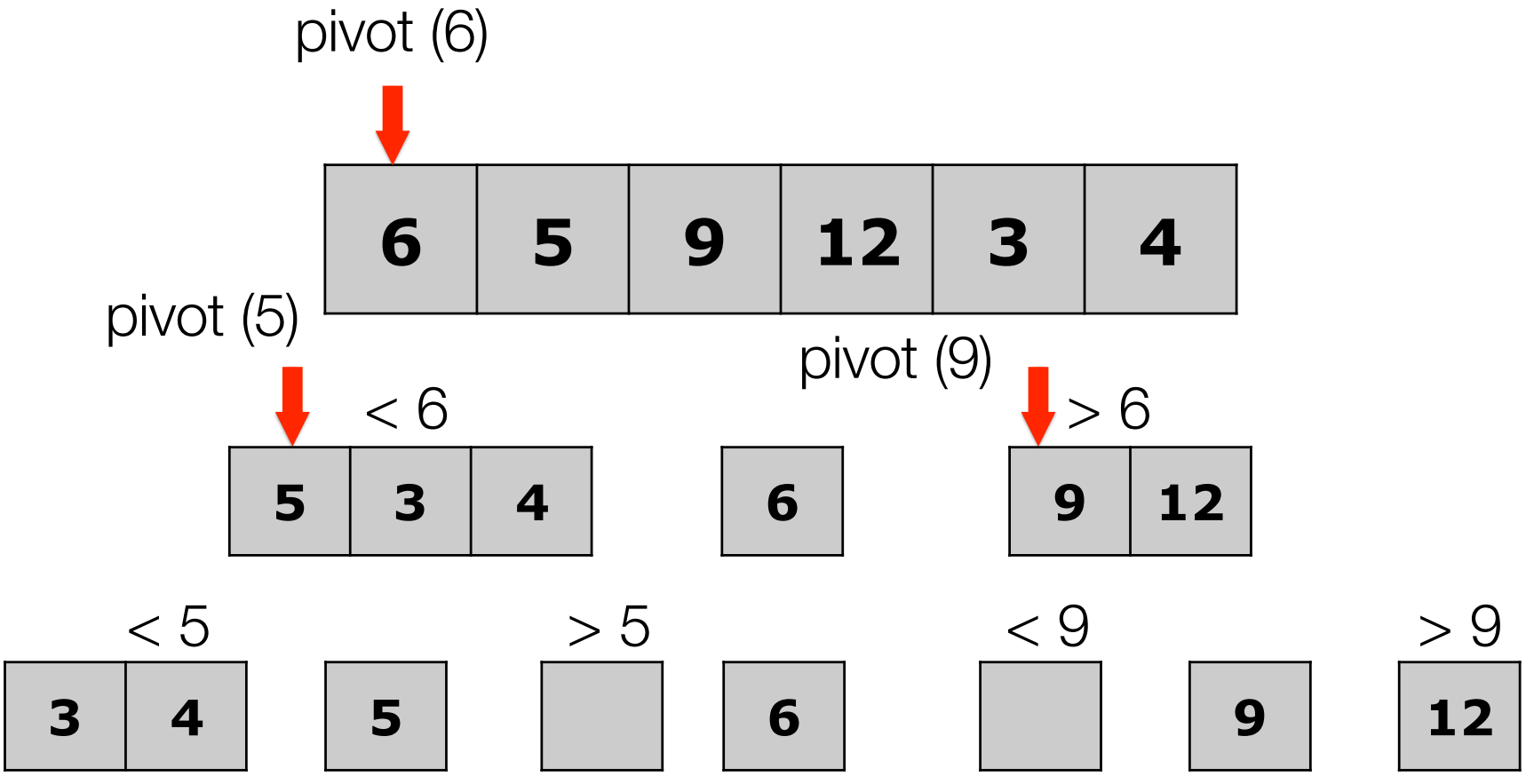
Quicksort



Partition into two new lists -- less than the pivot on the left, and greater than the pivot on the right.

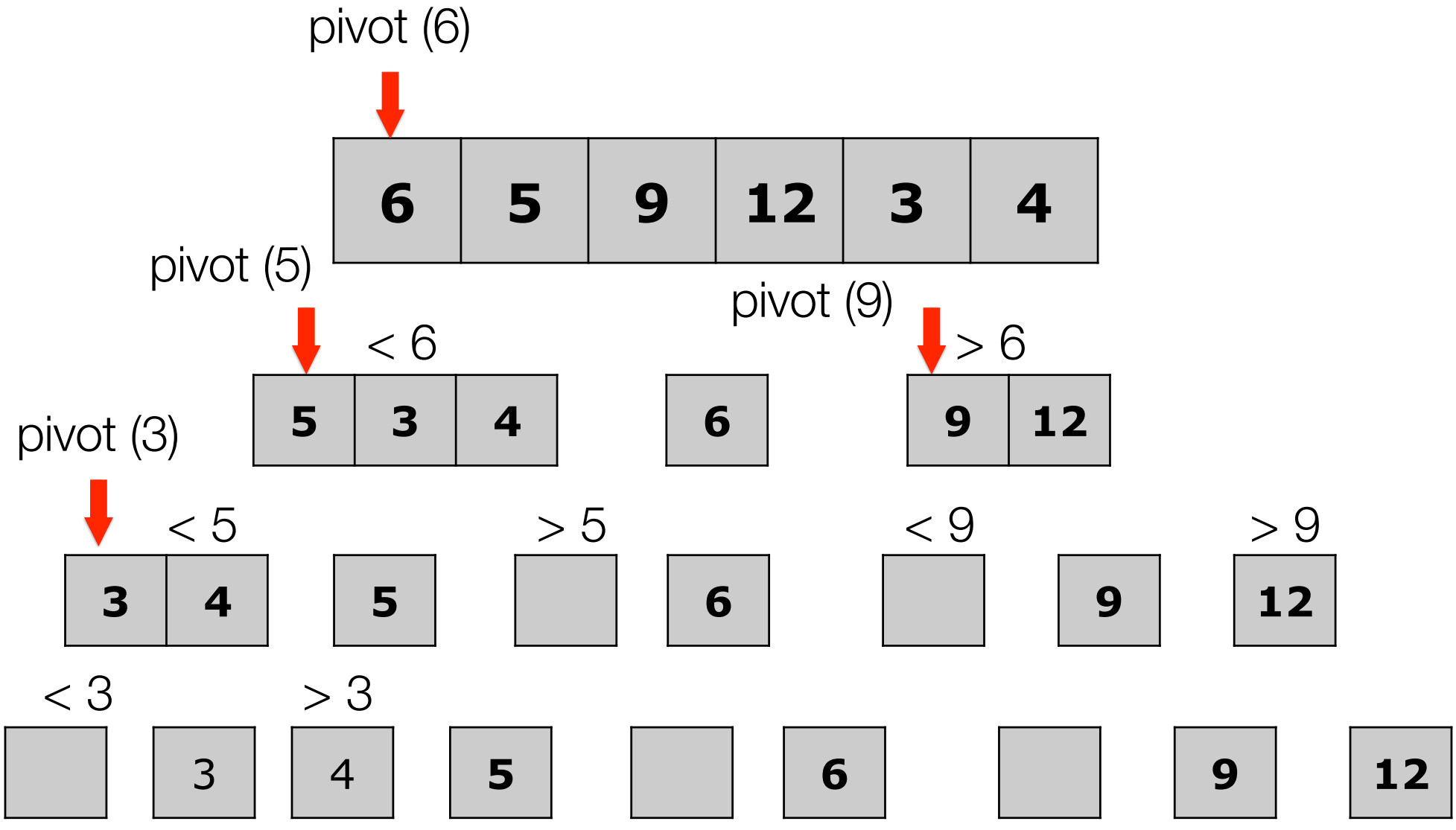
Even if all elements go into one list, that was just a poor partition.

Quicksort

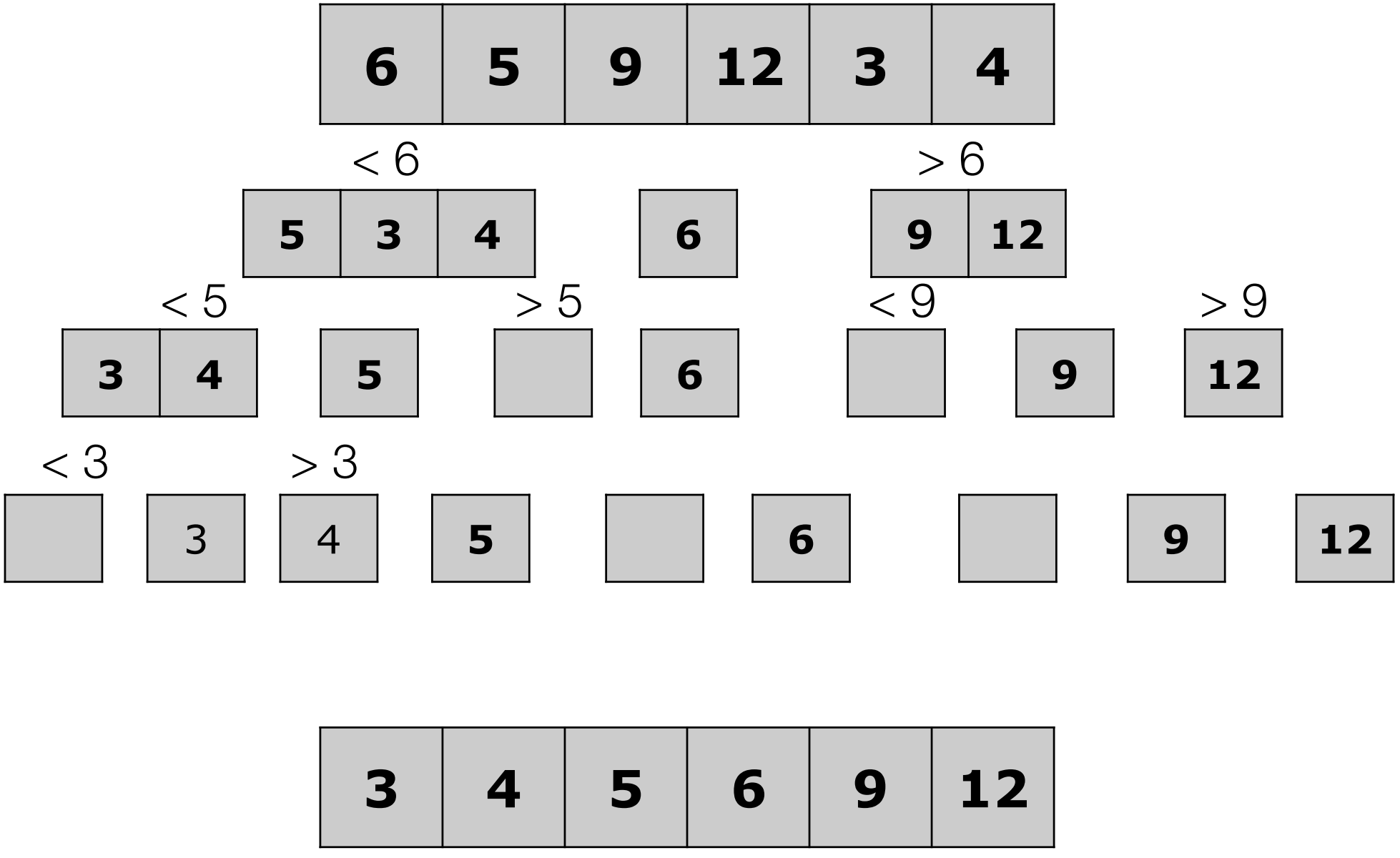


Keep partitioning the sub-lists

Quicksort


pivot (6)
6 5 9 12 3 4
pivot (5)
< 6
5 3 4
6
pivot (9)
> 6
9 12
pivot (3)
< 5
> 5
< 9
> 9
3 4
5
6
9
12
< 3
> 3
3
4
5
6
9
12

Quicksort



Quicksort

```
int* QuickSort(int* arr, int n) {
    if (n < 2) return arr;
    int pivot = arr[0]
    int* left, right;

    for(int i=1; i<= n) {
        if (arr[i] <= pivot){
            add_end(left, v[i]);
        } else {
            add_end(right, v[i]);
        }
    }

    left = QuickSort(left, len(left));
    right = QuickSort(right, len(right));
    add_end(left, pivot)
    return add_end(left, right);
```

Quicksort Algorithm: Choosing the Pivot

- One interesting issue with quicksort is the decision about choosing the pivot.
- If the left-most element is always chosen as the pivot, already-sorted arrays will have $O(n^2)$ behavior (why?)
- Therefore, choosing a pivot that is random works well, or choosing the middle item as the pivot.

0	1	2	3	4	5
4	5	3	6	9	12

Quicksort Algorithm: Repeated Elements

- Repeated elements also cause quicksort to slow down.
- If the whole list was the same value, each recursion would cause all elements to go into one partition, which degrades to $O(n^2)$

Quicksort Algorithm: Big-O

- Best-case time complexity: O(n log n)
- Worst-case time complexity: $O(n^2)$
- Average time complexity: O(n log n)
- Space complexity: O(n) extra

Summary

Sorting Big-O Cheat Sheet			
Sort	**Worst Case**	**Best Case**	**Average Case**
Insertion	$O(n^2)$	$O(n)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$