Searching and Sorting I

Simple “linear” algorithms for searching
Insertion Sort algorithm
More Ruby methods for strings and arrays

Reading: Searching and Sorting tutorial (download from class web site)

There are many situations in every day life where we need to find something
- look for a book, either on a bookshelf at home or in a library
- find a name in a phone book or a word in a dictionary
- search a file drawer to find customer information or student records

What these problems have in common:
- we have a large collection of items
- we need to search the collection to find a single item that matches a certain condition (e.g. name of book, name of a person)

Important idea: how the collection is organized will influence how we carry out the search
- if a bookcase is a haphazard collection we have to look at random or systematically scan each shelf
- if things are sorted or carefully cataloged we can look them up

Not all searches fit this “paradigm” of looking for a specified item
- Perform a search using different information
  - files may be organized by student name, but we need to search by file contents, e.g. find students whose advisor is Prof. X
- Search for items matching a general description
  - select all customer records where the customer’s name starts with “A”
  - find all customers with overdue accounts
- “I’ll know it when I see it” searches
  - look for a good book to read (bookcase, bookstore, library, ...)
  - scan a menu at a restaurant
Search Algorithms

- Searching is one of the most common jobs for a computer
  - on-line dictionaries and catalogs
  - almost every application has a "search" or "find" menu option

As in real life, there are many variations and many different types of searches
- search a list to see if it contains a specified number
- search a list to find the largest number
- find all strings in a list that start with "a"
- find all strings that match a pattern (e.g. numbers like \(\text{xx-xxx-xxxx}\))
- search an abstract "space" of solutions (e.g. best move in chess or shortest tour)
- machines can be taught to scan images for interesting items (no exact representation of the item to search for)
- "I'll know it when I see it" -- "data mining" for interesting or unusual patterns

We'll return to algorithms for pattern-based searches and abstract searches later in the term

Linear Search

- The simplest search algorithm is known as linear search
  - As the name implies, the strategy is to start at the beginning of a collection and compare one by one

Some terminology:
- the item we are looking for is known as the key
- this type of search is also sometimes called a scan
- if the key is not found the search fails

Aside: Conditional Expressions

- To do a search in Ruby we need what is known as a conditional expression
  - Ruby allows us to include a modifier with any statement
    - the modifier consists of the keyword if followed by an expression that evaluates to true or false
  - There are many ways to write conditions -- we'll see others later
Linear Search in Ruby

- It's easy to write a method that tells us if an array contains a specified value.
- We can use the `each` iterator to check every item in the array.

```ruby
def search?(a,k)
a.each { |x| return true if x == k }
return nil
end
```

... if the key matches an item in the array

A convention in Ruby: if a method name ends with `?` the return value will be either `true` or `false`.

Let's try it out on an array of strings:

```ruby
load "linear.rb"

a = ["8", "K", "9", "2", "A", "2", "K"]

search?(a,"A") => true
search?(a,"Q") => nil
```

The search methods shown in this lecture are all in a file named `linear.rb`.

Array Indices

- In many algorithms we want to know where an item was found.
- Instead of simply returning "true" or "false" we want a method to say something like "the thing you are looking for is in the second location in the array".
- An array location is known as an address or an index.
- Computer scientists start counting at 0.
- If an array has \( n \) items the addresses are 0 to \( n - 1 \).

```
8  K  9  2  A  2  K
0  1  2  3  4  5  6
```

Each `with_index`

- Ruby has a version of the `each` iterator that will tell us where an item can be found.

```ruby
a = ["apple", "lime", "kiwi", "orange", "ugli"]

(a.each_with_index) { |x,i| puts "#{x} is at address #{i}" }
apple is at address 0
lime is at address 1
kiwi is at address 2
orange is at address 3
ugli is at address 4
```

each_with_index "walks through" the array; an item is put in \( x \) and its address is put in \( i \).
Linear Search in Ruby Revisited

Here is a slightly different version of the Ruby method for linear search:

```ruby
def search(a,k)
a.each_with_index { |x,i| return i if x == k }
return nil
end
```

```ruby
> search(a,"kiwi")
=> 2
> search(a,"banana")
=> nil
```

This version of the method is named `search` instead of `search?`. It returns `nil` if the item is not found.

Index Expressions

Referring to an individual item in an array is very common if `a` is an array the expression `a[i]` means “the item at address `i` in the array” pronounced “a sub i”, from the mathematical notation `a_i`.

```ruby
>> a = ["apple", "lime", "kiwi", "orange", "ugli"]
>> a[3]
=> "orange"
>> a[3] = "tangerine"
>> a
=> ["apple", "lime", "kiwi", "tangerine", "ugli"]
>> a[5]
=> nil
```

Aside: the `index` Method

Searching is such a common operation that it comes built in to Ruby

If `a` is an array object, we can call a method named `index` to do a search through the array.

```ruby
>> a
 => ["apple", "lime", "kiwi", "orange", "ugli"]
>> a.index("kiwi")
 => 2
>> a.index("banana")
 => nil
```

Other types of objects have index methods, too:

```ruby
>> s = "supercalifragilisticexpialidocious"
>> s.index("list")
 => 14
```

Question: What About Duplicates?

What will the linear search algorithm do if an array contains duplicates?

```ruby
>> a = ["8","K","9","2","A","2","K"]
```

```ruby
>> search(a,"2")
 => 3
```

A: the algorithm returns the index of the first match.

The `index` method behaves the same way:

```ruby
>> a.index("2")
 => 3
```
Searching for the Largest Item

Another type of linear search is one that looks for the largest item in an array.

- Initialize a “place holder” variable \( x \).
- Compare each item in the array to \( x \).
- When any item is greater than \( x \), update \( x \) to this new value.

```
Searching for the Largest Item
```

```
x = 0

x = 3

x = 7

x = 7
```

A Ruby method that implements this form of linear search:

```
def max(a)
  x = 0
  a.each { |n| x = n if n > x }
  return x
end
```

```
17 3 7 6 2 9 2 5

18

17 3 7 6 2 9 2 5
```

Questions

- What do you think will happen in these examples?

```
>> load "linear.rb"
=> true

>> a = [ ]
=> []

>> 10.times { a << rand(10) }
=> 10

>> a
=> [0, 6, 2, 0, 8, 9, 9, 4, 5, 4]

>> max(a)
=> 9
```

```
A: since all the items are negative (less than 0) the method never found a value greater than 0 and returned 0.
```

```
>> a = ["apple", "lime", "kiwi", "orange", "ugli"]
>> max(a)
ArgumentError: comparison of String with 0 failed
```

```
A: the method assumed all the array values would be integers when it set the initial value of \( x \) to 0.
```
An Improved \textit{max}:

- Here is a solution that fixes both the problems shown on the previous slide:
  - initialize the place holder variable \(x\) to the first item in the array
  - start the iteration at the second location

\begin{itemize}
  \item \(x: \begin{array}{c}
  5 \\
  4 \\
  7
  \end{array}
  \begin{array}{c}
  3 \\
  7 \\
  6 \\
  2 \\
  9 \\
  2 \\
  5
  \end{array}
  \begin{array}{c}
  7 \\
  3 \\
  7 \\
  6 \\
  2 \\
  9 \\
  2 \\
  5
  \end{array}
  \)
\end{itemize}

\textbf{for Loops}:

- How do we start the iteration at the second location in the array?
  - the \textit{each} iterator always starts at the first location

- Here is a new Ruby construct that does what we want:
  \begin{verbatim}
  for i in j..k
    # any expression(s)
  end
  \end{verbatim}

- This is what is known as a “for loop”
  - it’s like a combination of \textit{each} and \textit{while}
  - the variable \(i\) is set to every value from \(j\) to \(k\)
  - after assigning a value to \(i\), Ruby executes the statements in the body of the loop

\textbf{Improved \textit{max} Using a \textit{for} Loop}:

- Here is the new definition of \textit{max}:
  \begin{verbatim}
  def max(a)
    x = a[0]
    for i in 1 .. a.length-1
      x = a[i] if a[i] > x
    end
    return x
  end
  \end{verbatim}

\textbf{Note}: the file named “linear.rb” has both versions of \textit{max}.
The method called \textit{max0} is the original version, which initializes \(x\) to 0.
The method named \textit{max} is the one shown here.
**Improved max Using a for Loop**

- Trying out the new version:
  ```ruby
global $a = ['apple', 'lime', 'kiwi', 'orange', 'ugli']
global $maxa
>>> $max0($a)
ArgumentError: comparison of String with 0 failed
>>> $max($a)
=> 'ugli'
```

- The new version works because Ruby knows how to compare two strings
  - It will actually work for any type of object (e.g., floats)

**Extra Credit Challenge**

- Rewrite the new version of max so that it uses a while loop instead of a for loop
  ```ruby
def max($a)
  $x = $a[0]
  $i = 1
  while ??
    ??
  end
  return $x
end
```

**Sorting**

- The search algorithms shown on the previous slides are examples of **linear** algorithms
  - Start at the beginning of a collection
  - Systematically progress through the collection, all the way to the end if necessary
- A similar strategy can be used to sort the items in an array
- The next set of slides will introduce a very simple sorting algorithm known as **insertion sort**

**Basic Idea:** Pick up an item, find the place it belongs, insert it back into the array

**Move to the next item and repeat**

**Insertion Sort**

- The important property of the insertion sort algorithm: at any point in this algorithm, **part of the array is already sorted**
- The item we currently want to find a place for will be called the **key**
  - Items to the left of the key are already sorted
  - The goal on each iteration is to insert the key at its proper place in the sorted part
- Example (shown at right):
  - When it is time to find a place for the J in this hand, the portion to the left is sorted
Insertion Sort

- Here is a more precise statement of the insertion sort algorithm:
  1. The initial key is the second item in the array (the Q in this example).
  2. Use your left hand to pick up the key.
  3. Scan left until you find an item lower than the one in your left hand, or the front of the array, whichever comes first.
  4. Insert the key back into the array at this location.
  5. The new key is the item to the right of the location of the previous key.
  6. Go back to step 2.

- This new version is precise enough that we can organize a Ruby method that will implement this algorithm.

Insertion Sort in Ruby

- Here is the insertion sort method, with a few things still left to fill in:

```ruby
def isort(a)
  for i in 1 .. a.length-1
    key = a[i]
    remove key from a
    j = location for key
    insert key at a[j+1]
  end
  return a
end
```

Example:
- When i = 3 key will be "J"
- j will be set to 0 (since a[0] = "9"
- "J" will be inserted at location 1

A description like this that is part Ruby and part English is known as pseudocode.

Pseudocode vs Real Code

- When algorithms require more than a few lines of Ruby I will use pseudocode in the lecture slides.
- The algorithms have all been fully implemented.
  - You can download them from the class web site.
  - You are welcome to look at the Ruby code if you want to see the "gory details."

```ruby
def isort(a)
  for i in 1 .. a.length-1
    key = a[i]
    remove key from a
    j = location for key
    insert key at a[j+1]
  end
  return a
end
```

See: isort.rb

Insertion Sort Example

- The following pictures show an example of how insertion sort works (using a list of numbers instead of cards).
  - Suppose the first 3 positions (0 through 2) have been sorted:
    - The first statements in the body of the loop set key to 7 and remove it from a:
      ```ruby
      a = [6, 12, 14, 4]
      key = 7
      a = remove key from a
      ```
    ```ruby
      j = 1
      while j >= 0 && a[j] > key
        j = j-1
      end
      a.insert(j+1, key)
    ```
    ```ruby
    a = [6, 12, 7, 4]
    ```
    - The first statements in the body of the loop set key to 7 and remove it from a:
Insertion Sort Example

- The algorithm looks to the left for a location to put the 7
- j is set to 0 since a[0] is the first item smaller than 7

```
def isort(a)
  for i in 1 .. a.length-1
    key = a[i]
    remove key from a
    j = location for key
    insert key at a[j+1]
  end
  return a
end
```

```ruby
def isort(a)
  for i in 1 .. a.length-1
    key = a[i]
    remove key from a
    j = location for key
    insert key at a[j+1]
  end
  return a
end
```

Since j = 0 key is inserted at a[1]

The next iteration will continue with i = 4

```
key
a

def isort(a)
  for i in 1 .. a.length-1
    key = a[i]
    remove key from a
    j = location for key
    insert key at a[j+1]
  end
  return a
end
```

```
key
a
```

```
def isort(a)
  for i in 1 .. a.length-1
    key = a[i]
    remove key from a
    j = location for key
    insert key at a[j+1]
  end
  return a
end
```

Sort Demo

- There are lots of web sites with animated demonstrations of different sorting algorithms
- Here is one with a Java “applet” that will run in a browser:

```
http://math.hws.edu/TMCM/java/xSortLab
```

View the insertion sort demo today -- we’ll come back later to watch demos for mergesort and quicksort

Nested Loops

- At first glance it might seem that insertion sort is a “linear” algorithm like search and max
  - it has a for loop that progresses through the array from left to right
- But it’s important to note what is happening inside the loop
  - the step that finds the proper location for the current item is also a loop
  - it scans left from location i, going all the way back to 0 if necessary
- An algorithm with one loop inside another is said to have nested loops
Nested Loops

- The diagram at right helps visualize how many steps the `isort` method takes.
  - A dot in a square indicates a potential comparison.
  - For any value of `i`, the inner loop might have to compare key to values from `i - 1` all the way down to 0.
- The number of dots in this diagram is `(6 × 5)/2 = 15`.
- In general, for an array with `n` items, the number of comparisons is
  \[(n × (n - 1))/2 = n^2/2\]

Summary

- This set of slides introduced two types of new algorithms based on iteration.
- **Simple linear search** algorithms scan an array from left to right.
  - Methods that implement these algorithms can use the each iterator or a `for` loop.
- A straightforward sorting algorithm known as **insertion sort** also involves a scan from left to right.
  - An important difference: there is a second loop inside the main loop.
  - The inner loop scans back from right to left to find the place for an item.
- New “technology” introduced for this topic:
  - `for` loops
  - Conditional execution (modifiers with the keyword `if`)
  - Pseudocode

What You Should Know

- The first part of the lab will be on the algorithms presented here.
  - The second half will be on new searching and sorting algorithms presented next week.
- The tutorial project will have instructions for downloading and running the `search`, `max`, and `isort` methods.
- The version of `isort` in this file will have options for printing the array as the algorithm progresses.
  - You will be able to see the sorted portion grow on each iteration.
- Lab questions will test your understanding, e.g., ask you to write out what the sort method would print for a given input list.