Problems, solutions, and algorithms

- What is the difference between an algorithm and a **problem**?

- What is the difference between an algorithm and a **solution to a problem**?

- What is the difference between an algorithm and a **program**?

Properties of a good algorithm

1. It clearly defines the state (variables)
2. It clearly defines the order of execution for instructions
3. Each instruction is clear to the intended audience
   - You can leave out details about instructions when you are sure that the reader can understand it without them.

Algorithms, revisited

- What is an algorithm?

  Wikipedia Definition: an **algorithm** is a definite list of well-defined instructions for completing a task that, given an initial state, will proceed through a well-defined series of successive states, eventually terminating in an end-state.
Algorithms: Why?

Why bother writing an algorithm? Why not just write the code?

An algorithm lets you leave out unimportant details of the program. That way, you can focus on the hard parts of solving the problem.

Example Algorithm: Washing Dishes

# Note: the indentation indicates where the while loop ends.
Stack dishes by sink.
Fill sink with hot soapy water.

while there are more dishes:
    Get dish from counter,
    Wash dish,
    Put dish in drain rack.

Wipe off counter.
Rinse out sink.

Example Algorithm: Greatest Common Divisor

# Compute the GCD using the brute force approach.
# Notation: x | y means x divides y evenly with no remainder.
# Inputs: m and n are two integers.
# Output: gcf.
factor = 1, gcf = 1

while factor <= minimum of m and n:
    if factor | m and factor | n:
        Let gcf = factor.
    Let factor = factor + 1.

The Sorting Problem

Problem Statement:

Given a set of data and a method for comparing two data points to determine which is greater or smaller (or equal),

Organize the data into a sequence so that the smallest element appears first, and every subsequent element is at least as big as the one before it.
The Sorting Problem

Problem Statement, revisited:

Input:
A set of data
A comparison method

Output:
Data organized in sequence, smallest to largest

A potential solution?

- Find the biggest element, move it to the end
- Of the (size-1) elements left, find the biggest one, and move it to second-to-last.
- Repeat.

Is this a good algorithm?
NO. It leaves out too much detail – how to find the biggest element? How to move it to the end?
BubbleSort

Iteration 1

compare

They're in the wrong order.
Swap them.

compare

They're in the wrong order.
Swap them.

compare

They're in the wrong order.
Swap them.

compare

They're in the wrong order.
Swap them.
BubbleSort

Iteration 1

They're in the wrong order.
Swap them.

-29  8  -2  -15  10  40  -6  -7  31  0

compare

compare

-29  8  -2  -15  10  40  -6  -7  31  0

They're in the wrong order.
Swap them.
BubbleSort

Iteration 1

-29  8  -2  -15  10  -6  -7  40  31  0

They're in the **wrong** order.
Swap them.
BubbleSort

Iteration 1

(compare)

They're in the **wrong** order.
Swap them.

Notice that the biggest element is now at the end.

Iteration 1
**BubbleSort**

**Iteration 2**
*Repeat the same process as iteration 1*

1. **compare**

   -29  -2   8   -15  10  -6   -7   31   0   40

   They're in the wrong order.
   Swap them.

2. **compare**

   -29  -2   8   -15  10  -6   -7   31   0   40

   They're in the wrong order.

3. **compare**

   -29  8   -2   -15  10  -6   -7   31   0   40

   They're in the right order.
   Don't do anything.

4. **compare**

   -29  8   -2   -15  10  -6   -7   31   0   40

   They're in the wrong order.
BubbleSort

Iteration 2
Repeat the same process as iteration 1

-29  -2  -15  8  10  -6  -7  31  0  40

They're in the wrong order.
Swap them.

Iteration 2
Repeat the same process as iteration 1

-29  -2  -15  8  -6  10  -7  31  0  40

They're in the wrong order.
Swap them.

Iteration 2
Repeat the same process as iteration 1

-29  -2  -15  8  10  -6  -7  31  0  40

They're in the wrong order.
Swap them.

Iteration 2
Repeat the same process as iteration 1

-29  -2  -15  8  10  -6  -7  31  0  40

They're in the right order.
Don’t do anything.
Iteration 2

Repeat the same process as iteration 1

compare

They’re in the wrong order.

Swap them.

Iteration 2

Repeat the same process as iteration 1

compare

They’re in the right order.

Don’t do anything.
**Iteration k**

*Repeat the same process as iteration 1*

*Stop at k-to-last element.*

The kth iteration will guarantee that the kth-biggest element ends up in the kth-to-last spot.

```
-29  -2  -15  8  -6  -7  10  0  31  40
```

---

**Iteration 2**

*Repeat the same process as iteration 1*

They're in the **wrong** order.

Swap them.

```
-29  -2  -15  8  -6  -7  10  0  31  40
```

---

**Iteration k**

*Repeat the same process as iteration 1*

*Stop at k-to-last element.*

The kth iteration will guarantee that the kth-biggest element ends up in the kth-to-last spot.

```
-29  -15  -2  -6  -7  8  0  10  31  40
```

Iteration 3 puts 3rd-biggest element here.

---

**Iteration 2**

*Repeat the same process as iteration 1*

*Stop at second-to-last element this time.*

Notice that the second-biggest element is now in the second-to-last place.

```
-29  -2  -15  8  -6  -7  10  0  31  40
```
BubbleSort

Iteration k

Repeat the same process as iteration 1

Stop at k-to-last element.

The kth iteration will guarantee that the kth-biggest element ends up in the kth-to-last spot.

BubbleSort stops when one full iteration performs zero swaps.

-29 -15 -7 -6 -2 0 8 10 31 40

Iteration 4 puts 4th-biggest element here.

In this case, iteration 4 also puts the 5th- and 6th-biggest elements in the right places, but that’s not guaranteed to happen in general.

A Sorting Algorithm: BubbleSort

# Input: an array of elements called D
# Output: D is sorted

performedSwap = true
while performedSwap is true:
    performedSwap = false
    for i goes from 0 to the end of D, less 1:
        if D[i] > D[i+1]:
            swap D[i] and D[i+1]
            performedSwap = true

Iteration 5 just swaps these two guys.

Iteration 6 leaves everything alone – everything is in the right spot!

BubbleSort

Iteration k

Repeat the same process as iteration 1

Stop at k-to-last element.

The kth iteration will guarantee that the kth-biggest element ends up in the kth-to-last spot.
First, what is “merging”?

- **Merging** two arrays means
  - The two arrays start off sorted
  - After merging, the resulting array contains all of the elements of the original two arrays, and they’re all sorted.

Efficient Sorting Algorithms

- There are a large number of algorithms that can sort efficiently, but the three most common are:
  - MergeSort
  - QuickSort
  - HeapSort

- **QuickSort** is often the quickest in practice, but somewhat difficult to understand

- We will study MergeSort, and you will implement a version on your homework

BubbleSort is Slow!!!

- BubbleSort is very inefficient on large arrays
  - Java does not use the BubbleSort algorithm for its Arrays.sort() and Collections.sort()
  - In fact, almost nobody does BubbleSort, except as a teaching device
  - We will talk later on about how you can tell that it’s not an efficient algorithm

Merging sub-Arrays that are sorted

- First, what is “merging”? Merging two arrays means
  - The two arrays start off sorted
  - After merging, the resulting array contains all of the elements of the original two arrays, and they’re all sorted.

- Efficient Sorting Algorithms
  - There are a large number of algorithms that can sort efficiently, but the three most common are:
    - MergeSort
    - QuickSort
    - HeapSort

- QuickSort is often the quickest in practice, but somewhat difficult to understand

- We will study MergeSort, and you will implement a version on your homework
Merging sub-Arrays that are sorted

Step 1: Check D[i1], D[i2], move smallest to Temp[next]
Merging sub-Arrays that are sorted

Repeat!

Step 2: Update next, i1, and i2

Repeat!
Exercise: The Merge Algorithm

function merge(D, left1, right1, left2, right2):
    Temp = an array big enough to hold both subarrays
    i1 = left1, i2 = left2, iTemp = 0
    while i1 <= right1 and i2 <= right2:
        if D[i1] < D[i2]:
            Temp[iTemp] = D[i1]
            increment i1, iTemp
        else:
            Temp[iTemp] = D[i2]
            increment i2, iTemp
    while i1<=right1:
        # for when i2 finishes first
        Temp[iTemp] = D[i1]
        increment i1, iTemp
    while i2<=right2:
        # for when i1 finishes first
        Temp[iTemp] = D[i2]
        increment i2, iTemp
    copy Temp[0 to length-1] to D[left1 to right2]
Using Merge to Sort

Step 2: Merge second two elements.

-29 40 -2 8 -15 10 -6 -7 31 0

Using Merge to Sort

Step 3: Merge third two elements.

-29 40 -2 8 -15 10 -6 -7 31 0

Using Merge to Sort

Step 4: Merge fourth two elements.

-29 40 -2 8 -15 10 -6 -7 31 0

Using Merge to Sort

Step 2: Merge second two elements.

-29 40 -2 8 -15 10 -6 -7 31 0

Using Merge to Sort

Step 2: Merge second two elements.

-29 40 -2 8 -15 10 -6 -7 31 0
Using Merge to Sort

Step 4: Merge fourth two elements.

Using Merge to Sort

Step 5: Merge fifth two elements.

End phase 1.

Notice: we now have 5 sub-arrays of length 2, each of which is sorted.

Phase 2 will be: merge sub-arrays of length 2.
Using Merge to Sort

Phase 2, step 1: merge the first two and second two elements

-29 40

Phase 2, step 2: merge the third two and fourth two elements

-29 -2 8 40 -15 -7 -6 0 31

Using Merge to Sort

Phase 2, step 1: merge the first two and second two elements

-29 -2 8 40 -15 10 -7 -6 0 31

Using Merge to Sort

Phase 2, step 2: merge the third two and fourth two elements

-29 -2 8 40 -15 -7 -6 10 0 31
Using Merge to Sort

Phase 3, step 1: Merge the first 4 and second 4 elements.

Finally, phase 4: Merge first 8 and last 2 elements.

End phase 2.
(The fifth two elements have nothing to merge with, so we leave those alone for now.)
Notice: we now have two sub-arrays of length 4, which are sorted.
Phase 3 will merge these.
What you’re expected to know

- Know how to implement the Comparable interface, and use the Arrays.sort() and Arrays.binarySearch() methods.
- Know the requirements for all algorithms
  - Linear search, bubble sort, merge sort: none
  - Binary search: whole array must be sorted
  - Merge: each sub-array must be sorted
- I expect you to know the search algorithms
  - Memorize or understand well enough to reproduce on a quiz or exam
- You DO NOT need to know the algorithms for bubble sort, merge, or merge sort by heart.
- You DO need to be able to simulate an iteration of each algorithm on a quiz or exam.
- If I give you the algorithm for one of these, I expect you to be able to write the code.

Using Merge to Sort

Done.

Exercise: The MergeSort Algorithm

```java
# input array D, output sorted D
mergeSize = 1
while mergeSize < length of D:
    left1 = 0
    while left1 + mergeSize < length of D:
        right1 = left1 + mergeSize - 1
        left2 = left1 + mergeSize
        right2 = the smaller of
            left1 + 2*mergeSize-1, length of D-1
        merge(D,left1,right1,left2,right2)
        left1 = right2 + 1
    mergeSize = mergeSize * 2
```

-29  -15  -7  -6  -2  0  8  10  31  40