Searching

Searching

- Problem: Given a collection of items, return whether a given item is in that collection
  - Two algorithms
    - Linear Search
    - Binary Search

Searching

- Problem:
  - Given:
    - Collection of objects to be searched (x)
      - In our examples, this collection will be stored in an array
    - Object that we are searching for (key)
  - Returns
    - Boolean value indicating if key was found in x

Evaluation

- Time Analysis
  - "Basic Operation" = comparison
  - Best case
  - Worst case
  - Average Case
- Data storage
  - Sorted vs. unsorted
  - Random-access (array) vs. List

Linear Search

- Basic idea
  - Go through the collection, one element at a time, and test if each element is equal to key.
    - If "yes", immediately return true
    - Continue until all elements have been tested.
    - After all elements have been tested unsuccessfully
      - Return false

Linear Search

```
public boolean linearSearch
   (int x[], int key) {
       for (int i=0; i < x.length; i++) {
            if (x[i] == key)
               return true;
       }
       return false;
}
```

Linear Search

- Example
  - Linear Search applet
    - Link

Linear Search

- Example
  - Things to note:
    - When key was in the array, the number of comparisons depends upon it's location in the array.
    - When key was not in the array, all elements had to be compared. Number of comparisons equaled the length of the array.

Linear Search

- Time Analysis
  - Best case
    - when key is located at x[0].
    - $T(n) = 1 = \Theta(1)$
  - Worst case
    - When key is not in array
    - $T(n) = n = \Theta(n)$

Linear Search

- Time Analysis
  - Average case
    - If you consider cases when key is in the array
      - Average case is when key is in middle of array
      - $T(n) = 0.5n = \Theta(n)$
    - However, if we consider that in the "average" case, key will not be in the array
      - Average case would required going through the entire array
      - $T(n) = n = \Theta(n)$
    - In either case $T(n) = \Theta(n)$

Linear Search

- Data Storage
  - Note that since LinearSearch handles each elements in sequential order:
    - Algorithm works:
      - Random access collections
      - Linear collections
      - When data is not sorted
      - When data is sorted

Linear Search

- Summary
  - Go through the collection, one element at a time, and test if each element is equal to key.
  - Time Analysis
    - Best case: $T(n) = \Theta(1)$
    - Worst case, avg case: $T(n) = \Theta(n)$
  - Works for
    - Sorted or unsorted data
    - Random access or linear access collections

Binary Search

- Basic idea
  - Major assumption: The collection is sorted!
  - Consider the element in the middle of the list (mid):
    - If it is equal to key, return true
    - Otherwise
      - if (key < mid)
        - » Run search on the first half of the collection
      - If (key > mid)
        - » Run same search on 2nd half of collection

Binary Search

```
public boolean binarySearch (int x[], int key) {
       int low =0; int high = x.length-1;
       while ( low < high )
       {
             mid = ( low + high ) / 2;
             else if (x[mid] < key )
                     low = mid + 1;
             else
                     high = mid - 1;
      }
      if (mid[low] == key) return true;
      else return false;
}
```

Binary Search

- Example
  - Binary Search Applet
    - Andrzej Czygrinow (Arizona State University)
    - Link

Binary Search

- Things to note
  - Algorithm will always continue until low == high.
    - This takes log n comparisons
    - Best case, worst case, average case:
      - $T(n) = \Theta(\log n)$
  - Algorithm only works if data is sorted
  - Array accesses are not in sequential order
    - Algorithm works best for a random access collection
  - Let's try to tweak the algorithm a bit

Binary Search

```
public boolean binarySearch (int x[], int key) {
       int low =0; int high = x.length-1;
       while ( low <= high )
       {
             mid = ( low + high ) / 2;
             if ( x[mid] == key ) return true;
             else if (x[mid] < key )
                     low = mid + 1;
             else
                     high = mid - 1;
      }
      return false;
}
```

Binary Search

- Things to note
  - If key is found at the middle of the array, it is returned immediately after only one comparison:
    - Best case:
      - $T(n) = 1 = \Theta(1)$
  - Otherwise, search is performed on half the array
    - worst case, average case:
      - $T(n) = \Theta(\log n)$

Binary Search

- Let's compare to Linear Search
  - Worst case:

	Linear	Binary
	O(n)	O(log n)
16	16	4
32	32	5
64	64	6
1,048,576	1,048,576	20

Binary Search

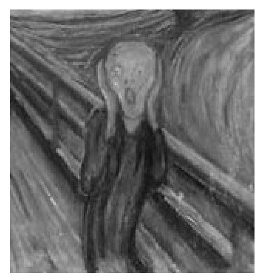
- Basic idea
  - Major assumption: The collection is sorted!
  - Consider the element in the middle of the list (mid):
    - If it is equal to key, return true
    - Otherwise
      - if (key < mid)
        - » Run search on the first half of the collection
      - If (key > mid)
        - » Run same search on 2nd half of collection
  - This algorithm just cries out for a recursive solution!

Binary Search

```
public boolean binarySearch
 (int x[], int key, int low, int high) {
    mid = ( low + high ) / 2;
    if ( x[mid] == key ) return true;
    else if (x[mid] < key )
       return binarySearch (x, key, mid+1, high);
    else
       return binarySearch (x, key, low, high-1);
}
```

Binary Search

- But won't this recursion go on forever?



Binary Search

- Yes!
  - Recursion stops when
    - The middle element of the array section we are looking at is equal to key (we check for this)
    - Or
    - When the array section we are looking at is only 1 element long

Binary Search

```
public boolean binarySearch
 (int x[], int key, int low, int high) {
    if (high == low)
        return (x[high] == key);

    mid = ( low + high ) / 2;
    if ( x[mid] == key ) return true;
    else if (x[mid] < key )
       return binarySearch (x, key, mid+1, high);
    else
       return binarySearch (x, key, low, high-1);
}
```

Recursive method

- Recall:
  - Three necessary components for a recursive method:
    1. A test to stop or continue the recursion
    2. An end case that stops the recursion
    3. A recursive call that continues the recursion.

Binary Search

```
public boolean binarySearch
 (int x[], int key, int low, int high) {
    if (high == low)
      return (x[high] == key);

    mid = ( low + high ) / 2;
   if ( x[mid] == key ) return true;
   else if (x[mid] < key )
      return binarySearch (x, key, mid+1, high);
   else
      return binarySearch (x, key, low, high-1);
}
```

Test

Stop

Continue

Binary Search

- Summary
  - Compare key with element at middle of collection and apply search to either the first half or second half of collection
    - Iterative and Recursive solutions
  - Time Analysis
    - Best case (when optimized): $T(n) = \Theta$ (1)
    - Worst case, avg case: $T(n) = \Theta$ (log n)
  - Faster than linear search but…
    - Works only when data is sorted
    - Will only perform as listed above for random access collections like arrays.

Searching

- Questions?

Linear Search: sample code

- Let's generalize to search for any object
  - Object class contains an equals method.
    - public boolean **equals**(Object obj)
      - Returns a boolean indicating whether some other object is "equal to" this one.

Linear Search

```
public boolean linearSearch
   (Object x[], Object key) {
       for (int i=0; i < x.length; i++){
             if (x[i].equals(key))
                 return true;
       }
       return false;
}
```

Binary Search: sample code

- Let's generalize to search for any object
  - Comparable interface:
    - public int compareTo(Object o)
      - Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
      - Assumes that o is of same class of object being compared to.

General Binary Search

```
public boolean binarySearch
 (Comparable x[], Comparable key, int low, int
  high) {
    if (high == low)
        return (x[high].equals(key));

    int mid = ( low + high ) / 2;
    if ( x[mid].equals(key) ) return true;
    else if ((x[mid].compareTo(key)) < 0 )
       return binarySearch (x, key, mid+1, high);
    else
       return binarySearch (x, key, low, high-1);
}
```

Sample code: Search class

- In main:
  - Creates an array of random Integer values guaranteed to have value 7 and not to have value 12.
    - Length of test array is a commandline argument
  - Performs binary and linear searches on arrays looking for 7 and 12
  - Counts the number of comparisons made

Sample code: Search class

- Method binarySearch()

```
public boolean binarySearch (Comparable x[],
  Comparable key)
{
  // clear number of compares
  n_comp = 0;

  // start a recursive binary search
  return binarySearch (x, key, 0, x.length - 1);
 }
```

private binarySearch class

Sample code: Search class

- Running the tests:
  - `java Search 100`
    - `Linear Search`
    - `Found 7 using 30 comparisons`
    - `Did not find 12 using 100 comparisons`
    - `Binary Search`
    - `Found 7 using 25 comparisons`
    - `Did not find 12 using 25 comparisons`

Sample code: Search class

- Running the tests:
  - `java Search 1000`
    - `Linear Search`
    - `Found 7 using 797 comparisons`
    - `Did not find 12 using 1000 comparisons`
    - `Binary Search`
    - `Found 7 using 49 comparisons`
    - `Did not find 12 using 49 comparisons`

Sample code: Search class

- Running the tests:
  - `java Search 1000000`
    - `Linear Search`
    - `Found 7 using 946218 comparisons`
    - `Did not find 12 using 1000000 comparisons`

    - `Binary Search`
    - `Found 7 using 127 comparisons`
    - `Did not find 12 using 127 comparisons`

Search

- Any questions?