

Lecture 8

Simple Sorting

Bubble Sort, Selection Sort and Insertion Sort

Last time:

We looked at Queue:

- A limited data structure that has two major operations, enqueue and dequeue.
- Enqueue inserts a new element into the back of the queue and dequeue **gets and removes** the first element from the queue.
- Enqueue and dequeue's time complexity is $O(1)$.
- Similar to Stack, it can also be used in applications as an aiding tool for a programmer.
- Java's LinkedList and ArrayDeque provide a few methods that enable a programmer to use them as Queue.
- Example usage of a queue is serving people in line or queue in a coffee shop or printing jobs in a printer.

Let's shift gears for a while!

So far, we have seen some data structures: arrays, ArrayList, LinkedList, Stack and Queue.

As we saw in lecture 3, suppose that I put all of your names (first name and last name) and Andrew id or student id in an array to have my roster.

Initially, I put them into the array randomly.

Now, it is time to grade your first homework. Oops! Well, I want to sort you by student id or Andrew id.

The question is **"how would you do it?"**

A computer program is NOT able to see the big picture. Thus, it needs to concentrate on the details and follow preset rules.

The three sorting algorithms in this lecture involve two simple steps executed over and over till the items are sorted:

- Compare two items
- Swap the two items, or copy one item or items.

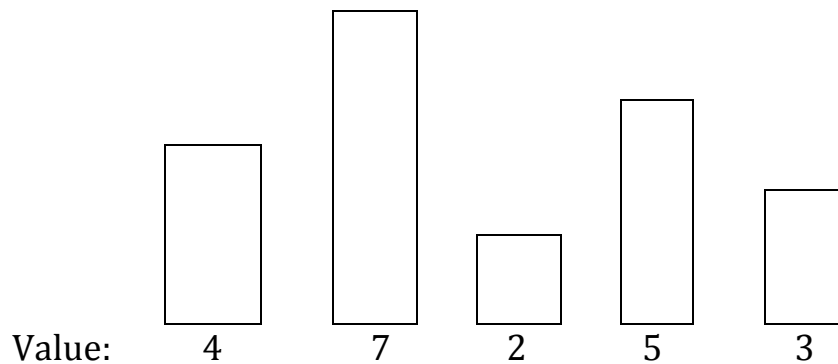
Bubble Sort: Very slow but simple

Step 1: Conceptual View

There are three basic steps in bubble sort.

- Compare two values at a time.
- If the one on the left is bigger, swap them to ***BUBBLE UP the bigger value to the right.***
- Move one position right.

Initial state



After first round

Number of comparisons? _____

Number of swaps? _____

After second round

Number of comparisons? _____

Number of swaps? _____

Now, do you see a pattern?

After third round

Number of comparisons? _____

Number of swaps? _____

After fourth round

Number of comparisons? _____

Number of swaps? _____

Step 2: Implementation View

```
int[] data = {4,7,2,5,3};
```

Swap method

```
// a helper method that swaps two values in an array
private void swap(int[] data, int one, int two) {
    int temp = data[one];
    data[one] = data[two];
    data[two] = temp;
}
```

Comparison method for one round

```
// forward from 0 to the end of the array
for(int in=0; in < data.length-1; in++) {
    if(data[in]>data[in+1]) { // if the left value is bigger
        swap(data, in, in+1); // then swap
    }
}
```

Multiple rounds

```
// backward from the last index till _____
for(int out=data.length-1; out>=1; out--) {
    for(int in=0; in < out; in++) {
        if(data[in]>data[in+1]) { // if left value is bigger
            swap(data, in, in+1);
        }
    }
}
```

Finally, putting the pieces all together

```
public void bubbleSort(int[] data) {
    for(int out=_____; out>=_____; out--) {
        for(int in=0; in < _____; in++) {
            if(data[in]>data[in+1]) {
                swap(_____, _____, _____);
            }
        }
    }
}
```

Results (Let's trace!)

First round (out: 4)

4,7,2,5,3 (in: 0, in+1: 1, swap?: N)

4,2,7,5,3 (in: 1, in+1: 2, swap?: Y)

4,2,5,7,3 (in: 2, in+1: 3, swap?: Y)

4,2,5,3,7 (in: 3, in+1: 4, swap?: Y)

Second round (out: 3)

4,2,5,3,7 (in: 0, in+1: 1, swap?: Y)

2,4,5,3,7 (in: 1, in+1: 2, swap?: N)

2,4,3,5,7 (in: 2, in+1: 3, swap?: Y)

Third round (out: 2)

2,4,3,5,7 (in: 0, in+1: 1, swap?: N)

2,3,4,5,7 (in: 1, in+1: 2, swap?: Y)

Fourth round (out: 1)

2,3,4,5,7 (in: 0, in+1: 1, swap?: N)

Invariants

Conditions that remain unchanged as the algorithm proceeds.

What is the invariant in bubbleSort?

Values after "out" are sorted.

Time complexity

For 5 items, there are 4 comparisons on the first pass, 3 comparisons on the second pass, and so on, which makes 10 comparisons total:

$$4+3+2+1 = 10$$

$$(N-1) + (N-2) + (N-3) + \dots + 1 = N*(N-1) / 2$$

We can say that the bubble sort algorithm makes about $N^2 / 2$ comparisons.

Also, keep in mind that there are swaps: $N^2 / 4$

When is the case that we need to swap in every comparison on every pass?

Because constants don't count in Big O notation, we can conclude that bubble sort runs in $O(N^2)$ time.

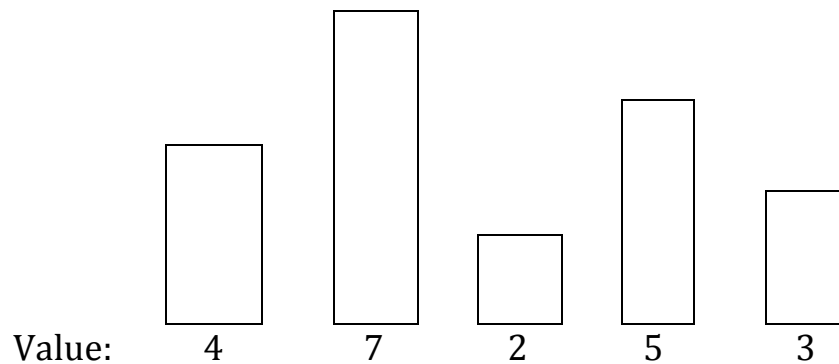
$$n C_k = \frac{n!}{k! (n-k)!} = \frac{n(n-1)}{2}$$

Selection Sort: Faster than Bubble Sort but still not enough**Step 1: Conceptual View**

There are two steps in selection sort:

- ***Pick or SELECT*** the minimum value
- Swap it with the element on the left end

Initial State



After first round

Number of comparisons? _____

Number of swaps? _____

After second round

Number of comparisons? _____

Number of swaps? _____

Now, do you see a pattern?

After third round

Number of comparisons? _____

Number of swaps? _____

After fourth round

Number of comparisons? _____

Number of swaps? _____

On each pass, where are the items that are sorted?

Where was it in Bubble Sort?

Step 2: Implementation View

```
int[] a = {4,7,2,5,3};
```

Swap method

```
// a helper method that swaps two values in an array
private void swap(int[] data, int one, int two) {
    int temp = data[one];
    data[one] = data[two];
    data[two] = temp;
}
```

Select the minimum

```
int min; // min variable
for(int out=0; out < data.length-1; out++) {
    min = out; // set initial min value's index

    // Select a new minimum value's index
    for(int in=1; in < data.length; in++) {
        if(data[in] < data[min]) // if new min value found
            min = in; // reset the new min index
    }

    swap(data, out, min); // time to put min value.
}
```

Finally, putting the pieces all together

```
public void selectionSort(int[] data) {
    int min; // min variable
    for(int out=0; out < _____; out++) {
        min = _____; // set initial min value's index

        // Select a new minimum value's index
        for(int in=_____; in < _____; in++) {
            if(data[in] < data[min]) // if new min value
                min = in; // reset the new min index
        }

        swap(_____, _____, min); // time to put min value.
    }
}
```

Results (Let's trace!)

First round (out: 1, min: 0)

4,7,2,5,3 (in: 1, min: 0)

4,7,2,5,3 (in: 2, min: 2)

4,7,2,5,3 (in: 3, min: 2)

4,7,2,5,3 (in: 4, min: 2)

swap(____, ____)

2,7,4,5,3

Second round (out: 1, min: ____)

2,7,4,5,3 (in: 2, min: 2)

2,7,4,5,3 (in: 3, min: 2)

2,7,4,5,3 (in: 4, min: 4)

swap(____, ____)

2,3,4,5,7

Third round (out: 2, min: ____)

2,3,4,5,7 (in: 3, min: 2)

2,3,4,5,7 (in: 4, min: 2)

swap(____, ____)

2,3,4,5,7

Fourth round (out: 3, min: ____)

2,3,4,5,7 (in: ____, min: 3)

swap(____, ____)

2,3,4,5,7

Invariants

The elements less than _____ variable are sorted.

Time Complexity

First of all, do you see any improvement?

Is the number of comparisons in the selection sort the same as the bubble sort?

Same

How about number of swaps?

smaller, less swaps

What is the time complexity?

Insertion Sort: Best among the three! *critical*

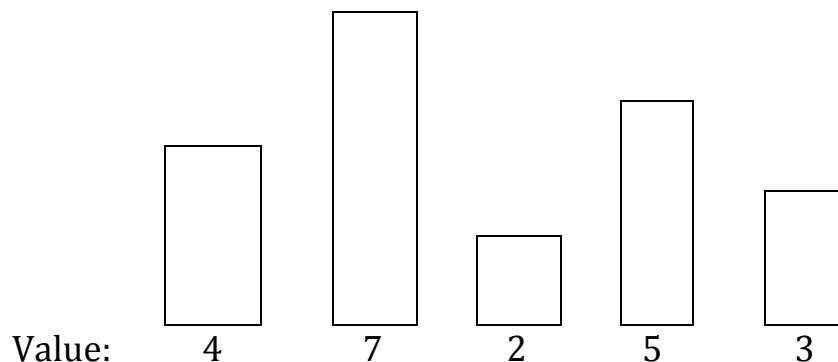
Step 1: Conceptual View

To me, this is the *most intuitive* sorting algorithm.

Most important thing in the insertion sort is that there is ***an imaginary dividing line***.

- Left hand side of the line is *partially* sorted
- The first element of the right hand side of the line should be inserted into the left hand side in a proper position
 - First, we keep the value of the first element into a temp place
 - Shift the items of the left hand side to the right so that there can be a space for the value that is stored in the temp place
 - When the position is found, ***INSERT the value into*** that position

Initial State



After first round

Where is the dividing line? _____

Which value should be kept in the temporary place? _____

After second round

Where is the dividing line? _____

Which value should be kept in the temporary place? _____

After third round

Where is the dividing line? _____

Which value should be kept in the temporary place? _____

After fourth round

Where is the dividing line? _____

Which value should be kept in the temporary place? _____

Step 2: Implementation View

```
int[] a = {4,7,2,5,3};
```

```
public static void insertionSort(int[] data) {
    // set and increase the division line
    for(int out=___1___; out < ___data.length___, out++) {
        int temp = data[out]; // store the value into temp
        int in = out;

        // go backward in the left side of the imaginary line
        // to find a place to insert temp value
        while(in > 0 && ___data[in-1] >= tmp___) {
            data[in] = data[in-1];
            in--;
        }

        data[___in___] = temp; // INSERT the temp value
    }
}
```

Results (Let's trace!)

First round (out: 1, tmp: 7)
 4, ,2,5,3 (in: 1, in-1: 0, shift?: N)
 insert
 4,7,2,5,3

Second round (out: 2, tmp: 2)
 4,7, ,5,3 (in: 2, in-1: 1, shift?: Y)
 4, ,7,5,3 (in: 1, in-1: 0, shift?: Y)
 ,4,7,5,3
 insert
 2,4,7,5,3

Third round (out: 3, tmp: 5)
 2,4,7, ,3 (in: 3, in-1: 2, shift?: Y)
 2,4, ,7,3 (in: 2, in-1: 1, shift?: Y)
 insert
 2,4,5,7,3

Fourth round (out: 4, tmp: 3)
 2,4,5,7, (in: 4, in-1: 3, shift?: Y)
 2,4,5, ,7 (in: 3, in-1: 2, shift?: Y)
 2,4, ,5,7 (in: 2, in-1: 1, shift?: Y)
 2, ,4,5,7 (in: 1, in-1: 0, shift?: N)
 insert
 2,3,4,5,7

Invariants

At the end of each round, the elements less than _____ variable are **PARTIALLY** sorted.

Time Complexity

First of all, do you see any improvement?

Is the number of comparisons in the insertion sort the same as the bubble sort? (Hint: how far do you go back to actually insert each round?)

Do you notice anything different in terms of swapping?

1 swap = 3 copies

What is the running time complexity?

There is a time that this insertion sort can run even faster as much as in $O(n)$ time. Can you think of it?

Now, how about the opposite case of the previous situation? Would it be faster than the bubble sort?

These three internal sorting algorithms, bubble sort, selection sort and insertion sort, all run in $O(n^2)$ time.

However, in most cases, insertion sort performs better than the other two because it requires less number of comparisons depending on the input values and uses copying instead of swapping.