Sorting: Selection Sort and Insertion Sort

Instructors: Sam McCauley and Dan Barowy
March 7, 2022
Admin

- Any questions?
Sorting
How can we sort a set of items?

- Goal: sequence of steps
How can we sort a set of items?

- Goal: sequence of steps
- Guarantee that the cards are sorted at the end
How can we sort a set of items?

- Goal: sequence of steps
- Guarantee that the cards are sorted at the end
- We want to be able to:
How can we sort a set of items?

- Goal: sequence of steps
- Guarantee that the cards are sorted at the end
- We want to be able to:
  - Code it up in Java
How can we sort a set of items?

• Goal: sequence of steps

• Guarantee that the cards are sorted at the end

• We want to be able to:
  • Code it up in Java
  • Analyze the running time
Specifics

- Have an array of numbers

| 10 | 21 | -3 | 40 | 17 | 13 | 11 | -4 |
Specifics

- Have an array of numbers
- Want to sort them \textit{in place} (without copying to a new array)
Specifics

- Have an array of numbers
- Want to sort them *in place* (without copying to a new array)
  - In other words: sort them using $O(1)$ extra space.
Specifics

10  21  -3  40  17  13  11  -4

-3  -4  10  11  13  17  21  40

• Have an array of numbers

• Want to sort them *in place* (without copying to a new array)

  • In other words: sort them using $O(1)$ extra space.
Where to Start?

- Is there any number we can put directly in the correct place?

| 10 | 21 | -3 | 40 | 17 | 13 | 11 | -4 |

• We can put the largest number in the last slot
• Scan through the array to find the maximum number
• Time?
• $O(n)$
• Swap that number with the last number
Where to Start?

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>21</th>
<th>-3</th>
<th>40</th>
<th>17</th>
<th>13</th>
<th>11</th>
<th>-4</th>
</tr>
</thead>
</table>

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
Where to Start?

10 21 -3 40 17 13 11 -4

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number

Time?

$O(n)$

Swap that number with the last number.
Where to Start?

| 10 | 21 | -3 | 40 | 17 | 13 | 11 | -4 |

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
Where to Start?

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$
Where to Start?

| 10 | 21 | -3 | 40 | 17 | 13 | 11 | -4 |

• Is there any number we can put directly in the correct place?
• We can put the largest number in the last slot
• Scan through the array to find the maximum number
  • Time?
  • $O(n)$
• *Swap* that number with the last number

Maximum so far:
Where to Start?

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
    - $O(n)$
- **Swap** that number with the last number

Maximum so far: 10 at pos 0
Where to Start?

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$
- *Swap* that number with the last number

| 10 | 21 | -3 | 40 | 17 | 13 | 11 | -4 |

Maximum so far: 21 at pos 1
### Where to Start?

*Is there any number we can put directly in the correct place?*

*We can put the largest number in the last slot*

*Scan through the array to find the maximum number*
  *Time?*
  *$O(n)$*

*Swap* that number with the last number

| 10 | 21 | -3 | 40 | 17 | 13 | 11 | -4 |

Maximum so far: 21 at pos 1
Where to Start?

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$
- *Swap* that number with the last number

```
10 21 -3 40 17 13 11 -4
```

Maximum so far: 40 at pos 3
Where to Start?

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$
- *Swap* that number with the last number

| 10 | 21 | -3 | 40 | 17 | 13 | 11 | -4 |

Maximum so far: 40 at pos 3
Where to Start?

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$
- **Swap** that number with the last number

| 10 | 21 | -3 | 40 | 17 | 13 | 11 | -4 |

Maximum so far: 40 at pos 3
Where to Start?

10 21 -3 40 17 13 11 -4

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
    - $O(n)$
  - Swapped that number with the last number

Maximum so far: 40 at pos 3
Where to Start?

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
    - $O(n)$
  - Swap that number with the last number

Maximum so far: 40 at pos 3
Where to Start?

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$
- *Swap* that number with the last number
Now what?

- Do it again! But now on all but the last element of the array
Now what?

- Do it again! But now on all but the last element of the array

- (This is essentially a recursive algorithm)
Selection Sort

Maximum so far: 10 at pos 0
### Selection Sort

| 10 | 21 | -3 | -4 | 17 | 13 | 11 | 40 |

**Maximum so far:** 21 at pos 1
Selection Sort

10 21 -3 -4 17 13 11 40

Maximum so far: 21 at pos 1
Selection Sort

10  21  -3  -4  17  13  11  40

Maximum so far: 21 at pos 1
### Selection Sort

| 10 | 21 | -3 | -4 | 17 | 13 | 11 | 40 |

Maximum so far: 21 at pos 1
Selection Sort

Maximum so far: 21 at pos 1
## Selection Sort

Maximum so far: 21 at pos 1
Selection Sort

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>10</th>
<th>11</th>
<th>-3</th>
<th>-4</th>
<th>17</th>
<th>13</th>
<th>21</th>
<th>40</th>
</tr>
</thead>
</table>

Maximum so far: 10 at pos 0
Selection Sort

| 10 | 11 | -3 | -4 | 17 | 13 | 21 | 40 |

Maximum so far: 11 at pos 1
## Selection Sort

| 10 | 11 | -3 | -4 | 17 | 13 | 21 | 40 |

**Maximum so far:** 11 at pos 1
Selection Sort

Maximum so far: 11 at pos 1
Selection Sort

| 10 | 11 | -3 | -4 | 17 | 13 | 21 | 40 |

Maximum so far: 17 at pos 4
## Selection Sort

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>-3</td>
<td>-4</td>
<td>17</td>
<td>13</td>
<td>21</td>
<td>40</td>
</tr>
</tbody>
</table>

Maximum so far: 17 at pos 4
Selection Sort

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>-3</td>
<td>-4</td>
<td>17</td>
<td>13</td>
<td>21</td>
<td>40</td>
</tr>
</tbody>
</table>

Maximum so far: 17 at pos 4
Selection Sort

-3  -4  10  11  13  17  21  40
Let’s look at the code

- We’ll do loops, not recursion
Let’s look at the code

- We’ll do loops, not recursion

- Let’s assume we have a `swap(int[], int, int)` method that swaps two indices of an array
Selection Sort Code

```java
public static void selectionSort(int data[], int n) {
    int numUnsorted = n;
    int index; // general index
    int max; // index of largest value
    while (numUnsorted > 0) {
        // determine maximum value in array
        max = 0;
        for (index = 1; index < numUnsorted; index++) {
            if (data[max] < data[index]) max = index;
        }
        swap(data, max, numUnsorted-1);
        numUnsorted--;
    }
}
```
How can we prove that this works?

- Why does it work?

Idea: after the loop iterates \( i \) times,
- The last \( i \) slots of the array contain the \( i \) largest elements of the array in sorted order
- When \( i = n \) we are done

Prove using induction. (Kind of like recursive algorithms.)
How can we prove that this works?

- Why does it work?

- Idea: after the loop iterates $i$ times,
  
  - The last $i$ slots of the array contain the $i$ largest elements of the array in sorted order
How can we prove that this works?

- Why does it work?

- Idea: after the loop iterates $i$ times,
  - The last $i$ slots of the array contain the $i$ largest elements of the array in sorted order

- When $i = n$ we are done
How can we prove that this works?

- Why does it work?

- Idea: after the loop iterates $i$ times,
  - The last $i$ slots of the array contain the $i$ largest elements of the array in sorted order

- When $i = n$ we are done

- Prove using induction. (Kind of like recursive algorithms.)
Proving Correctness by Induction

To show: for all $k \leq n$, after the loop iterates $k$ times, the last $k$ slots of the array contain the $k$ largest elements of the array in sorted order.
To show: for all $k \leq n$, after the loop iterates $k$ times, the last $k$ slots of the array contain the $k$ largest elements of the array in sorted order.

- Base case: $k = 0$. Already satisfied
Proving Correctness by Induction

To show: for all $k \leq n$, after the loop iterates $k$ times, the last $k$ slots of the array contain the $k$ largest elements of the array in sorted order.

- Base case: $k = 0$. Already satisfied

- Inductive hypothesis: for some $k$, after the loop iterates $k$ times, the last $k$ slots of the array contain the $k$ largest elements of the array in sorted order.
To show: for all $k \leq n$, after the loop iterates $k$ times, the last $k$ slots of the array contain the $k$ largest elements of the array in sorted order.

- Base case: $k = 0$. Already satisfied

- Inductive hypothesis: for some $k$, after the loop iterates $k$ times, the last $k$ slots of the array contain the $k$ largest elements of the array in sorted order.

- Inductive step: by the inductive hypothesis, after the $k$th iteration of the outer loop, the last $k$ slots of the array contain the $k$ largest array items in sorted order. We scan through the array and find the largest element excluding the last $k$ slots; this is the $k + 1$st largest item. The swap moves it into the $k + 1$st slot from the end of the array.
Wrapping up selection sort

- Fill up the array from right to left with largest element

- How long does finding the maximum take?
  - $O(n - i + 1)$ time for the $i$th loop

- Summing:
  - $\sum_{i=1}^{n} O(n - i + 1) = \sum_{j=1}^{n} O(j) = O(n^2)$
Wrapping up selection sort

- Fill up the array from right to left with largest element

- How long does finding the maximum take?
Wrapping up selection sort

- Fill up the array from right to left with largest element

- How long does finding the maximum take?
  
  - $O(n - i + 1)$ time for the $i$th loop
Wrapping up selection sort

- Fill up the array from right to left with largest element

- How long does finding the maximum take?
  - $O(n - i + 1)$ time for the $i$th loop

- Summing: $\sum_{i=1}^{n} O(n - i + 1) = \sum_{j=1}^{n} O(j) = O(n^2)$
Insertion Sort
Insertion Sort

- Similar to Selection Sort
Insertion Sort

- Similar to Selection Sort

- Significantly more efficient in practice (we’ll come back to this)
Insertion Sort

• Similar to Selection Sort

• Significantly more efficient in practice (we’ll come back to this)

• This time we’ll start with why it works, and derive the algorithm
Insertion Sort

- A different approach to sorting
Insertion Sort

- A different approach to sorting
- After the $k$th loop, the first $k$ items in the array are sorted
  - The first $k$ items may not be the smallest—but they are in sorted order

- How can we guarantee this for $k = 1$?
  - Don't need to do anything
- Let's say it works for $k$. What does the $k+1$st loop need to accomplish to maintain the invariant?
  - Needs to insert the $k+1$st item among the first $k$ items in sorted order.
Insertion Sort

- A different approach to sorting
- After the $k$th loop, the first $k$ items in the array are sorted
  - The first $k$ items may not be the smallest—but they are in sorted order
- How can we guarantee this for $k = 1$?
Insertion Sort

- A different approach to sorting
- After the $k$th loop, the first $k$ items in the array are sorted
  - The first $k$ items may not be the smallest—but they are in sorted order
- How can we guarantee this for $k = 1$?
  - Don’t need to do anything
Insertion Sort

- A different approach to sorting
- After the $k$th loop, the first $k$ items in the array are sorted
  - The first $k$ items may not be the smallest—but they are in sorted order
- How can we guarantee this for $k = 1$?
  - Don’t need to do anything
- Let’s say it works for $k$. What does the $k + 1$st loop need to accomplish to maintain the invariant?

| -3 | 10 | 21 | 40 | 17 | 13 | 11 | -4 |
Insertion Sort

- A different approach to sorting
- After the $k$th loop, the first $k$ items in the array are sorted
  - The first $k$ items may not be the smallest—but they are in sorted order
- How can we guarantee this for $k = 1$?
  - Don’t need to do anything
- Let’s say it works for $k$. What does the $k + 1$st loop need to accomplish to maintain the invariant?
  - Needs to insert the $k + 1$st item among the first $k$ items in sorted order.
A Beautiful Way to Accomplish This

-3 10 17 21 40 13 11 -4

- Want to take the new item and move it into sorted position
A Beautiful Way to Accomplish This

-3  10  17  21  40  13  11  -4

- Want to take the new item and move it into sorted position

- Idea: need to move it down until the previous element is smaller
A Beautiful Way to Accomplish This

-3 10 17 21 40 13 11 -4

- Want to take the new item and move it into sorted position.

- Idea: need to move it down until the previous element is smaller.

- Inner loop: store element we are trying to insert. Shift elements down while it is smaller.
public static void insertionSort(int data[], int n) {
    int numSorted = 1; // number of values in place
    int index; // general index
    while (numSorted < n) {
        int temp = data[numSorted]; // first unsorted value
        for (index = numSorted; index > 0; index--) {
            if (temp < data[index-1]) {
                data[index] = data[index-1];
            } else {
                break;
            }
        }
        data[index] = temp; // reinsert value
        numSorted++;
    }
}
Insertion Sort Code

```java
public static void insertionSort(int data[], int n) {
    int numSorted = 1; // number of values in place
    int index; // general index
    while (numSorted < n) {
        int temp = data[numSorted]; // first unsorted value
        for (index = numSorted; index > 0; index--) {
            if (temp < data[index-1]) {
                data[index] = data[index-1];
            } else {
                break;
            }
        }
        data[index] = temp; // reinsert value
        numSorted++;
    }
}
```

Can we get rid of the `break` command in this code?
public static void insertionSort(int data[], int n) {
    int numSorted = 1; // number of values in place
    while (numSorted < n) {
        int temp = data[numSorted]; // first unsorted value
        int index = numSorted;
        while (index > 0 && temp < data[index - 1]) {
            data[index] = data[index-1];
            index--;
        }
        data[index] = temp; // reinsert value
        numSorted++;
    }
}
Tradeoff with Selection Sort

- No swap method needed
Tradeoff with Selection Sort

- No swap method needed
- Code is a little shorter
Tradeoff with Selection Sort

- No swap method needed
- Code is a little shorter

Efficiency?
- Both take $n$ iterations of the outer loop. What about the inner loop?
- Selection sort *always* iterates through $n - i$ elements on the $i$th iteration
- Insertion sort may stop early! Can lead to better performance in practice (and is never worse)
Tradeoff with Selection Sort

- No swap method needed
- Code is a little shorter

**Efficiency?**
- Both take $n$ iterations of the outer loop. What about the inner loop?
- Selection sort *always* iterates through $n - i$ elements on the $i$th iteration
- Insertion sort may stop early! Can lead to better performance in practice (and is never worse)

- To be clear: both are still $O(n^2)$ in terms of worst-case performance. Insertion sort just has better constants, and better best-case performance
Sorting Objects
What do we need

- Reminder: we interact with objects using methods
What do we need

- Reminder: we interact with objects using methods

- What methods do we need in order to sort objects?
What do we need

- Reminder: we interact with objects using **methods**

- What methods do we need in order to sort objects?
  
  - Need to be able to determine if one item is less than another
What do we need

• Reminder: we interact with objects using methods

• What methods do we need in order to sort objects?
  • Need to be able to determine if one item is less than another

• Two ways that this may work. Both are good depending on use case.
What do we need

• Reminder: we interact with objects using methods

• What methods do we need in order to sort objects?
  
  • Need to be able to determine if one item is less than another

• Two ways that this may work. Both are good depending on use case.
  
  • First: only sort objects of a type with a `compareTo()` method, allowing two objects of that type to be compared
What do we need

- Reminder: we interact with objects using methods

- What methods do we need in order to sort objects?
  - Need to be able to determine if one item is less than another

- Two ways that this may work. Both are good depending on use case.
  - First: only sort objects of a type with a `compareTo()` method, allowing two objects of that type to be compared
  - Second: create a new method that allows us to compare the objects
Sorting with `compareTo`

- Let’s add a `compareTo()` method to `Student`
Sorting with `compareTo`

- Let’s add a `compareTo()` method to `Student`

- This method compares the name of this student
Sorting with `compareTo`

- Let’s add a `compareTo()` method to `Student`
- This method compares the name of this student
- How does this choice affect what a sorted vector looks like?
Sorting with `compareTo`

- Let’s add a `compareTo()` method to `Student`

- This method compares the name of this student

- How does this choice affect what a sorted vector looks like?

- Let’s try sorting `Students` with a `compareTo` method
Making InsertionSort generic

- We never used the fact that this is a vector of students (other than the `compareTo()` method)
Making InsertionSort generic

- We never used the fact that this is a vector of students (other than the `compareTo()` method)

- What kind of types can we sort?
Making InsertionSort generic

- We never used the fact that this is a vector of students (other than the `compareTo()` method)

- What kind of types can we sort?

- We want this class to have a `compareTo()` method. How can we require this?
Making InsertionSort generic

- We never used the fact that this is a vector of students (other than the `compareTo()` method)

- What kind of types can we sort?

- We want this class to have a `compareTo()` method. How can we require this?

- With an interface!
Comparable\(<T>\) Interface

- This is a Java interface, *not structure5*. (Built-in; don’t need to import anything.)
Comparable<T> Interface

- This is a Java interface, *not structure5*. (Built-in; don’t need to import anything.)

- Comparable<T> has only one method: `public int compareTo(T other)`
Comparable\(<T>\) Interface

- This is a Java interface, *not structure5*. (Built-in; don’t need to import anything.)

- Comparable\(<T>\) has only one method: `public int compareTo(T other)`

- Let’s tell Java that our `Student` class implements this interface
Creating a generic sorting method

- We can make the InsertionSort class generic, but that seems a bit nonspecific.
Creating a generic sorting method

- We can make the `InsertionSort` class generic, but that seems a bit nonspecific.

- Really: want to make one method generic. Can we do this in Java?
Creating a generic sorting method

• We can make the InsertionSort class generic, but that seems a bit nonspecific.

• Really: want to make one method generic. Can we do this in Java?

• Yes! Looks something like this:
  • public static void <E> insertionSort(Vector<E> vec)
Creating a generic sorting method

- We can make the InsertionSort class generic, but that seems a bit nonspecific.

- Really: want to make one method generic. Can we do this in Java?

- Yes! Looks something like this:
  - public static void <E> insertionSort(Vector<E> vec)

- Problem: can’t use *any* E. Needs to be comparable with other objects of type E
Generic Upper Bounds

- Way to tell Java that a generic type needs to meet certain requirements

These are called upper bounds. Let's say we only want to accept objects that meet the requirements of the `List` interface. Rather than `<E>`, we write something like `<E extends List>`. (Yes, it's `extends` and not `implements`. There are some good back-end reasons for this.)

What do we want for our `insertionSort` method?

Want `<E extends Comparable<E>>`. That is to say: we want a type `E` that implements `Comparable<E>`. That is to say: need that objects of type `E` have a `compareTo` method that takes objects of type `E` as argument.
Generic Upper Bounds

- Way to tell Java that a generic type needs to meet certain requirements
- That way, at compile time, Java can make sure our types match up
Generic Upper Bounds

- Way to tell Java that a generic type needs to meet certain requirements
- That way, at compile time, Java can make sure our types match up
- These are called *upper bounds*
Generic Upper Bounds

- Way to tell Java that a generic type needs to meet certain requirements
- That way, at compile time, Java can make sure our types match up
- These are called **upper bounds**
- Let’s say we only want to accept objects that meet the requirements of the List interface. Rather than `<E>`, we write something like `<E extends List>`
  - *(Yes, it’s extends and not implements. There are some good back-end reasons for this.)*
Generic Upper Bounds

• Way to tell Java that a generic type needs to meet certain requirements
• That way, at compile time, Java can make sure our types match up
• These are called upper bounds
• Let’s say we only want to accept objects that meet the requirements of the List interface. Rather than <E>, we write something like <E extends List>
  • (Yes, it’s extends and not implements. There are some good back-end reasons for this.)
• What do we want for our insertionSort method?
  • Want <E extends Comparable<E>>
  • That is to say: we want a type E that implements Comparable<E>. That is to say: need that objects of type E have a compareTo method that takes objects of type E as argument
Where we are

- Can sort any object so long as it implements Comparable<E>
Where we are

- Can sort any object so long as it implements Comparable<E>
- What are the downsides of this?
Where we are

- Can sort any object so long as it implements `Comparable<E>`

- What are the downsides of this?

  - What if we want to sort objects that aren’t already comparable and we don’t want to modify the class?
Where we are

- Can sort any object so long as it implements `Comparable<E>`

- What are the downsides of this?

  - What if we want to sort objects that aren’t already comparable and we don’t want to modify the class?

  - Can only sort objects one way. (What if we want to sort `Student` objects by grade? Would need to rewrite the `Student` class!)
Where we are

- Can sort any object so long as it implements `Comparable<E>`

- What are the downsides of this?
  - What if we want to sort objects that aren’t already comparable and we don’t want to modify the class?
  - Can only sort objects one way. (What if we want to sort `Student` objects by grade? Would need to rewrite the `Student` class!)

- There are upsides as well; we’ll come back to this after we talk about Comparators