ARRAYS S0RTING

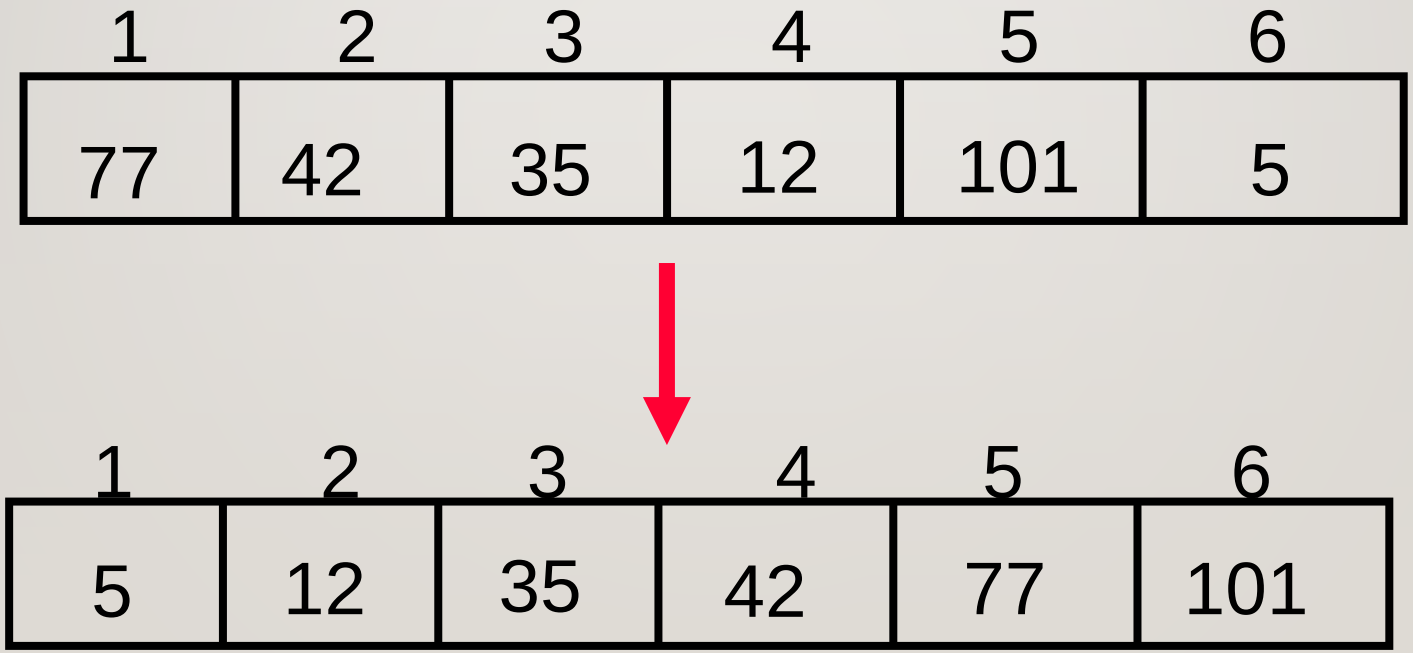
NATIONAL UNIVERSITY OF TECHNOLOGY (NUTECH)

DR. SAMAN RIAZ

LECTURE # 5

SORTING

- **Sorting takes an unordered collection and makes it an ordered one (default Ascending).**



1	2	3	4	5	6
77	42	35	12	101	5

1	2	3	4	5	6
5	12	35	42	77	101

SORTING

- To arrange a set of items in sequence.
- It is estimated that 25~50% of all computing power is used for sorting activities.
- Possible reasons:
  - Many applications require sorting;
  - Many applications use inefficient sorting algorithms.

SORTING APPLICATIONS

- To prepare a list of student ID, names, and scores in a table (sorted by ID or name) for easy checking.
- To prepare a list of scores before letter grade assignment.
- To produce a list of horses after a race (sorted by the finishing times) for payoff calculation.
- To prepare an originally unsorted array for ordered binary searching.

SOME SORTING METHODS

- Bubble sort
- Selection sort
- Insertion sort
- Merge sort
- Quick sort (a very efficient sorting method for most applications)

BUBBLE SORT

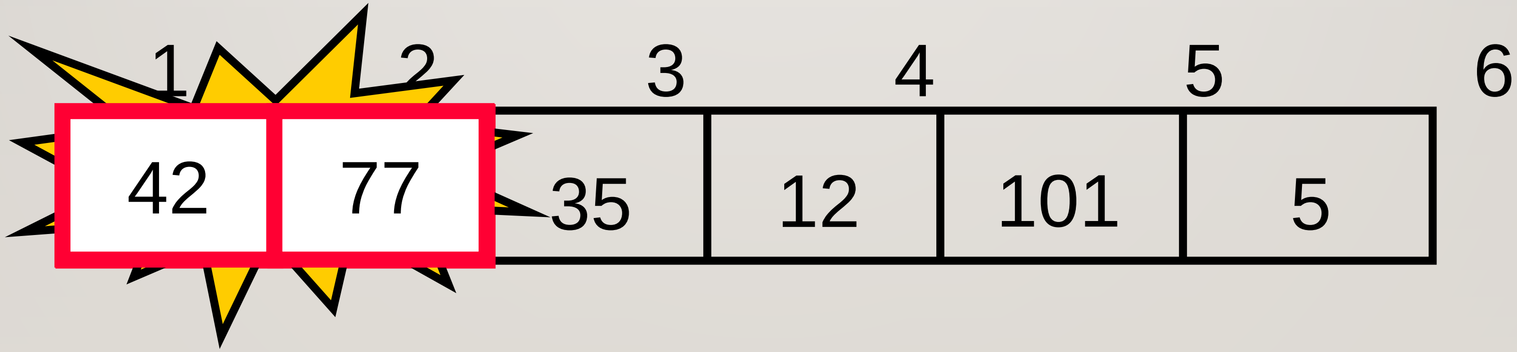
"BUBBLING UP" THE LARGEST ELEMENT

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping

1	2	3	4	5	6
77	42	35	12	101	5

"BUBBLING UP" THE LARGEST ELEMENT

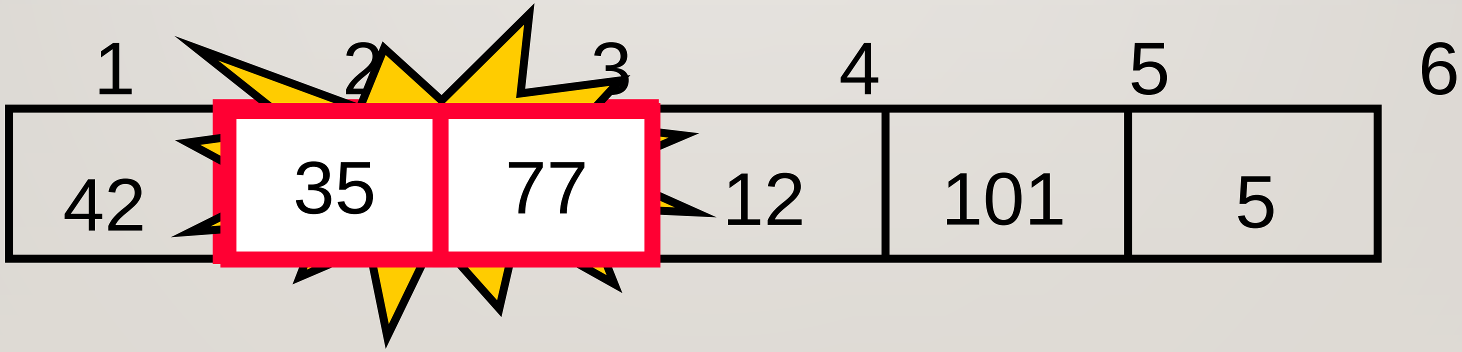
- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



1	2	3	4	5	6
42	77	35	12	101	5

"BUBBLING UP" THE LARGEST ELEMENT

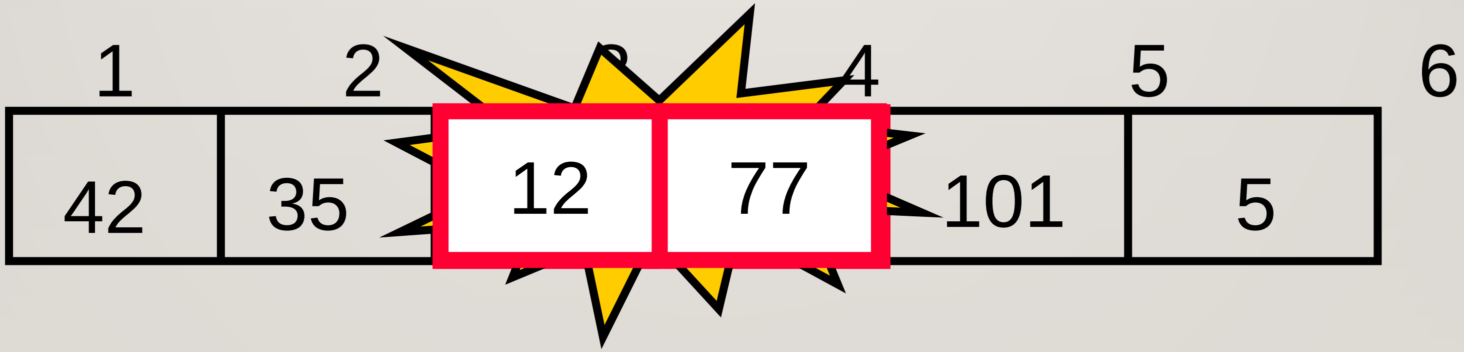
- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



1	2	3	4	5	6
42	35	77	12	101	5

"BUBBLING UP" THE LARGEST ELEMENT

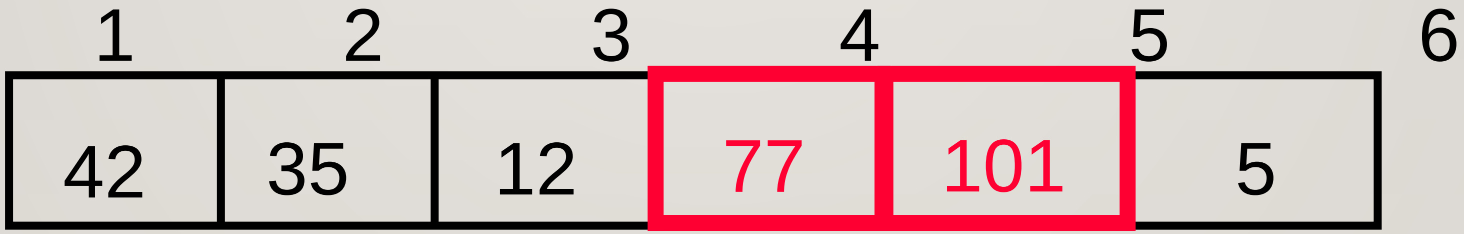
- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



1	2	3	4	5	6
42	35	12	77	101	5

"BUBBLING UP" THE LARGEST ELEMENT

- **Traverse a collection of elements**
  - **Move from the front to the end**
  - **“Bubble” the largest value to the end using pair-wise comparisons and swapping**

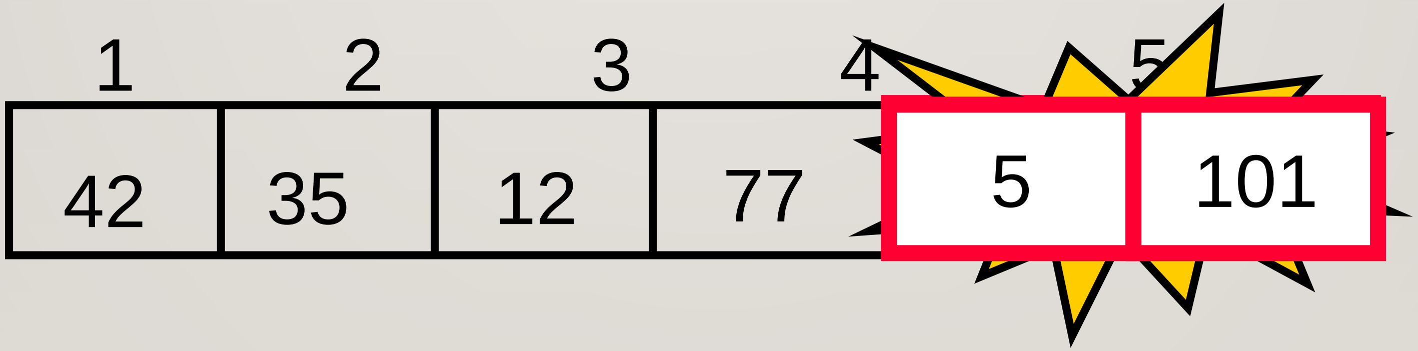


1	2	3	4	5	6
42	35	12	77	101	5

No need to swap

"BUBBLING UP" THE LARGEST ELEMENT

- **Traverse a collection of elements**
  - **Move from the front to the end**
  - **“Bubble” the largest value to the end using pair-wise comparisons and swapping**



1	2	3	4	5	6
42	35	12	77	5	101

"BUBBLING UP" THE LARGEST ELEMENT

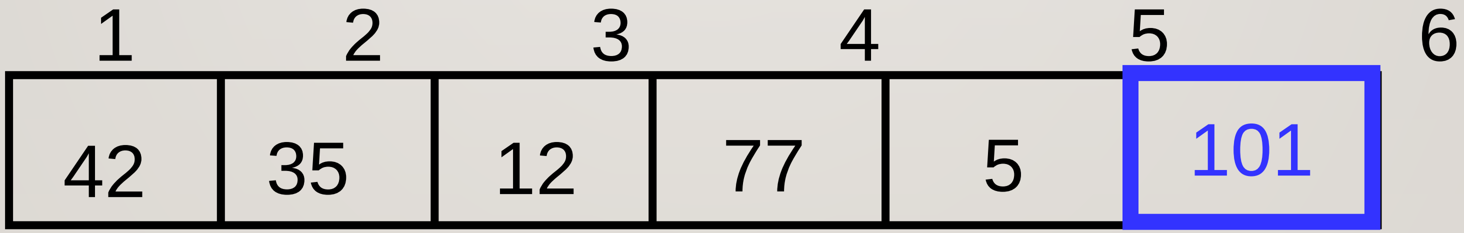
- **Traverse a collection of elements**
  - **Move from the front to the end**
  - **“Bubble” the largest value to the end using pair-wise comparisons and swapping**

1	2	3	4	5	6
42	35	12	77	5	101

Largest value correctly placed

ITEMS OF INTEREST

- **Notice that only the largest value is correctly placed**
- **All other values are still out of order**
- **So we need to repeat this process**



1	2	3	4	5	6
42	35	12	77	5	101

Largest value correctly placed

REPEAT "BUBBLE UP" HOW MANY TIMES?

- **If we have N elements…**
- **And if each time we bubble an element, we place it in its correct location…**
- **Then we repeat the "bubble up" process N – 1 times. HOW?**
- **This guarantees we'll correctly place all N elements.**

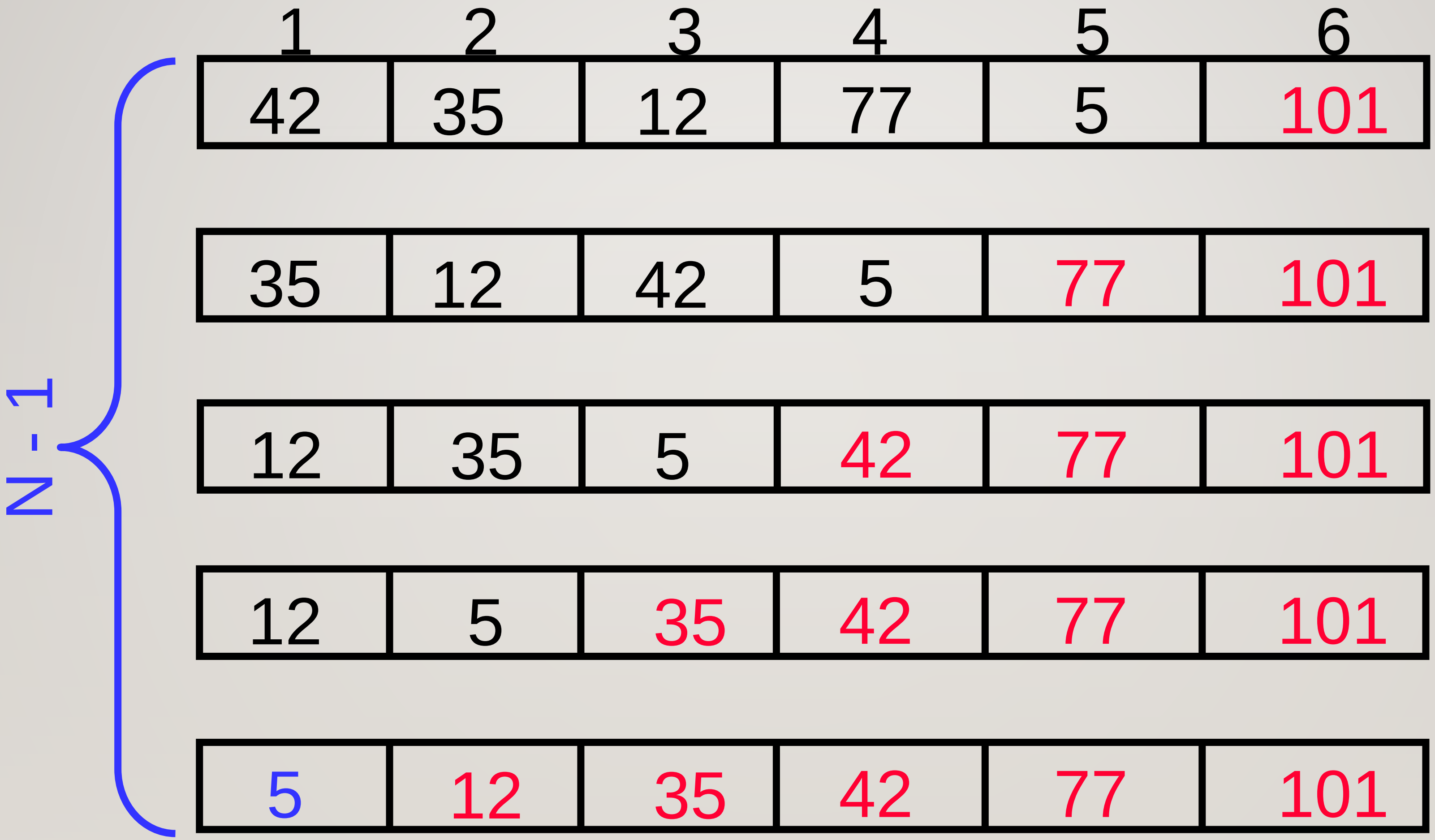
BUBBLE SORT

```
void BubbleSort(int a[], const int ARRAY_SIZE)
{
    for(int pass = 1; pass < ARRAY_SIZE; pass++)// N – 1 passes
    {
        for(int i = 0; i < ARRAY_SIZE - pass; i++)//0–>(SIZE-PASS) steps
        {
            if (a[i] > a[i+1]) // swap
            {
                int tmp = a[i];
                a[i] = a[i+1];
                a[i+1] = tmp;
            }
        }
    }
}
```

“Bubbling” All the Elements

1	2	3	4	5	6
77	42	35	12	101	5

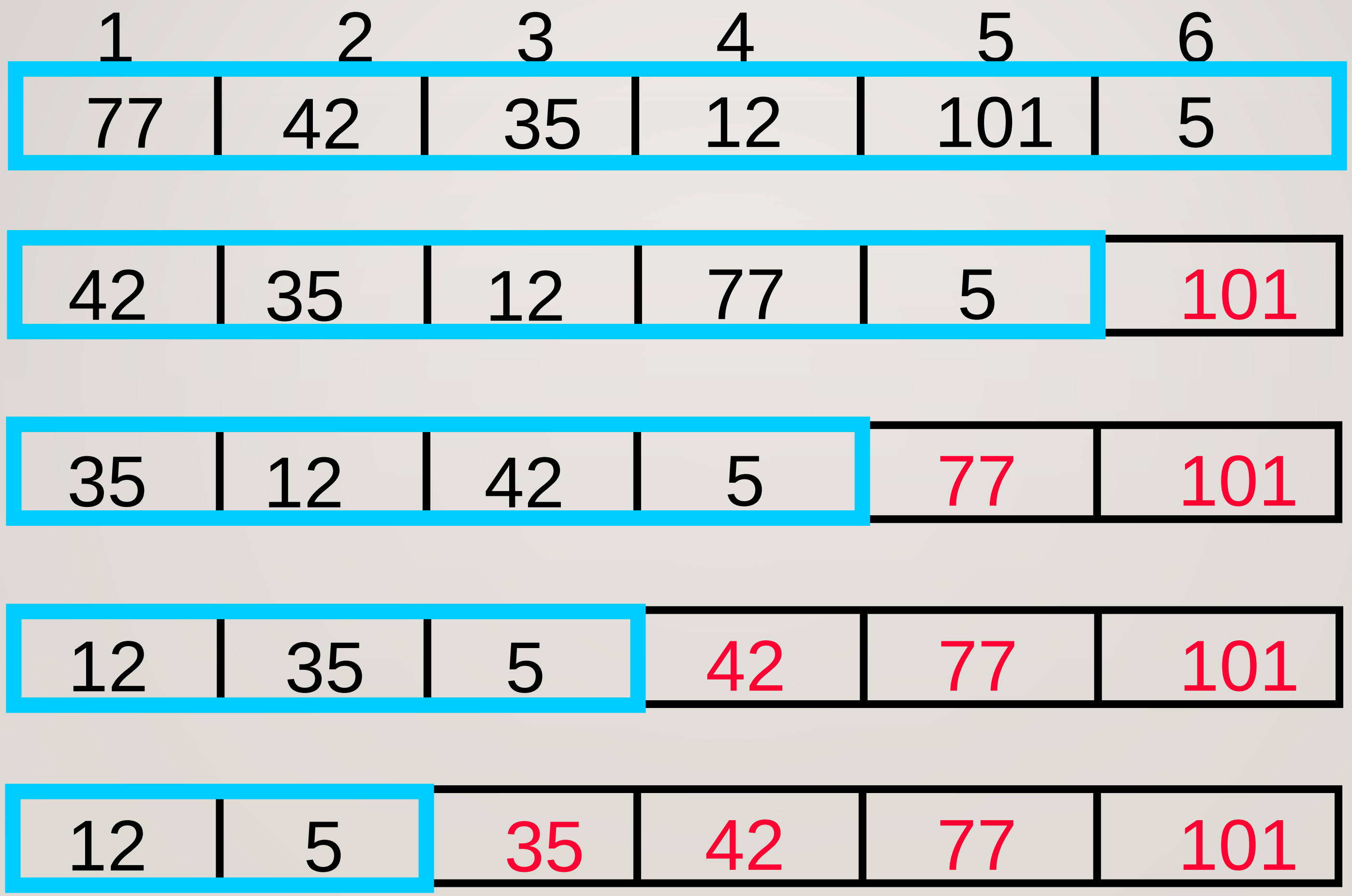
“BUBBLING” ALL THE ELEMENTS



1	2	3	4	5	6
42	35	12	77	5	101
35	12	42	5	77	101
12	35	5	42	77	101
12	5	35	42	77	101
5	12	35	42	77	101

N - 1

REDUCING THE NUMBER OF COMPARISONS

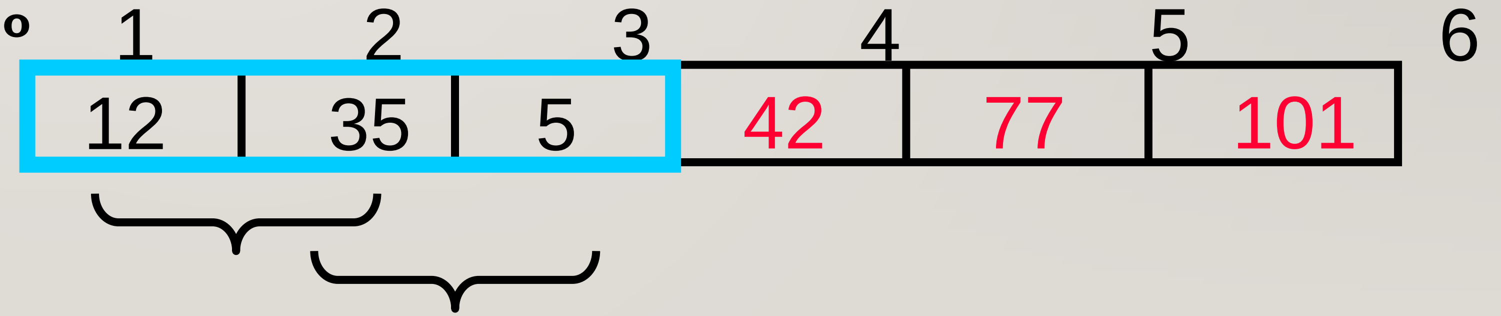


1	2	3	4	5	6
77	42	35	12	101	5
42	35	12	77	5	101
35	12	42	5	77	101
12	35	5	42	77	101
12	5	35	42	77	101

REDUCING THE NUMBER OF COMPARISONS

- **On the N^th^ "bubble up", we only need to do MAX - N comparisons.**

- **For example:**
  - **This is the 4th "bubble up"**
  - **MAX is 6**
  - **Thus we have 2 comparisons to do**



1	2	3	4	5	6
12	35	5	42	77	101

ALREADY SORTED COLLECTIONS?

- **What if the collection was already sorted?**
- **What if only a few elements were out of place and after a couple of "bubble ups," the collection was sorted?**

- **We want to be able to detect this and "stop early"!**

1	2	3	4	5	6
5	12	35	42	77	101

USING A BOOLEAN “FLAG”

- **We can use a boolean variable to determine if any swapping occurred during the “bubble up”**

- **If no swapping occurred, then we know that the collection is already sorted!**

- **This boolean “flag” needs to be reset after each “bubble up.”**

BUBBLE SORT

```
int pass = 1;
boolean exchanges;
do {
  exchanges = false;
  for (int i = 0; i < ARRAY_SIZE-pass; i++)
    if (a[i] > a[i+1]) {
      T tmp = a[i];
      a[i] = a[i+1];
      a[i+1] = tmp;
      exchanges = true;
    }
pass++;
} while (exchanges);
```

SELECTION SORT

SELECTION SORT

- Selection sort performs sorting by repeatedly putting the largest element in the unsorted portion of the array to the end of this unsorted portion until the whole array is sorted.

- It is similar to the way that many people do their sorting.

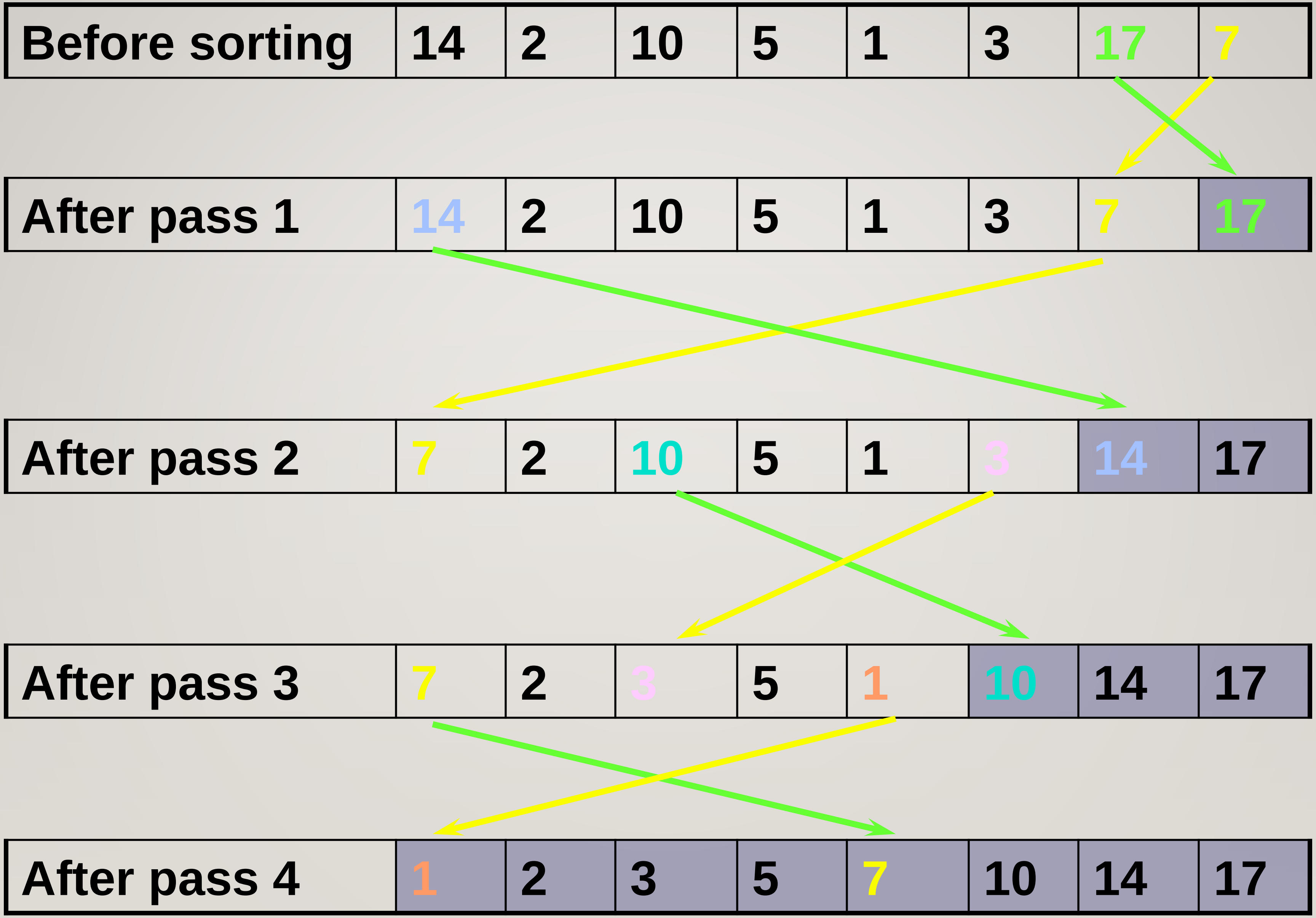
SELECTION SORT

- Algorithm
  1. Define the entire array as the unsorted portion of the array
  2. While the unsorted portion of the array has more than one element:

     ⇒ Find its largest element.

     ⇒ Swap with last element (assuming their values are different).

     ⇒ Reduce the size of the unsorted portion of the array by 1.



Before sorting	14	2	10	5	1	3	17	7
After pass 1	14	2	10	5	1	3	7	17
After pass 2	7	2	10	5	1	3	14	17
After pass 3	7	2	3	5	1	10	14	17
After pass 4	1	2	3	5	7	10	14	17

```
// Sort array of integers in ascending order
void select(int data[], // in/output: array
            int size){  // input: array size
  int temp;       // for swap
  int max_index;  // index of max value
  for (int rightmost=size-1; rightmost>0; rightmost--){
  //find the largest item in the unsorted portion
  //rightmost is the end point of the unsorted part of array
          max_index = 0; //points the largest element
          for ( int current=1; current<=rightmost; current++)
                    if (data[current] > data[max_index])
                              max_index = current;
          //swap the largest item with last item if necessary
          if (data[max_index] > data[rightmost]){
                    temp = data[max_index];   // swap
                    data[max_index] = data[rightmost];
                    data[rightmost] = temp;
          }
  }
}
```

SELECTION SORT VS. BUBBLE SORT

- The bubble sort is inefficient for large arrays because items only move by one element at a time.
- The selection sort moves items immediately to their final position in the array so it makes fewer exchanges.

INSERTION SORT

INSERTION SORT

- Insertion sort is a simple sorting algorithm that is appropriate for small inputs.
  - Most common sorting technique used by card players.
- The list is divided into two parts: sorted and unsorted.
- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted at the appropriate place.
- A list of *n* elements will take at most *n-1* passes to sort the data.

Sorted **Unsorted**

23	78	45	8	32	56	Original List
23	78	45	8	32	56	After pass 1
23	45	78	8	32	56	After pass 2
8	23	45	78	32	56	After pass 3
8	23	32	45	78	56	After pass 4
8	23	32	45	56	78	After pass 5

INSERTION SORT ALGORITHM

```
template <class Item>
void insertionSort(Item a[], int n)
{
   for (int i = 1; i < n; i++)
   {
      Item tmp = a[i];

      for (int j=i; j>0 && tmp < a[j-1]; j--)
         a[j] = a[j-1];
      a[j] = tmp;
   }
}
```