Admin

Compression assignment

Lab tomorrow

Sorting

Insertion sort

Selection sort

How do they work? Best, worst, average case runtime?

Selection sort

Divide the array into two parts: a sorted part on the left and an unsorted part on the right

- Find the smallest element in the unsorted part
- Swap it with the leftmost element of the unsorted array
- The sorted array is now one element larger
Selection sort

Divide the array into two parts: a sorted part on the left and an unsorted part on the right.

Repeat:
- Find the smallest element in the unsorted part
- Swap it with the leftmost element of the unsorted array
- The sorted array is now one element larger

Selection sort: overall runtime

Best case = worst case = average case = \(O(n^2)\)

Insertion sort

Divide the array into two parts:
- left part: left elements in sorted order
- right part: right elements in unsorted order

Repeat:
- Look at the next element in the unsorted part
- Find the correct location in the sorted part (by sliding each item right one at a time)
- The sorted array is now one element larger

Insertion sort

Divide the array into two parts:
- left part: left elements in sorted order
- right part: right elements in unsorted order

Repeat:
- Look at the next element in the unsorted part
- Find the correct location in the sorted part (by sliding each item right one at a time)
- The sorted array is now one element larger
Insertion sort: overall runtime

Best case: $O(n)$, the array is already sorted.

Worst case: $O(n^2)$, the array is reverse sorted (same sum as before).

Average case: $O(n^2)$, $n$ iterations and still have to move $n/2$ entries on average.

Divide the array into two parts:
left part: left elements in sorted order
right part: right elements in unsorted order

Repeat:
- Look at the next element in the unsorted part
- Find the correct location in the sorted part (by sliding each item right one at a time)
- The sorted array is now one element larger

Sorting algorithm properties

Stable sorting algorithms

If there are ties, the elements occur in their original order.

Excel demo!

Selection sort

Divide the array into two parts: a sorted part on the left and an unsorted part on the right.

Repeat:
- Find the smallest element in the unsorted part
- Swap it with the leftmost element of the unsorted array
- The sorted array is now one element larger

Are these stable?

Insertion sort

Divide the array into two parts:
left part: left elements in sorted order
right part: right elements in unsorted order

Repeat:
- Look at the next element in the unsorted part
- Find the correct location in the sorted part (by sliding each item right one at a time)
- The sorted array is now one element larger

Selection sort

Divide the array into two parts: a sorted part on the left and an unsorted part on the right.

Repeat:
- Find the smallest element in the unsorted part
- Swap it with the leftmost element of the unsorted array
- The sorted array is now one element larger

Insertion sort is stable

Divide the array into two parts:
left part: left elements in sorted order
right part: right elements in unsorted order

Repeat:
- Look at the next element in the unsorted part
- Find the correct location in the sorted part (by sliding each item right one at a time)
- The sorted array is now one element larger

Are these stable?
## Sorting algorithm properties

### In-place sorting

Can be done without additional memory, i.e., another array

### Insertion sort

- Divide the array into two parts: a sorted part on the left and an unsorted part on the right
- Repeat:
  - Find the smallest element in the unsorted part
  - Swap it with the leftmost element of the unsorted array
  - The sorted array is now one element larger

### Selection sort

- Divide the array into two parts:
  - left part: left elements in sorted order
  - right part: right elements in unsorted order
- Repeat:
  - Look at the next element in the unsorted part
  - Find the correct location in the sorted part (by sliding each item right one at a time)
  - The sorted array is now one element larger

### Are these in-place?

- Insertion sort is in-place
- Selection sort is in-place

### What questions do we ask about the data?

- Selection sort
  - Divide the array into two parts: a sorted part on the left and an unsorted part on the right
  - Repeat:
    - Find the smallest element in the unsorted part
    - Swap it with the leftmost element of the unsorted array
    - The sorted array is now one element larger

- Insertion sort
  - Divide the array into two parts:
    - left part: left elements in sorted order
    - right part: right elements in unsorted order
  - Repeat:
    - Look at the next element in the unsorted part
    - Find the correct location in the sorted part (by sliding each item right one at a time)
    - The sorted array is now one element larger

- Selection sort
  - Divide the array into two parts: a sorted part on the left and an unsorted part on the right
  - Repeat:
    - Find the smallest element in the unsorted part
    - Swap it with the leftmost element of the unsorted array
    - The sorted array is now one element larger

- Insertion sort
  - Divide the array into two parts:
    - left part: left elements in sorted order
    - right part: right elements in unsorted order
  - Repeat:
    - Look at the next element in the unsorted part
    - Find the correct location in the sorted part (by sliding each item right one at a time)
    - The sorted array is now one element larger
Insertion sort
Divide the array into two parts: a sorted part on the left and an unsorted part on the right
Repeat:
- Find the smallest element in the unsorted part
- Swap it with the leftmost element of the unsorted array
- The sorted array is now one element larger

Selection sort
Divide the array into two parts: a sorted part on the left and an unsorted part on the right
Repeat:
- Look at the next element in the unsorted part
- Find the correct location in the sorted part (by sliding each item right one at a time)
- The sorted array is now one element larger

What questions do we ask about the data?
- Compare to other elements

Comparable interface
https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

Interface Comparable<T>

int compareTo(T other)
- -1: this object is less than other (technically, any negative number)
  0: this object is equal to other
  1: this object is greater than other (technically, any positive number)
Which algorithm is this?

```java
public static void sortComparable(int[] a) {
    for (int i = 0; i < a.length; ++i) {
        int smallestIndex = i;
        for (int j = i + 1; j < a.length; ++j) {
            if (a[j].compareTo(a[smallestIndex]) < 0) {
                smallestIndex = j;
            }
        }
        Comparable temp = a[i];
        a[i] = a[smallestIndex];
        a[smallestIndex] = temp;
    }
}
```

Which algorithm is this?

```java
public static void sortComparable(int[] a) {
    for (int i = 0; i < a.length; ++i) {
        int smallestIndex = i;
        for (int j = i + 1; j < a.length; ++j) {
            if (a[j].compareTo(a[smallestIndex]) < 0) {
                smallestIndex = j;
            }
        }
        Comparable temp = a[i];
        a[i] = a[smallestIndex];
        a[smallestIndex] = temp;
    }
}
```

A better way

We can constrain the type variable to only allow for classes that implement Comparable<E>.

```java
public static <E extends Comparable<E>> void sortBetter(E[] a) {
    for (int i = 0; i < a.length; ++i) {
        int smallestIndex = i;
        for (int j = i + 1; j < a.length; ++j) {
            if (a[j].compareTo(a[smallestIndex]) < 0) {
                smallestIndex = j;
            }
        }
        E temp = a[i];
        a[i] = a[smallestIndex];
        a[smallestIndex] = temp;
    }
}
```

Merge

Assuming left (L) and right (R) are sorted already, merge the two to create a new, single sorted array

| L: 1 3 5 8 | R: 2 4 6 7 |

How can we do this?
Create a new array to hold the result that is the combined length

What item is first? How did you know?

Compare the first two elements in the lists!

What item is second? How did you know?
Compare the smallest element that hasn’t been used yet in each list
- For L, this is next element in the list
- For R, this is still the first element

General algorithm:
- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we’re done:
  - Compare current elements
  - Copy smaller one down and increment that index
General algorithm:
- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index
General algorithm:
- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index
Merge

L: 1 3 5 8  
R: 2 4 6 7

1 2 3 4 5

General algorithm:
- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index

Merge

L: 1 3 5 8  
R: 2 4 6 7

1 2 3 4 5 6

General algorithm:
- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index

What do we do now?
Merge

If we run off the end of either array, just copy the remaining from the other array

Merge in code

```java
public static <E extends Comparable<E>> E[] merge(E[] left, E[] right) {
    int size = left.length + right.length;
    E[] result = (E[])(new Object[size]);
    int leftIndex = 0;
    int rightIndex = 0;
    for (int i = 0; i < size; ++i) {
        if (leftIndex >= left.length) { // done with left
            result[i] = right[rightIndex++];
        } else if (rightIndex >= right.length) { // done with right
            result[i] = left[leftIndex++];
        } else if (left[leftIndex].compareTo(right[rightIndex]) < 0) {
            result[i] = left[leftIndex++];
        } else {
            result[i] = right[rightIndex++];
        }
    }
    return result;
}
```

MergeSort

Divide the data in half

Call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

If the two halves are sorted, does MergeSort work?
**MergeSort**

Divide the data in half

Call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

What are we missing? Why does this work?

---

**MergeSort**

Divide the data in half

Call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

**MergeSort** is recursive. We're missing a base case!

---

**MergeSort: base case**

7

Is this array sorted?

---

**MergeSort: base case**

7

If the array is of size 1 (or 0), it's sorted
MergeSort

$ms(7 \ 1 \ 4 \ 2 \ 6 \ 5 \ 3 \ 8)$

53

MergeSort

$ms(7 \ 1 \ 4 \ 2 \ 6 \ 5 \ 3 \ 8)$

$7 \ 1 \ 4 \ 2 \ 6 \ 5 \ 3 \ 8$

54

split in half

MergeSort

$ms(7 \ 1 \ 4 \ 2 \ 6 \ 5 \ 3 \ 8)$

$ms(7 \ 1 \ 4 \ 2)$

$6 \ 5 \ 3 \ 8$

55

sort left side

MergeSort

$ms(7 \ 1 \ 4 \ 2 \ 6 \ 5 \ 3 \ 8)$

$ms(7 \ 1 \ 4 \ 2)$

$6 \ 5 \ 3 \ 8$

$7 \ 1 \ 4 \ 2$

56

split in half
MergeSort

$ms(7 \ 1 \ 4 \ 2 \ 6 \ 5 \ 3 \ 8)$

$ms(7 \ 1 \ 4 \ 2) \quad 6 \ 5 \ 3 \ 8$

$ms(7 \ 1) \quad 4 \ 2$

sort left side

MergeSort

$ms(7 \ 1 \ 4 \ 2 \ 6 \ 5 \ 3 \ 8)$

$ms(7 \ 1 \ 4 \ 2) \quad 6 \ 5 \ 3 \ 8$

$ms(7 \ 1) \quad 4 \ 2$

$7 \ 1$

split in half

MergeSort

$ms(7 \ 1 \ 4 \ 2 \ 6 \ 5 \ 3 \ 8)$

$ms(7 \ 1 \ 4 \ 2) \quad 6 \ 5 \ 3 \ 8$

$ms(7 \ 1) \quad 4 \ 2$

$ms(7) \quad 1$

sort left side

MergeSort

$ms(7 \ 1 \ 4 \ 2 \ 6 \ 5 \ 3 \ 8)$

$ms(7 \ 1 \ 4 \ 2) \quad 6 \ 5 \ 3 \ 8$

$ms(7 \ 1) \quad 4 \ 2$

$ms(7) \quad 1$

what now?
MergeSort

ms(7 1 4 2 6 5 3 8)

ms(7 1 4 2) 6 5 3 8

ms(7 1) 4 2

ms(1) 7

what now?

merge!

65

66

MergeSort

ms(7 1 4 2 6 5 3 8)

ms(7 1 4 2) 6 5 3 8

ms(7 1) 4 2

ms(1) 7

merge!

67

68

MergeSort

ms(7 1 4 2 6 5 3 8)

ms(7 1 4 2) 6 5 3 8

ms(7 1) 4 2

ms(4 2) 1 7

sort right
MergeSort

ms(7 1 4 2 6 5 3 8)

ms(7 1 4 2) 6 5 3 8

ms(7) 1 4 2

split in half
sort left
sort right

merge!

Now what?
MergeSort

1 2 3 4 5 6 7 8
ms(7 1 4 2 6 5 3 8)

1 2 4 7
ms(7 1 4 2)

3 5 6 8
ms(6 5 3 8)

1 7
ms(7 1)

2 4
ms(4 2)

…

7
ms(7)

1
ms(1)

4
ms(4)

2
ms(2)

merge!

MergeSort: implementation 1

```java
public static <E extends Comparable<E>> E[] mergeSort(E[] a) {
    if (a.length <= 1) {
        return a;
    } else {
        int mid = a.length/2;
        E[] left = Arrays.copyOfRange(a, 0, mid);
        E[] right = Arrays.copyOfRange(a, mid, a.length);
        E[] sortedLeft = mergeSort(left);
        E[] sortedRight = mergeSort(right);
        return merge(sortedLeft, sortedRight);
    }
}
```