SORTING ALGORITHMS

OVERVIEW

OVERVIEW

- **Sorting is one of the oldest and most studied areas of computer science**
  - Problem is to take in unsorted data in an array or a file
  - Rearrange data so it is in ascending/descending order based on value of selected fields
  - Store sorted data in an output array or file

- **Key issues to consider**
  - How hard is the algorithm to implement?
  - How much CPU time will the algorithm take?
  - How much data storage will be needed?
  - Will this algorithm work for all types of data or orderings?

OVERVIEW

- **Objectives of this lesson:**

- **Learn about algorithm analysis**
  - How to estimate the speed of an algorithm
  - Examination of code, solving recurrence relationships
  - Learn about best case, worst case, and average case

- **Learn seven classic sorting algorithms**
  - How these classic algorithms work
  - How to implement them
  - Perform speed analysis

OVERVIEW

- **A long list of sorting algorithms have been invented**
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Merge sort
  - Quick sort
  - Bucket sort
  - Radix sort

  - And many more…

OVERVIEW

- **A long list of sorting algorithms have been invented**
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Merge sort
  - Quick sort
  - Bucket sort
  - Radix sort

  - And many more…

Slow but simple to implement

OVERVIEW

- **A long list of sorting algorithms have been invented**
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Merge sort
  - Quick sort
  - Bucket sort
  - Radix sort

  - And many more…

Fast but complex to implement

OVERVIEW

- **A long list of sorting algorithms have been invented**
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Merge sort
  - Quick sort
  - Bucket sort
  - Radix sort

Very fast but only work on some types of data

  - And many more…

OVERVIEW

- **A long list of sorting algorithms have been invented**
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Merge sort
  - Quick sort
  - Bucket sort
  - Radix sort
  - And many more…

See wikipedia for the history of sorting and many examples

SORTING ALGORITHMS

ALGORITHM ANALYSIS

ALGORITHM ANALYSIS

- **To compare two algorithms it is helpful to know how many instructions are executed to process N data values**

- **For example, to calculate sum of N integers we could use:**

```
int sum=0;
for (int i=0; i<N; i++)
    sum += data[i];
```

- **Here the loop is executed N times**
- **This is an O(N) algorithm**

ALGORITHM ANALYSIS

- **Similarly if we wanted to print the product of all possible pairs of numbers between 0 and N-1 we could use:**

```
for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
        cout << data[i] * data[j] << endl;
```

- **The outer loop will execute N times**
- **The inner loop will execute N * N = N^2 times**
- **This is an $O(N^2)$ algorithm**

ALGORITHM ANALYSIS

- **Often the loops are more complex.**

```
int count=0;
for (int i=0; i<N; i++)
    for (int j=i; j<N; j++);
        count++;
```

- **The outer loop executes N times**
- **How many times is the inner loop executed?**

ALGORITHM ANALYSIS

- **Often the loops are more complex.**

```
int count=0;
for (int i=0; i<N; i++)
  for (int j=i; j<N; j++);
    count++;
```

- **The inner loop executes N + N-1 + ... + 2 + 1 times**
- **This equals (N+1) * N/2 = $N^2/2 + N/2$**
- **This is less than N^2 but only differs by a constant**
- **This is an $O(N^2)$ algorithm**

ALGORITHM ANALYSIS

- **Sometimes a loop can execute less than N times**
- **We saw this with binary search and the power function**
- **Here is a similar example:**

```
int num = N;
while (num > 0)
    num = num / 2;
```

- **If $N = 2^P$, the loop will execute $P = \log_2 N$ times**
- **This is an $O(\log_2 N)$ algorithm**

ALGORITHM ANALYSIS

- **Sometimes a $\log_2 N$ calculation is inside another loop:**

```
for (int i=0; i<N; i++)
{
    int num = N;
    while (num > 0)
        num = num / 2;
}
```

- **The inner loop will execute $N * \log_2 N$ times**
- **This is an $O(N \log_2 N)$ algorithm**

ALGORITHM ANALYSIS

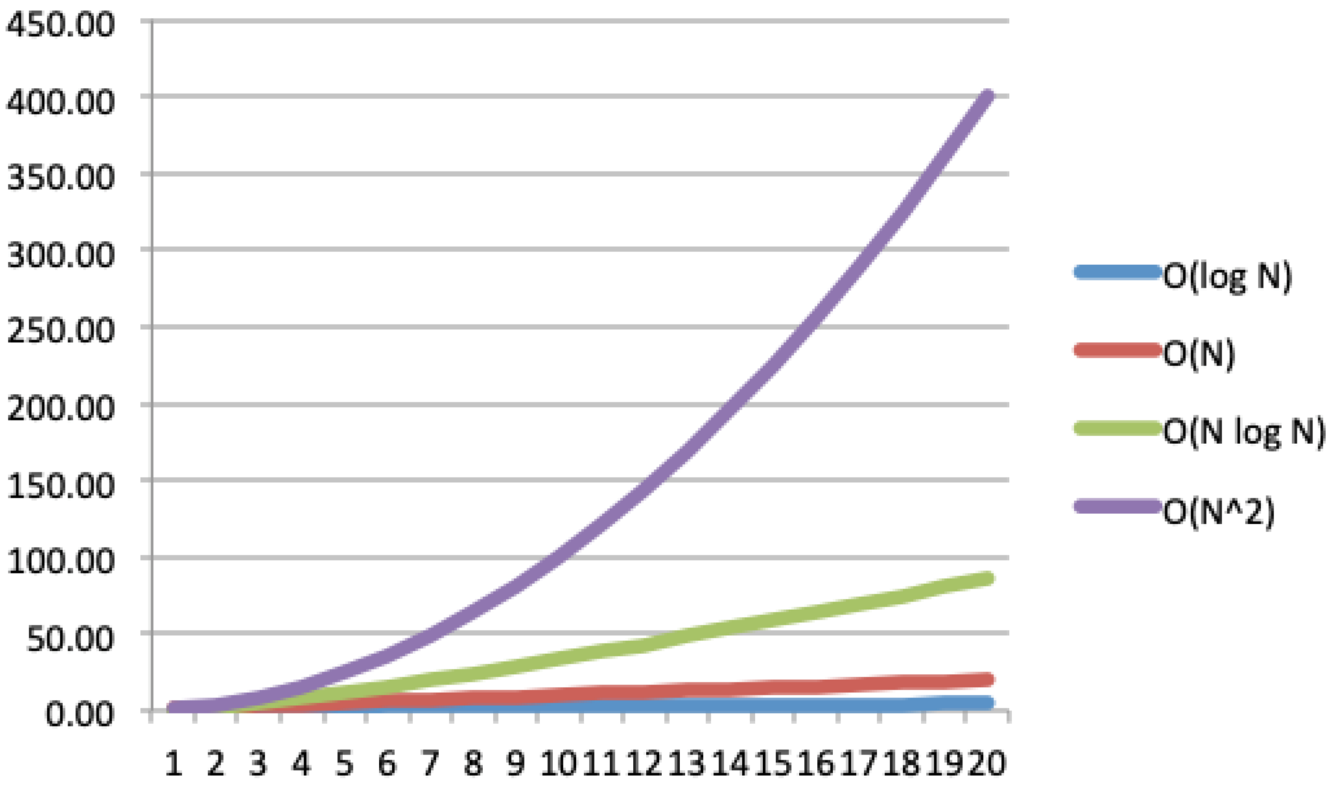
- **Linear search is O(N) but binary search is O(log N)**
  - For N = 1000 binary search takes only 10 steps
  - This is 100 times faster than linear search
  - For N = 1,000,000 binary search takes only 20 steps
  - This is 50,000 times faster than linear search

- **Most sorting algorithms are O(N log N) or O(N^2)**
  - The speed difference for sorting is equally dramatic
  - For N = 1000, the O(N log N) sort is 100 times faster
  - For N = 1,000,000 the fast sort is 50,000 times faster

ALGORITHM ANALYSIS

O(log N)	O(N)	O(N log N)	O(N^2)
0.00	1	0.00	1
1.00	2	2.00	4
1.58	3	4.75	9
2.00	4	8.00	16
2.32	5	11.61	25
2.58	6	15.51	36
2.81	7	19.65	49
3.00	8	24.00	64
3.17	9	28.53	81
3.32	10	33.22	100
3.46	11	38.05	121
3.58	12	43.02	144
3.70	13	48.11	169
3.81	14	53.30	196
3.91	15	58.60	225
4.00	16	64.00	256
4.09	17	69.49	289
4.17	18	75.06	324
4.25	19	80.71	361
4.32	20	86.44	400

ALGORITHM ANALYSIS



SORTING ALGORITHMS

SELECTION SORT

SELECTION SORT

- **Selection sort is a very simple sorting algorithm**
- **The idea is to iteratively select the smallest value from an unsorted array, and put this at the end of a sorted array**
- **Loop N times**
  - Select the smallest value in unsorted array
  - Mark this value as "taken" in the unsorted array
  - Store smallest value at end of sorted array
- **When this loop finishes, the unsorted array will be empty, and the sorted array will have N values in ascending order**

SELECTION SORT

Move the smallest value

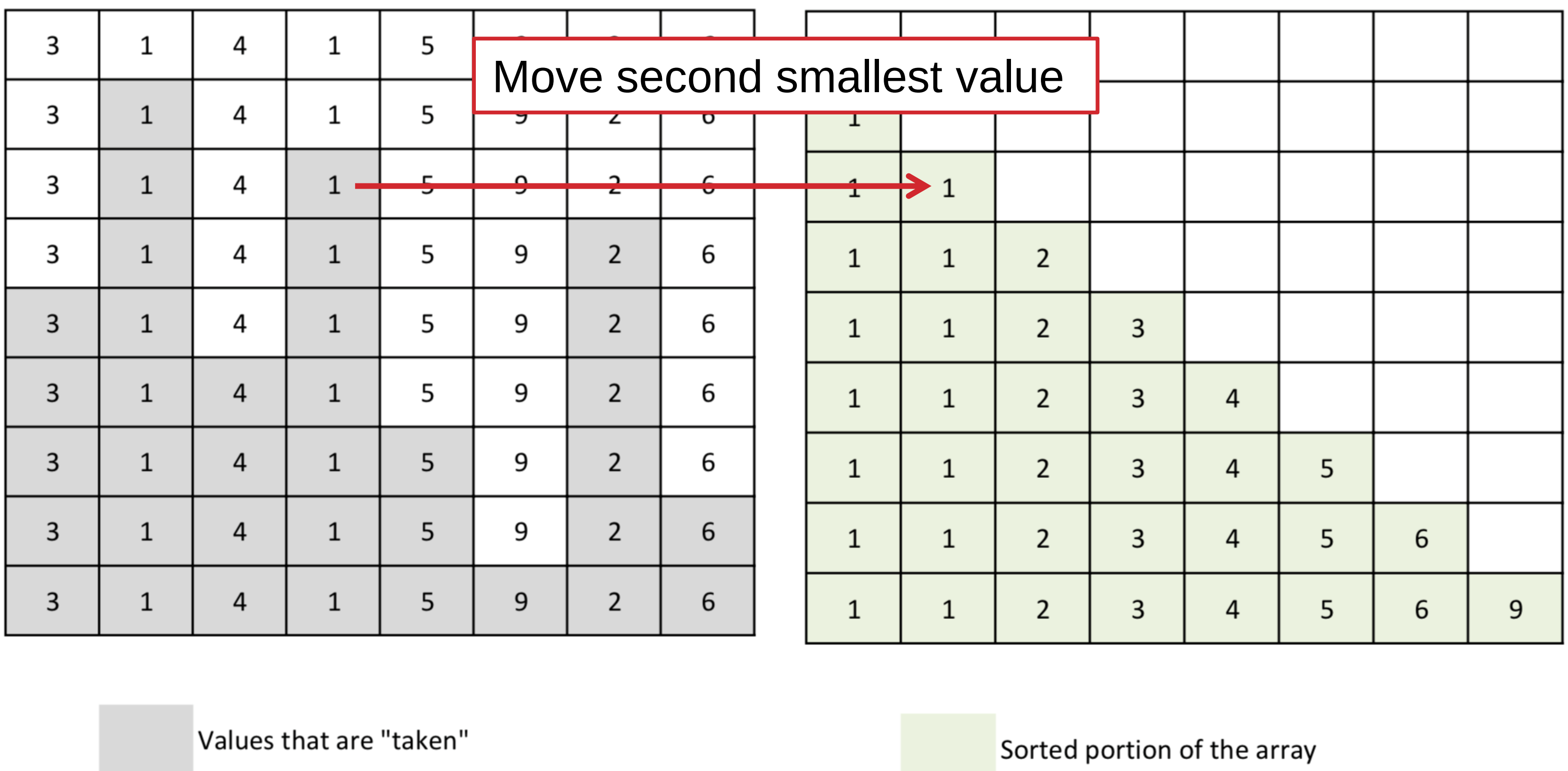
3	1	4	1	5	9	2	6
3	1	4	1	5	9	2	6
3	1	4	1	5	9	2	6
3	1	4	1	5	9	2	6
3	1	4	1	5	9	2	6
3	1	4	1	5	9	2	6
3	1	4	1	5	9	2	6
3	1	4	1	5	9	2	6
3	1	4	1	5	9	2	6

1							
1	1						
1	1	2					
1	1	2	3				
1	1	2	3	4			
1	1	2	3	4	5		
1	1	2	3	4	5	6	
1	1	2	3	4	5	6	9

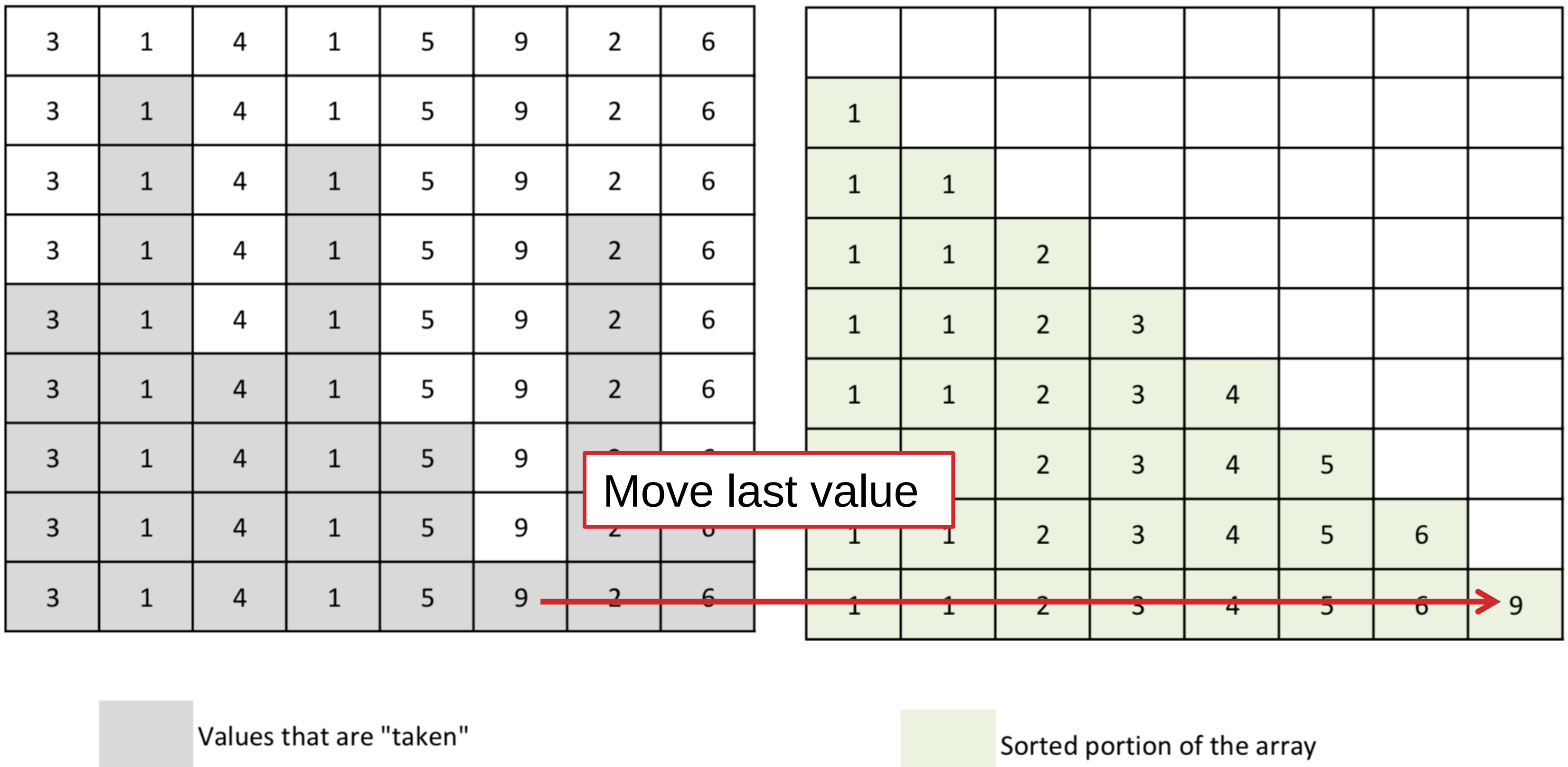
Values that are "taken"

Sorted portion of the array

SELECTION SORT



SELECTION SORT



SELECTION SORT

3	1	4	1	5	9	2	6
3	1	4	1	5	9	2	6
3	1	4	1	5	9	2	6
3	1	4	1	5	9	2	6
3	1	4	1	5	9	2	6
3	1	4	1	5	9	2	6
3	1	4	1	5	9	2	6
3	1	4	1	5	9	2	6
3	1	4	1	5	9	2	6

1							
1	1						
1	1	2					
1	1	2	3				
1	1	2	3	4			
1	1	2	3	4	5		
1	1	2	3	4	5	6	
1	1	2	3	4	5	6	9

Values that are "taken"

Sorted portion of the array

SELECTION SORT

- **One of the most expensive steps in selection sort is finding the next smallest value in the unsorted array**
  - First we must find the location of the first “untaken” value
  - Then we have to loop over the rest of the array to see if any other “untaken” value is smaller
  - Since the data array is N long this search takes N steps
  - This search loop is inside a loop that executes N times
  - Hence selection sort is an $O(N^2)$ algorithm
- **In practice, using two arrays is inefficient, so most implementations use one array and move data around**

SELECTION SORT

3	1	4	1	5	9	2	6
1	3	4	1	5	9	2	6
1	1	4	3	5	9	2	6
1	1	2	3	5	9	4	6
1	1	2	3	5	9	4	6
1	1	2	3	4	9	5	6
1	1	2	3	4	5	9	6
1	1	2	3	4	5	6	9
1	1	2	3	4	5	6	9

Sorted portion of the array

Next smallest value in array

In each pass of this algorithm we swap the smallest value in the unsorted part of array with leftmost unsorted value and increase the size of sorted part

SELECTION SORT

3	1	4	1	5	9	2	6
1	3	4	1	5	9	2	6
1	1	4	3	5	9	2	6
1	1	2	3	5	9	4	6
1	1	2	3	5	9	4	6
1	1	2	3	4	9	5	6
1	1	2	3	4	5	9	6
1	1	2	3	4	5	6	9
1	1	2	3	4	5	6	9

Sorted portion of the array

Next smallest value in array

In the second pass we find and swap the second smallest value

SELECTION SORT

3	1	4	1	5	9	2	6
1	3	4	1	5	9	2	6
1	1	4	3	5	9	2	6
1	1	2	3	5	9	4	6
1	1	2	3	5	9	4	6
1	1	2	3	4	9	5	6
1	1	2	3	4	5	9	6
1	1	2	3	4	5	6	9
1	1	2	3	4	5	6	9

Sorted portion of the array

Next smallest value in array

Then we swap the third smallest value in into its correct location

SELECTION SORT

3	1	4	1	5	9	2	6
1	3	4	1	5	9	2	6
1	1	4	3	5	9	2	6
1	1	2	3	5	9	4	6
1	1	2	3	5	9	4	6
1	1	2	3	4	9	5	6
1	1	2	3	4	5	9	6
1	1	2	3	4	5	6	9
1	1	2	3	4	5	6	9

Sorted portion of the array

Next smallest value in array

In some cases the smallest value does not need to be swapped because it is already in the correct location

SELECTION SORT

3	1	4	1	5	9	2	6
1	3	4	1	5	9	2	6
1	1	4	3	5	9	2	6
1	1	2	3	5	9	4	6
1	1	2	3	5	9	4	6
1	1	2	3	4	9	5	6
1	1	2	3	4	5	9	6
1	1	2	3	4	5	6	9
1	1	2	3	4	5	6	9

Sorted portion of the array

Next smallest value in array

The selection sort loop ends after N passes over the data

SELECTION SORT

- **The selection sort algorithm can be adapted to search for the next largest value in the unsorted part of the array**
  - In this case, the unsorted portion is on the left side and the sorted portion is on the right side of the array

- **Questions:**
  - Do you think the selection sort algorithm is faster if the input data is already in sorted order?
  - Do you think the selection sort algorithm is slower if the input data is in reverse sorted order?
  - Do you think selection sort is always $O(N^2)$?

SELECTION SORT

- **The selection sort algorithm can be adapted to search for the next largest value in the unsorted part of the array**
  - In this case, the unsorted portion is on the left side and the sorted portion is on the right side of the array

- **Questions:**
  - Do you think the selection sort algorithm is faster if the input data is already in sorted order? NO
  - Do you think the selection sort algorithm is slower if the input data is in reverse sorted order? NO
  - Do you think selection sort is always $O(N^2)$? YES

SELECTION SORT

```
void selection_sort(int data[], int low, int high)
{
   // Loop over input array N times
   for (int last = high; last > low; last--)
   {
      // Find index of largest value in unsorted array
      int largest = low;
      for (int index = low + 1; index <= last; index++)
         if (data[index] > data[largest])
            largest = index;

      // Swap with last element in unsorted array
      int temp = data[last];
      data[last] = data[largest];
      data[largest] = temp;
   }
}
```

Notice that this sorting function has two nested for loops

SELECTION SORT

Experimental results:

```
Enter number of data values:100
CPU time = 5.5e-05 sec

Enter number of data values:1000
CPU time = 0.004088 sec

Enter number of data values:10000
CPU time = 0.24972 sec

Enter number of data values:100000
CPU time = 14.2292 sec
```

SORTING ALGORITHMS

BUBBLE SORT

BUBBLE SORT

- **Bubble sort is a widely known sorting algorithm because it is simple to explain and implement**
  - Unfortunately, this simplicity comes at a cost – speed

- **The idea is to iteratively scan the data array from left to right, and swap any adjacent values that are out of order**
  - Each pass over the array "bubbles" the largest data value to the right, and smaller data values shift one to the left
  - After N iterations over the input array, the data values will be in sorted order

BUBBLE SORT

Pass 1

3	1	4	1	5	9	2	6	original data
1	3	4	1	5	9	2	6	swap values
1	3	4	1	5	9	2	6	don't swap
1	3	1	4	5	9	2	6	swap values
1	3	1	4	5	9	2	6	don't swap
1	3	1	4	5	9	2	6	don't swap
1	3	1	4	5	2	9	6	swap values
1	3	1	4	5	2	6	9	swap values

BUBBLE SORT

Pass 2

1	3	1	4	5	2	6	9	data after pass 1
1	3	1	4	5	2	6	9	don’t swap
1	1	3	4	5	2	6	9	swap values
1	1	3	4	5	2	6	9	don’t swap
1	1	3	4	5	2	6	9	don’t swap
1	1	3	4	2	5	6	9	swap values
1	1	3	4	2	5	6	9	don’t swap
1	1	3	4	2	5	6	9	don’t swap

BUBBLE SORT

Pass 3

1	1	3	4	2	5	6	9	data after pass 2
1	1	3	4	2	5	6	9	don’t swap
1	1	3	4	2	5	6	9	don’t swap
1	1	3	4	2	5	6	9	don’t swap
1	1	3	2	4	5	6	9	swap values
1	1	3	2	4	5	6	9	don’t swap
1	1	3	2	4	5	6	9	don’t swap
1	1	3	2	4	5	6	9	don’t swap

BUBBLE SORT

Pass 4

1	1	3	2	4	5	6	9	data after pass 3
1	1	3	2	4	5	6	9	don’t swap
1	1	3	2	4	5	6	9	don’t swap
1	1	2	3	4	5	6	9	swap values
1	1	2	3	4	5	6	9	don’t swap
1	1	2	3	4	5	6	9	don’t swap
1	1	2	3	4	5	6	9	don’t swap
1	1	2	3	4	5	6	9	don’t swap

BUBBLE SORT

```
void bubble_sort(int data[], int low, int high)
{
   // Loop over array N times
   for (int count = low; count < high; count++)
   {
      // Loop over N elements in array
      for (int index = low; index < high; index++)
      {
           // Swap two data values if out of order
           if (data[index] > data[index + 1])
           {
              int temp = data[index];
              data[index] = data[index + 1];
              data[index + 1] = temp;
           }
      }
   }
```

Notice that this sorting function has two nested for loops

BUBBLE SORT

- **How long does bubble sort take to sort an array?**
  - There are N passes over the array
  - For each pass N-1 pairs of adjacent values are compared
  - In total there are N * (N-1) = N^2 – N comparisons
  - Hence this is an $O(N^2)$ sorting algorithm

- **What happens if the input array is already sorted?**
  - The current algorithm will do N passes over the data
  - No data values will be swapped in each pass
  - This is a massive waste of CPU time

BUBBLE SORT

- **To improve bubble sort we can stop when no swaps occur**
  - Initialize swap counter to zero before each pass
  - When swap counter is still zero after the comparison pass the array is in sorted order and we can stop looping
- **In the best case, when the data is already sorted, this improved bubble sort is only O(N) steps**
  - This algorithm will still be $O(N^2)$ on average, but with a smaller run time constant than the basic bubble sort

BUBBLE SORT

```
void bubble_sort(int data[], int low, int high)
{
   // Bubble largest value to the right N times
   int pass = 1;
   int exchange = 1;
   int count = high - low + 1;
   while ((pass < count) && (exchange > 0))
   {
      // Scan unsorted part of data array
      exchange = 0;
      for (int index = low; index <= high - pass; index++)
      ...
```

By checking the number of exchanges, we can stop the outer loop when data is sorted

BUBBLE SORT

```
      {
            // Swap two data values if out of order
            if (data[index] > data[index + 1])
            {
                int temp = data[index];
                data[index] = data[index + 1];
                data[index + 1] = temp;
                exchange++;
            }
      }
      pass++;
   }
}
```

BUBBLE SORT

Experimental results with basic bubble sort:

```
Enter number of data values:100
CPU time = 0.000144 sec

Enter number of data values:1000
CPU time = 0.009901 sec

Enter number of data values:10000
CPU time = 0.503489 sec

Enter number of data values:100000
CPU time = 43.8939 sec
```

BUBBLE SORT

Experimental results with improved bubble sort:

```
Enter number of data values:100
CPU time = 3.5e-05 sec

Enter number of data values:1000
CPU time = 0.004836 sec

Enter number of data values:10000
CPU time = 0.402202 sec

Enter number of data values:100000
CPU time = 30.8026 sec
```

Although this is much faster than the basic bubble sort algorithm it is still very slow for large values of N

SORTING ALGORITHMS

INSERTION SORT

INSERTION SORT

- **Insertion sort is another simple (but slow) algorithm**
- **The idea is to gradually create a sorted array by inserting unsorted data values one-by-one into a sorted array**
- **Loop N times**
  - Look at the next data value in the unsorted array
  - Move sorted data to make room for this value
  - Insert data value into correct location in sorted array
- **When this loop finishes, the unsorted array will be empty, and the sorted array will have N values in ascending order**

INSERTION SORT

- **The tricky part of the insertion sort algorithm is making room for the new data**

- **Set array index to current length of sorted array**
- **While data[index-1] greater than insert_value**
  - Shift data using “data[index] = data[index-1]”
  - Decrement index using “index=index-1”
- **Store insert_value in data[index]**

- **This process is the most expensive step in insertion sort**

INSERTION SORT

- **Example inserting value 2 into sorted array**
  - We have to shift 4 values to make room for the 2

0	1	2	3	4	5	6	7	
1	1	3	4	5	9			index = 6
1	1	3	4	5		9		shift 9, index = 5

INSERTION SORT

- **Example inserting value 2 into sorted array**
  - We have to shift 4 values to make room for the 2

0	1	2	3	4	5	6	7	
1	1	3	4	5	9			index = 6
1	1	3	4	5		9		shift 9, index = 5
1	1	3	4		5	9		shift 5, index = 4

INSERTION SORT

- **Example inserting value 2 into sorted array**
  - We have to shift 4 values to make room for the 2

0	1	2	3	4	5	6	7	
1	1	3	4	5	9			index = 6
1	1	3	4	5		9		shift 9, index = 5
1	1	3	4		5	9		shift 5, index = 4
1	1	3		4	5	9		shift 4, index = 3

INSERTION SORT

- **Example inserting value 2 into sorted array**
  - We have to shift 4 values to make room for the 2

0	1	2	3	4	5	6	7	
1	1	3	4	5	9			index = 6
1	1	3	4	5		9		shift 9, index = 5
1	1	3	4		5	9		shift 5, index = 4
1	1	3		4	5	9		shift 4, index = 3
1	1		3	4	5	9		shift 3, index = 2

INSERTION SORT

- **Example inserting value 2 into sorted array**
  - We have to shift 4 values to make room for the 2

0	1	2	3	4	5	6	7	
1	1	3	4	5	9			index = 6
1	1	3	4	5		9		shift 9, index = 5
1	1	3	4		5	9		shift 5, index = 4
1	1	3		4	5	9		shift 4, index = 3
1	1		3	4	5	9		shift 3, index = 2
1	1	**2**	3	4	5	9		insert 2 at index 2

INSERTION SORT

- **Example inserting N unsorted data into sorted array**
  - We repeat the insert process N times

Sorted data								Actions taken	Unsorted data							
-	-	-	-	-	-	-	-		3	1	4	1	5	9	2	6

INSERTION SORT

- **Example inserting N unsorted data into sorted array**
  - We repeat the insert process N times

Sorted data								Actions taken	Unsorted data							
-	-	-	-	-	-	-	-		3	1	4	1	5	9	2	6
3	-	-	-	-	-	-	-	insert 3	-	1	4	1	5	9	2	6

INSERTION SORT

- **Example inserting N unsorted data into sorted array**
  - We repeat the insert process N times

Sorted data								Actions taken	Unsorted data							
-	-	-	-	-	-	-	-		3	1	4	1	5	9	2	6
3	-	-	-	-	-	-	-	insert 3	-	1	4	1	5	9	2	6
1	3	-	-	-	-	-	-	shift 3, insert 1	-	-	4	1	5	9	2	6

INSERTION SORT

- **Example inserting N unsorted data into sorted array**
  - We repeat the insert process N times

Sorted data								Actions taken	Unsorted data							
-	-	-	-	-	-	-	-		3	1	4	1	5	9	2	6
3	-	-	-	-	-	-	-	insert 3	-	1	4	1	5	9	2	6
1	3	-	-	-	-	-	-	shift 3, insert 1	-	-	4	1	5	9	2	6
1	3	**4**	-	-	-	-	-	insert 4	-	-	-	1	5	9	2	6

INSERTION SORT

- **Example inserting N unsorted data into sorted array**
  - We repeat the insert process N times

Sorted data								Actions taken	Unsorted data							
-	-	-	-	-	-	-	-		3	1	4	1	5	9	2	6
3	-	-	-	-	-	-	-	insert 3	-	1	4	1	5	9	2	6
1	3	-	-	-	-	-	-	shift 3, insert 1	-	-	4	1	5	9	2	6
1	3	**4**	-	-	-	-	-	insert 4	-	-	-	1	5	9	2	6
1	**1**	3	4	-	-	-	-	shift 3 4, insert 1	-	-	-	-	5	9	2	6

INSERTION SORT

- **Example inserting N unsorted data into sorted array**
  - We repeat the insert process N times

Sorted data								Actions taken	Unsorted data							
-	-	-	-	-	-	-	-		3	1	4	1	5	9	2	6
3	-	-	-	-	-	-	-	insert 3	-	1	4	1	5	9	2	6
1	3	-	-	-	-	-	-	shift 3, insert 1	-	-	4	1	5	9	2	6
1	3	**4**	-	-	-	-	-	insert 4	-	-	-	1	5	9	2	6
1	**1**	3	4	-	-	-	-	shift 3 4, insert 1	-	-	-	-	5	9	2	6
1	1	3	4	5	-	-	-	insert 5	-	-	-	-	-	9	2	6

INSERTION SORT

- **Example inserting N unsorted data into sorted array**
  - We repeat the insert process N times

Sorted data								Actions taken	Unsorted data							
-	-	-	-	-	-	-	-		3	1	4	1	5	9	2	6
3	-	-	-	-	-	-	-	insert 3	-	1	4	1	5	9	2	6
1	3	-	-	-	-	-	-	shift 3, insert 1	-	-	4	1	5	9	2	6
1	3	**4**	-	-	-	-	-	insert 4	-	-	-	1	5	9	2	6
1	**1**	3	4	-	-	-	-	shift 3 4, insert 1	-	-	-	-	5	9	2	6
1	1	3	4	5	-	-	-	insert 5	-	-	-	-	-	9	2	6
1	1	3	4	5	**9**	-	-	insert 9	-	-	-	-	-	-	2	6

INSERTION SORT

- **Example inserting N unsorted data into sorted array**
  - We repeat the insert process N times

Sorted data								Actions taken	Unsorted data							
-	-	-	-	-	-	-	-		3	1	4	1	5	9	2	6
3	-	-	-	-	-	-	-	insert 3	-	1	4	1	5	9	2	6
1	3	-	-	-	-	-	-	shift 3, insert 1	-	-	4	1	5	9	2	6
1	3	**4**	-	-	-	-	-	insert 4	-	-	-	1	5	9	2	6
1	**1**	3	4	-	-	-	-	shift 3 4, insert 1	-	-	-	-	5	9	2	6
1	1	3	4	5	-	-	-	insert 5	-	-	-	-	-	9	2	6
1	1	3	4	5	**9**	-	-	insert 9	-	-	-	-	-	-	2	6
1	1	**2**	3	4	5	9	-	shift 3 4 5 9, insert 2	-	-	-	-	-	-	-	6

INSERTION SORT

- **Example inserting N unsorted data into sorted array**
  - We repeat the insert process N times

Sorted data								Actions taken	Unsorted data							
-	-	-	-	-	-	-	-		3	1	4	1	5	9	2	6
3	-	-	-	-	-	-	-	insert 3	-	1	4	1	5	9	2	6
1	3	-	-	-	-	-	-	shift 3, insert 1	-	-	4	1	5	9	2	6
1	3	**4**	-	-	-	-	-	insert 4	-	-	-	1	5	9	2	6
1	**1**	3	4	-	-	-	-	shift 3 4, insert 1	-	-	-	-	5	9	2	6
1	1	3	4	**5**	-	-	-	insert 5	-	-	-	-	-	9	2	6
1	1	3	4	5	**9**	-	-	insert 9	-	-	-	-	-	-	2	6
1	1	**2**	3	4	5	9	-	shift 3 4 5 9, insert 2	-	-	-	-	-	-	-	6
1	1	2	3	4	5	**6**	9	shift 9, insert 6	-	-	-	-	-	-	-	-

INSERTION SORT

- **Insertion sort is normally implemented with one array and we keep track of which half is sorted (white and green) and which half is unsorted (pink).**

3	1	4	1	5	9	2	6	Unsorted data

INSERTION SORT

- **Insertion sort is normally implemented with one array and we keep track of which half is sorted (white and green) and which half is unsorted (pink).**

3	1	4	1	5	9	2	6	
3	1	4	1	5	9	2	6	insert 3

INSERTION SORT

- **Insertion sort is normally implemented with one array and we keep track of which half is sorted (white and green) and which half is unsorted (pink).**

3	1	4	1	5	9	2	6	
3	1	4	1	5	9	2	6	insert 3
1	3	4	1	5	9	2	6	shift 3, insert 1

INSERTION SORT

- **Insertion sort is normally implemented with one array and we keep track of which half is sorted (white and green) and which half is unsorted (pink).**

3	1	4	1	5	9	2	6	
3	1	4	1	5	9	2	6	insert 3
1	3	4	1	5	9	2	6	shift 3, insert 1
1	3	**4**	1	5	9	2	6	insert 4

INSERTION SORT

- **Insertion sort is normally implemented with one array and we keep track of which half is sorted (white and green) and which half is unsorted (pink).**

3	1	4	1	5	9	2	6	
3	1	4	1	5	9	2	6	insert 3
1	3	4	1	5	9	2	6	shift 3, insert 1
1	3	**4**	1	5	9	2	6	insert 4
1	**1**	3	4	5	9	2	6	shift 3 4, insert 1

INSERTION SORT

- **Insertion sort is normally implemented with one array and we keep track of which half is sorted (white and green) and which half is unsorted (pink).**

3	1	4	1	5	9	2	6	
3	1	4	1	5	9	2	6	insert 3
1	3	4	1	5	9	2	6	shift 3, insert 1
1	3	**4**	1	5	9	2	6	insert 4
1	**1**	3	4	5	9	2	6	shift 3 4, insert 1
1	1	3	4	5	9	2	6	insert 5

INSERTION SORT

- **Insertion sort is normally implemented with one array and we keep track of which half is sorted (white and green) and which half is unsorted (pink).**

3	1	4	1	5	9	2	6	
3	1	4	1	5	9	2	6	insert 3
1	3	4	1	5	9	2	6	shift 3, insert 1
1	3	**4**	1	5	9	2	6	insert 4
1	**1**	3	4	5	9	2	6	shift 3 4, insert 1
1	1	3	4	5	9	2	6	insert 5
1	1	3	4	5	**9**	2	6	insert 9

INSERTION SORT

- **Insertion sort is normally implemented with one array and we keep track of which half is sorted (white and green) and which half is unsorted (pink).**

3	1	4	1	5	9	2	6	
3	1	4	1	5	9	2	6	insert 3
1	3	4	1	5	9	2	6	shift 3, insert 1
1	3	**4**	1	5	9	2	6	insert 4
1	**1**	3	4	5	9	2	6	shift 3 4, insert 1
1	1	3	4	5	9	2	6	insert 5
1	1	3	4	5	**9**	2	6	insert 9
1	1	**2**	3	4	5	9	6	shift 3 4 5 9, insert 2

INSERTION SORT

- **Insertion sort is normally implemented with one array and we keep track of which half is sorted (white and green) and which half is unsorted (pink).**

3	1	4	1	5	9	2	6	
3	1	4	1	5	9	2	6	insert 3
1	3	4	1	5	9	2	6	shift 3, insert 1
1	3	**4**	1	5	9	2	6	insert 4
1	**1**	3	4	5	9	2	6	shift 3 4, insert 1
1	1	3	4	5	9	2	6	insert 5
1	1	3	4	5	**9**	2	6	insert 9
1	1	**2**	3	4	5	9	6	shift 3 4 5 9, insert 2
1	1	2	3	4	5	**6**	9	shift 9, insert 6

INSERTION SORT

- **Notice that in the previous example**
  - Some insertions caused a lot of data shifting
  - Other insertions caused no data shifting

- **On average we can expect half the sorted data to shift**
  - The data insertion loop iterates N times
  - On iteration K we shift K/2 values on average
  - Total data shifts $= (0+1+2+3+\ldots+N-1)/2$

    $= (N * (N-1)/2)/2$

    $= (N^2 - N)/2$
  - Hence insertion sort is an $O(N^2)$ algorithm on average

INSERTION SORT

- **If the "unsorted data" is in reverse sorted order, we must shift all the data for each insert**
  - This is the worst case for insertion sort
  - Total data shifts = $(N^2 – N)$
  - Hence algorithm is $O(N^2)$ in worst case
- **If the "unsorted data" is in sorted order, we do no shifting**
  - This is the best case for insertion sort
  - Total data shifts = 0 for N insertions
  - Hence algorithm is $O(N)$ in best case

INSERTION SORT

```
void insertion_sort(int data[], int low, int high)
{
   // Insert each element of unsorted list into sorted list
   for (int unsorted = low + 1; unsorted <= high; unsorted++)
   {
      // Select unsorted value to be inserted
      int value = data[unsorted];
      int posn = unsorted;

      // Make room for new data value
      while ((posn > 0) && (data[posn - 1] > value))
      { data[posn] = data[posn - 1]; posn--; }

      // Put new value into array
      data[posn] = value;
   }
}
```

Notice that this sorting function has two nested loops

INSERTION SORT

Experimental results:

```
Enter number of data values: 100
CPU time = 3.6e-05 sec

Enter number of data values: 1000
CPU time = 0.002247 sec

Enter number of data values: 10000
CPU time = 0.150965 sec

Enter number of data values: 100000
CPU time = 8.23081 sec
```

INSERTION SORT

Experimental results:

```
Enter number of data values: 100
CPU time = 3.6e-05 sec

Enter number of data values: 1000
CPU time = 0.002247 sec

Enter number of data values: 10000
CPU time = 0.150965 sec

Enter number of data values: 100000
CPU time = 8.23081 sec
```

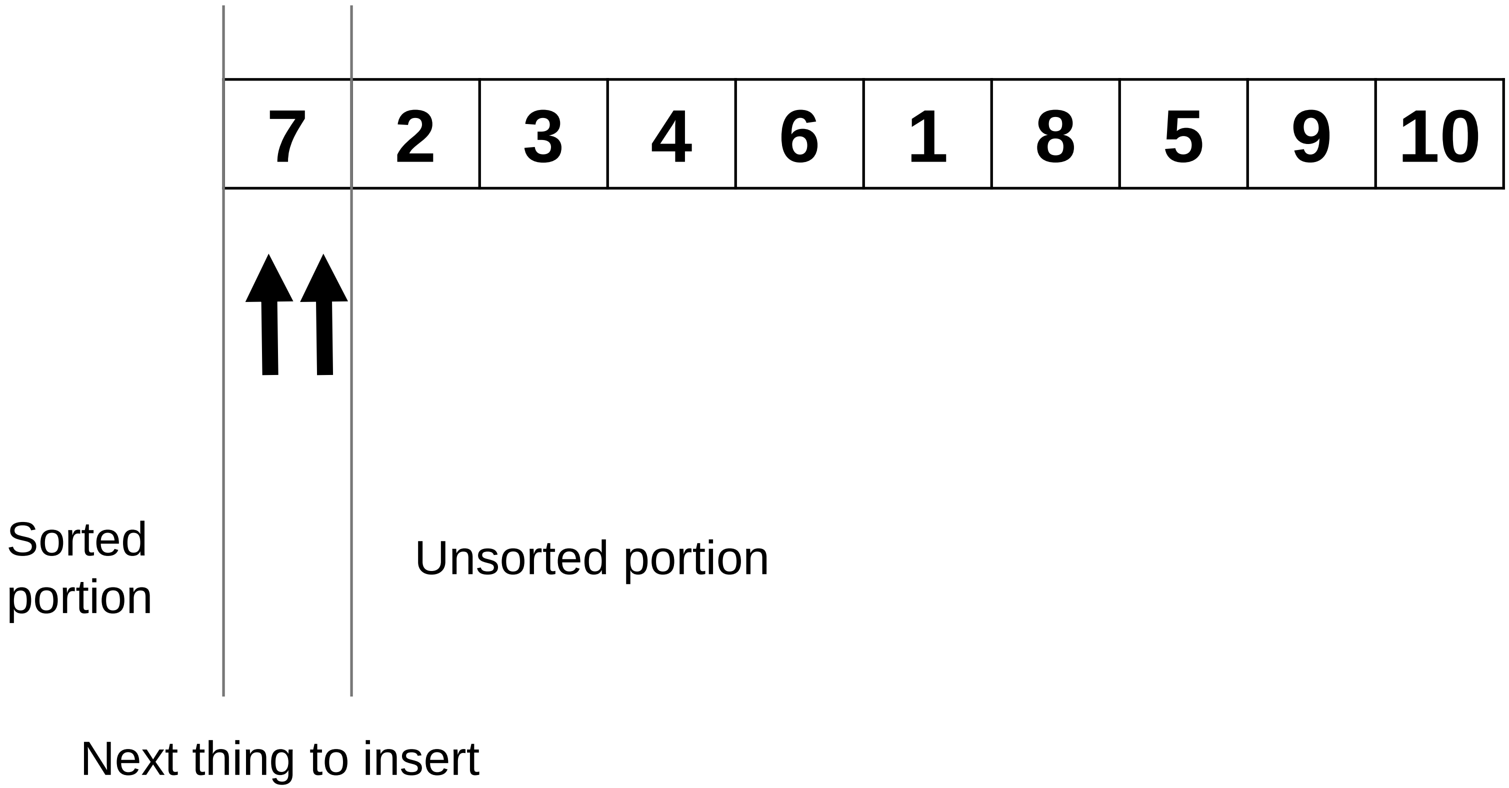
This is faster than selection sort (14 sec) or bubble sort (30 sec)

INSERTION SORT

- **Sometimes the data insertion phase is implemented with an element-by-element swap that is similar to bubble sort.**
  - Instead of “bubbling” the largest value to right, we “bubble” the inserted value to the left until it is in correct location
  - This approach requires slightly more CPU time because we keep moving the inserted value over and over

- **This approach is illustrated in the example below**
  - We use one index for location of data being inserted
  - We use second index to keep track of bubble location

INSERTION SORT



INSERTION SORT

7	2	3	4	6	1	8	5	9	10

Sorted portion

Unsorted portion

Next thing to insert

INSERTION SORT

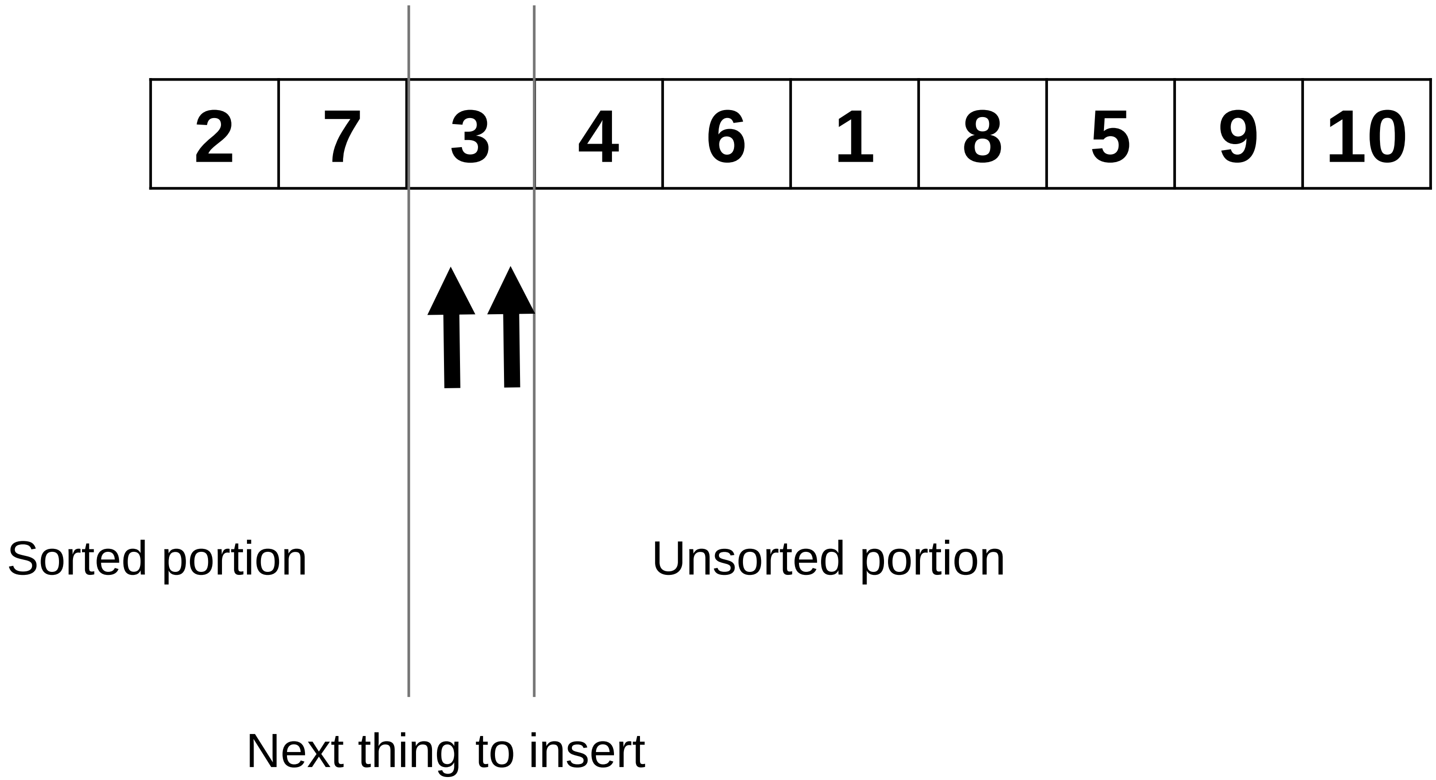
2	7	3	4	6	1	8	5	9	10

Sorted portion

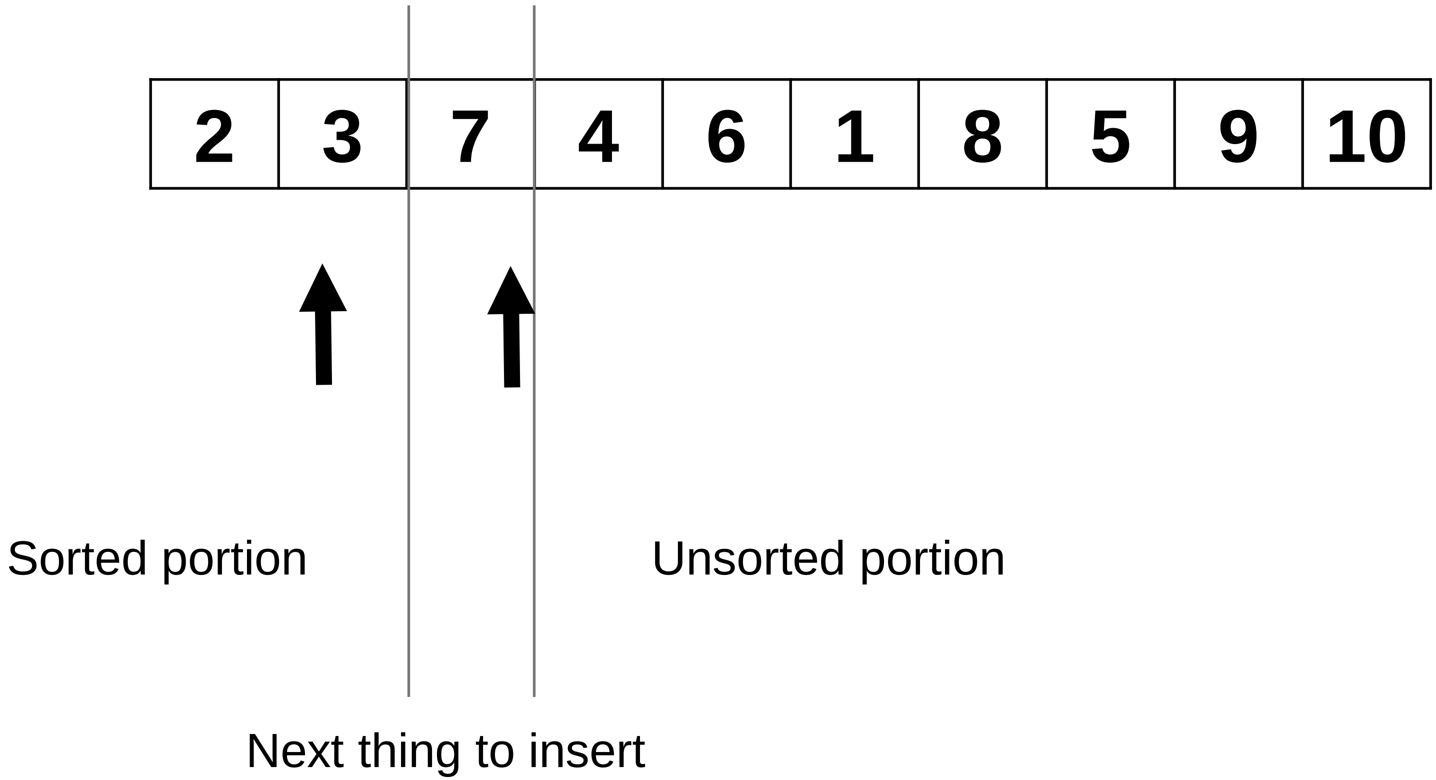
Unsorted portion

Next thing to insert

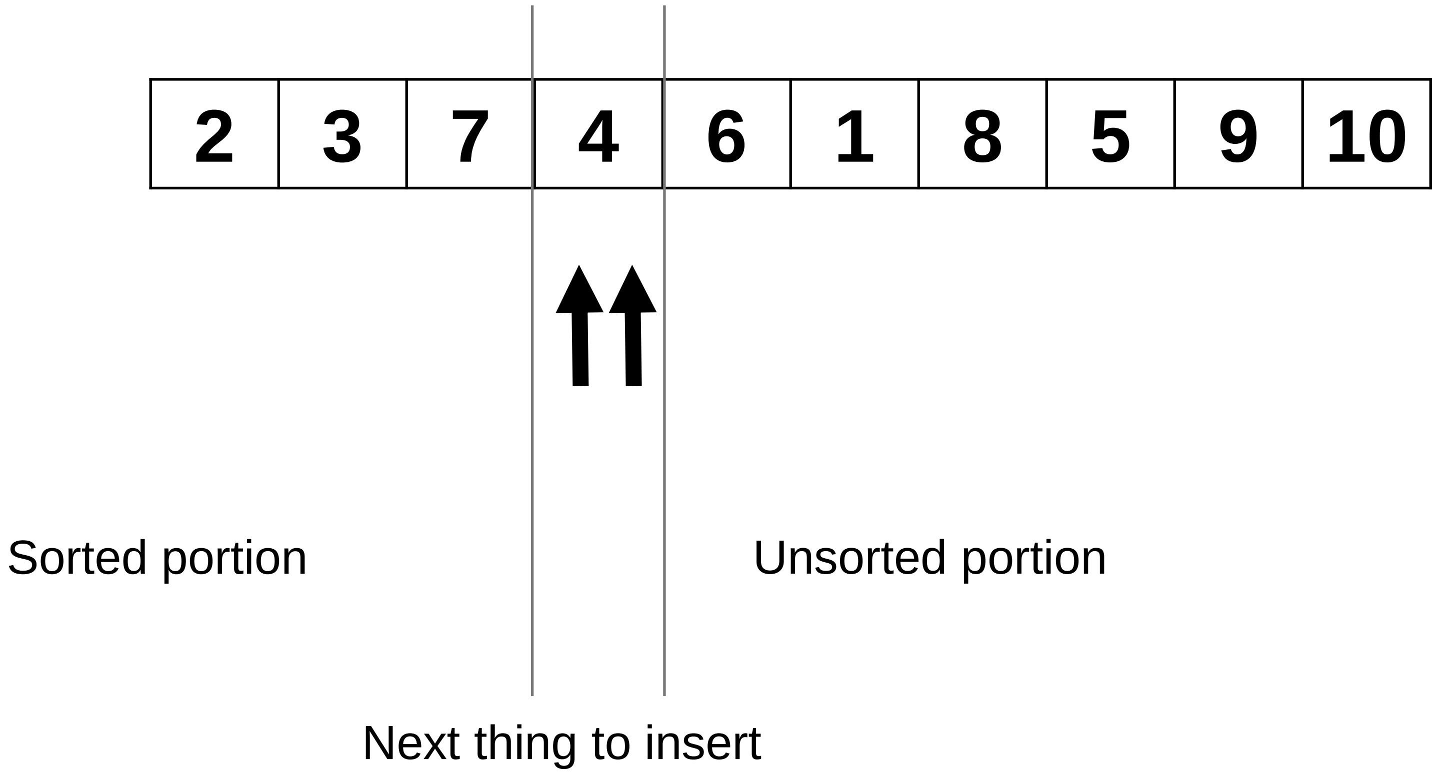
INSERTION SORT



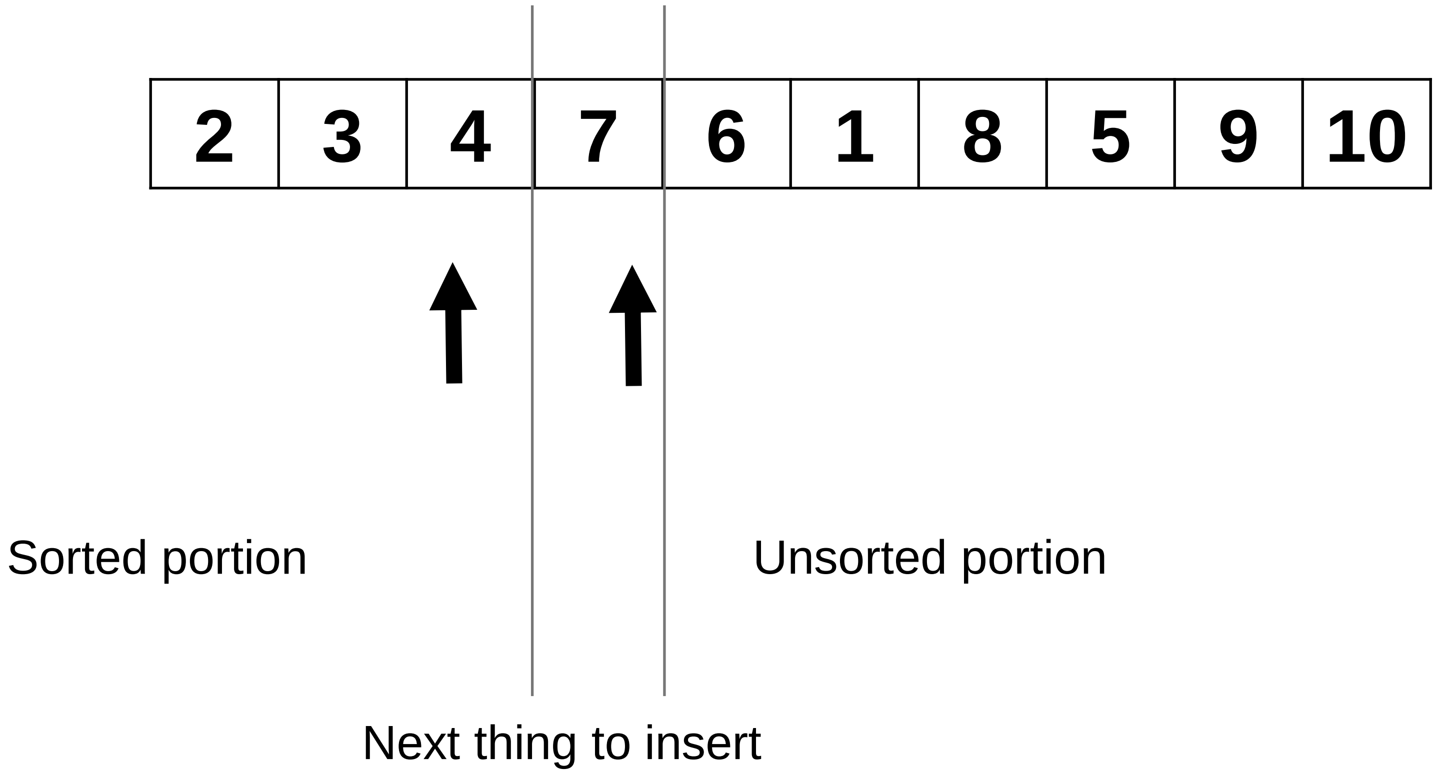
INSERTION SORT



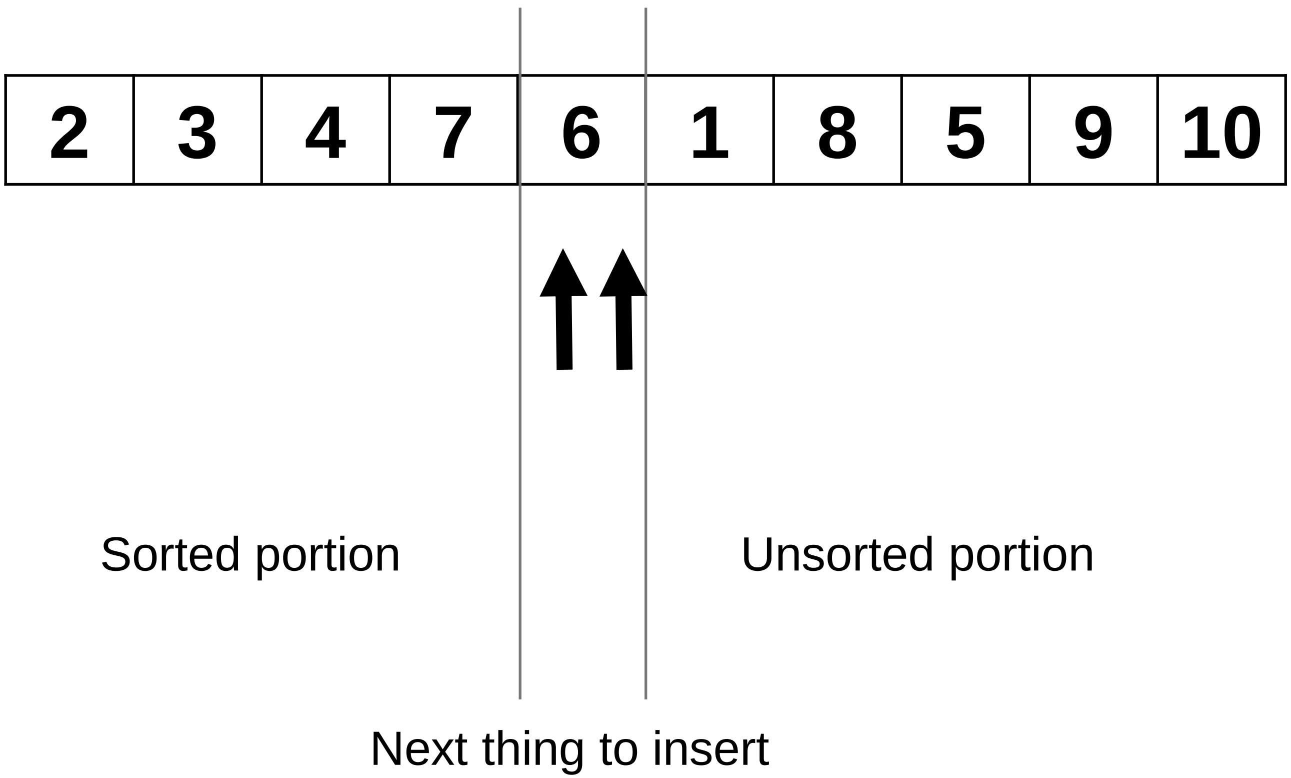
INSERTION SORT



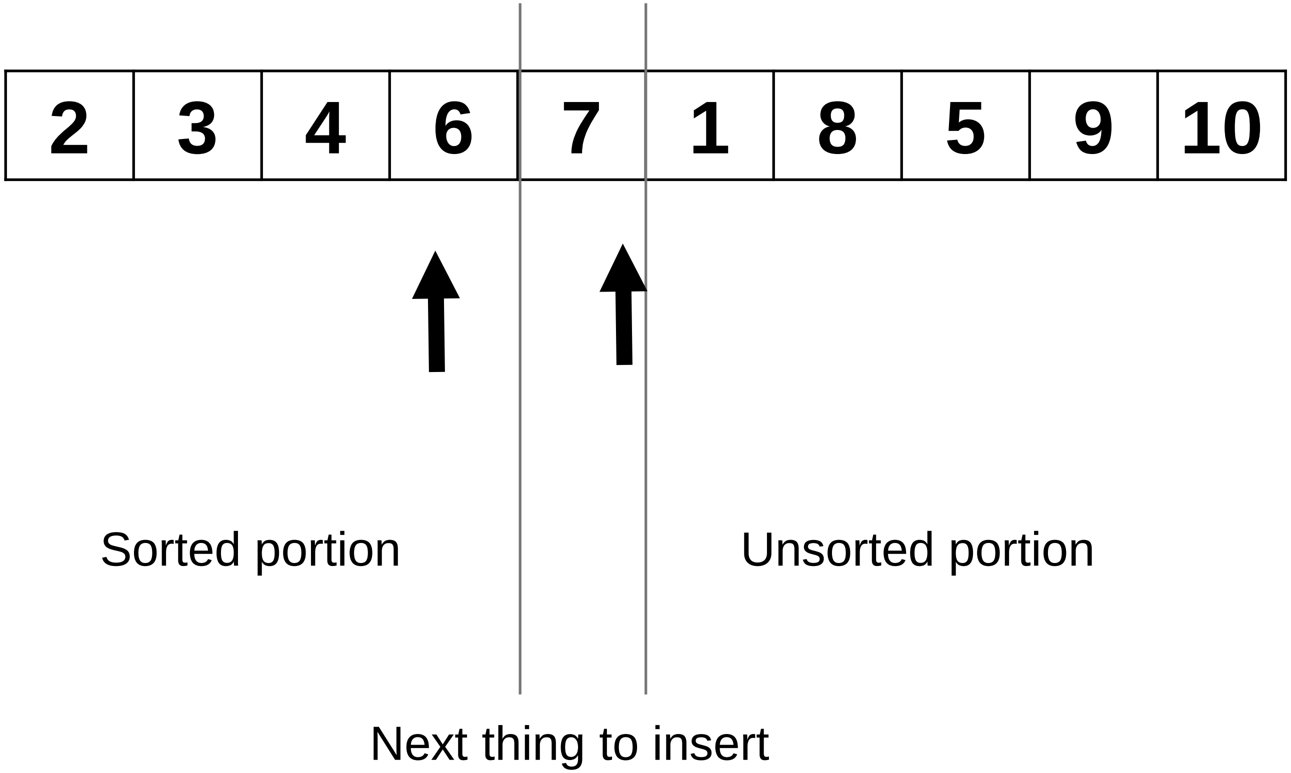
INSERTION SORT



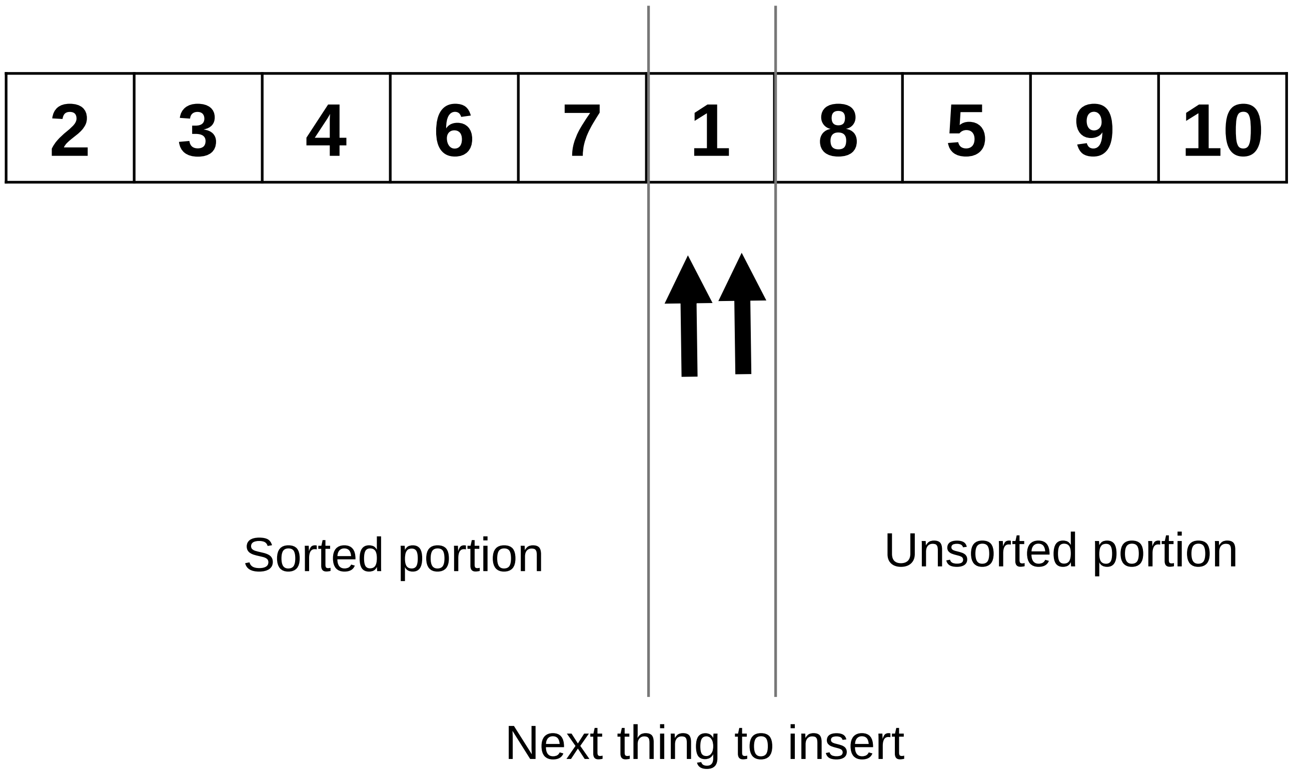
INSERTION SORT



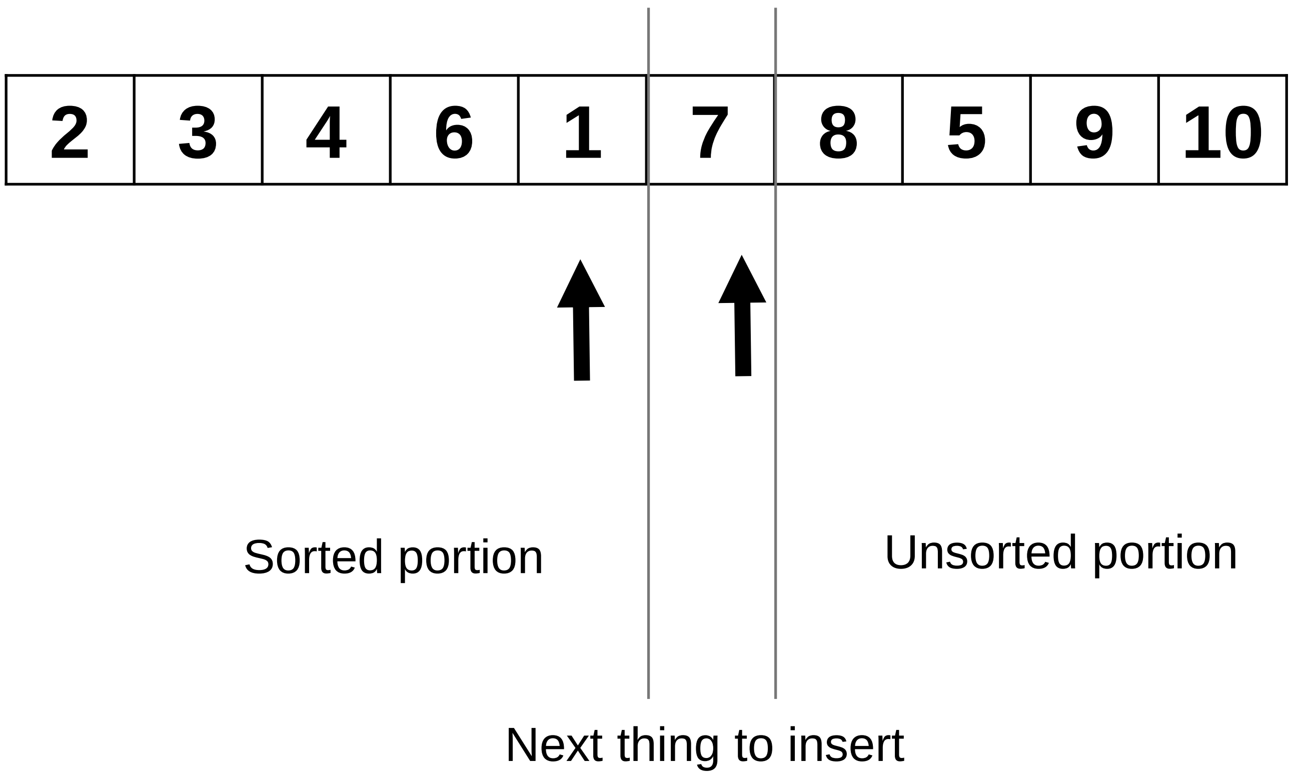
INSERTION SORT



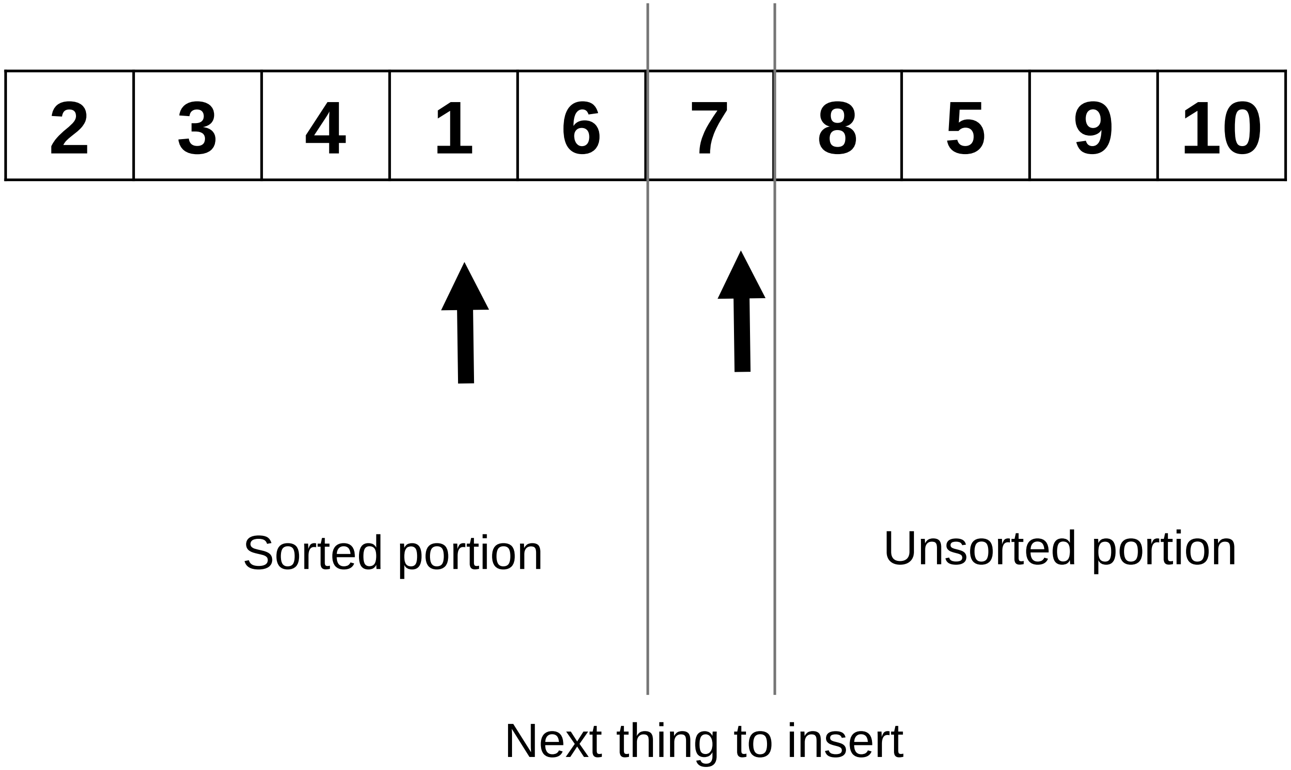
INSERTION SORT



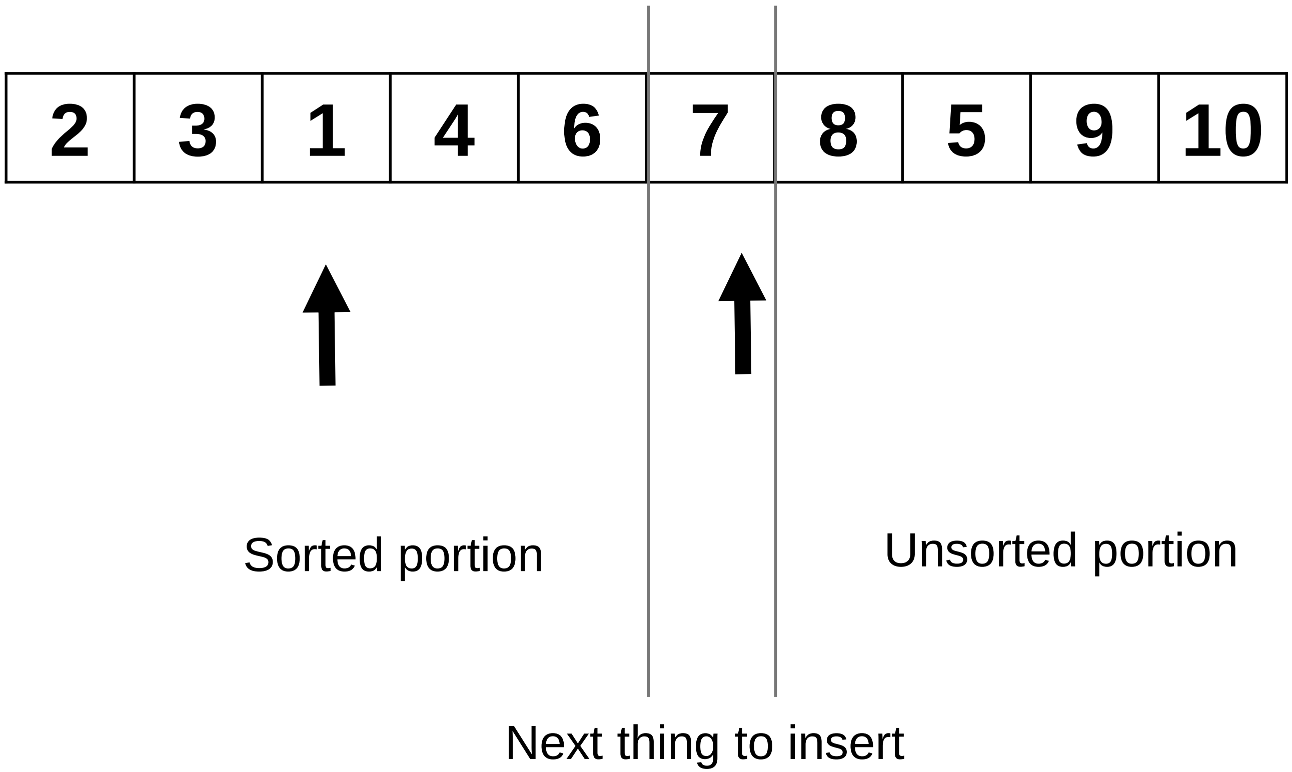
INSERTION SORT



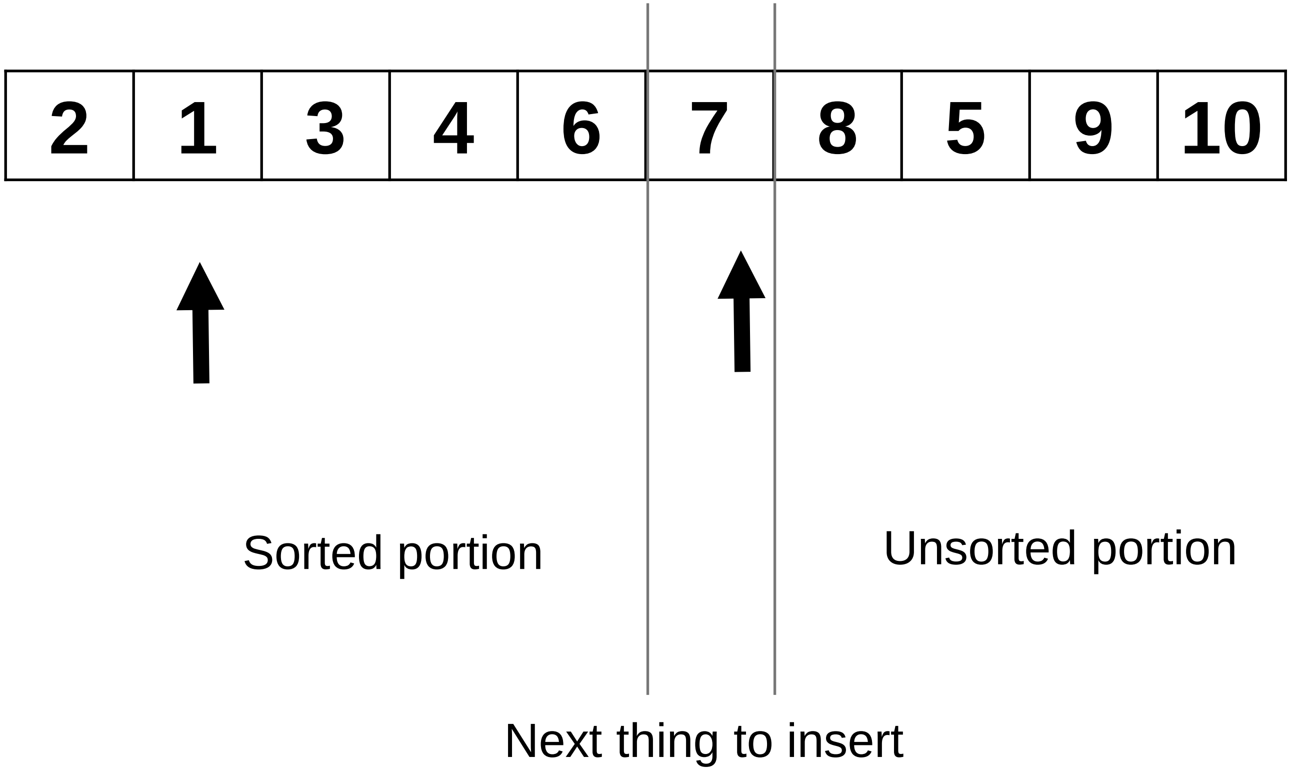
INSERTION SORT



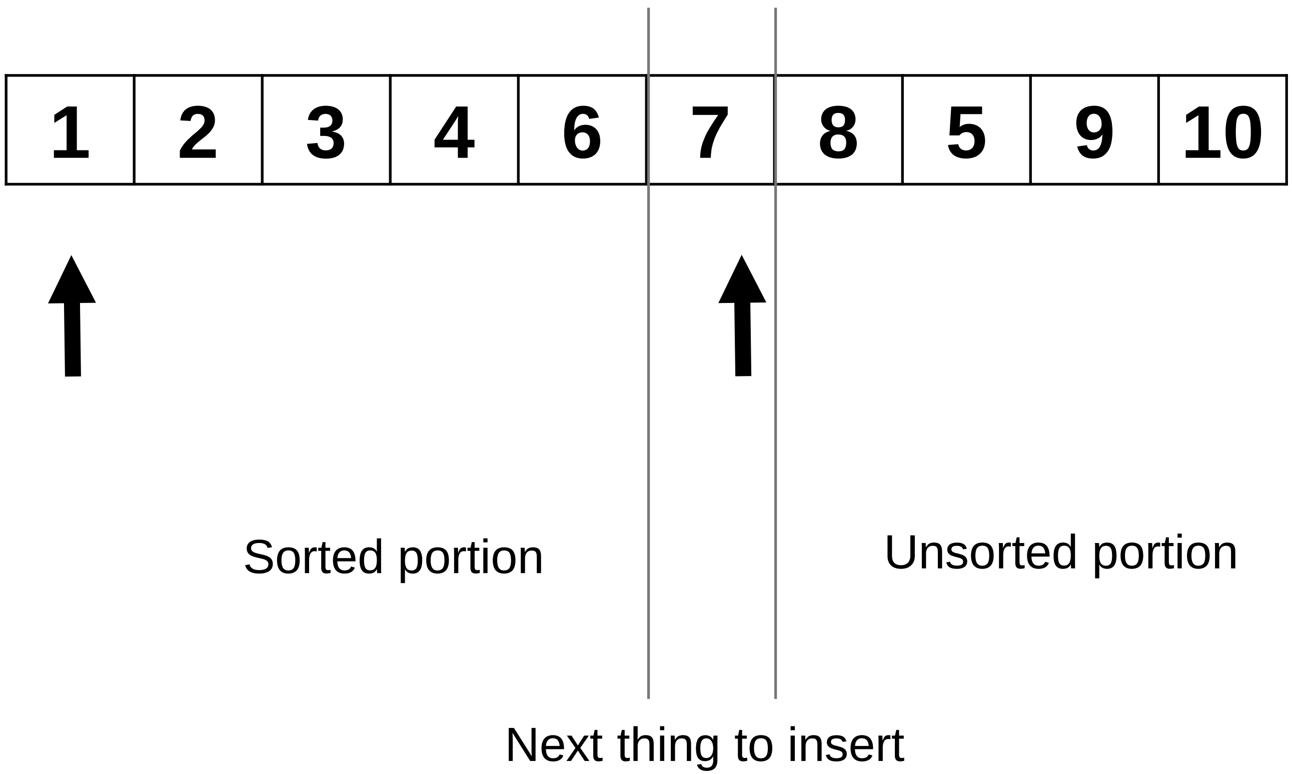
INSERTION SORT



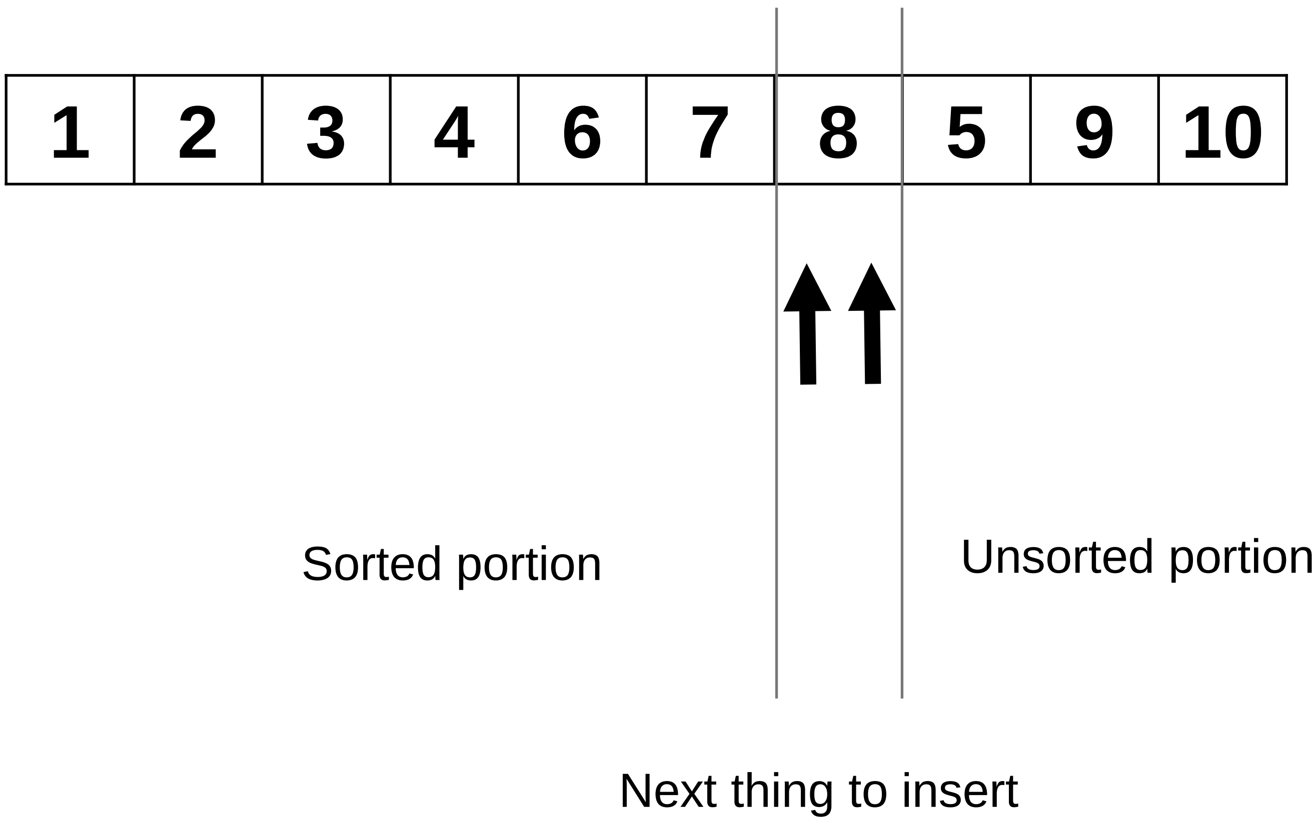
INSERTION SORT



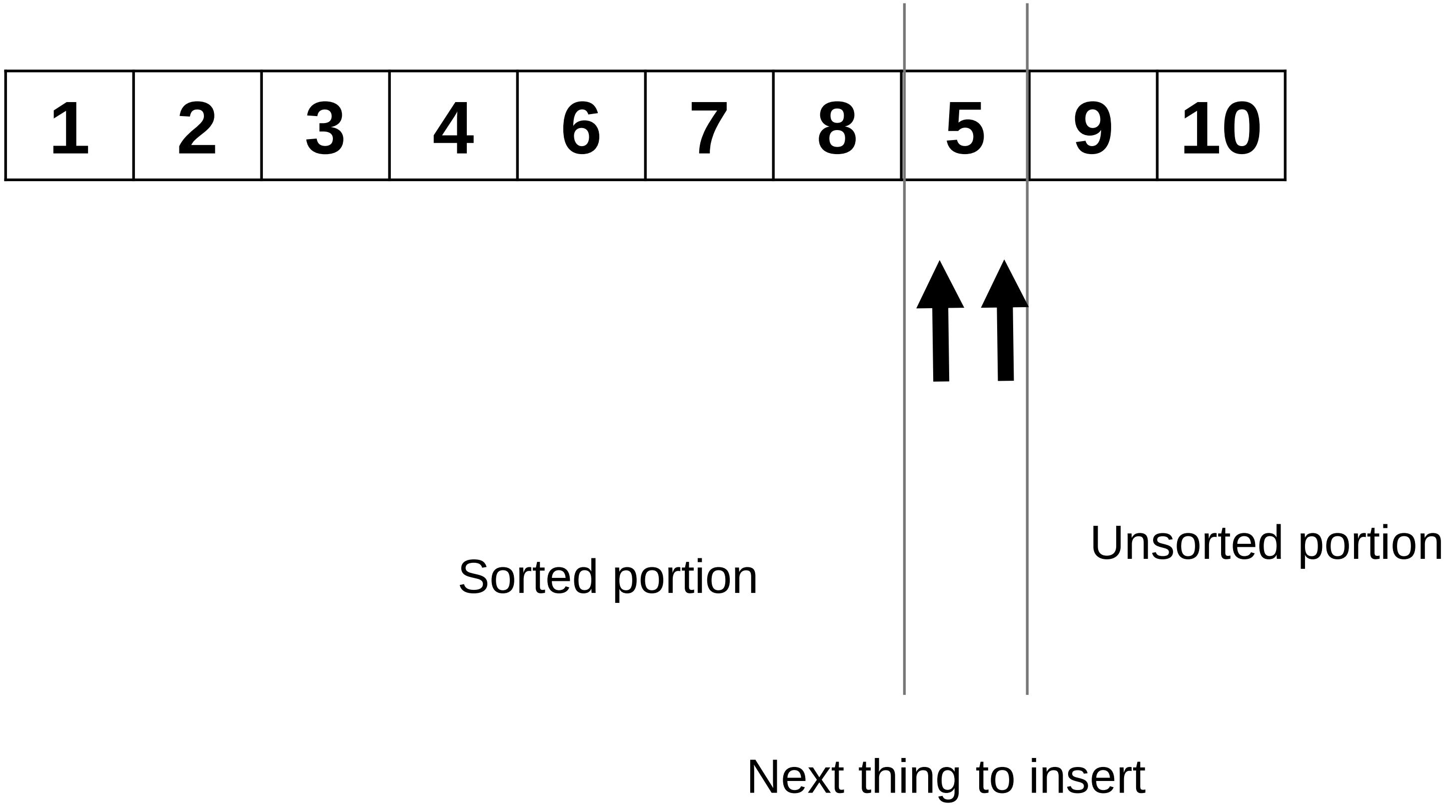
INSERTION SORT



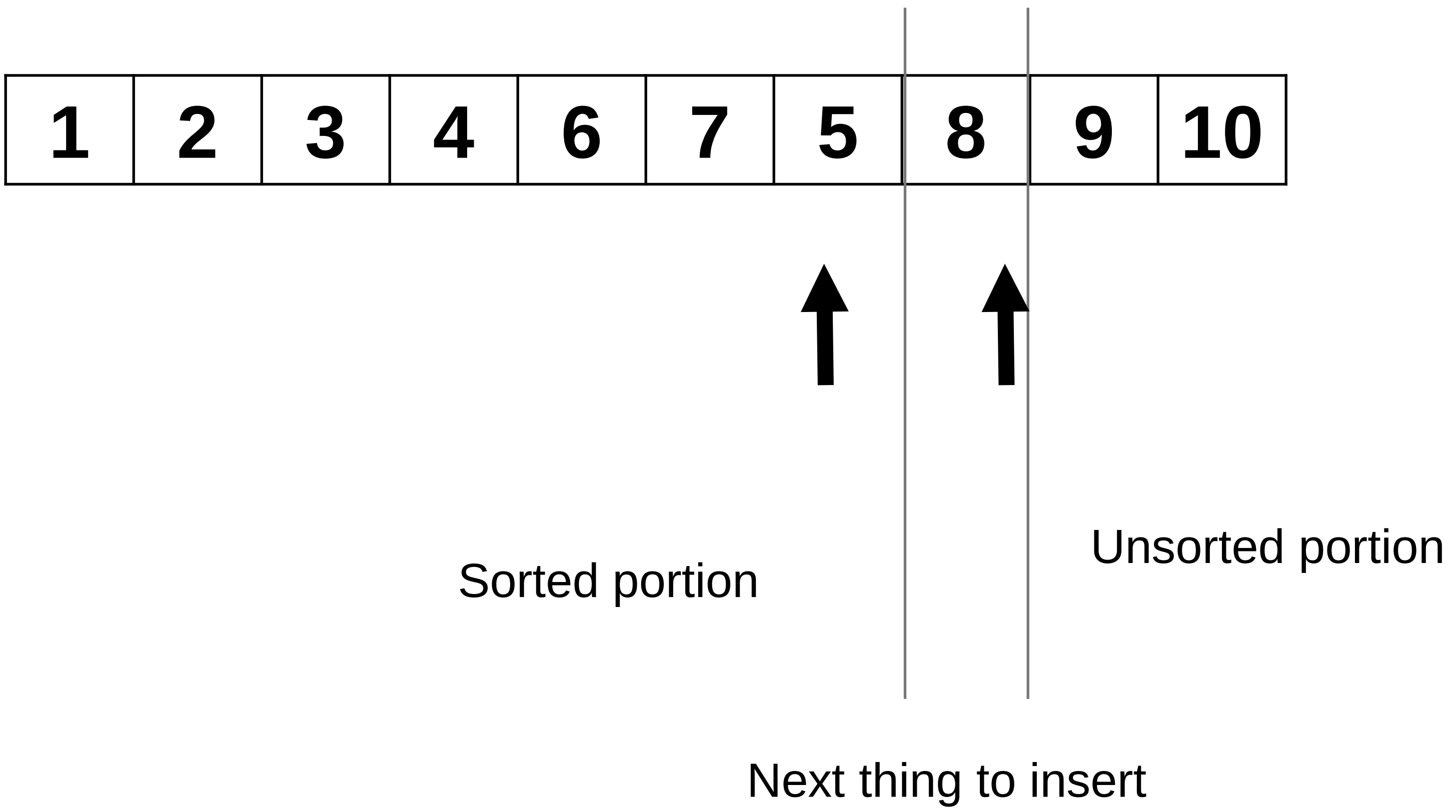
INSERTION SORT



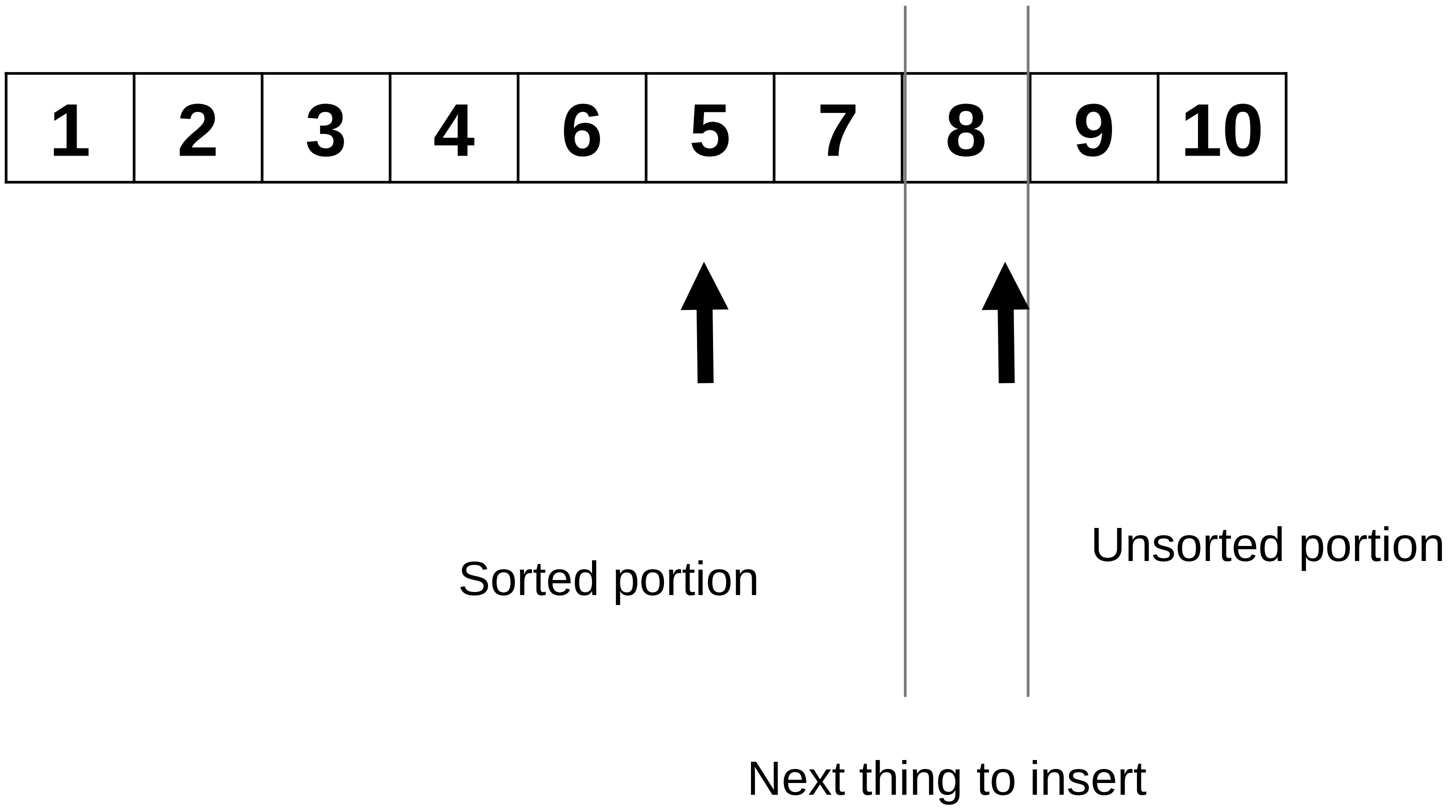
INSERTION SORT



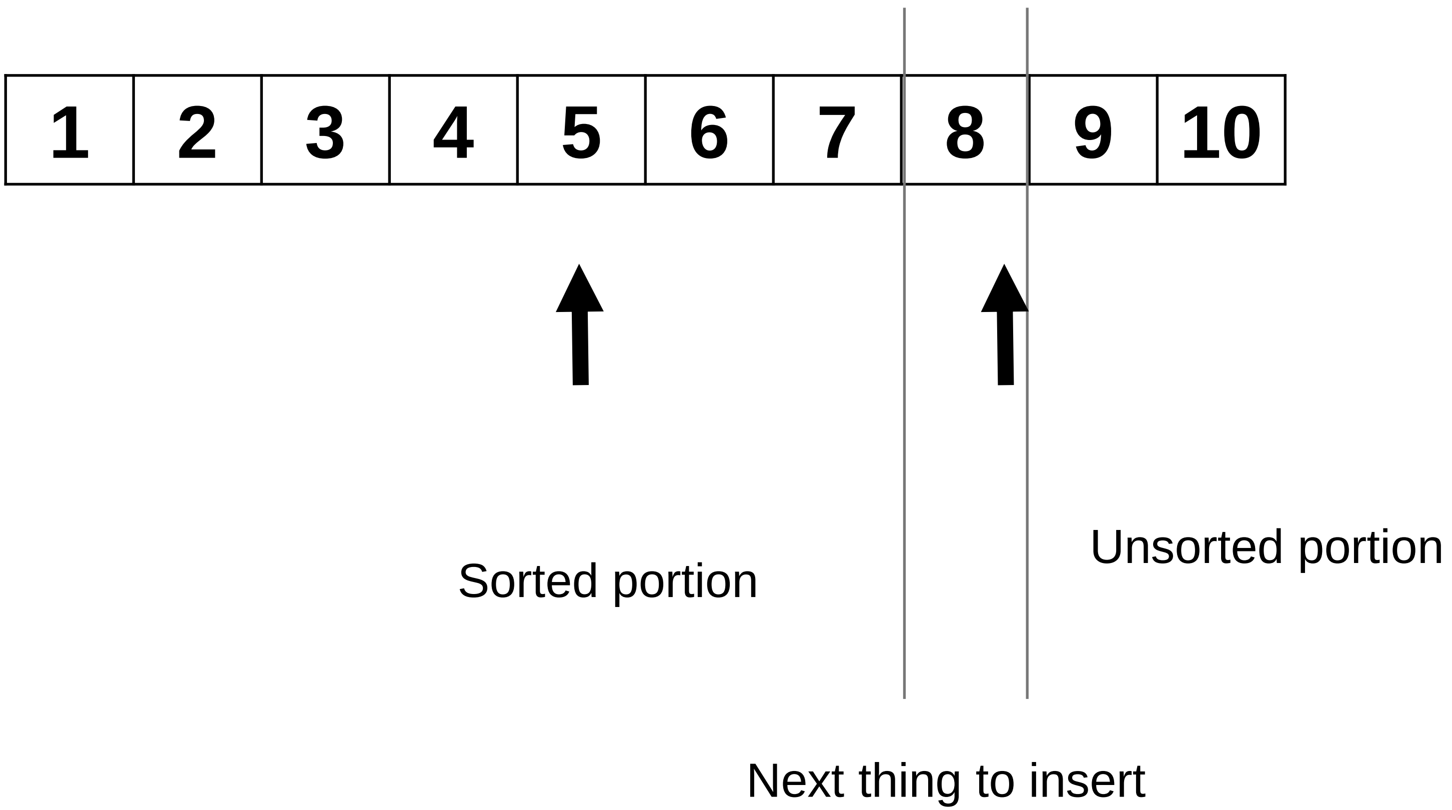
INSERTION SORT



INSERTION SORT

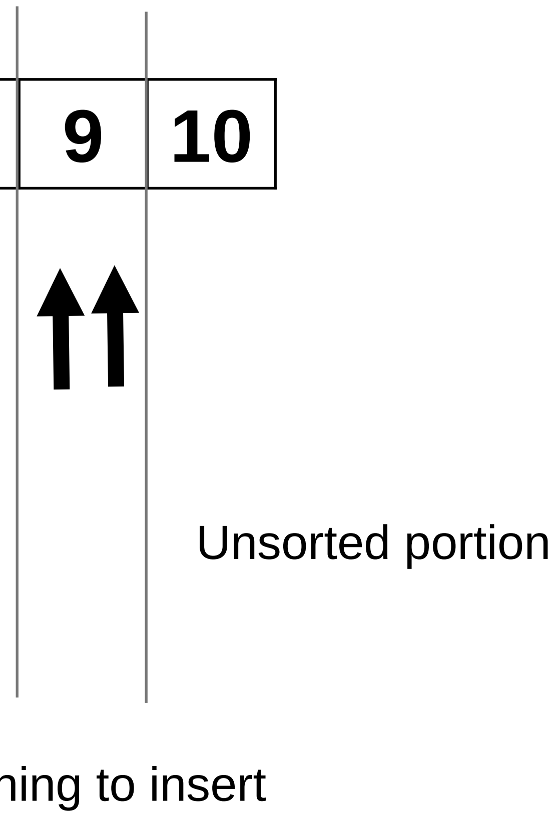


INSERTION SORT

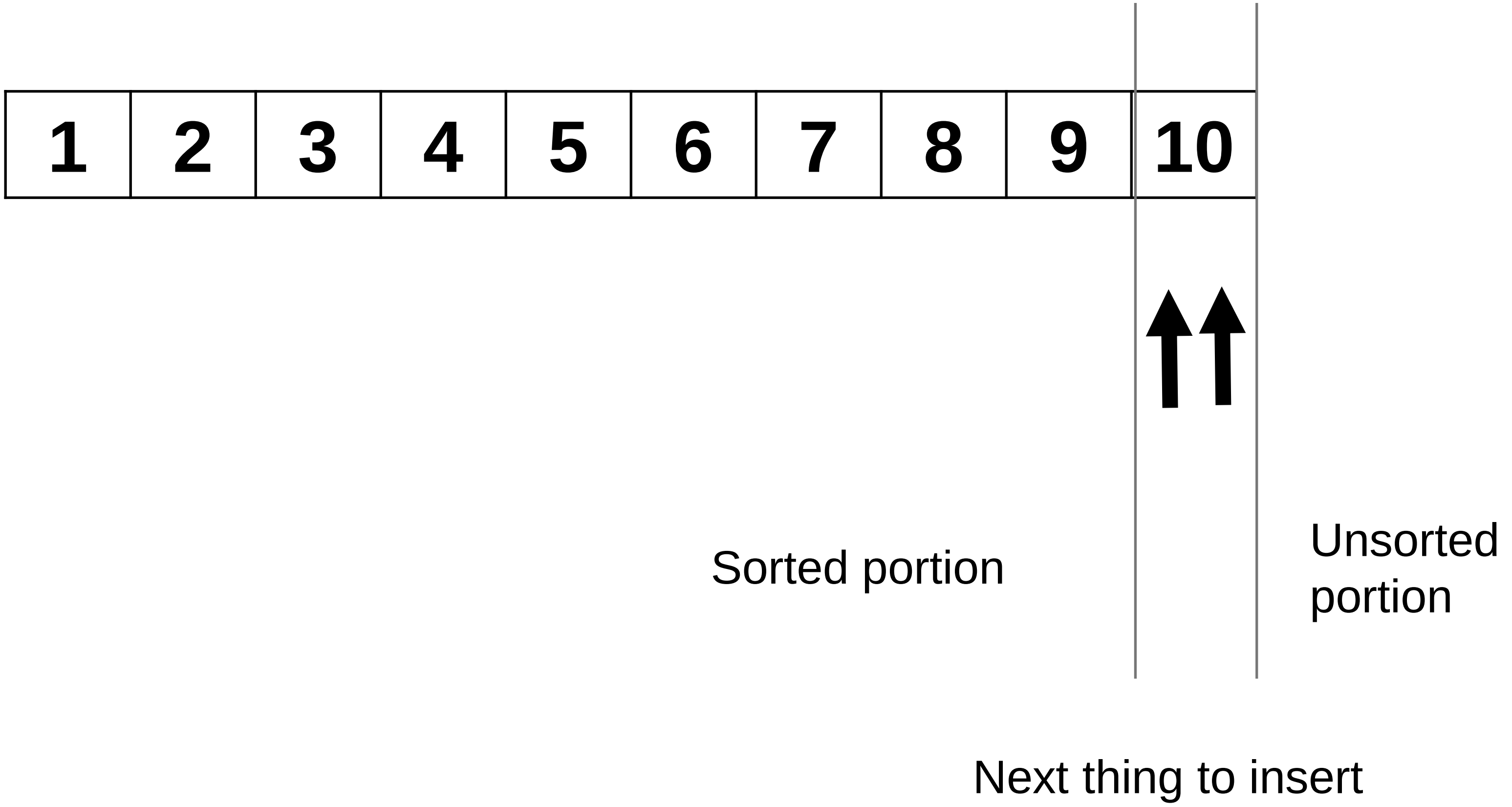


INSERTION SORT

1	2	3	4	5	6	7	8	9	10



INSERTION SORT



INSERTION SORT

1	2	3	4	5	6	7	8	9	10

Sorted portion

Unsorted portion

SORTING ALGORITHMS

MERGE SORT

MERGE SORT

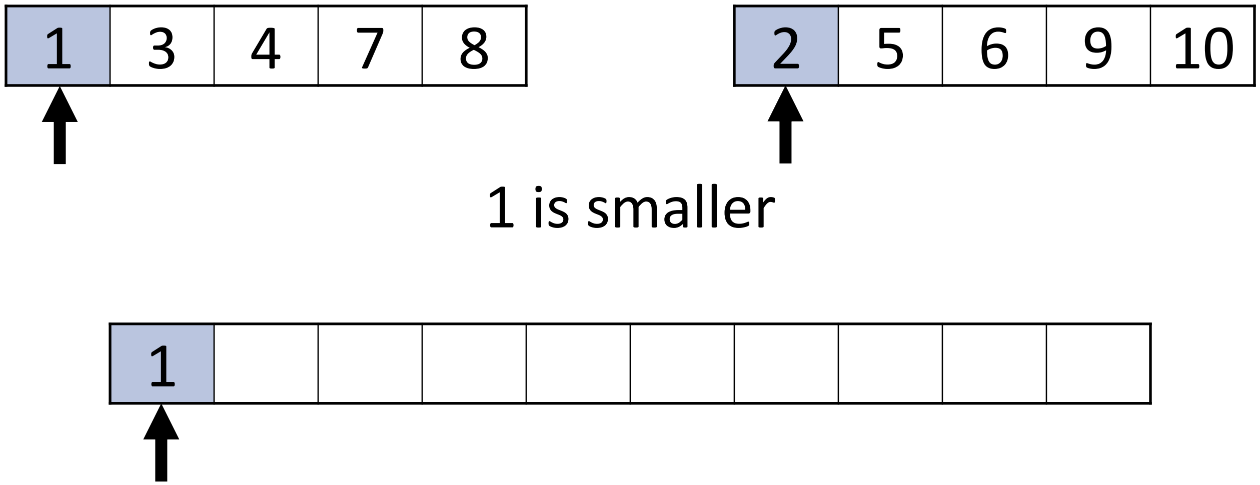
- **Merge sort is a very clever "divide and conquer" sorting algorithm that is much faster than the previous methods**
- **The key idea is that sorting N data values can be broken into three steps**
  - Divide the input data into two parts that are N/2 long
  - Sort the two arrays of N/2 values
  - Merge the two arrays of N/2 values to get N sorted values
- **First we demonstrate the merging step with the following:**

1	3	4	7	8

2	5	6	9	10

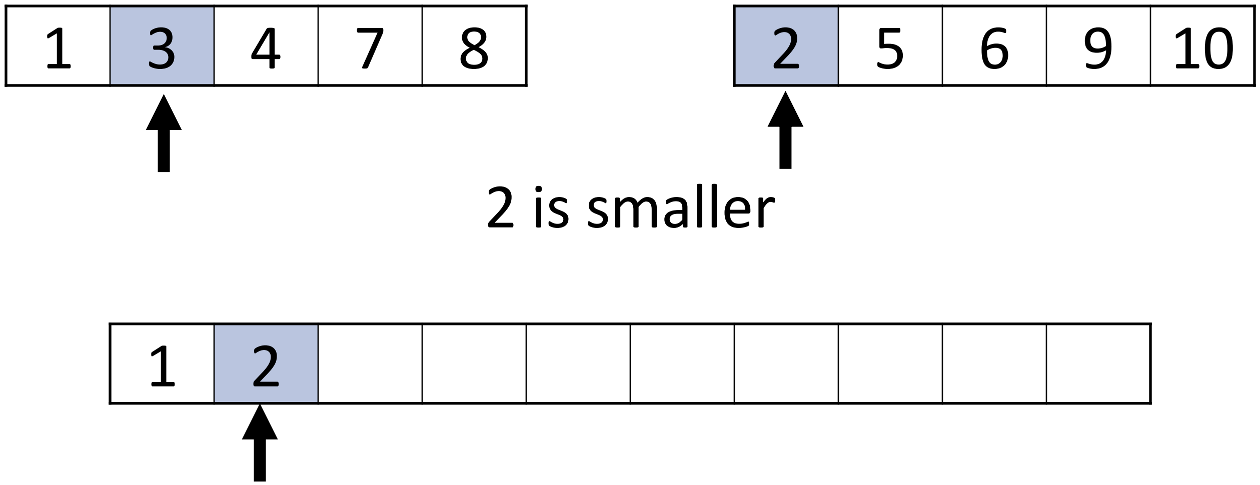
MERGE SORT

- **Start with two sorted arrays, start two indices at smallest values in each array, copy smallest value to merged array**



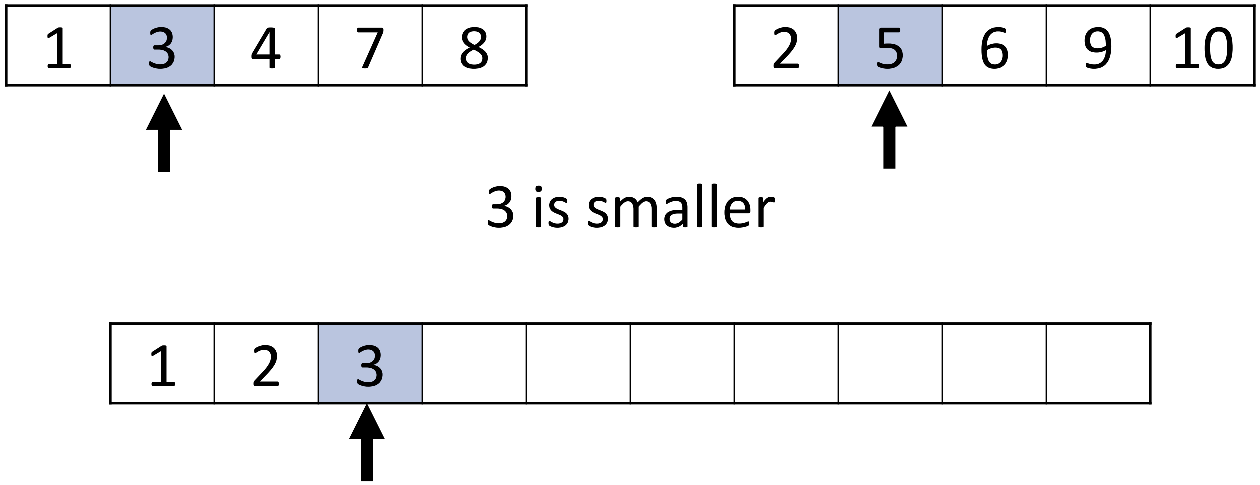
MERGE SORT

- **Advance the array index on one array, select smallest value and copy into merged array**



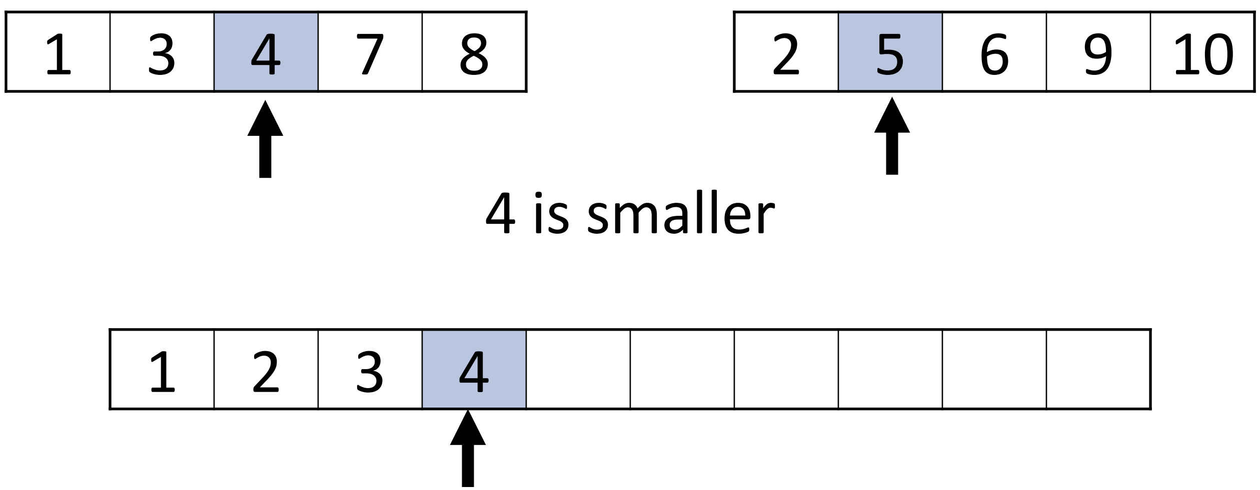
MERGE SORT

- **Advance the array index on one array, select smallest value and copy into merged array**



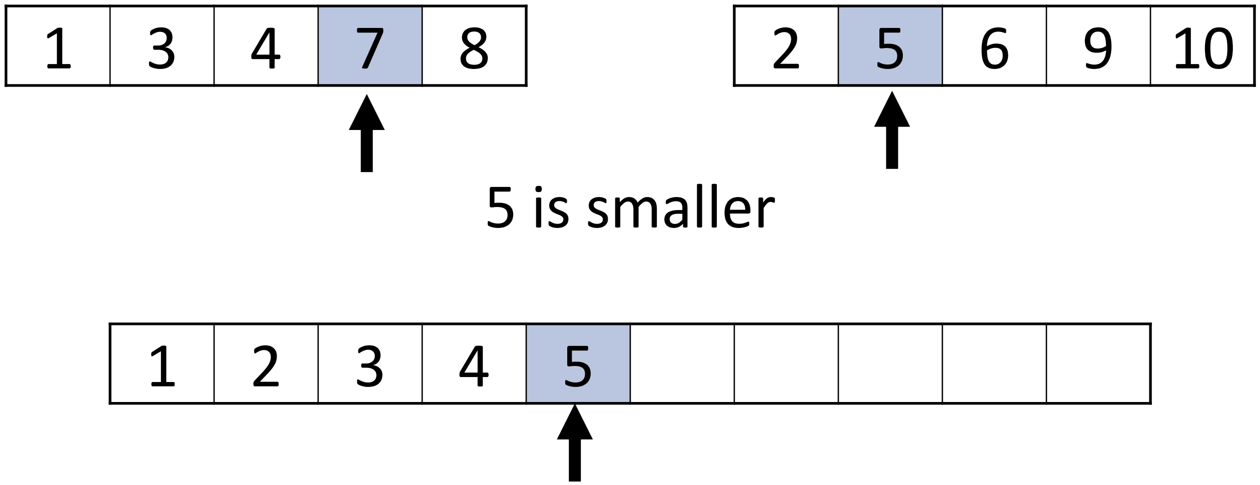
MERGE SORT

- **Advance the array index on one array, select smallest value and copy into merged array**



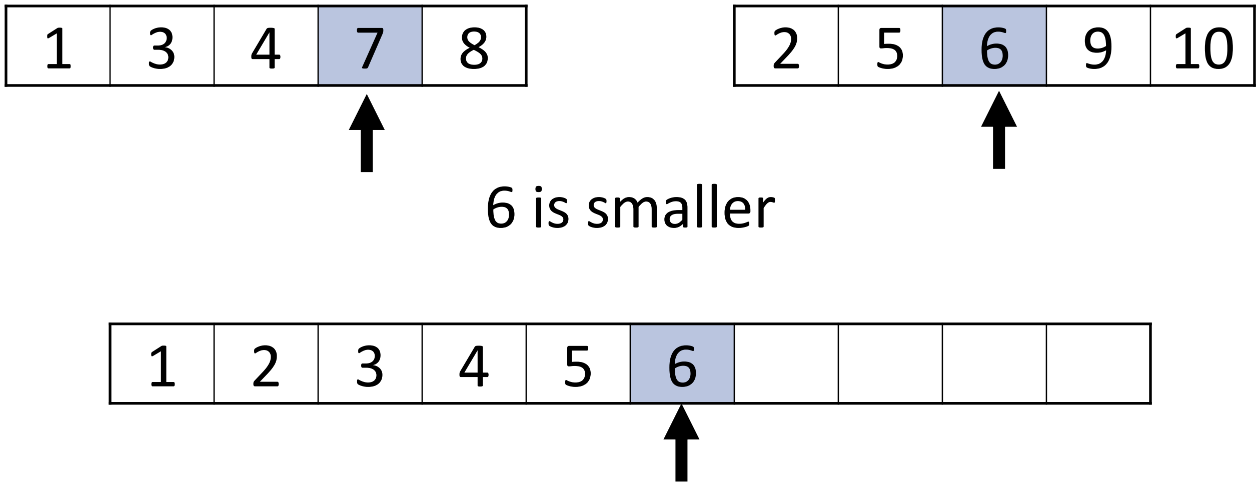
MERGE SORT

- **Advance the array index on one array, select smallest value and copy into merged array**



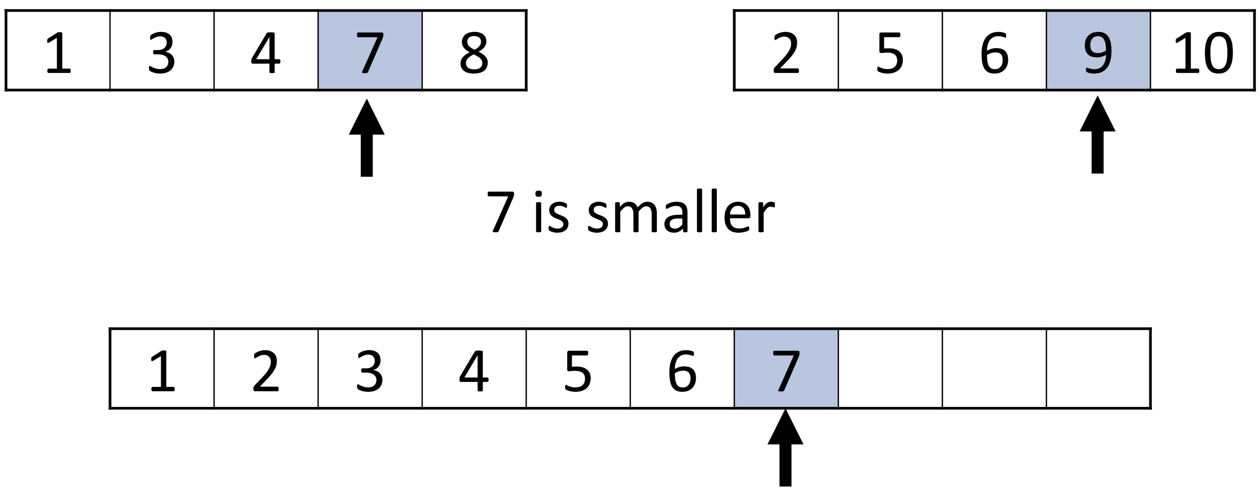
MERGE SORT

- **Advance the array index on one array, select smallest value and copy into merged array**



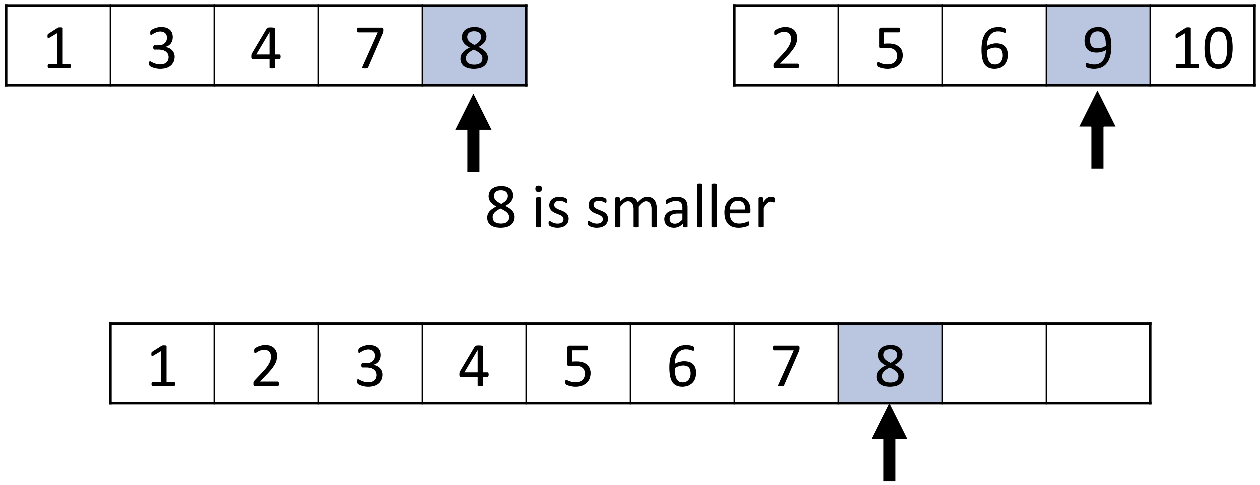
MERGE SORT

- **Advance the array index on one array, select smallest value and copy into merged array**



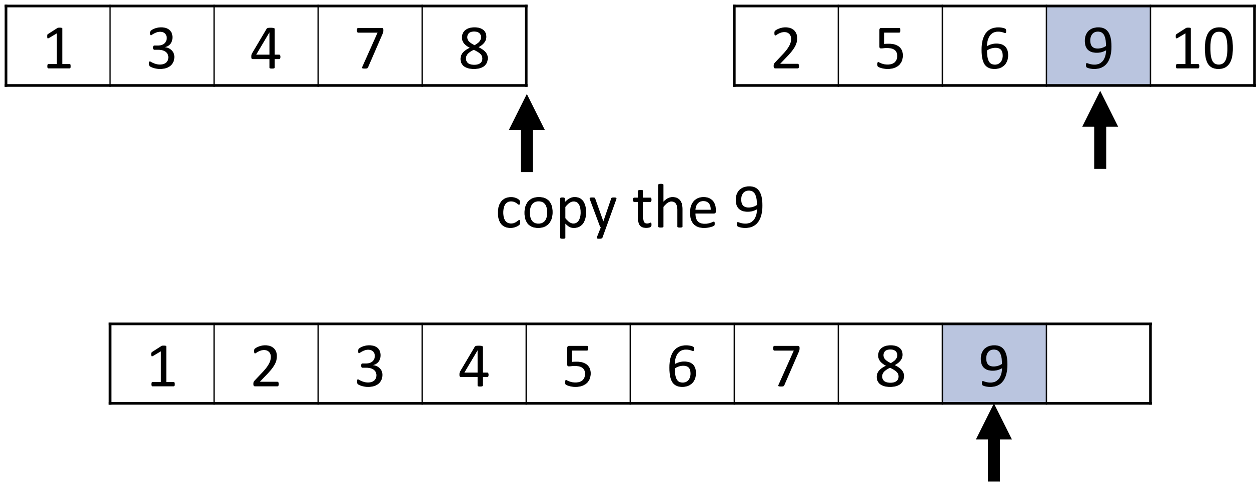
MERGE SORT

- **Advance the array index on one array, select smallest value and copy into merged array**



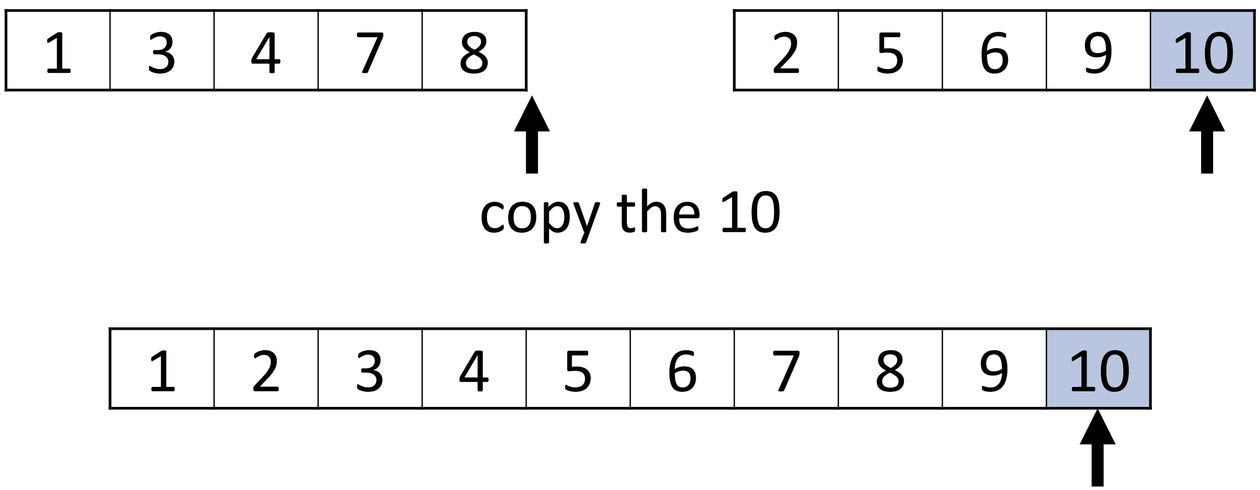
MERGE SORT

- **Index has reached end of one array, copy remaining values from second array into merged array**



MERGE SORT

- **Index has reached end of one array, copy remaining values from second array into merged array**



MERGE SORT

- **How are we going to sort the two arrays of N/2 values?**
  - Divide both arrays of N/2 values into arrays of N/4 values
  - Sort the arrays of N/4 values
  - Merge arrays of N/4 values to create array of N/2 values

- **How are we going to sort the two arrays of N/4 values?**
  - Divide both arrays of N/4 values into arrays of N/8 values
  - Sort the arrays of N/8 values
  - Merge arrays of N/8 values to create array of N/4 values

- **We continue this recursive “divide and conquer” process until the array being divided is only one element long**

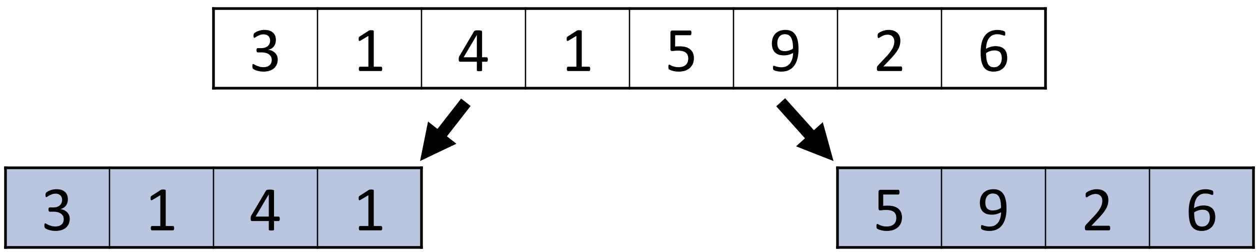
MERGE SORT

Start with an unsorted array of length N=8

3	1	4	1	5	9	2	6

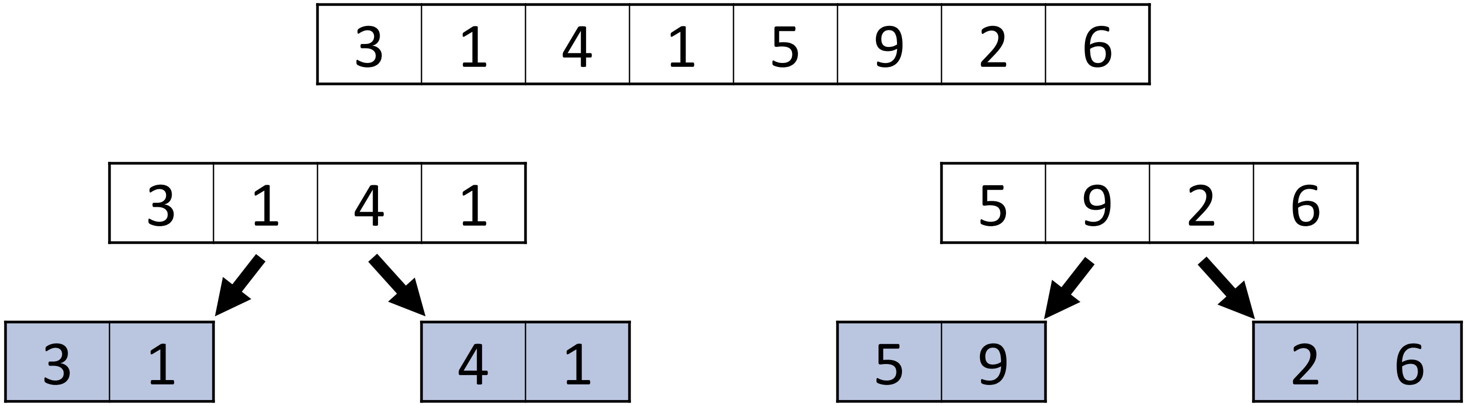
MERGE SORT

Split into 2 arrays of length N/2=4



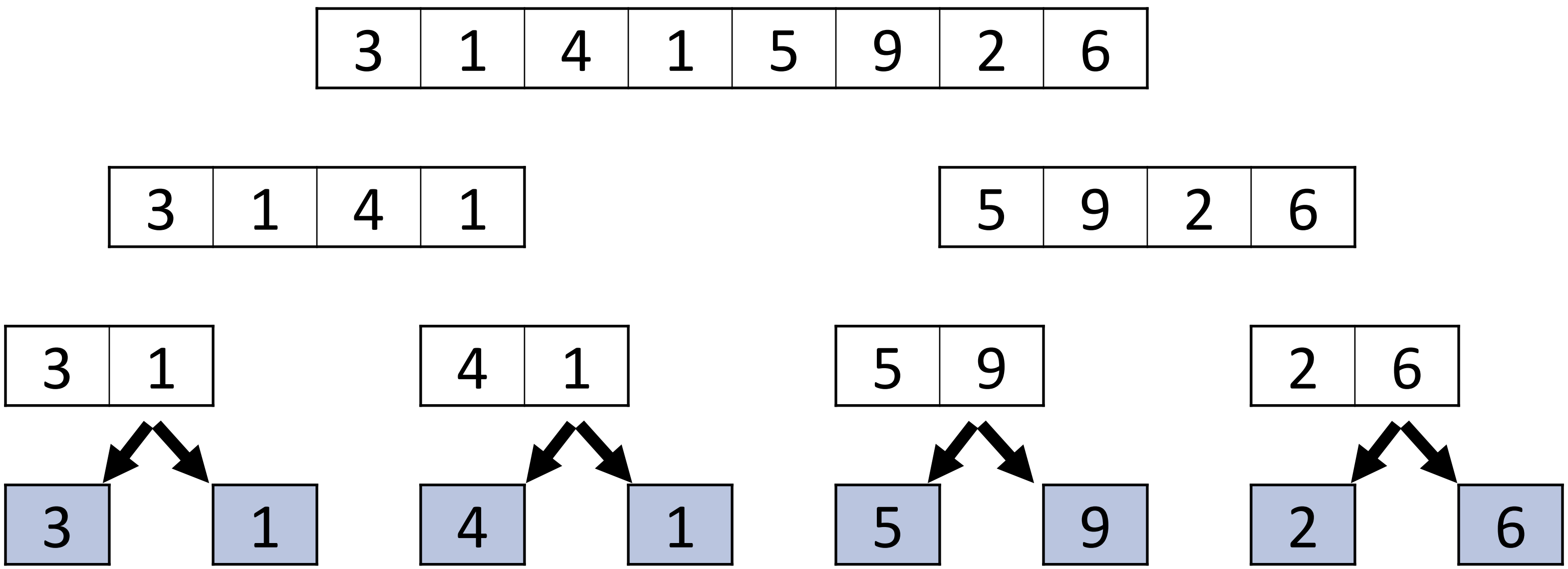
MERGE SORT

Split into 4 arrays of length N/4=2



MERGE SORT

Split into N=8 arrays of length 1



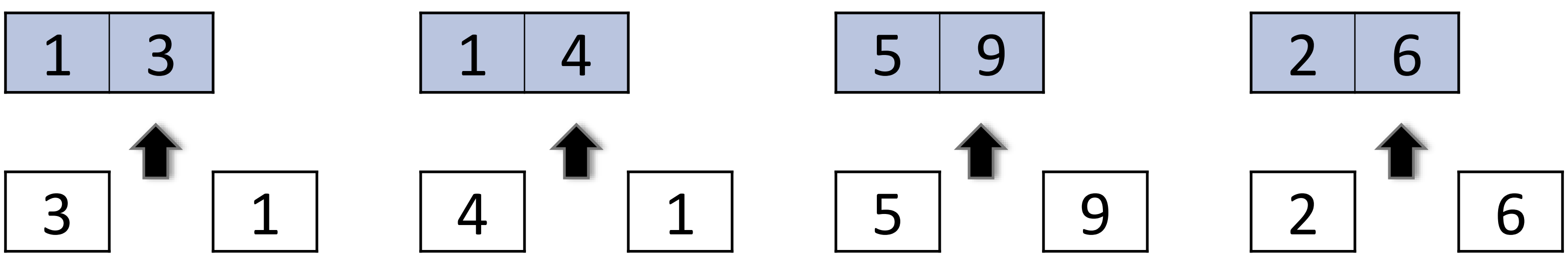
MERGE SORT

Start merging phase with N=8 arrays of length 1

3	1	4	1	5	9	2	6

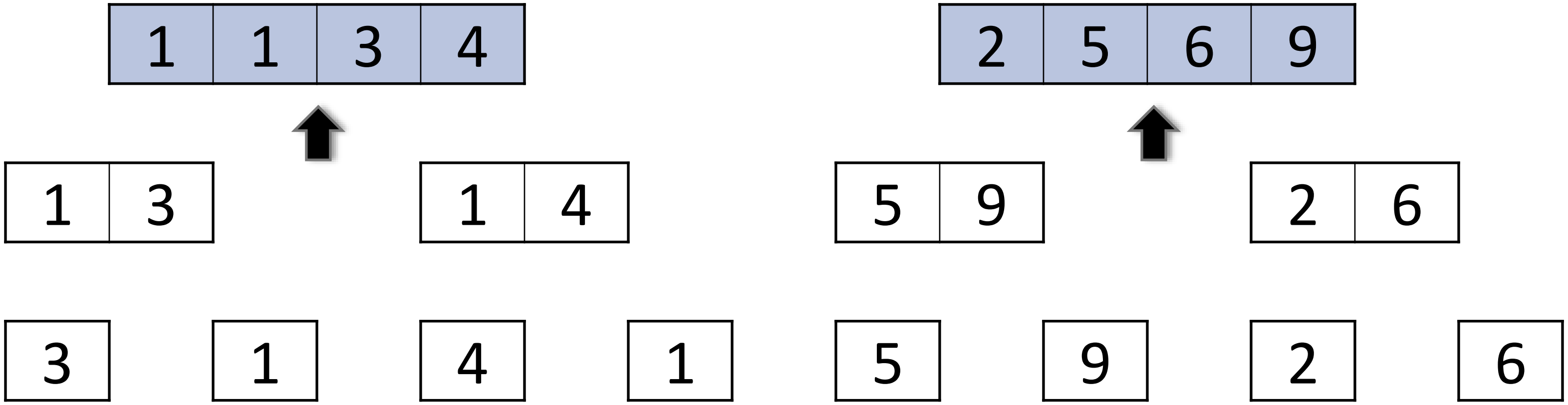
MERGE SORT

Merge to create N/2=4 sorted arrays of length 2



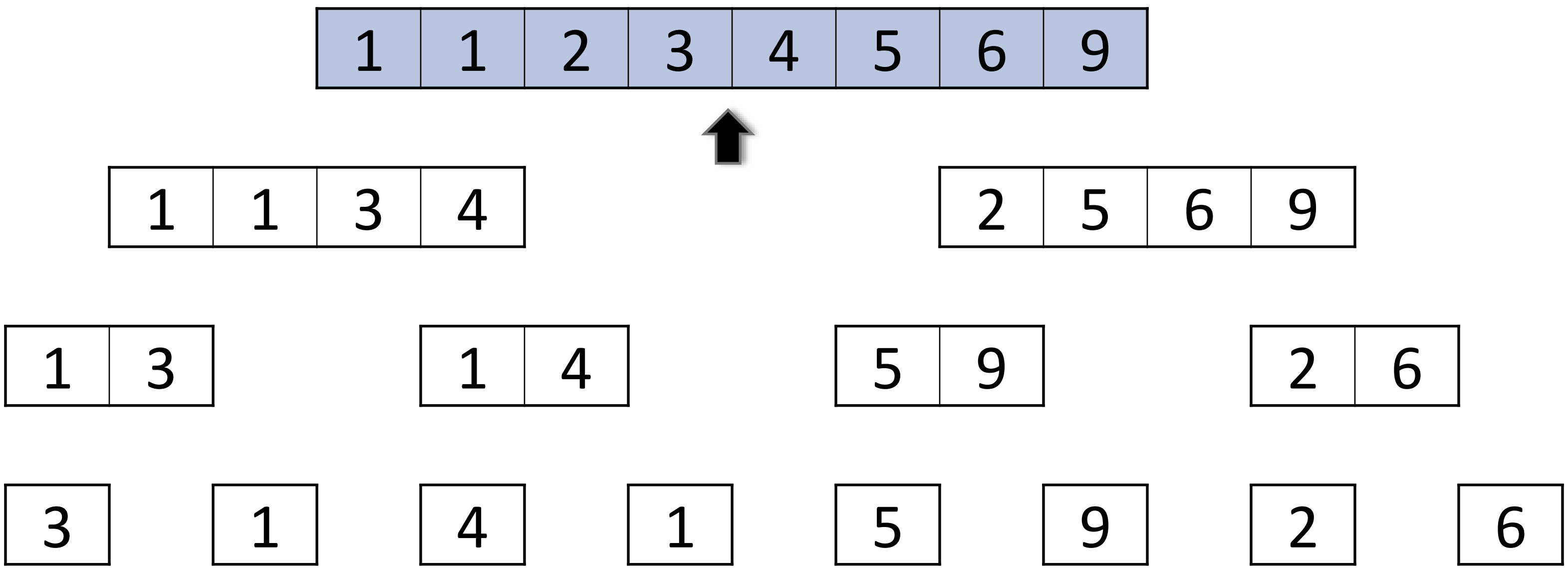
MERGE SORT

Merge to create N/4=2 sorted arrays of length 4



MERGE SORT

Merge to create N/8=1 sorted array of length 8



MERGE SORT

```
void merge_sort(int data[], int low, int high)
{
   // Check terminating condition
   int count = high - low + 1;
   if (count > 1)
   {
      // Divide the array and sort both halves
      int mid = (low + high) / 2;
      merge_sort(data, low, mid);
      merge_sort(data, mid + 1, high);

      // Merge sorted arrays
      ...
```

The terminating condition is when the array is <= 1 long

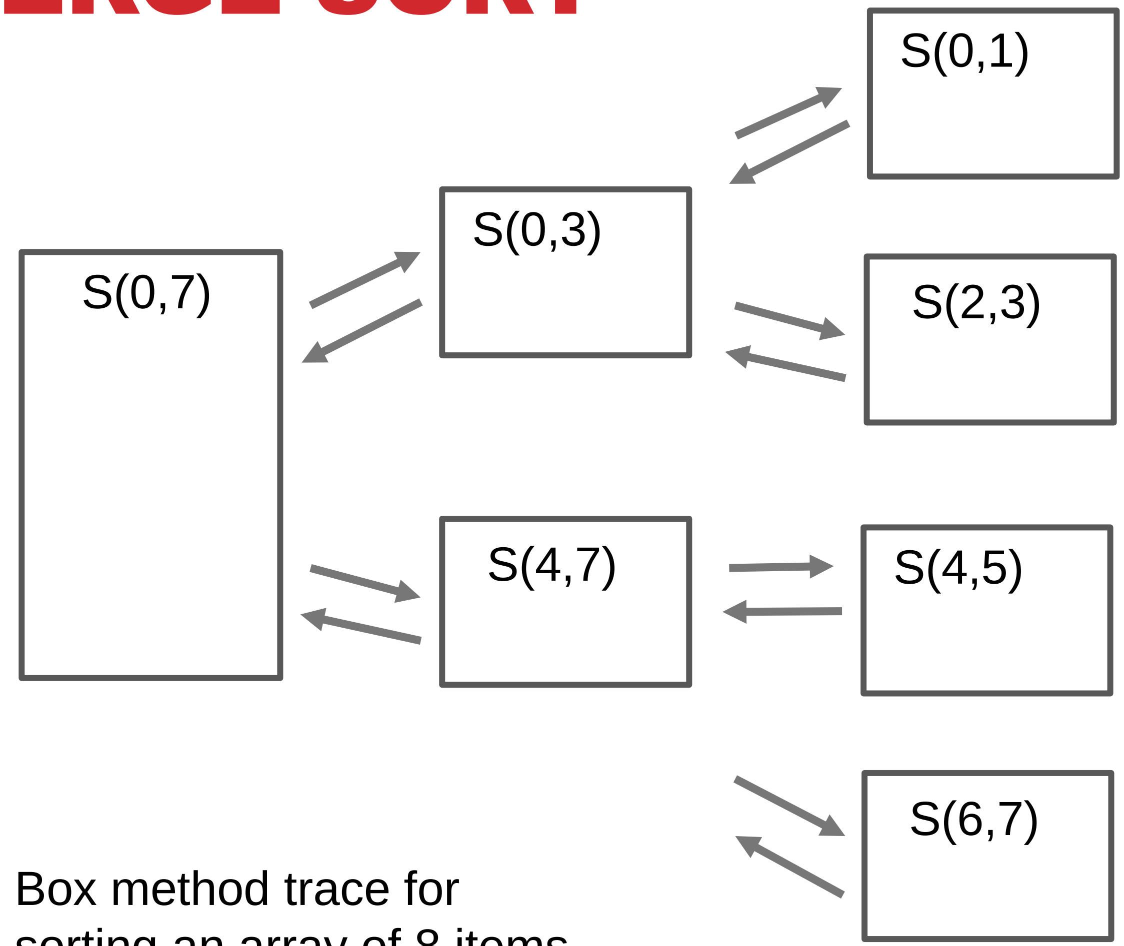
MERGE SORT

```
void merge_sort(int data[], int low, int high)
{
   // Check terminating condition
   int count = high - low + 1;
   if (count > 1)
   {
      // Divide the array and sort both halves
      int mid = (low + high) / 2;
      merge_sort(data, low, mid);
      merge_sort(data, mid + 1, high);

      // Merge sorted arrays
      ...
```

We make two recursive calls to sort the left and right halves of the input array

MERGE SORT



Box method trace for
sorting an array of 8 items

MERGE SORT

```
// Create temporary array for merged data
int *copy = new int[range];

// Initialize array indices
int index1 = low;
int index2 = mid + 1;
int index = 0;

// Merge smallest data elements into copy array
while (index1 <= mid && index2 <= high)
{
    if (data[index1] <= data[index2])
        copy[index++] = data[index1++];
    else
        copy[index++] = data[index2++];
}
...
```

Next, we merge the two sorted arrays into a temporary array

MERGE SORT

```
    ...
    // Copy any remaining entries from the first half
    while (index1 <= mid)
        copy[index++] = data[index1++];

    // Copy any remaining entries from the second half
    while (index2 <= high)
        copy[index++] = data[index2++];

    // Copy data back from the temporary array
    for (index = 0; index < range; index++)
        data[low + index] = copy[index];
    delete[]copy;
  }
}
```

Finally, we copy temporary array back into original array

MERGE SORT

Experimental results:

```
Enter number of data values: 100
CPU time = 5.4e-05 sec

Enter number of data values: 1000
CPU time = 0.000439 sec

Enter number of data values: 10000
CPU time = 0.004654 sec

Enter number of data values: 100000
CPU time = 0.046654 sec
```

MERGE SORT

Experimental results:

```
Enter number of data values: 100
CPU time = 5.4e-05 sec

Enter number of data values: 1000
CPU time = 0.000439 sec

Enter number of data values: 10000
CPU time = 0.004654 sec

Enter number of data values: 100000
CPU time = 0.046654 sec
```

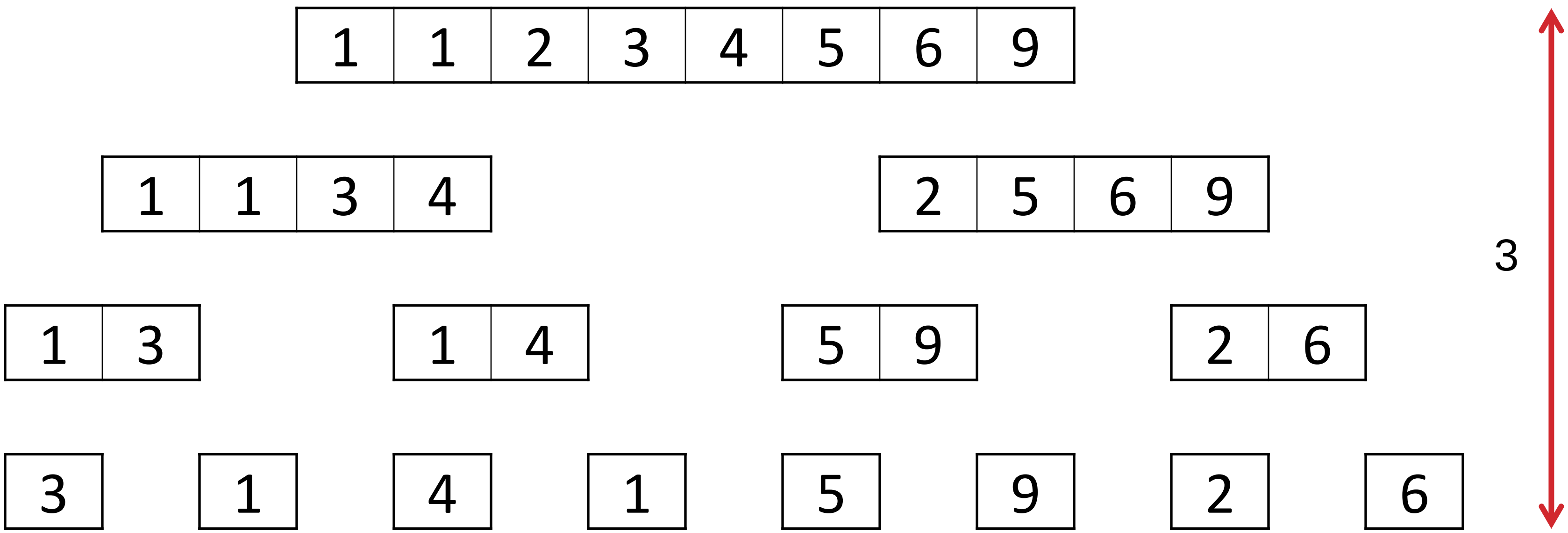
This is much faster than insertion sort (8 sec) selection sort (14 sec) or bubble sort (30 sec)

SORTING ALGORITHMS

MERGE SORT ANALYSIS

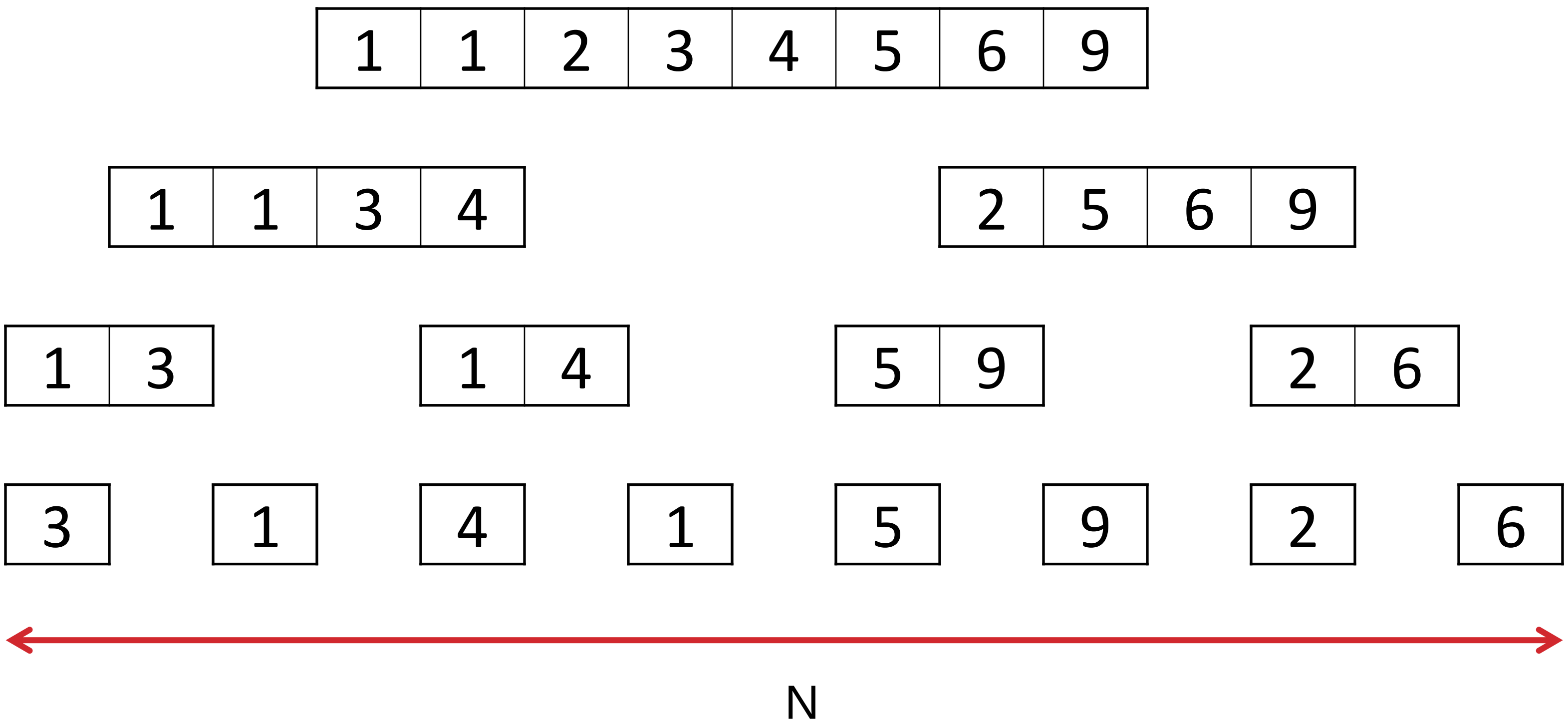
MERGE SORT ANALYSIS

To merge N=8 values takes $\log_2 N$=3 levels of merging



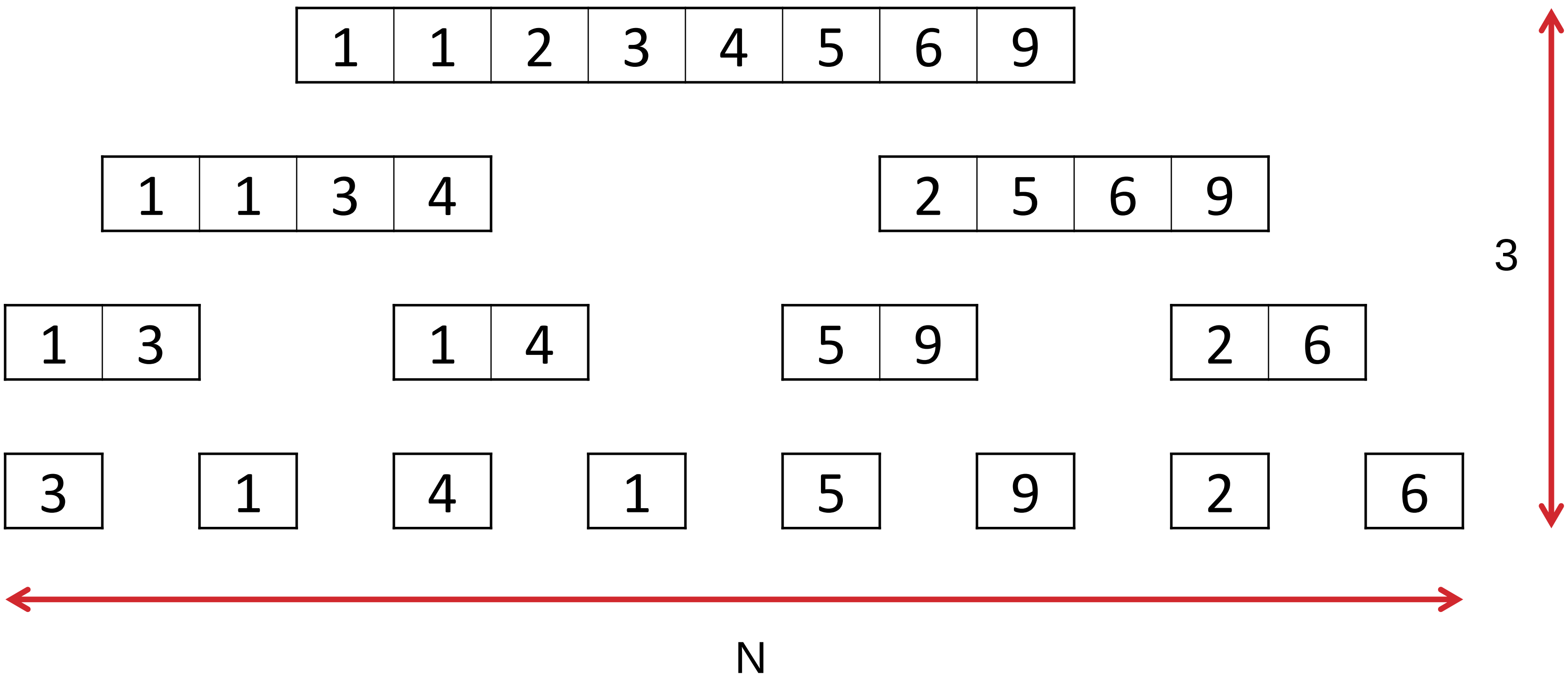
MERGE SORT ANALYSIS

Each merge step processes all N values



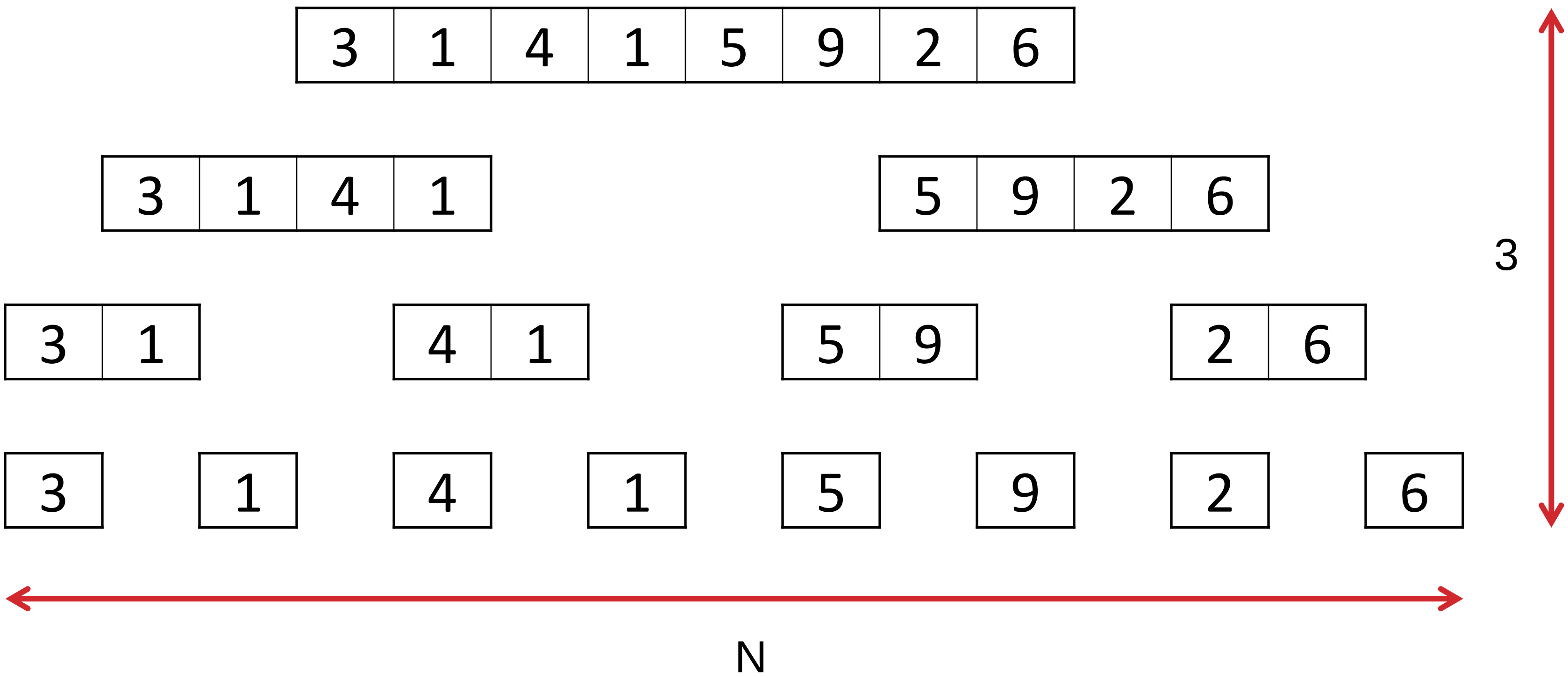
MERGE SORT ANALYSIS

Total work **merging** **sorted arrays is** **$O(N \log_2 N)$**



MERGE SORT ANALYSIS

Similarly, total work splitting unsorted arrays is O(N $\log_2$N)



MERGE SORT ANALYSIS

- The slitting and merging phases are both $O(N \log_2 N)$
- Hence, the merge sort algorithm is $O(N \log_2 N)$
- This is a tremendous speed improvement over $O(N^2)$

$O(N)$	$O(N \log_2 N)$	$O(N^2)$
10	33	100
100	664	10,000
1,000	9,966	1,000,000
10,000	132,877	100,000,000
100,000	1,660,964	10,000,000,000
1,000,000	19,931,569	1,000,000,000,000

MERGE SORT ANALYSIS

- **Let S(N) be amount of work to sort N values**
  - S(1) = 1 - a single data value
  - S(N) = 2 * S(N/2) + N - 2 recursive sorts and merge

- **Substituting the recurrence relationship into itself**
  - S(N) = 2 * S(N/2) + N
  - S(N) = 2 * (2 * S(N/4) + N/2) + N
  - S(N) = 4 * S(N/4) + 2 * N
  - S(N) = 4 * (2 * S(N/8) + N/4) + 2 * N
  - S(N) = 8 * S(N/8) + 3 * N
  - …

MERGE SORT ANALYSIS

- **Notice the pattern?**
  - S(N) = $\boxed{2}$ * S(N/2)   + N
  - S(N) = $\boxed{4}$ * S(N/4)   + 2 * N
  - S(N) = $\boxed{8}$ * S(N/8)   + 3 * N
  - S(N) = $\boxed{16}$ * S(N/16) + 4 * N
  - ...

These values are powers of 2

MERGE SORT ANALYSIS

- **Notice the pattern?**
  - $S(N) = 2 * S(N/2) + N$
  - $S(N) = 4 * S(N/4) + 2 * N$
  - $S(N) = 8 * S(N/8) + 3 * N$
  - $S(N) = 16 * S(N/16) + 4 * N$
  - …

These are the powers

- **If we let** **k be the power**, **the recurrence formula becomes**
  - $S(N) = 2^k * S(N/2^k) + k * N$

MERGE SORT ANALYSIS

- **Assume that $N = 2^k$ where $k = \log_2 N$**
- **Substituting for 2^k and k in the recurrence formula we get**
  - $S(N) = 2^k * S(N/2^k) + k * N$
  - $S(N) = N * S(N/N) + \log_2 N * N$
  - $S(N) = N * S(1) + \log_2 N * N$
  - $S(N) = N * 1 + \log_2 N * N$
  - $S(N) = N + N \log_2 N$

- **Since N is smaller than $N \log_2 N$ we can ignore this term**
- **Hence the merge sort algorithm is $O(N \log_2 N)$**

MERGE SORT ANALYSIS

- **What happens if the input array is already sorted?**
  - The splitting is not affected
  - The merging is not affected
  - The algorithm is still $O(N \log_2 N)$

- **The number of splitting and merging steps in this algorithm do not depend on the data values in the array**
  - Best case is $O(N \log_2 N)$
  - Worst case is $O(N \log_2 N)$
  - Average case is $O(N \log_2 N)$

SORTING ALGORITHMS

QUICKSORT

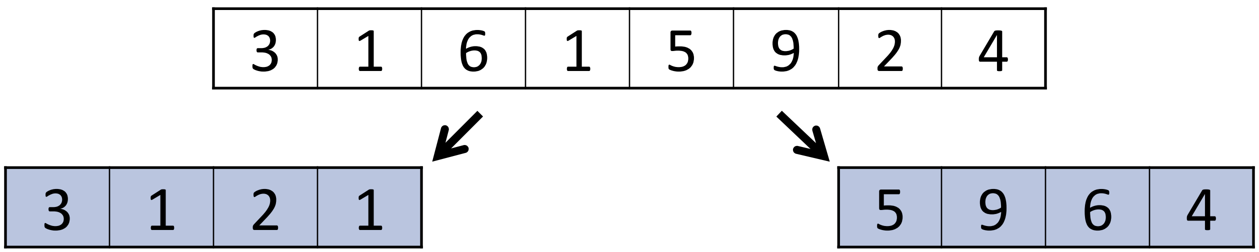
QUICKSORT

- **Quicksort is another "divide and conquer" algorithm that is famous for being fast (hence the name)**
  - It was invented in 1960 by Tony Hoare

- **The key idea is to partition the unsorted array into two parts, sort the two parts, and combine to get sorted result**
  - The really clever idea is to partition the data with small values in one part and large values in the other part
  - This way the combine step takes no work!

QUICKSORT

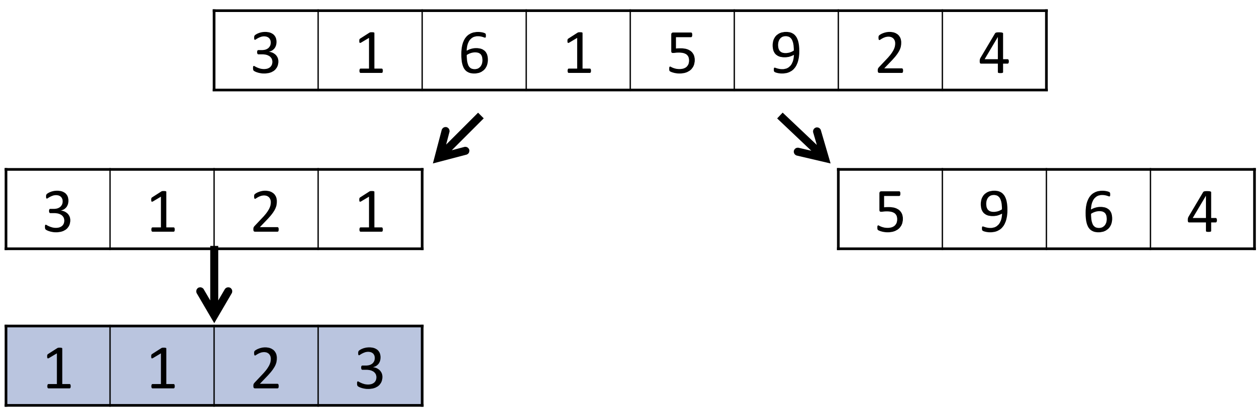
Step 1: Partition unsorted data into two parts



We put all small values on the left and all large values on the right

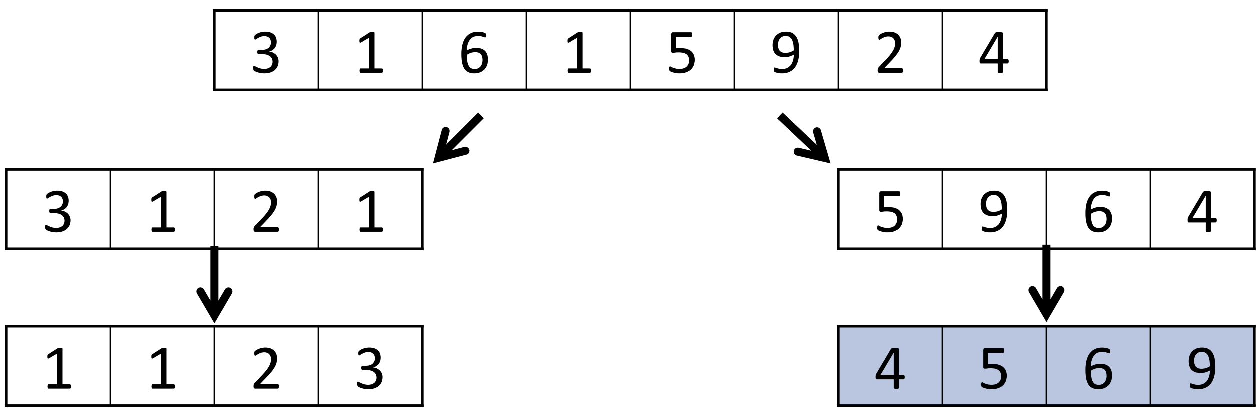
QUICKSORT

Step 2: Sort the small values on the left (recursively)



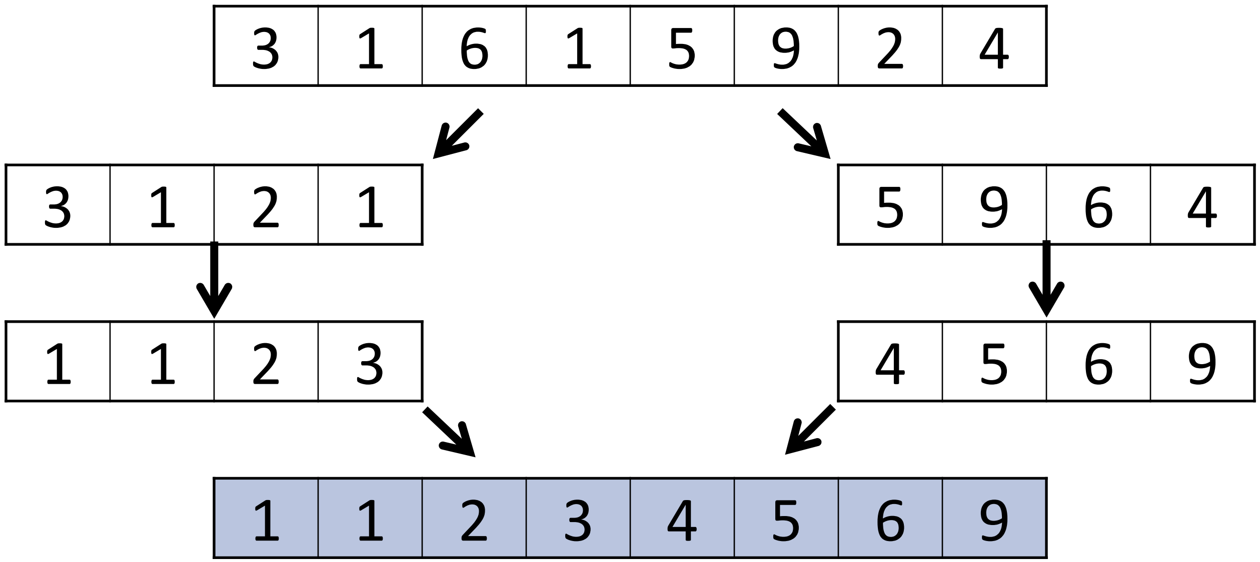
QUICKSORT

Step 3: Sort the large values on the right (recursively)



QUICKSORT

Step 4: Combine the two sorted halves (no work)



QUICKSORT

- **How can we partition the input array so all small values are on the left and all large values are on the right?**
  - Hoare's solution was to select a "pivot value" from array and use this value to decide what is "small" and "large"
  - Simple choice is to use rightmost array location

- **Hoare's partition algorithm:**
  - Scan the unsorted array from left to right until we find a data value that is greater than the pivot
  - Scan the unsorted array from right to left until we find a data value that is less than the pivot
  - Swap these two values, repeat until the L-R and R-L scans cross each other in the middle of the array

QUICKSORT

Step 1: Select rightmost array value as pivot value

3	1	6	1	5	9	2	4

QUICKSORT

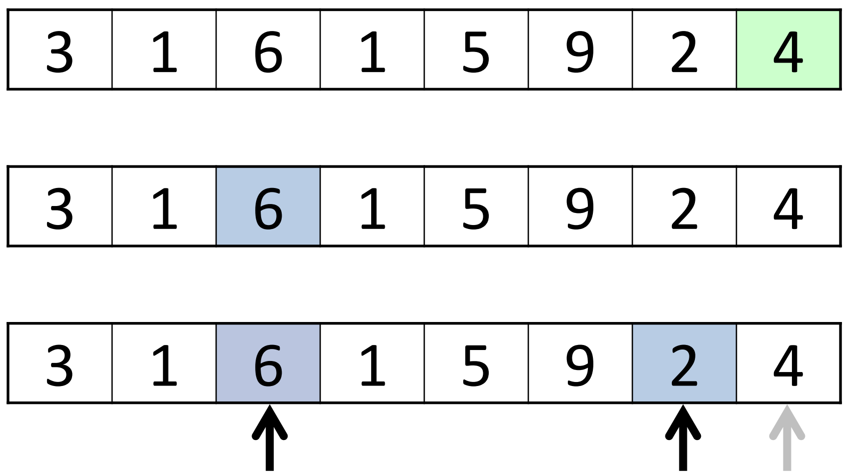
Step 2: Scan L-R to find value greater than pivot value

3	1	6	1	5	9	2	4

3	1	6	1	5	9	2	4

QUICKSORT

Step 3: Scan R-L to find value smaller than pivot value



QUICKSORT

Step 4: Swap the values if left value > right value

3	1	6	1	5	9	2	4

3	1	6	1	5	9	2	4

3	1	6	1	5	9	2	4

3	1	2	1	5	9	6	4

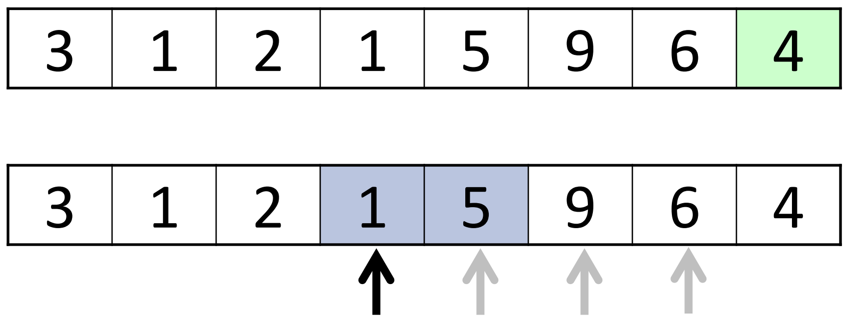
QUICKSORT

Repeat Step 2: Scan L-R to find value greater than pivot

3	1	2	1	5	9	6	4

QUICKSORT

Repeat Step 3: Scan R-L to find value smaller than pivot



QUICKSORT

Repeat Step 4: Swap the values if left value > right value

3	1	2	1	5	9	6	4

3	1	2	1	5	9	6	4

Since the left value < right value
we do NOT swap these values

QUICKSORT

Now we have partitioned the array into two parts

3	1	2	1	5	9	6	4

3	1	2	1	5	9	6	4

All of these values are
less than the pivot value

QUICKSORT

Now we have partitioned the array into two parts

3	1	2	1	5	9	6	4

3	1	2	1	5	9	6	4

All of these values are
greater than or equal
to the pivot value

QUICKSORT

After recursively sorting both halves the array is sorted

3	1	2	1	5	9	6	4

1	1	2	3	4	5	6	9

No data movement is needed
because all small values are
already left of all large values

QUICKSORT

```
void quick_sort(int data[], int low, int high)
{
   // Check terminating condition
   if (low < high)
   {
      // Partition data into two parts
      int mid = 0;
      partition(data, low, high, mid);

      // Recursive calls to sort array
      quick_sort(data, low, mid - 1);
      quick_sort(data, mid + 1, high);
   }
}
```

We call partition to divide the array into two parts

QUICKSORT

```
void quick_sort(int data[], int low, int high)
{
   // Check terminating condition
   if (low < high)
   {
      // Partition data into two parts
      int mid = 0;
      partition(data, low, high, mid);

      // Recursive calls to sort array
      quick_sort(data, low, mid - 1);
      quick_sort(data, mid + 1, high);
   }
}
```

We make two recursive calls to sort the parts of array

QUICKSORT

```
void partition(int data[], int low, int high, int &mid)
{
   // Use data[high] for pivot value
   int pivot = data[high];

   // Partition array into two parts
   int left = low;
   int right = high;
   while (left < right)
   {
      // Scan left to right
      while ((left < right) && (data[left] < pivot))
         left++;
```

First, we do L-R scan to find value >= pivot value

QUICKSORT

```
...
      // Scan right to left
      while ((left < right) && (data[right] >= pivot))
         right--;

      // Swap data values
      int temp = data[left];
      data[left] = data[right];
      data[right] = temp;
   }
   // Swap pivot to mid
   mid = left;
   data[high] = data[mid];
   data[mid] = pivot;
}
```

Next, we do R-L scan to find value < pivot value

QUICKSORT

```
...
      // Scan right to left
      while ((left < right) && (data[right] >= pivot))
         right--;

      // Swap data values
      int temp = data[left];
      data[left] = data[right];
      data[right] = temp;
   }
   // Swap pivot to mid
   mid = left;
   data[high] = data[mid];
   data[mid] = pivot;
}
```

Then we swap the two data values

QUICKSORT

```
...
      // Scan right to left
      while ((left < right) && (data[right] >= pivot))
         right--;

      // Swap data values
      int temp = data[left];
      data[left] = data[right];
      data[right] = temp;
   }
   // Swap pivot to mid
   mid = left;
   data[high] = data[mid];
   data[mid] = pivot;
}
```

Finally we swap pivot value to middle of array

QUICKSORT

Experimental results for random data:

```
Enter number of data values: 100
CPU time = 2.0e-05 sec

Enter number of data values: 1000
CPU time = 0.00025 sec

Enter number of data values: 10000
CPU time = 0.003042 sec

Enter number of data values: 100000
CPU time = 0.034606 sec
```

QUICKSORT

Experimental results for random data:

```
Enter number of data values: 100
CPU time = 2.0e-05 sec

Enter number of data values: 1000
CPU time = 0.00025 sec

Enter number of data values: 10000
CPU time = 0.003042 sec

Enter number of data values: 100000
CPU time = 0.034606 sec
```

This is slightly faster than merge sort (0.046654 sec)

SORTING ALGORITHMS

QUICKSORT ANALYSIS

QUICKSORT ANALYSIS

- **The run time performance of quicksort for random data is very similar to the merge sort algorithm**
  - The input array is partitioned into two arrays N/2 long
  - These arrays are partitioned into four arrays N/4 long
  - These arrays are partitioned into eight arrays N/8 long
  - This partitioning process stops after $\log_2 N$ steps
  - Each partition step must look at N array values
  - Hence quicksort is $O(N \log_2 N)$ for random data

- **In practice quicksort is slightly faster then merge sort because there is less data copying and no merge step**

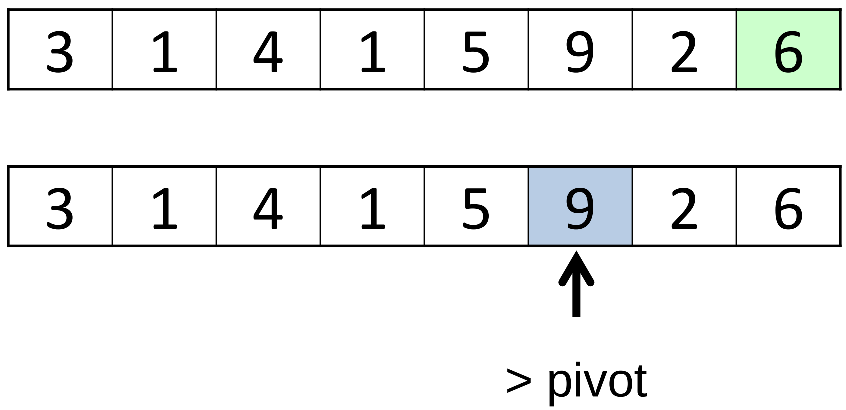
QUICKSORT ANALYSIS

- **Let S(N) be amount of work to sort N random values**
  - $S(1) = 1$
  - $S(N) = 2 * S(N/2) + N$
  - …
  - $S(N) = 2^k * S(N/2^k) + k * N$
  -
- **Assume that $N = 2^k$ where $k = \log_2 N$**
  - $S(N) = N * S(N/N) + \log_2 N * N$
  - $S(N) = N * S(1) + \log_2 N * N$
  - $S(N) = N \log_2 N + N$

- **Hence quicksort is $O(N \log_2 N)$ for random data**

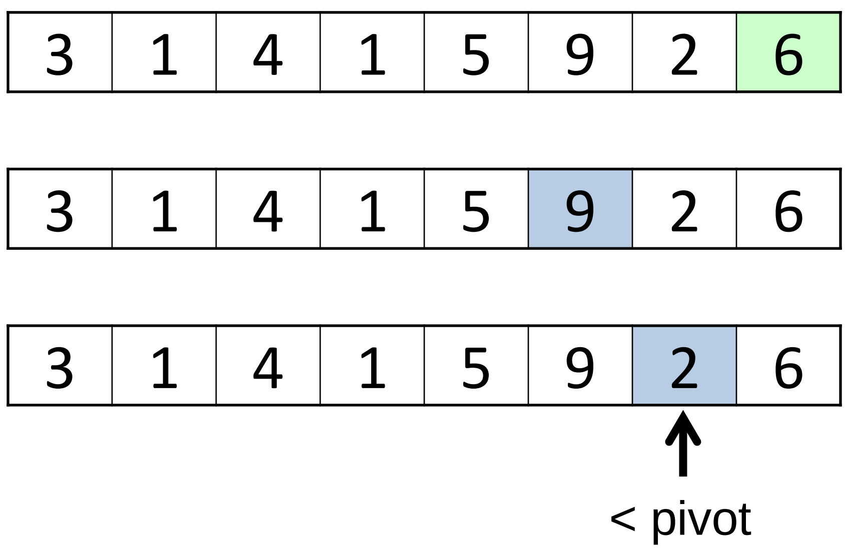
QUICKSORT ANALYSIS

- **Potential problem: What happens if the pivot value is not in the middle of the range of data values?**
  - We will partition the array into two unequal halves



QUICKSORT ANALYSIS

- **Potential problem: What happens if the pivot value is not in the middle of the range of data values?**
  - We will partition the array into two unequal halves



QUICKSORT ANALYSIS

- **Potential problem: What happens if the pivot value is not in the middle of the range of data values?**
  - We will partition the array into two unequal halves

3	1	4	1	5	9	2	6

3	1	4	1	5	9	2	6

3	1	4	1	5	9	2	6

3	1	4	1	5	2	9	6

Swap

QUICKSORT ANALYSIS

- **Potential problem: What happens if the pivot value is not in the middle of the range of data values?**
  - We will partition the array into two unequal halves

3	1	4	1	5	9	2	6

3	1	4	1	5	9	2	6

3	1	4	1	5	9	2	6

3	1	4	1	5	2	9	6

The left is 6 long

QUICKSORT ANALYSIS

- **Potential problem: What happens if the pivot value is not in the middle of the range of data values?**
  - We will partition the array into two unequal halves

3	1	4	1	5	9	2	6

3	1	4	1	5	9	2	6

3	1	4	1	5	9	2	6

3	1	4	1	5	2	9	6

The right is 2 long

QUICKSORT ANALYSIS

- **The worst case for pivot selection happens when the data is already in sorted order**
  - The rightmost value in array will be larger than all others
  - 1st partition will produce arrays N-1 long and 1 long
  - 2nd partition will produce arrays N-2 long and 1 long
  - 3rd partition will produce arrays N-3 long and 1 long
  - This partitioning stops after N steps
  - Each partition step looks at N/2 values on average
  - Hence the worst case for quicksort is an $O(N^2)$

QUICKSORT ANALYSIS

- **Let S(N) be amount of work to sort N sorted values**
  - S(1) = 1
  - S(N) = S(N-1) + S(1) + N
  - S(N) = S(N-2) + 2 * S(1) + N + N-1
  - S(N) = S(N-k) + k * S(1) + N + N-1 + … + N-k-1
- **The partitioning stops when k = N-1**
  - S(N) = S(1) + (N-1) * S(1) + N + N-1 + … + 1
  - S(N) = 1 + (N-1) + (N+1)*N/2
  - S(N) = $N^2/2$ + 3*N/2
- **Hence the worst case for quicksort is O(N^2)**

QUICKSORT

Experimental results for sorted data:

```
Enter number of data values: 100
CPU time = 5.4e-05 sec

Enter number of data values: 1000
CPU time = 0.003985 sec

Enter number of data values: 10000
CPU time = 0.22371 sec

Enter number of data values: 100000
CPU time = 13.6309 sec
```

This is slower than insertion sort (8 sec) and similar to selection sort (14 sec)

QUICKSORT ANALYSIS

- **The selection of quicksort pivots has been widely studied**
  - Robert Sedgewick did his PhD dissertation on this topic
  - He has also written several excellent algorithms books

- **Common pivot choices:**
  - Selecting the last value as pivot is bad for sorted data
  - Selecting the first value as pivot is bad for sorted data
  - Selecting the middle value as pivot is good for sorted data
  - Selecting the median of first, middle, last values is the most expensive choice, but also the most robust
  - See sort.cpp on class website for implementation details

SORTING ALGORITHMS

COUNTING SORT

COUNTING SORT

- **All of the sorting techniques we have discussed so far are general purpose “comparison based” algorithms**
  - These algorithms will work for any data type that can be compared to each other (floats, integers, chars, strings)
  - We rearrange data in the array based on comparisons

- **Counting sort is a “non-comparison based” algorithm that was invented in 1954 by Harold Seward**
  - Instead of comparing elements, we simply count them and use this information to output sorted data
  - This approach works for integers and characters but it does not work for floats or strings

COUNTING SORT

- **The counting sort algorithm has the following steps**
  - Create an array to contain the count information
  - Initialize this count array to all zeros
  - Loop over the data array and increment the counters
  - Loop over the count array to create sorted output

- **We demonstrate counting sort by sorting 30 integers between the values of 0 and 9**
  - We use the first 30 digits of PI just for fun

COUNTING SORT

The unsorted data is shown below

1	4	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

COUNTING SORT

First, we create and initialize the counting array to zeros

1	4	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	0	0	0	0	0	0	0	0	0

COUNTING SORT

Next, we loop over the digit array and increment counters

1	4	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	1	0	0	0	0	0	0	0	0

COUNTING SORT

Next, we loop over the digit array and increment counters

1	4	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	1	0	0	1	0	0	0	0	0

COUNTING SORT

Next, we loop over the digit array and increment counters

1	4	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	0	0	1	0	0	0	0	0

COUNTING SORT

Next, we loop over the digit array and increment counters

1	4	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	0	0	1	1	0	0	0	0

COUNTING SORT

Next, we loop over the digit array and increment counters

1	4	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	0	0	1	1	0	0	0	1

COUNTING SORT

Next, we loop over the digit array and increment counters

1	4	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	1	0	1	1	0	0	0	1

COUNTING SORT

Next, we loop over the digit array and increment counters

1	4	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	1	0	1	1	1	0	0	1

COUNTING SORT

Next we loop over the digit array and increment counters

1	4	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	1	0	1	2	1	0	0	1

COUNTING SORT

Next, we loop over the digit array and increment counters

1	4	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	1	1	1	2	1	0	0	1

COUNTING SORT

After 10 digits we have the following counts

1	4	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	1	1	1	3	1	0	0	1

COUNTING SORT

After 20 digits we have the following counts

1	4	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	2	3	2	3	2	1	2	3

COUNTING SORT

After 30 digits we have the following counts

1	4	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	4	6	3	3	3	2	3	4

COUNTING SORT

Now we can create the sorted output array

1	4	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	4	6	3	3	3	2	3	4

COUNTING SORT

First, we output zero 0's

1	4	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	4	6	3	3	3	2	3	4

COUNTING SORT

Then we output two 1's

1	1	1	5	9	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	4	6	3	3	3	2	3	4

COUNTING SORT

Then we output four 2’s

1	1	2	2	2	2	6	5	3	5
8	9	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	4	6	3	3	3	2	3	4

COUNTING SORT

Then we output six 3's

1	1	2	2	2	2	3	3	3	3
3	3	7	9	3	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	4	6	3	3	3	2	3	4

COUNTING SORT

Then we output three 4's

1	1	2	2	2	2	3	3	3	3
3	3	4	4	4	2	3	8	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	4	6	3	3	3	2	3	4

COUNTING SORT

Then we output three 5's

1	1	2	2	2	2	3	3	3	3
3	3	4	4	4	5	5	5	4	6
2	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	4	6	3	3	3	2	3	4

COUNTING SORT

Then we output three 6's

1	1	2	2	2	2	3	3	3	3
3	3	4	4	4	5	5	5	6	6
6	6	4	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	4	6	3	3	3	2	3	4

COUNTING SORT

Then we output two 7's

1	1	2	2	2	2	3	3	3	3
3	3	4	4	4	5	5	5	6	6
6	7	7	3	3	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	4	6	3	3	3	2	3	4

COUNTING SORT

Then we output three 8's

1	1	2	2	2	2	3	3	3	3
3	3	4	4	4	5	5	5	6	6
6	7	7	8	8	8	3	2	7	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	4	6	3	3	3	2	3	4

COUNTING SORT

Finally, we output four 9's

1	1	2	2	2	2	3	3	3	3
3	3	4	4	4	5	5	5	6	6
6	7	7	8	8	8	9	9	9	9

index	0	1	2	3	4	5	6	7	8	9
count	0	2	4	6	3	3	3	2	3	4

COUNTING SORT

- **How much work was needed for this example?**
  - Create and initialize count array (10 steps)
  - Loop over data array to get counts (30 steps)
  - Loop over count array to use counts (10 steps)
  - Output sorted values in data array (30 steps)

- **To generalize:**
  - Assume the input array is N long
  - Assume the data has a range of M values
  - Total work for counting sort = 2 * N + 2 * M = O(N + M)

COUNTING SORT

- **When should we use counting sort?**
  - When the data values can be counted (int, char)
  - When the value of M is small compared to N
  - Sorting first 30 digits of PI: N=30, M=10
  - Counting sort is excellent in this case

- **When should we not use counting sort?**
  - When the data values can not be counted (float, string)
  - When the value of M is large compared to N
  - Sorting 100 UofA student IDs: N=100, M=1,000,000,000
  - Counting sort is terrible this case

COUNTING SORT

```
void counting_sort(int data[], int low, int high, int range)
{
   // Initialize data count array
   int *datacount = new int[range];
   for (int cindex = 0; cindex < range; cindex++)
      datacount[cindex] = 0;

   // Count number of occurrences of each data value
   for (int dindex = low; dindex <= high; dindex++)
      datacount[data[dindex]]++;

   ...
```

First we initialize the counting array to zeros

COUNTING SORT

```
void counting_sort(int data[], int low, int high, int range)
{
   // Initialize data count array
   int *datacount = new int[range];
   for (int cindex = 0; cindex < range; cindex++)
      datacount[cindex] = 0;

   // Count number of occurrences of each data value
   for (int dindex = low; dindex <= high; dindex++)
      datacount[data[dindex]]++;

   ...
```

Then we loop over input array incrementing the counters

COUNTING SORT

```
void counting_sort(int data[], int low, int high, int range)
{
   // Initialize data count array
   int *datacount = new int[range];
   for (int cindex = 0; cindex < range; cindex++)
      datacount[cindex] = 0;

   // Count number of occurrences of each data value
   for (int dindex = low; dindex <= high; dindex++)
      datacount[data[dindex]]++;

   ...
```

Note: we assume all data values are between [0..range-1] or an array bounds error will occur

COUNTING SORT

```
...

   // Generate output array
   int dindex = low;
   for (int cindex = 0; cindex < range; cindex++)
   {
      for (int index = 0; index < datacount[cindex]; index++)
         data[dindex + index] = cindex;
      dindex += datacount[cindex];
   }
   delete[]datacount;
}
```

Finally we loop over the count array and produce the sorted output array

COUNTING SORT

Experimental results for random data:

```
Enter number of data values: 100
Enter range of data values: 100
CPU time = 1.8e-05 sec

Enter number of data values: 1000
Enter range of data values: 100
CPU time = 4.5e-05 sec

Enter number of data values: 10000
Enter range of data values: 100
CPU time = 0.000319 sec
```

COUNTING SORT

Experimental results for random data:

```
Enter number of data values: 100000
Enter range of data values: 100
CPU time = 0.001492 sec

Enter number of data values: 1000000
Enter range of data values: 100
CPU time = 0.015416 sec

Enter number of data values: 1000000
Enter range of data values: 1000000
CPU time = 0.047364 sec
```

Quicksort takes 0.034606 sec

Increasing the data range makes counting sort run slower

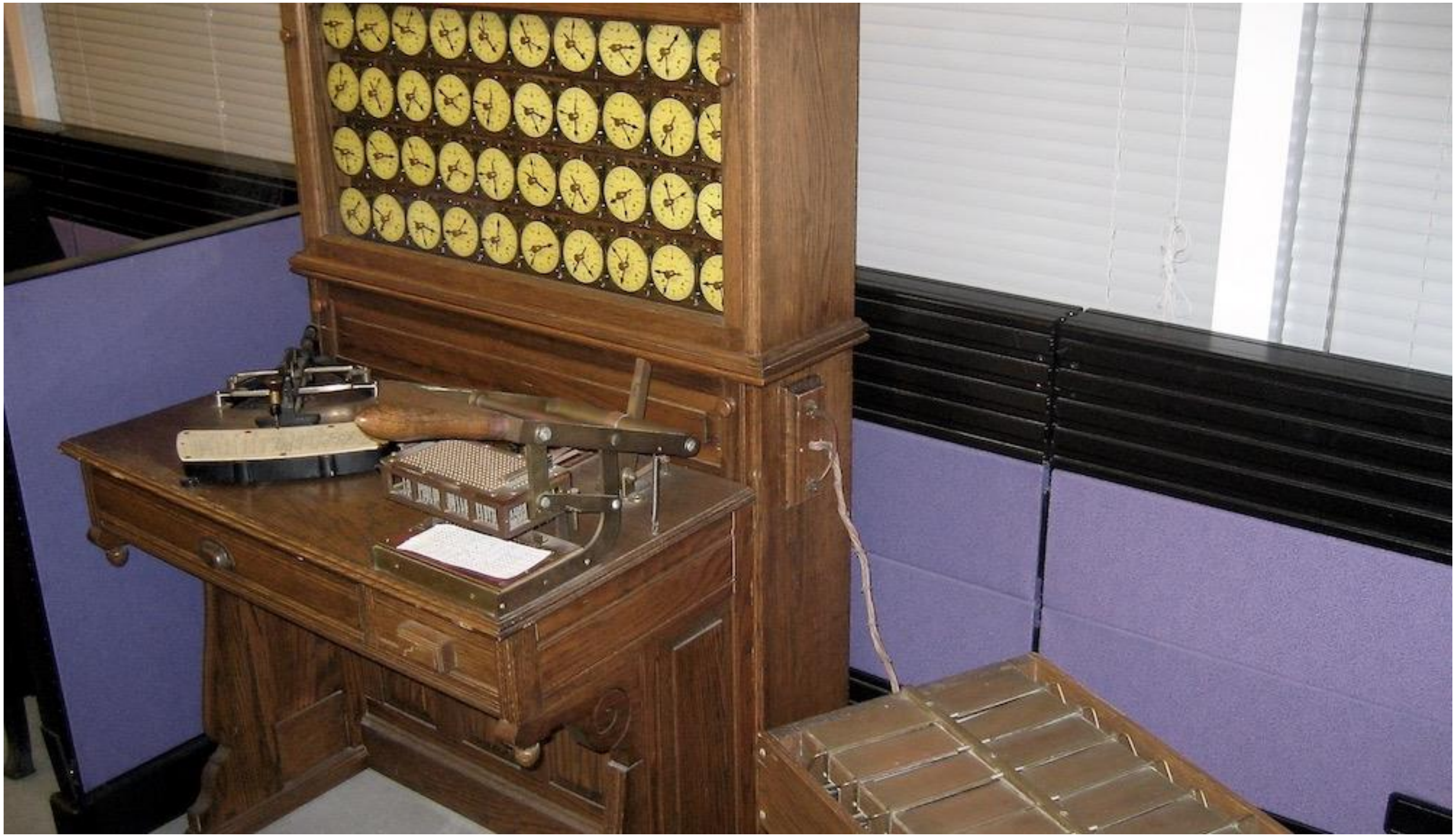
SORTING ALGORITHMS

RADIX SORT

RADIX SORT

- **Radix sort is a "non-comparison based" algorithm that was invented in 1887 by Herman Hollerith**
  - Hollerith used this algorithm in his mechanical tabulating machine to sort punched cards for the 1890 US census
  - The algorithm was implemented in software in 1954 by Herman Seward (who invented counting sort in the process)
  - This sorting algorithm works for all most common data types by processing values one digit or letter at a time
  - The algorithm works for any base (2 for binary, 10 for digits, 26 for letters) so it is called a radix sort

RADIX SORT



Replica of Hollerith's tabulating machine with sorting box (from Wikipedia)

RADIX SORT

IBM card sorting machine that uses radix sort (from Wikipedia)

RADIX SORT

- **The radix sort algorithm has the following steps:**
  - Assume there are N data values with D digits in base R
  - Create R buckets (arrays or linked lists) for storing data values
  - Perform D passes over the data array
  - Each pass will look at one digit of the data value from least significant digit to most significant digit
  - Based on value of digit, move data into corresponding bucket
  - Combine all R buckets after each pass
  - After D passes over the data will be in sorted order

RADIX SORT

Example with eight 3-digit integers

Original	170	045	075	090	002	802	002	066

RADIX SORT

Place data into buckets based on 1's digit

Original	170	045	075	090	002	802	002	066
1's	170	090	002	802	002	045	075	066

The 0 bucket

RADIX SORT

Place data into buckets based on 1's digit

Original	170	045	075	090	002	802	002	066
1's	170	090	002	802	002	045	075	066

The 2 bucket

RADIX SORT

Place data into buckets based on 1’s digit

Original	170	045	075	090	002	802	002	066
1's	170	090	002	802	002	045	075	066

The 5 bucket

RADIX SORT

Place data into buckets based on 1's digit

Original	170	045	075	090	002	802	002	066
1's	170	090	002	802	002	045	075	066

The 6 bucket

RADIX SORT

Place data into buckets based on 10's digit

Original	170	045	075	090	002	802	002	066
1's	170	090	002	802	002	045	075	066
10's	002	802	002	045	066	170	075	090

The 0 bucket

RADIX SORT

Place data into buckets based on 10's digit

Original	170	045	075	090	002	802	002	066
1's	170	090	002	802	002	045	075	066
10's	002	802	002	045	066	170	075	090

The 4 bucket

RADIX SORT

Place data into buckets based on 10’s digit

Original	170	045	075	090	002	802	002	066
1's	170	090	002	802	002	045	075	066
10's	002	802	002	045	066	170	075	090

The 6 bucket

RADIX SORT

Place data into buckets based on 10's digit

Original	170	045	075	090	002	802	002	066
1's	170	090	002	802	002	045	075	066
10's	002	802	002	045	066	170	075	090

The 7 bucket

RADIX SORT

Place data into buckets based on 10's digit

Original	170	045	075	090	002	802	002	066
1's	170	090	002	802	002	045	075	066
10's	002	802	002	045	066	170	075	090

The 9 bucket

RADIX SORT

Place data into buckets based on 100's digit

Original	170	045	075	090	002	802	002	066
1's	170	090	002	802	002	045	075	066
10's	002	802	002	045	066	170	075	090
100's	002	002	045	066	075	090	170	802

The 0 bucket

RADIX SORT

Place data into buckets based on 100’s digit

Original	170	045	075	090	002	802	002	066
1's	170	090	002	802	002	045	075	066
10's	002	802	002	045	066	170	075	090
100's	002	002	045	066	075	090	170	802

The 1 bucket

RADIX SORT

Place data into buckets based on 100’s digit

Original	170	045	075	090	002	802	002	066
1's	170	090	002	802	002	045	075	066
10's	002	802	002	045	066	170	075	090
100's	002	002	045	066	075	090	170	802

The 8 bucket

RADIX SORT

Place data into buckets based on 100's digit

Original	170	045	075	090	002	802	002	066
1's	170	090	002	802	002	045	075	066
10's	002	802	002	045	066	170	075	090
100's	002	002	045	066	075	090	170	802

After 3 passes the input data is now in sorted order

RADIX SORT

- **Essential implementation details:**
  - We can implement buckets using linked lists or arrays
  - For arrays, we must know in advance the size and staring point for each of the R buckets for each pass
  - This can be calculated by one pass over the data that counts the number of times each digit occurs
  - We must maintain the original ordering of data within each bucket by filling buckets from the right
  - We must make ensure that all data is D digits long by padding integers to left and strings to the right

RADIX SORT

- **How much work is done by radix sort?**
  - Assume there are N data values with D digits in base R
  - There are D passes over the array
  - We must move N data values in each pass
  - Hence radix sort is O(N*D)

- **Radix sort is fast when D is small compared to N**
  - Sorting 1000 3-digit integers

- **Radix sort is slow when D is greater than or equal to N**
  - Sorting 3 1000-digit integers

SORTING ALGORITHMS

SUMMARY

SUMMARY

- **In this section, we introduced algorithm analysis for searching and sorting, and the differences between O(logN), O(N), O(N logN), and O(N^2) algorithms**

- **We discussed three O(N^2) sorting techniques:**
  - We described the Selection sort algorithm and its implementation and run time performance
  - We described two versions of the Bubble sort algorithm and compared their implementations
  - We described the Insertion sort algorithm and its implementation and run time performance

SUMMARY

- **We discussed two O(N logN) sorting methods:**
  - We described the recursive merge sort algorithm and its implementation and run time performance
  - We did an analysis of merge sort and demonstrated that this is an O(N logN) algorithm
  - Quicksort is a divide and conquer algorithm that is faster then most other sorting algorithms most of the time
  - We did an analysis of quicksort and demonstrated that this algorithm is O(N logN) on average but O(N^2) in worst case

- **These sorting algorithms demonstrate that slightly more complex algorithms can outperform simple algorithms**

SUMMARY

- **Finally, we described two specialized sorting algorithms**
  - Counting sort is a “non-comparison based” sort that is well suited for sorting large arrays of small integers
  - Radix sort is a “non-comparison based” algorithm that sorts fixed size data one digit at a time using buckets

- **These sorting algorithms have very different best case and worst-case behaviors so we have to be careful when deciding what sorting algorithm to use**

SUMMARY

Algorithm	Best Case	Average Case	Worst Case
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Basic Bubble Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Bubble Sort	**O(N)**	$O(N^2)$	$O(N^2)$
Insertion Sort	**O(N)**	$O(N^2)$	$O(N^2)$
Merge Sort	O(N logN)	O(N logN)	O(N logN)
Quicksort	O(N logN)	O(N logN)	$O(N^2)$
Counting Sort	O(N+M)	O(N+M)	O(N+M)
Radix Sort	O(N*D)	O(N*D)	O(N*D)

The $O(N^2)$ algorithms have relatively slow run times, insertion sort is often the fastest, especially for mostly sorted data

SUMMARY

Algorithm	Best Case	Average Case	Worst Case
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Basic Bubble Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Bubble Sort	$O(N)$	$O(N^2)$	$O(N^2)$
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
Quicksort	$O(N \log N)$	$O(N \log N)$	$\mathbf{O(N^2)}$
Counting Sort	$O(N+M)$	$O(N+M)$	$O(N+M)$
Radix Sort	$O(N*D)$	$O(N*D)$	$O(N*D)$

The O(N logN) algorithms have similar run times, but quicksort is generally the fastest, except when the input data is sorted

SUMMARY

Algorithm	Best Case	Average Case	Worst Case
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Basic Bubble Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Bubble Sort	O(N)	$O(N^2)$	$O(N^2)$
Insertion Sort	O(N)	$O(N^2)$	$O(N^2)$
Merge Sort	O(N logN)	O(N logN)	O(N logN)
Quicksort	O(N logN)	O(N logN)	$O(N^2)$
Counting Sort	O(N+M)	O(N+M)	O(N+M)
Radix Sort	O(N*D)	O(N*D)	O(N*D)

The non-comparison based algorithms can be faster than all other sort algorithms, but they only work for limited data types

SUMMARY

Algorithm	History
Selection Sort	Unknown*
Basic Bubble Sort	Unknown*
Bubble Sort	Unknown*
Insertion Sort	Unknown*
Merge Sort	Invented 1945 by John von Neumann
Quicksort	Invented 1960 by Tony Hoare
Counting Sort	Invented 1954 by Harold Seward
Radix Sort	Invented 1887 by Herman Hollerith

* Because no one wants to take credit for $O(N^2)$ sort algorithms