Searching

Also: Logarithms

Searching an array of integers

- If an array is not sorted, there is no better algorithm than linear search for finding an element in it

```
static final int NONE = -1;  // not a legal index

static int linearSearch(int target, int[] a) {
    for (int p = 0; p < a.length; p++) {
        if (target == a[p]) return p;
    }
    return NONE;
}
```

Searching an array of Strings

- Searching an array of Strings is just like searching an array of integers, *except*
  - Instead of `int1==int2` we need to use `string1.equals(string2)`

```
static final int NONE = -1;  // not a legal index

static int linearSearch(String target, String[] a) {
    for (int p = 0; p < a.length; p++) {
        if (target.equals(a[p])) return p;
    }
    return NONE;
}
```

Searching an array of Objects

- Searching an array of Objects is just like searching an array of Strings, *provided*
  - The operation `equals` has been defined appropriately

```
static final int NONE = -1;  // not a legal index

static int linearSearch(Object target, Object[] a) {
    for (int p = 0; p < a.length; p++) {
        if (target.equals(a[p])) return p;
    }
    return NONE;
}
```

Templates

- There is no way, in Java, to write a *general* linear search method for any data type
- We can write a method that works for an array of objects (because `Object` defines the `equals` method)
  - For arbitrary objects, `equals` is just `==`
  - For your own objects, you may want to override `public boolean equals(Object o)`
    - The parameter must be of type `Object`!
  - The method we defined for `Object`s also works fine for `String`s (because `String` overrides `equals`)
- A search method for objects *won't work* for primitives
  - Although an `int` can be autoboxed to an Integer, an `int[]` *cannot* be autoboxed to an `Integer[]`

Review: Overriding methods

- To override a method means to replace an inherited method with one of your own
- Your new method must be a *legal replacement* for the inherited version
  - Consequences:
    - Your new method must have the exact **same signature** (name, order and types of parameters—but parameter names are irrelevant)
    - Your new method must have the **same return type**
    - Your new method must be **at least as public** as the method it replaces
    - Your new method can throw **no new exceptions** that the method being overridden doesn't already throw
  - In Java 5 and later, you should put `@Override` in front of your method
    - This lets the compiler check that you got the signature right

Java review: equals

- The Object class defines
  ```
  public boolean equals(Object obj)
  ```
- For most objects, this just tests *identity:* whether the two objects are really one and the same
- This is *not* generally what you want
- The String class overrides this method with a method that is more appropriate for Strings
- You can override equals for your own classes
  - If you override equals, there are some rules you should follow

Overriding `equals`

- If you override `equals`, your method should have the following properties (for your objects `x`, `y`, `z`)
  - Reflexive: for any `x`, `x.equals(x)` should return `true`
  - Symmetric: for any non-null objects `x` and `y`, `x.equals(y)` should return the same result as `y.equals(x)`
    - For any non-null `x`, `x.equals(null)` should return `false`
  - Transitive: if `x.equals(y)` and `y.equals(z)` are `true`, then `x.equals(z)` should also be `true`
  - Consistent: `x.equals(y)` should always return the *same* answer (unless you modify `x` or `y`, of course)
- Java cannot check to make sure you follow these rules

Reference implementation for `equals`

```
public class Person {
    String name;

    public Person(String name) {
        this.name = name;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (! (o instanceof Person)) return false;
        Person p = (Person)o;
        return (name.equals(p.name));
    }
}
```

Overriding `hashCode`

- Whenever you override `equals`, you should also override `public int hashCode()`
  - This method "makes hash" of its instance, producing a value that looks random
  - The purpose of this function is not discussed here
- There is only one rule that the `hashCode` method *must* follow:
  - If `object1.equals(object2)`, then it must be true that `object1.hashCode() == object2.hashCode()`
  - Note that the reverse is *not* necessarily true—unequal objects may have equal hash codes

About sorted arrays

- An array is sorted in ascending order if each element is no smaller than the preceding element
- An array is sorted in descending order if each element is no larger than the preceding element
- When we just say an array is "sorted," by default we mean that it is sorted in ascending order
- An array of `Object` ***cannot be in sorted order !***
  - There is no notion of "smaller" or "larger" for arbitrary objects
  - We can *define* an ordering for some of our objects

The `Comparable` interface

- `java.lang` provides a `Comparable` interface with the following method:
  - `public int compareTo(Object that)`
  - This method should return
    - A negative integer if `this` is less than `that`
    - Zero if `this` equals `that`
    - A positive integer if `this` is greater than `that`
- Reminder: you *implement* an interface like this:

```
class MyObject implements Comparable {
        public int compareTo(Object that) {...}
}
```

Rules for implementing `Comparable`

- You *must* ensure:
  - `x.compareTo(y)` and `y.compareTo(x)` either are both zero, or else one is positive and the other is negative
  - `x.compareTo(y)` throws an exception if and only if `y.compareTo(x)` throws an exception
  - The relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`
  - if `x.compareTo(y)==0`, then `x.compareTo(z)` has the same sign as `y.compareTo(z)`
- You *should* ensure:
  - `compareTo` is consistent with `equals`

Consistency with `equals`

- `compareTo` is consistent with equals if:

  `x.compareTo(y)==0`

  gives the same boolean result as

  `x.equals(y)`
- *Therefore:* if you implement `Comparable`, you really should override `equals` as well
- Java doesn’t actually require consistency with equals, but sooner or later you’ll get into trouble if you don’t meet this condition

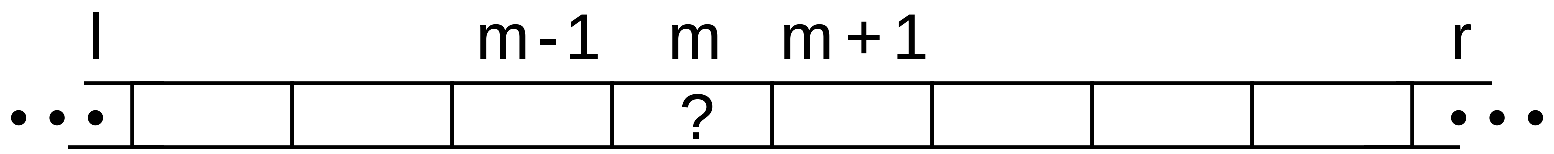
Binary search

- *Linear search* has linear time complexity:
  - Time `n` if the item is not found
  - Time `n/2`, on average, if the item is found
- If the array is sorted, we can write a faster search
- How do we look up a name in a phone book, or a word in a dictionary?
  - Look somewhere in the middle
  - Compare what's there with the thing you're looking for
  - Decide which half of the remaining entries to look at
  - Repeat until you find the correct place
  - This is the binary search algorithm

Binary search algorithm (p. 43)

- To find which (if any) component of `a[left..right]` is equal to `target` (where `a` is sorted):

```
Set l = left, and set r = right
While l <= r, repeat:
    Let m be an integer about midway between l and r
    If target is equal to a[m], terminate with answer m
    If target is less than a[m], set r = m–1
    If target is greater than a[m], set l = m+1
Terminate with answer none
```



Example of binary search

Search the following array a for 36:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	5	7	10	13	13	15	19	19	23	28	28	32	32	37	41	46

1. (0+15)/2=7; a[7]=19;
   too small; search 8..15
2. (8+15)/2=11; a[11]=32;
   too small; search 12..15
3. (12+15)/2=13; a[13]=37;
   too large; search 12..12
4. (12+12)/2=12; a[12]=32;
   too small; search 13..12...but 13>12, so quit: 36 not found

Binary search in Java (p. 45)

```
static int binarySearch(Comparable target,
                    Comparable[] a, int left, int right) {
    int l = left, r = right;
    while (l <= r) {
        int m = (l + r) / 2;
        int comp = target.compareTo(a[m]);
        if (comp == 0) return m;
        else if (comp < 0) r = m – 1;
        else /* comp > 0 */  l = m + 1;
    }
    return NONE; // As before, NONE = -1
}
```

Recursive binary search in Java

```
static int binarySearch(Comparable target,
                        Comparable[] a, int left, int right) {
    if (left > right) return NONE;
    int m = (left + right) / 2;
    int comp = target.compareTo(a[m]);
    if (comp == 0) return m;
    else if (comp < 0)
        return binarySearch(target, a, left, m-1);
    else {
        assert comp > 0;
        return binarySearch(target, a, m+1, right);
    }
}
```

Strings of bits

- There is only one possible zero-length sequence of bits
- There are two possible "sequences" of a single bit: 0, 1
- There are four sequences of two bits: 00 01, 10 11
- There are eight sequences of three bits: 000 001, 010 011, 100 101, 110 111
- Each time you add a bit, you double the number of possible sequences
  - Add 0 to the end of each existing sequence, and do the same for 1
- "Taking the logarithm" is the inverse of exponentiation
- $2^0 = 1$ $2^1 = 2$ $2^2 = 4$ $2^3 = 8$, etc.
- $\log_2 1 = 0$ $\log_2 2 = 1$ $\log_2 4 = 2$ $\log_2 8 = 3$, etc.

Logarithms

- In computer science, we almost always work with logarithms base 2, because we work with bits
- $\log_2 n$ (sometimes written as lg n) tells us how many bits we need to represent n possibilities
  - Example: To represent 10 digits, we need lg 10 = 3.322 bits
  - Since we can't have fractional bits, we need 4 bits, with some bit patterns not used: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, and not 1010, 1011, 1100, 1101, 1110, 1111
- Logarithms also tell us how many times we can cut a positive integer in half before reaching 1
  - Example: 16/2=8, 8/2=4, 4/2=2, 2/2=1, and lg 16 = 4
  - Example: 10/2=5, 5/2=2.5, 2.5/2=1.25, and lg 10 = 3.322

Relationships

- Logarithms of the same number to different bases differ by a constant factor
- $\log_2(2) = 1.000$ $\log_{10}(2) = 0.301$ $\log_2(2) / \log_{10}(2) = 3.322$
- $\log_2(3) = 1.585$ $\log_{10}(3) = 0.477$ $\log_2(3) / \log_{10}(3) = 3.322$
- $\log_2(4) = 2.000$ $\log_{10}(4) = 0.602$ $\log_2(4) / \log_{10}(4) = 3.322$
- $\log_2(5) = 2.322$ $\log_{10}(5) = 0.699$ $\log_2(5) / \log_{10}(5) = 3.322$
- $\log_2(6) = 2.585$ $\log_{10}(6) = 0.778$ $\log_2(6) / \log_{10}(6) = 3.322$
- $\log_2(7) = 2.807$ $\log_{10}(7) = 0.845$ $\log_2(7) / \log_{10}(7) = 3.322$
- $\log_2(8) = 3.000$ $\log_{10}(8) = 0.903$ $\log_2(8) / \log_{10}(8) = 3.322$
- $\log_2(9) = 3.170$ $\log_{10}(9) = 0.954$ $\log_2(9) / \log_{10}(9) = 3.322$
- $\log_2(10) = 3.322$ $\log_{10}(10) = 1.000$ $\log_2(10) / \log_{10}(10) = 3.322$

Logarithms—a summary

- Logarithms are exponents
  - if $b^x = a$, then $\log_b a = x$
  - if $10^3 = 1000$, then $\log_{10} 1000 = 3$
  - if $2^8 = 256$, then $\log_2 256 = 8$
- If we start with $x=1$ and multiply x by 2 eight times, we get 256
- If we start with $x=256$ and divide x by 2 eight times, we get 1
- $\log_2$ is how many times we halve a number to get 1
- $\log_2$ is the number of bits required to represent a number in binary (fractions are rounded up)

Binary search takes log n time

- In binary search, we choose an index that cuts the remaining portion of the array in half
- We repeat this until we either find the value we are looking for, or we reach a subarray of size 1
- If we start with an array of size n, we can cut it in half $\log_2 n$ times
- Hence, binary search has logarithmic (log n) time complexity
- For an array of size 1000, this is 100 times faster than linear search ($2^{10} \sim= 1000$)

Conclusion

- Linear search has linear time complexity
- Binary search has logarithmic time complexity
- For large arrays, binary search is far more efficient than linear search
  - However, binary search requires that the array be *sorted*
  - If the array *is* sorted, binary search is
    - 100 times faster for an array of size 1000
    - 50 000 times faster for an array of size 1 000 000
- *This* is the kind of speedup that we care about when we analyze algorithms

The End