Data Structures

6. Array Sorting

Sorting

- Sorting takes an unordered collection and makes its an ordered one

- Let A be a list of n elements $A_1,\ A_2,\ \ldots\ A_n$ in memory
- Sorting A refers to the operation of rearranging the contents of A
  - Ascending order (numerically or lexicographically)
  - Descending order (numerically or lexicographically)

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 – Example 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

Sorting Algorithms

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

Bubble Sort

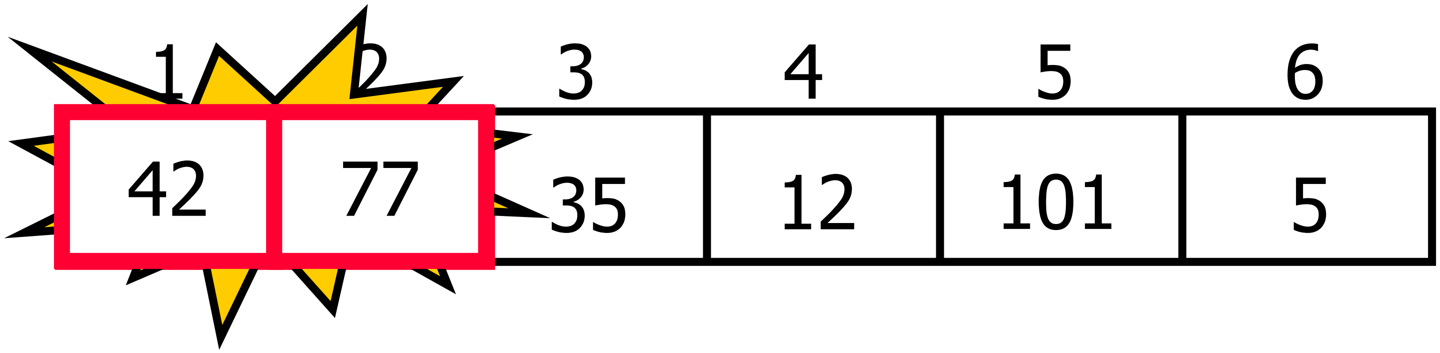
Idea: Bubbling Up the Largest Element

- Traverse a collection of elements
- Move from the front to the end
- "Bubble" the largest value to the end using the operations
  - Pair-wise comparison
  - Swapping

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

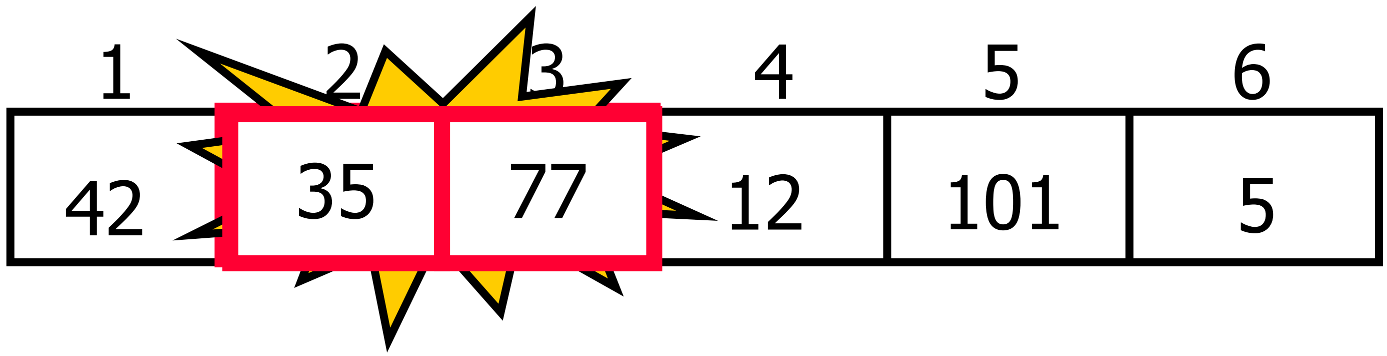
Idea: Bubbling Up the Largest Element

- Traverse a collection of elements
- Move from the front to the end
- "Bubble" the largest value to the end using the operations
  - Pair-wise comparison
  - Swapping



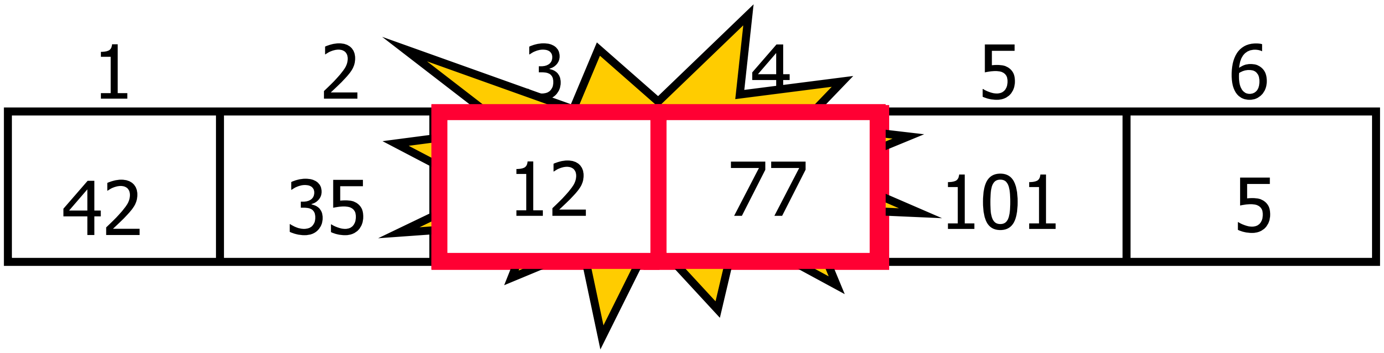
Idea: Bubbling Up the Largest Element

- Traverse a collection of elements
- Move from the front to the end
- "Bubble" the largest value to the end using the operations
  - Pair-wise comparison
  - Swapping



Idea: 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 the operations
  - Pair-wise comparison
  - Swapping



Idea: Bubbling Up the Largest Element

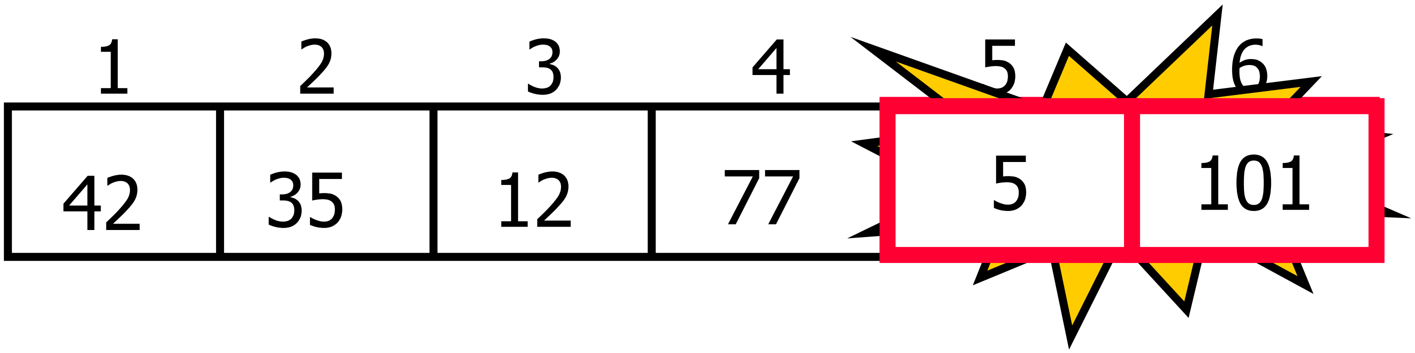
- Traverse a collection of elements
- Move from the front to the end
- "Bubble" the largest value to the end using the operations
  - Pair-wise comparison
  - Swapping

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

No need to swap

Idea: 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 the operations
  - Pair-wise comparison
  - Swapping



Idea: 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 the operations
  - Pair-wise comparison
  - Swapping

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

Largest value correctly placed

Repeat "Bubble Up"

- Notice that only the largest value is correctly placed
- All other values are still out of order
- So we need to repeat this process ,i.e., repeat "bubble up"

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

Largest value correctly placed

How Many Times to Repeat "Bubble Up"

- Each time we bubble an element, we place it in its correct location
- If we have n elements…
  – Then we repeat the "bubble up" process n – 1 times
  – This guarantees all n elements are correctly placed
- Why?

“Bubbling” All the Elements

N - 1

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

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

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

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

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

Reducing Number of Comparisons

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

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

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

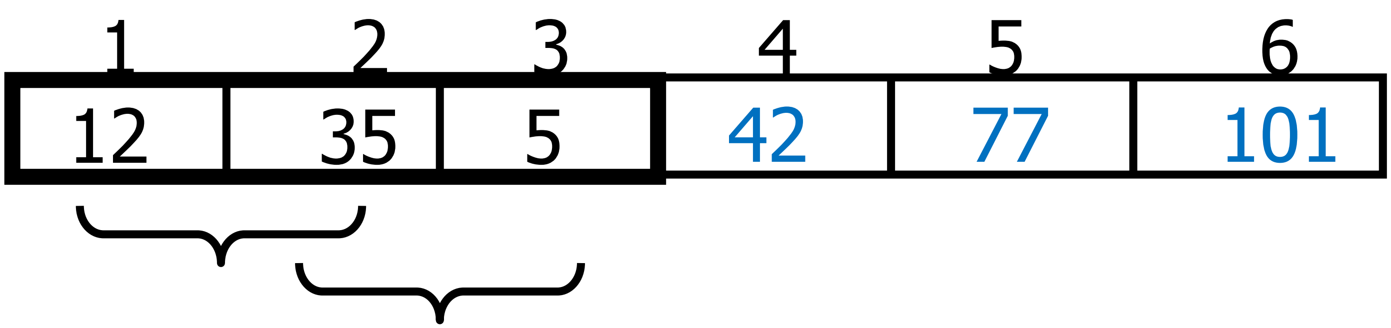
1	2	3	4	5	6
12	35	5	42	77	101

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

Reducing Number of Comparisons

- On the kth "bubble up", we only need to do SIZE - k comparisons

- Example
  - This is the 4th "bubble up"
  - SIZE of the array is 6
  - Thus we have 2 comparisons to do



Bubble Sort Algorithm

```
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;
         }
      }
  }
}
```

Already Sorted Elements

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

```
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

- Define the entire array as the unsorted portion of the array

- 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.

Selection Sort - Example

Original Array	14	2	10	5	1	3	**17**	7
Pass 1	**14**	2	10	5	1	3	7	17
Pass 2	7	2	**10**	5	1	3	14	17
Pass 3	**7**	2	3	5	1	10	14	17
Pass 4	1	2	3	5	7	10	14	17

Selection Sort Algorithm

```
// Sort array of integers in ascending order
void select(int data[], int 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
  - Items only move by one element at a time

- The selection sort moves items immediately to their final position
  - Makes fewer exchanges

Insertion Sort

Insertion Sort

- The list is divided into two parts: sorted and unsorted
- In each pass, the following steps are performed
  - First element of the unsorted part (i.e., sub-list) is picked up
  - Transferred to the sorted sub-list
  - Inserted at the appropriate place
- A list of n elements will take at most n-1 passes to sort the data

Insertion Sort – Example

Sorted Array Unsorted Array

23	78	45	8	32	56	Original Array
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) {
    int i, j;
    for ( i = 1; i < n; i++) {
        Item tmp = a[i];

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

Selection Sort vs. Insertion Sort

- Insertion sort will perform less comparisons than selection sort, depending on the degree of "sortedness" of the array
  - Selection sort must scan the remaining unsorted part of the array when placing an element
  - Insertion sort only scans as many elements as necessary
  - When the array is already sorted or almost sorted, insertion sort performs in O(n) time

- Number of swaps by Selection sort is in O(n), while in insertion sort it is in $O(n^2)$

Comparison of Sorting Algorithms

Algorithm	Number of Comparisons	Number of Swaps
Bubble sort	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{4} = O(n^2)$
Selection sort	$\frac{n(n-1)}{2} = O(n^2)$	$3(n-1) = O(n)$
Insertion sort	$\frac{1}{4}n^2 + O(n) = O(n^2)$	$\frac{1}{4}n^2 + O(n) = O(n^2)$

Any Question So Far?