CSC 222: Object-Oriented Programming
Spring 2012

Searching and sorting

- sequential search
- algorithm analysis: big-Oh, rate-of-growth
- binary search
- insertion sort, selection sort

Searching a list

suppose you have a list, and want to find a particular item, e.g.,
- lookup a word in a dictionary
- find a number in the phone book
- locate a student's exam from a pile

searching is a common task in computing

- searching a database
- checking a login password
- lookup the value assigned to a variable in memory

if the items in the list are unordered (e.g., added at random)
- desired item is equally likely to be at any point in the list
- need to systematically search through the list, check each entry until found

⇒ sequential search
Sequential search

sequential search traverses the list from beginning to end
- check each entry in the list
- if matches the desired entry, then FOUND (return its index)
- if traverse entire list and no match, then NOT FOUND (return -1)

recall: the ArrayList class has an indexOf method

```java
/**
 * Performs sequential search on the array field named items
 * @param desired item to be searched for
 * @returns index where desired first occurs, -1 if not found
 */
public int indexOf(Object desired) {
    for(int k=0; k < items.length; k++) {
        if (desired.equals(items[k])) {
            return k;
        }
    }
    return -1;
}
```

How efficient is sequential search?

for this algorithm, the dominant factor in execution time is checking an item
- the number of checks will determine efficiency

in the worst case:
- the item you are looking for is in the last position of the list (or not found)
- requires traversing and checking every item in the list
  - if 100 or 1,000 entries ➔ NO BIG DEAL
  - if 10,000 or 100,000 entries ➔ NOTICEABLE

in the average case?

in the best case?
Big-Oh notation

to represent an algorithm’s performance in relation to the size of the problem, computer scientists use Big-Oh notation

an algorithm is O(N) if the number of operations required to solve a problem is proportional to the size of the problem

sequential search on a list of N items requires roughly N checks (+ other constants) \( \Rightarrow O(N) \)

for an O(N) algorithm, doubling the size of the problem requires double the amount of work (in the worst case)

- if it takes 1 second to search a list of 1,000 items, then
  - it takes 2 seconds to search a list of 2,000 items
  - it takes 4 seconds to search a list of 4,000 items
  - it takes 8 seconds to search a list of 8,000 items
  - ...

Search an ordered list

when the list is unordered, can’t do any better than sequential search

- but, if the list is ordered, a better alternative exists

  e.g., when looking up a word in the dictionary or name in the phone book

  - can take ordering knowledge into account
  - pick a spot – if too far in the list, then go backward; if not far enough, go forward

binary search algorithm

- check midpoint of the list
- if desired item is found there, then DONE
- if the item at midpoint comes after the desired item in the ordering scheme, then repeat the process on the left half
- if the item at midpoint comes before the desired item in the ordering scheme, then repeat the process on the right half
**Binary search**

The *Collections* utility class contains a `binarySearch` method

- takes a `List` of `Comparable` items and the desired item
  - `List` is an interface that specifies basic list operations (`ArrayList` implements)
  - `Comparable` is an interface that requires `compareTo` method (`String` implements)

```java
/**
 * Performs binary search on a sorted list.
 * @param items sorted list of Comparable items
 * @param desired item to be searched for
 * @returns index where desired first occurs, -(insertion point)-1 if not found
 */
public static <T extends Comparable<? super T>> int
binarySearch(List<T> items, Comparable desired) {
    int left = 0; // initialize range where desired could be
    int right = items.length-1;
    while (left <= right) {
        int mid = (left+right)/2; // get midpoint value and compare
        int comparison = desired.compareTo(items[mid]);
        if (comparison == 0) { // if desired at midpoint, then DONE
            return mid;
        } else if (comparison < 0) { // if less than midpoint, focus on left half
            right = mid-1;
        } else { // otherwise, focus on right half
            left = mid + 1;
        }
    }
    return /* CLASS EXERCISE */; // if reduce to empty range, NOT FOUND
}
```

**Visualizing binary search**

Note: each check reduces the range in which the item can be found by half

- see [http://balance3e.com/Ch8/search.html](http://balance3e.com/Ch8/search.html) for demo
How efficient is binary search?

again, the dominant factor in execution time is checking an item

- the number of checks will determine efficiency

in the worst case:

- the item you are looking for is in the first or last position of the list (or not found)

start with N items in list

  after 1\textsuperscript{st} check, reduced to \(N/2\) items to search
  after 2\textsuperscript{nd} check, reduced to \(N/4\) items to search
  after 3\textsuperscript{rd} check, reduced to \(N/8\) items to search

  \ldots

  after \(\log_2 N\) checks, reduced to 1 item to search

in the average case?

in the best case?

---

Big-Oh notation

an algorithm is \(O(\log N)\) if the number of operations required to solve a problem is proportional to the logarithm of the size of the problem

binary search on a list of \(N\) items requires \(\text{roughly } \log_2 N\) checks (+ other constants)

\(\Rightarrow\) \(O(\log N)\)

for an \(O(\log N)\) algorithm, doubling the size of the problem adds only a constant amount of work

- if it takes 1 second to search a list of 1,000 items, then
  searching a list of 2,000 items will take time to check midpoint + 1 second
  searching a list of 4,000 items will take time for 2 checks + 1 second
  searching a list of 8,000 items will take time for 3 checks + 1 second

\ldots
Comparison: searching a phone book

<table>
<thead>
<tr>
<th>Number of entries in phone book</th>
<th>Number of checks performed by sequential search</th>
<th>Number of checks performed by binary search</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>8</td>
</tr>
<tr>
<td>400</td>
<td>400</td>
<td>9</td>
</tr>
<tr>
<td>800</td>
<td>800</td>
<td>10</td>
</tr>
<tr>
<td>1,600</td>
<td>1,600</td>
<td>11</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>20,000</td>
<td>20,000</td>
<td>15</td>
</tr>
<tr>
<td>40,000</td>
<td>40,000</td>
<td>16</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000</td>
<td>20</td>
</tr>
</tbody>
</table>

to search a phone book of the United States (~310 million) using binary search?

to search a phone book of the world (7 billion) using binary search?

Dictionary w/ binary search

reconsider the Dictionary class
  - load the 117K dictionary.txt & perform several searches

modify Dictionary to utilize binary search
  - import the Collections utility class

```java
import java.util.Collections;
```

  - modify the contains method so that it uses Collections.binarySearch

```java
return (Collections.binarySearch(this.dict, desiredWord) >= 0);
```

compare performance
  - is it noticeably faster when performing searches?
Dictionary revisited

binary search works as long as the list of words is sorted

- `dictionary.txt` is sorted, so can load the dictionary and do searches
- to ensure correct behavior, must also make sure that add methods maintain sorting

```java
public class Dictionary {
    private ArrayList<String> words;
    . . .
    public boolean addWord(String newWord) {
        int index = Collections.binarySearch(this.words, newWord.toLowerCase());
        this.words.add(Math.abs(index)-1, newWord.toLowerCase());
        return true;
    }
    public boolean addWordNoDupes(String newWord) {
        int index = Collections.binarySearch(this.words, newWord.toLowerCase());
        if (index < 0) {
            this.words.add(Math.abs(index)-1, newWord.toLowerCase());
            return true;
        }
        return false;
    }
    public boolean findWord(String desiredWord) {
        return (Collections.binarySearch(this.words, desiredWord.toLowerCase()) >= 0);
    }
}
```

In the worst case...

suppose words are added in reverse order: "zoo", "moo", "foo", "boo"

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>zoo</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

to add "moo", must first shift "zoo" one spot to the right

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>moo</td>
<td>zoo</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>moo</td>
<td>zoo</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>boo</td>
<td>foo</td>
<td>moo</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>boo</td>
<td>foo</td>
</tr>
</tbody>
</table>
Worst case (in general)

if inserting N items in reverse order

- 1\(^{st}\) item inserted directly
- 2\(^{nd}\) item requires 1 shift, 1 insertion
- 3\(^{rd}\) item requires 2 shifts, 1 insertion
- . . .
- N\(^{th}\) item requires N-1 shifts, 1 insertion

\[
(1 + 2 + 3 + \ldots + N-1) = N(N-1)/2 = (N^2 - N)/2 \text{ shifts} + N \text{ insertions}
\]

this approach is called "insertion sort"

- insertion sort builds a sorted list by repeatedly inserting items in correct order

since an insertion sort of N items can take roughly \(N^2\) steps, it is an O(\(N^2\)) algorithm

Timing the worst case

\textbf{System.currentTimeMillis} method accesses the system clock and returns the time (in milliseconds)

- we can use it to time repeated \texttt{adds} to a dictionary

```java
public class TimeDictionary {
    public static int timeAdds(int numValues) {
        Dictionary dict = new Dictionary();
        long startTime = System.currentTimeMillis();
        for (int i = numValues; i > 0; i--) {
            String word = String.format("0000000000", i);
            dict.addWord(word.substring(word.length()-10));
        }
        long endTime = System.currentTimeMillis();
        return (int)(endTime-startTime);
    }
}
```

<table>
<thead>
<tr>
<th># items (N)</th>
<th>time in msec</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>15</td>
</tr>
<tr>
<td>10,000</td>
<td>49</td>
</tr>
<tr>
<td>20,000</td>
<td>162</td>
</tr>
<tr>
<td>40,000</td>
<td>651</td>
</tr>
<tr>
<td>80,000</td>
<td>2270</td>
</tr>
<tr>
<td>160,000</td>
<td>9168</td>
</tr>
<tr>
<td>320,000</td>
<td>36463</td>
</tr>
</tbody>
</table>
O(N²) performance

as the problem size doubles, the
time can quadruple

makes sense for an O(N²) algorithm
  • if X items, then X² steps required
  • if 2X items, then (2X)² = 4X² steps

QUESTION: why is the factor of 4
not realized immediately?

Big-Oh captures rate-of-growth behavior in the long run
  • when determining Big-Oh, only the dominant factor is significant (in the long run)

<table>
<thead>
<tr>
<th># items (N)</th>
<th>time in msec</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>15</td>
</tr>
<tr>
<td>10,000</td>
<td>49</td>
</tr>
<tr>
<td>20,000</td>
<td>162</td>
</tr>
<tr>
<td>40,000</td>
<td>651</td>
</tr>
<tr>
<td>80,000</td>
<td>2270</td>
</tr>
<tr>
<td>160,000</td>
<td>9168</td>
</tr>
<tr>
<td>320,000</td>
<td>36463</td>
</tr>
</tbody>
</table>

Best case for insertion sort

while insertion sort can require ~N²
steps in worst case, it can do much better
  • BEST CASE: if items are added in order, then no shifting is required
  • only requires N insertion steps, so O(N)
    → if double size, roughly double time

<table>
<thead>
<tr>
<th>list size (N)</th>
<th>time in msec</th>
</tr>
</thead>
<tbody>
<tr>
<td>40,000</td>
<td>32</td>
</tr>
<tr>
<td>80,000</td>
<td>79</td>
</tr>
<tr>
<td>160,000</td>
<td>194</td>
</tr>
<tr>
<td>320,000</td>
<td>400</td>
</tr>
</tbody>
</table>

on average, might expect to shift only half the time
  • \((1 + 2 + \ldots + N-1)/2 = N(N-1)/4 = (N² - N)/4\) shifts, so still \(O(N²)\)
    → would expect faster timings than worst case, but still quadratic growth
Timing insertion sort (average case)

can use a Random object to pick random numbers and add to a String

<table>
<thead>
<tr>
<th>list size (N)</th>
<th>time in msec</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>87</td>
</tr>
<tr>
<td>20,000</td>
<td>119</td>
</tr>
<tr>
<td>40,000</td>
<td>397</td>
</tr>
<tr>
<td>80,000</td>
<td>1420</td>
</tr>
<tr>
<td>160,000</td>
<td>5306</td>
</tr>
<tr>
<td>320,000</td>
<td>20442</td>
</tr>
</tbody>
</table>

A more generic insertion sort

we can code insertion sort independent of the Dictionary class

- could use a temporary list for storing the sorted numbers, but not needed
- don't stress about `<T extends Comparable<? super T>>`
- specifies that the parameter must be an ArrayList of items that either implements or extends a class that implements the Comparable interface (???)
- more later, for now, it ensures the class has a compareTo method

```java
public static <T extends Comparable<? super T>> void insertionSort(ArrayList<T> items) {
    for (int i = 1; i < items.size(); i++) { // for each index i,
        T itemToPlace = items.get(i); // save the value at index i
        int j = i;                       // starting at index i,
        while (j > 0 && itemToPlace.compareTo(items.get(j-1)) < 0) { // shift values to the right
            items.set(j, items.get(j-1)); // until find spot for the value
            j--;
        }
        items.set(j, itemToPlace); // store the value in its spot
    }
}
```
Other $O(N^2)$ sorts

alternative algorithms exist for sorting a list of items
e.g., selection sort:
- find smallest item, swap into the 1st index
- find next smallest item, swap into the 2nd index
- find next smallest item, swap into the 3rd index
- ...

```java
public static <T extends Comparable<? super T>> void selectionSort(ArrayList<T> items) {
    for (int i = 0; i < items.size()-1; i++) {      // for each index i,
        int indexOfMin = i;                           //   find the ith smallest item
        for (int j = i+1; j < items.size(); j++) {    //   for (int j = i+1; j < items.size(); j++) {
            if (items.get(j).compareTo(items.get(indexOfMin)) < 0) {
                indexOfMin = j;                      //     if (items.get(j).compareTo(items.get(indexOfMin)) < 0) {
                    //     indexOfMin = j;  
            }
        }
        T temp = items.get(i);                        //   swap the ith smallest
        items.set(i, items.get(indexOfMin));          //   item into position i
        items.set(indexOfMin, temp);                 //   items.set(indexOfMin, temp);
    }
}
```

HW5: Hunt the Wumpus

you are to implement a text-based adventure game from the 70's

HUNT THE WUMPUSS: Your mission is to explore the maze of caves and destroy all of the wumpus (without getting yourself killed). To move to an adjacent cave, enter 'M' and the tunnel number. To toss a grenade into a cave, enter 'G' and the tunnel number.

You are currently in The Fountainhead
(1) unknown
(2) unknown
(3) unknown

What do you want to do? m 2

You are currently in The Silver Mirror
(1) The Fountainhead
(2) unknown
(3) unknown

What do you want to do? m 3

You are currently in Sherlock's Lair
(1) The Silver Mirror
(2) unknown
(3) unknown

You smell an awful stench coming from somewhere nearby.

What do you want to do? t 2

Missed, dagnabbit!
A startled wumpus charges into your cave... CHOMP CHOMP CHOMP GAME OVER
Cave class

you must implement a class that models a single cave

- each cave has a name & number, and is connected to three other caves via tunnels
- by default, caves are empty & unvisited (although these can be updated)

how do we represent the cave contents?

- we could store the contents as a string: "EMPTY", "WUMPUS", "BATS", "PIT"

```java
Cave c = new Cave("Cavern of Doom", 0, 1, 2, 3);
c.setContents("WUMPUS");
```

- potential problems?

there are only 4 possible values for cave contents

- the trouble with using a String to represent these is no error checking

```java
c.setContents("WUMPAS"); // perfectly legal, but ???
```

Enumerated types

there is a better alternative for when there is a small, fixed number of values

- an enumerated type is a new type (class) whose value are explicitly enumerated

```java
public enum CaveContents {
    EMPTY, WUMPUS, PIT, BATS
}
```

- note that these values are NOT Strings – they do not have quotes

- you specify a enumerated type value by ENUMTYPE.VALUE

```java
c.setContents(CaveContents.WUMPUS);
```

since an enumerated type has a fixed number of values, any invalid input would be caught by the compiler
the CaveMaze class reads in & stores a maze of caves

- since the # of caves is set, simpler to use an array

- provided version only allows limited movement

- you must add functionality

```java
public class CaveMaze {
    private Cave[] caves;
    private Cave currentCave;
    private boolean alive;

    public CaveMaze(String filename) throws java.io.FileNotFoundException {
        Scanner infile = new Scanner(new File(filename));
        int numCaves = infile.nextInt();
        this.caves = new Cave[numCaves];
        for (int i = 0; i < numCaves; i++) {
            int num1 = infile.nextInt();
            int num2 = infile.nextInt();
            int num3 = infile.nextInt();
            int num4 = infile.nextInt();
            String name = infile.nextLine().trim();
            this.caves[num1] = new Cave(name, num1, num2, num3, num4);
        }
        this.alive = true;
        this.currentCave = this.caves[0];
        this.currentCave.markAsVisited();
    }
}
```