CSCI 136:
Data Structures
and
Advanced Programming

Lecture 15

Sorting, part 3

Instructor: Dan Barowy

Williams

Announcements

Midterm exam:

- During your normal lab period
- You will have 1 hour, 45 minutes if you come on time.
- Closed book
- Covers readings from Bailey Ch. 1-7 & 9
- Stuff in class up to today.

Regular class on Monday

Exam review session: Monday 7-8pm in TCL 202

Short class on Wed: last chance to ask questions.

Outline

Visibility modifiers

Counting sort

Radix sort

Imagine that you are a cycling purist

who owns a bike shop.

Tricycles make you angry.

"They don't belong in a bike shop."

From before: Inheritance

Inheritance is a **mechanism** for defining a class in terms of another class. It is a labor-saving device employed to reduce **code duplication**. Inheritance allows programmers to specify a new implementation while :

1. **maintaining the same behavior**,
2. **reusing code**, and
3. **extending the functionality** of existing software.

How can we prevent programmers from changing essential behavior?

```
abstract class Cycle {
    int numWheels;
    int numGears;
    String color;
    String brand;

    Cycle(String color, String brand) {
        this.color = color;
        this.brand = brand;
        this.numWheels = 2;
        this.numGears = 2;
    }

    public String toString() {
        return
            "number of wheels: " + numWheels + "\n" +
            "number of gears: " + numGears + "\n" +
            "color: " + color + "\n" +
            "brand: " + brand + "\n";
    }
}
```

How can we prevent programmers from changing essential behavior?

```
class Tricycle extends Cycle {
    String flagColor;

    public Tricycle(String color, String brand, String flagColor) {
        super(color, brand);
        numWheels = 3;
        this.flagColor = flagColor;
    }

    public String toString() {
        return
            super.toString() +
            "flag color: " + flagColor + "\n";
    }
}
```

Can we make it **impossible** to change `numWheels` in subclass?

How can we prevent programmers from changing essential behavior?

```
abstract class Cycle {
    int numWheels;
    int numGears;
    String color;
    String brand;

    Cycle(String color, String brand) {
        this.color = color;
        this.brand = brand;
        this.numWheels = 2;
        this.numGears = 2;
    }

    public String toString() {
        return
            "number of wheels: " + numWheels + "\n" +
            "number of gears: " + numGears + "\n" +
            "color: " + color + "\n" +
            "brand: " + brand + "\n";
    }
}
```

Can we make it **impossible** to override `numWheels`?

Yes. Use **visibility modifiers**.

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

The default is "no modifier," aka *package-private*.

Modifiers control **who has access** to a class/method/variable and under what circumstances.

Yes. Use **visibility modifiers**.

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

E.g., a class always has access to a member (variable or method), regardless of modifier.

Note: Subclass means "subclass in a different package."

How can we prevent programmers from changing essential behavior?

```
abstract class Cycle {
    private int numWheels;
    int numGears;
    String color;
    String brand;

    Cycle(String color, String brand) {
        this.color = color;
        this.brand = brand;
        this.numWheels = 2;
        this.numGears = 2;
    }

    public String toString() {
        return
            "number of wheels: " + numWheels + "\n" +
            "number of gears: " + numGears + "\n" +
            "color: " + color + "\n" +
            "brand: " + brand + "\n";
    }
}
```

Can we make it **impossible** to override `numWheels`? **Yes.**

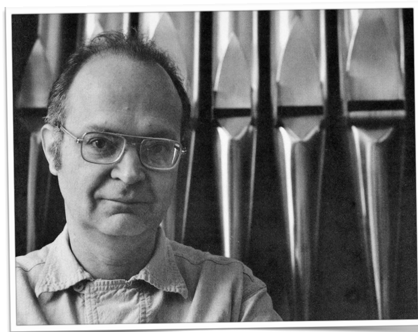
Hipster mission accomplished

```
$ javac *.java
Tricycle.java:6: error: numWheels has private access in Cycle
    numWheels = 3;
    ^
1 error
```

Sorting faster than **O(n log n)**

Donald Knuth proved that **comparison sorting** can **never** be faster than **O(n log n)**.

(nice proof in the "CLRS" textbook for the curious)



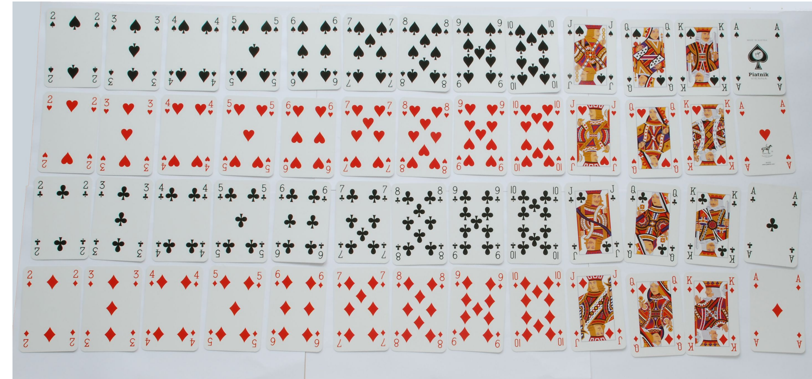
But the question remains: can we sort faster than **O(n log n)**?

Answer: Yes, as long as we can make assumptions about data.

Counting sort

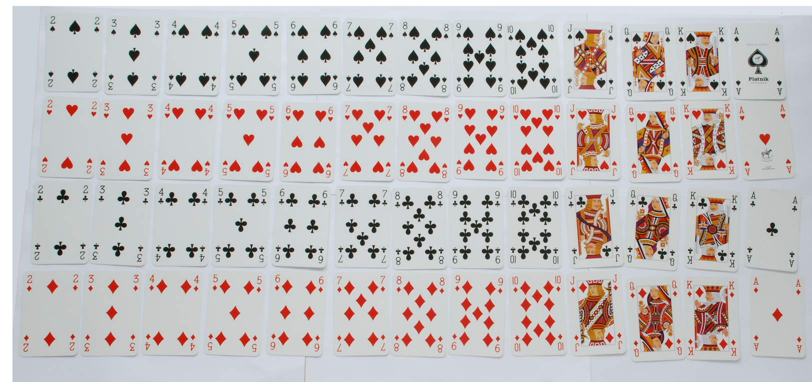
How many cards in a deck of cards?

52



Suppose I dropped them on the floor

To sort them, I could use insertion sort, quick sort, etc.



But unlike other things I might sort, I know exactly where these things *should* go.

(code)

Counting sort

```
public static int[] sort(int[] A, int k) {
    int[] B = new int[A.length];
    int[] C = new int[k];
    // initialize counting array
    for (int i = 0; i < k; i++) {
        C[i] = 0;
    }
    // count instances of values stored in A[j]
    for (int j = 0; j < A.length; j++) {
        C[A[j]] += 1;
    }
    // produce cumulative totals in C so that
    // C[i] contains the number of elements <= i
    for (int i = 1; i < k; i++) {
        C[i] += C[i-1];
    }
    // sort using position dictated by C[i]
    for (int j = A.length - 1; j >= 0; j--) {
        B[C[A[j]] - 1] = A[j];       // put A[j] in B in position C[A[j]] - 1
        C[A[j]] = C[A[j]] - 1;       // update count for C[A[j]]
    }
}
```

Cool fact: counting sort is $O(n + k)$

Cool fact: counting sort is stable

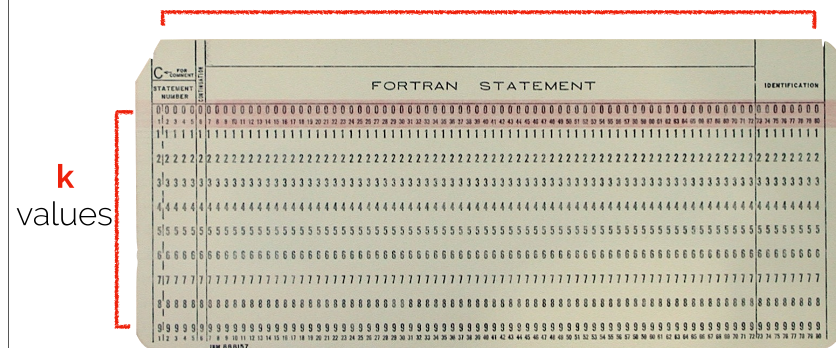
It is $O(n)$ when $k << n$.

What if we want to sort bigger numbers?

Can we still do it in $O(n)$ time?

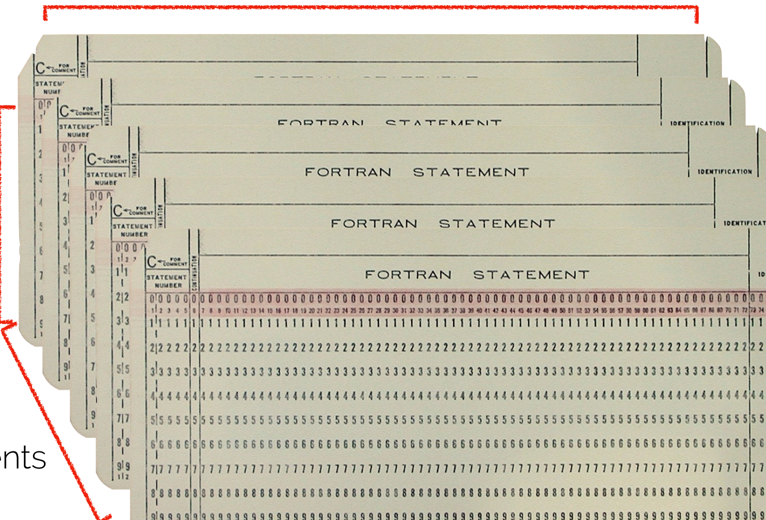
A multi-digit number

d digits



(a Hollerith punch card)

A multi-digit number



Radix sort

- Insight: sort each element by its least significant digit.
- Then sort by the next significant digit.
- …
- Finally, sort by most significant digit.
- This will take **O(d * g(n))** time
- If **g(n) = O(n + k)**, then **O(d(n + k))** time
- E.g., d = 80, k = 10 then **O(80(n + 10))** time = **O(n)** time

Radix sort

```
public static int[] sort(int[] A, int k, int d) {
    for (int i = 0; i < d; i++) {
         // suppose dthDigit(A) returns a new array
         // consisting of the dth digit of every
         // element in A.
        int[] D = dthDigit(A);
        return CountingSort.sort(D, k);
    }
}
```

Recap & Next Class

Today we learned:

Visibility modifiers

Counting sort

Radix sort

Next class:

Search