Lecture 39: Searching

Last time:

1. Overloading and overriding revisited
2. Interfaces and class inheritance
3. Interface hierarchies

Today:

1. Project #8 assigned!
2. Course Reviews
3. Linear search
4. Binary search
5. Time complexity analysis

Project #8 Assigned!

- Project due 5/10 at 11pm
- Project is open
- Start before now!
  - Read entire assignment from beginning to end before starting to code
  - Check out assignment now from CVS
  - Follow the instructions *exactly*, as much of grading is automated

Course reviews

- www.courses.umd.edu/online_evaluation/

- IMPORTANT: Please fill this out for your instructor/professor AND for your TA's (There are two different online evaluations for you to fill out.)

Searching

- A basic operation in computer science
- Problem formulation
  - Given:
    - Collection (array) of values
    - A property on values
  - Find: a value having the property in the array
- Examples
  - Find a particular element
  - Find the least element
- We will study different algorithms for solving this problem

The Search Game

- Pick an element in the range 0 … 1,023
- How long does it take to guess?
  - This is a search problem
  - The collection: numbers 0 … 1,023
  - The “property”: number chosen by the other party

The Search Game: Linear Search

- Start at 0
- Guess until you find the element!
  - Answer to guess is “yes / no”
  - Next guess is one more than previous guess
  - How long will it take?
- This kind of algorithm is called linear search

Implementing Linear Search

- Given: String array, String
- Return: index of String in array, if String is in array, or -1 if String is not in array

```
public static int linSearch (String[] a, String elt) {
  for (int i = 0; i < a.length; i++)
    if (elt.equals(a[i])) return i;
  return -1; // return -1 if elt is not in a
}
```

The Search Game Revised: Binary Search

- Remember lower, upper bound
- Guess middle element
  - Answer is “yes / higher / lower”
  - If “higher”, adjust lower bound
  - If “lower”, adjust higher bound
  - How long does it take?
- This kind of algorithm is called binary search

Implementing Binary Search

- Given: Sorted String array, String
- Return: index of String in array, if String is in array, or -1 if String is not in array

```
public static int binSearch (String[] a, String elt) {
  int lower = 0;                    // First possible position of elt
  int upper = a.length;             // First impossible position of elt
  int middle;                       // Middle position between lower, upper
  while (lower < upper) {
      middle = (lower + upper) / 2;
      if (elt.compareTo(a[middle]) == 0)
        return middle;
      else if (elt.compareTo(a[middle]) < 0)
        upper = middle;
      else
        lower = middle+1;
  }
  return -1;    // Element not found
}
```

Which Search Is Better?

- For linear search: no need for sorted array
- For binary search: *faster* … or is it?
- How many guesses needed in search game (worst case) for
  - Linear search?
  - Binary search?

Answers

- For linear search: 1,024
  - Number chosen may be 1,023
  - This would require each number to be guessed
- For binary search: 10 (wow!)
  - Each guess rules out half of the remaining possibilities
  - Total number of possibilities: 1,024
  - This can be cut in half at most 10 times

Search Game Continued

- Suppose number of possible values is n: 0, 1, … n-1.
- How many guesses needed in worst case for:
  - Linear search?
  - Binary search?

Answers

- Linear search: n
- Binary search: $\log_2 n$
  - $\log_2 n$ = number of times n can be cut in half
  - $\log_{10} n$ = number of times n can be divided by 10

Number of Guesses as n Grows

guesses

70
60
50
40
30
20
10
0

0 8 16 24 32 40 48 56 64

n

linSearch
binSearch

Linear Search: As data size n grows, doubling the data doubles the runtime

- Run time for linear search: $T(n) = an + b$, where n is the data size
- Prove: $\lim_{n\to\infty} T(2n) / T(n) = 2$
- Proof:
  - So we must show:
    - $\lim_{n\to\infty} a(2n) + b / an + b = 2$
  - In taking limit, constants (b) drop out, so we must show:
    - $\lim_{n\to\infty} a(2n) / an = 2$
  - Take quotient of coefficients on LHS and cancel out a:
    - $\lim_{n\to\infty} a(2) / a = \lim_{n\to\infty} 2 = 2$ √
    - Thus, as data size n grows, the runtime doubles
- Example: $T(n) = n/2$

n	T(n)	
64	32	
		doubles
128	64	
		doubles
256	128	
		doubles
n	n/2	

Binary Search: As data size n grows, doubling the data yields runtimes that increment by a constant amount

- Run time for binary search: $T(n) = a \log_b cn + d$, where n is the data size
- Prove: $\lim_{n\to\infty} T(2n) = T(n) + k$, where k is a constant
- Proof:
  - First, note that $T(n) = a \log_b cn + d$ and $T(2n) = a \log_b c(2n) + d$
  - Next, examine T(2n) more closely:
    - $T(2n) = a \log_b c(2n) + d$
    - $= a \log_b 2(cn) + d$
    - $= a[\log_b 2 + \log_b (cn)] + d$
    - $= a \log_b 2 + a \log_b (cn) + d$
    - $= a \log_b 2 + [\, a \log_b (cn) + d \,]$
    - $= a \log_b 2 + T(n)$
  - Since $a \log_b 2$ is simply a constant, we can set $k = a \log_b 2$
  - So $\lim_{n\to\infty} T(2n) = T(n) + k$, where k is a constant √
- Note that this means each time we double the data size, the runtime increases by the same constant value k!
- Example: $T(n) = \log_2 n$

n	T(n)	
64	6	Increments by 1
128	7	Increments by 1
256	8	
n	$\log_2 n$	Increments by 1

Categories of Algorithms

- We say:
  - “Linear Search is $\theta(n)$”
  - “Binary Search is $\theta(\log_2 n)$
- $\theta(1)$
  $\theta(\log (\log n))$
  $\theta(\log n)$
  $\theta(\log n * \log n)$
  $\theta(n)$
  $\theta(n \log n)$
  $\theta(n \log (\log n))$
  $\theta(n \log n * \log n)$
  $\theta(n^{1.1})$
  $\theta(n^2)$
  $\theta(n^{100000000})$
  $\theta(2^n)$
  $\theta(3^n)$
  $\theta(3.05^n)$
  $\theta(4^n)$
  $\theta(n!)$
  $\theta(n^n)$

Huge chasm between polynomial and exponential

Big-O?

- So what, then, is “Big-O”?
- If we have a function that is $O(n^2)$, then the function is NO WORSE than $\theta(n^2)$.
- It might even be BETTER than $\theta(n^2)$, e.g., $\theta(n \log n)$ or $\theta(n)$.
- If an algorithm is $\theta(n^2)$, we say it is $O(n^2)$, $O(n^3)$, $O(n^4)$, …

"Asymptotic Worst-Case Time Complexity"

- Asymptotic = "as n gets large"
- Worst-case = "how many guesses in the worst case?"
- Time complexity = "how much 'time' is consumed?"
  - Giving accurate "to-the-millisecond" characterizations not possible because algorithms can run on different computers
  - Time complexity instead refers to "number of basic operations"
  - Basic operation for search game = guess
- If n is a number, then f(n) is a function on that number (e.g. f(n) = 3n).
  - If an algorithm has worst-case processing time of f(n) for input size is n, then it is said to have asymptotic worst-case time complexity O(f(n)) ("big-oh of f(n)")
  - Linear search is O(n)
  - Binary search is $O(\log_2 n)$

Analyzing Asymptotic Worst-Case Time Complexity: Code

```
public static int linSearch (String[] a, String elt) {
  for (int i = 0; i < a.length; i++)
    if (elt.equals(a[i])) return i;
  return -1; // return -1 if elt is not in a
}
```

- Analysis based on coarse assumptions
  - Comparisons are “one operation”
  - Assignments are one operation
- What about `linSearch`?
  - Loop executes n times, where n is number of elements in `a`
  - Each loop iteration:
    - Comparison involving `i`
    - Comparison involving `elt`, `a[i]`
    - One assignment to `i`
    - So three operations per loop iteration
  - So worst-case time complexity is: O(3n)
- By convention, constants omitted, so O(n)

What about binSearch?

```
public static int binSearch (String[] a, String elt) {
  int lower = 0;                                  // First possible position of elt
  int upper = a.length;                           // First impossible position of elt
  int middle;                                     // Middle position between lower, upper
  while (lower < upper) {
        middle = (lower + upper) / 2;
        if (elt.compareTo(a[middle]) == 0)
          return middle;
        else if (elt.compareTo(a[middle]) < 0)
          upper = middle;
        else
          lower = middle + 1;
  }
  return -1;          // Element not found
}
```

- Loop iteration analysis
  - One assignment to `middle`
  - One assignment to `lower` or `upper` (not both!)
  - Three comparisons
  - So five operations / iteration
- How many iterations of loop in worst case?
  - Each iteration: half of possible values discarded
  - So: $\log_2 n$
- Thus `binSearch` code is $O(5 \log_2 n) = O(\log_2 n)$