CS100M/CIS121/EAS121

Introduction to Computer Programming

Spring 2004

Lecture 12

MATLAB: Searching

Announcements

- Prelim 1 was handed back in section
  - leftovers in Carpenter B101
  - regrades? see Syllabus, deadline next Thurs
- Prelim 2 on Thurs March 18
  - wait until next week for announcement on conflicts
- Java coming up: see lecture schedule on-line

Motivation

- Practice using functions
- A3:
  - different ways to solve the same problem
  - which to choose? how to choose?
- Ideal Considerations:
  - easy to implement
  - easy to understand
  - easy to modify
  - runs fast
  - runs correctly
  - do all approaches share these features?

Focus on Searching/Sorting

- Basic introduction to study of algorithms
  - CS majors: study of construction and analysis
  - others: understanding of which to use and why
- ***Searching***:
  - look for something in data
- ***Sorting***: (saving for Java)
  - organize data in a certain fashion (ascend, descend,...)
  - helps make searching easier (how? you will see)
- Connection to A3?
  - searching for root on numberline! (continuous problem)
  - in lecture, we'll show the discrete version

Naïve Searching

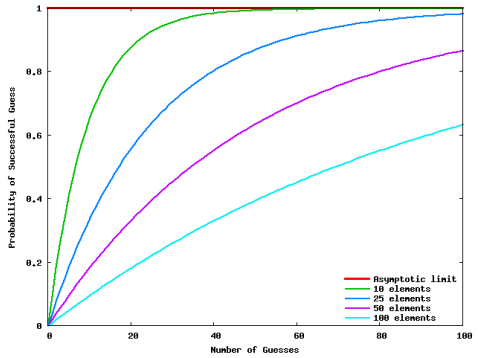
- Algorithm:
  Get collection of data.
  Randomly pick an element.
  Check if correct.
  If not correct, repeat.
- Code Example:

```
% RANDOMSEARCH
low = 0; high = 10; size = 5; target = 7;
data = randIntVec(low,high,size);
guess = readInt(low,high,'Guess! ');
while guess ~= target
   disp('Wrong!');
   guess = readInt(low,high,'Guess! ');
end
disp('Right!');
```


Analysis

- Design:
  - pretty simple
  - easy to understand
- Efficiency?
  - bad! why? how long could it run?
  - Probability of guessing in one try... $\frac{1}{n}$
  - Probability of **not** guessing in one try... $\frac{n-1}{n}$
  - Probability of **not** guessing in **k** tries... $(\frac{n-1}{n})^k$
  - Probability of guessing in at most **k** tries... $1-(\frac{n-1}{n})^k$

How to Improve?



control how and what is guessed
cover entire collection (don't skip anything)

Linear Search

- Easiest way to control search and still cover everything
- Algorithm:
  Get collection of data.
  Get first element.
  If element is target, stop.
  Else, get next element, and repeat.
- Shorter:
  For each element, starting from first,
  check if current element equal to target.
  If found, stop.

Implementation

- Inside another function: use **`while`**

  get initial element
  found ← false
  while not found
  **   if found, found ← true**
  **   else get next element in collection**

  (implementation left as self-exercise)

- As a function: use **`for`**

  for each element
  **   if equal to target, return true**
  **   else, keep searching**
  return false

  (see **`linearSearch.m`**)

Using/Testing Linear Search

- Examples:

```
>> linearSearch(3, [1 2 3])

>> result = linearSearch(3, [1 2 3])

>> linearSearch(3, [1])

>> linearSearch(3, [])

>> linearSearch(3, randIntVec(0,5,10))
```


Analysis

- Think number guessing
- Worst case is number of elements in data
  - eg) guess from 1 to 100
  - could take 100 guesses!
- Time to solve: called ***linear time***
  - doing each element from start to finish, one at a time
  - each element takes about the same amount of time to check
- Efficiency? Can we do better?

Binary Search

- Motivation/The gist
  - think number guessing
  - best place to guess is always in middle!
  - 8 guesses in worst case if assume rounding down (target=100):
    - start with [1, 100]: guess 50 (too low!)
    - next interval [50, 100]: guess 75 (too low!)
    - next interval [75, 100]: guess 87 (too low!)
    - next interval [87, 100]: guess 93 (too low!)
    - next interval [93, 100]: guess 96 (too low!)
    - next interval [96, 100]: guess 98 (too low!)
    - next interval [98, 100]: guess 99 (too low!)
    - 100!
  - using linear search, it's 100!

Development

- The gist:
  start with data:
  low (L), high (H), middle (M), target (T)
- Need to consider 3 cases:
  - case of M > T: L < T < M < H (high gets mid)
  - case of M < T: L < M < T < H (low gets mid)
  - case of M = T: done!
- Example (use integers and floor function):
  - Let L=1, H=10, and T = 7
  - To find T, pick M=(H+L)/2
  - Since M=5 < T, make new L=M=5.
  - Find new M=(5+10)/2=7! It worked!

Formal Algorithm

- Basic Algorithm:
  Get data; assign endpoints (low, high); determine middle;
  Check if middle is target
  if middle is too high, middle becomes new high
  else if middle is too low, middle becomes new low
  determine new middle
  Repeat
- What to do if initial interval too small?
  - eg, find 11 in 0:10
  - need to see if L eventually crosses H
  - use indices instead of actual values in array
  - see next slide...

Indices

- Suppose T=11 and D=0:10
- L=0, H=10. Since M=5<T, L=5+1=6
  (if we make L=5, the floor operation will never let us make L > H!)
- L=6, H=10. Since M=8<T, L=8+1=9
- L=9, H=10. Since M=9<T, L=9+1=10
- L=10, H=10. Since M=10<T, L=10+1=11
- L > H, so stop. We didn't find 11 in 0:10. Good!

Modified Binary Search

- Don't change values, change the indices
- New algorithm:
  Get 1st and last indices (left, right or low, high)
  If left <= right, interval might contain target
  find middle index (mid)
  if target equal to data at mid, success!
  else if target > data(mid) (target to right of mid)
  left ← mid + 1 (need index to creep right)
  else if target < data(mid) (target to left of mid)
  right ← mid – 1 (need index to creep left)
  Repeat

Implementation/Examples

- see **binarySearch.m**
- examples:

```
>> binarySearch(3, [1 2 3])

>> binarySearch(1, [1 2 3])

>> binarySearch(11, 1:10)

>> binarySearch(0, 1:10)
```


Analysis of Binary Search

- We start with an array of length N;
- At every step the length of the array is halved;
- We stop when the interval is of length 1 (if element is found), or 0 (if element is not found).
- The total number of steps k is such that $2^k = N$
  Thus $k = \log(N)$
  Note: we ignore some rounding issues here.
- **Approximate** number of steps for a successful search:

N	100	1000	10000	100000	1000000
K (approx)	7	10	14	17	20

Sorting

- What happens if data is not sorted?
  - `binarySearch(3, [3 4 1 1 2 4 2])`
- Binary Search assumes sorted order!
- How to sort? New problem...
- For now, use MATLAB's **sort**
- A3:
  - LHS/RHS similar to ... ?
  - Bisection similar to ... ?
  - Bisection does not require sorting because real number line has property of ... ?

MATLAB Searching

- **help find**

```
FIND   Find indices of nonzero elements.
   I = FIND(X) returns the indices of the vector X that are
   non-zero. For example, I = FIND(A>100), returns the indices
   of A where A is greater than 100. See RELOP.
   [I,J] = FIND(X) returns the row and column indices of
   the nonzero entries in the matrix X.  This is often used
   with sparse matrices.
   [I,J,V] = FIND(X) also returns a vector containing the
   nonzero entries in X.  Note that find(X) and find(X~=0)
   will produce the same I and J, but the latter will produce
   a V with all 1's.
   See also SPARSE, IND2SUB.
```

MATLAB Sorting

- **help sort**

```
SORT   Sort in ascending order.

For vectors, SORT(X) sorts the elements of X in ascending
  order. For matrices, SORT(X) sorts each column of X in
  ascending order. For N-D arrays, SORT(X) sorts the along the
  first non-singleton dimension of X. When X is a cell array of
  strings, SORT(X) sorts the strings in ASCII dictionary order.

SORT(X,DIM) sorts along the dimension DIM.

[Y,I] = SORT(X) also returns an index matrix I. If X is a
  vector, then Y = X(I).  If X is an m-by-n matrix, then
  for j = 1:n, Y(:,j) = X(I(:,j),j); end

When X is complex, the elements are sorted by ABS(X).  Complex
  matches are further sorted by ANGLE(X).

When more than one element has the same value, the order of
  the elements are preserved in the sorted result and the
  indexes of equal elements will be ascending in any index
  matrix.
```