* Searching and Sorting

Reading

The text for this chapter has not been written yet. You can find all the information you need for this project in the lecture notes. “Searching and Sorting I” discusses algorithms based on a straightforward iteration (linear search and insertion sort). “Searching and Sorting II” presented divide and conquer algorithms: binary search, QuickSort, and MergeSort.

Tutorial Project

Before you begin this project you need to download two files. Methods that search through arrays are in `searchlab.rb`:

- `search(a,x)` does a simple linear search through array `a` and returns the index of the item `x` if it is in the array, or `nil` if `x` is not in `a`.
- `max(a)` returns the largest item in `a`.
- `bsearch(a,x)` does a binary search through `a`, returning either the index where `x` is found or `nil` if `x` is not found.

Methods that sort arrays are in `sortlab.rb`:

- `isort(a)` returns a copy of array `a`, sorted using the insertion sort algorithm.
- `qsort(a)` same as above, but using QuickSort.
- `msort(a)` same, but using MergeSort.

Both files also have a method that can be used to create sample data for testing the searching and sorting algorithms:

- `randoms(n,m)` makes a list of `n` random numbers. The second parameter `m` is optional; if you use it, the values in the list will be between 0 and `m - 1`.

Note: The methods in `searchlab.rb` and `sortlab.rb` have all been extended with Ruby code that is useful for running experiments. For example, you can pass an extra parameter that tells a method to print a detailed trace at each step in the algorithm, or to record the
amount of time it takes to search or sort an array. If you want to look at the Ruby code to see how something has been implemented it would be best to download the original unmodified code shown in the lecture slides. These methods can be found in files named for each method, e.g. the QuickSort algorithm is in qsort.rb.

**Random Lists**

The first thing is to explore the methods that make lists of random numbers to use in the experiments.

1. Load the file with search algorithms:

   ```ruby
   >> load "searchlab.rb"
   => true
   ```

2. The `randoms` method makes a list of random numbers. The simplest way to call this method is to pass the size of the array you want:

   ```ruby
   >> a = randoms(10)
   => [18, 52, 8, 17, 77, 61, 32, 52, 47, 72]
   ```

   Obviously, since the method uses a random number generator, the actual array you get won’t be the same as the one above, but you should see a list of 10 numbers between 0 and 99.

3. If you pass a second parameter to `randoms` you will get a list of numbers between 0 and that second parameter. Make a list of numbers between 0 and 9:

   ```ruby
   >> a = randoms(10,10)
   => [9, 6, 6, 3, 8, 4, 2, 0, 6, 3]
   ```

4. Since the `randoms` method was written to test searching and sorting algorithms it tries to minimize the odds of duplicate values. If you don’t pass an upper limit, `randoms` chooses one for you, based on the size of the array you want to build. The default upper limit is 10 times the size of the array: if you ask for an array of 100 items, they will be between 0 and 999, and if you ask for an array of 1,000,000 items they will be between 0 and 9,999,999. Test this by asking for an array of 100 items:

   ```ruby
   >> a = randoms(100)
   => [772, 968, 173, ... ]
   ```

5. When we start dealing with very large arrays the terminal window will be overcrowded if we let Ruby print the arrays built by `randoms`. There is a trick to prevent Ruby from printing the array. We can ask Ruby to evaluate two expression by writing the first expression, a semicolon, and the second expression. Ruby evaluates both, but only prints the value of the second. Try this out by defining two variables with a single line of input to IRB:

   ```ruby
   >> x = 7; y = 11
   => 11
   ```

6. Note that Ruby only printed the value of the second expression. Verify that it in fact evaluated both expressions by asking for the value of `x`:

   ```ruby
   >> x
   => 7
   ```

7. If you want to use this trick when you’re making very big arrays, enter a line that has a call to `randoms` followed by the word nil, e.g.

   ```ruby
   >> a = randoms(1000); nil
   => nil
   >> a.length
   => 1000
   ```
8. The slides on iterative search algorithms introduced index expressions. To see what is at a particular location in an array, type the name of the array and then the location enclosed in square brackets. To see the first element in the new array of 1,000 numbers type this:

   >>> a[0]
   => 5935

   (once again, the actual value you see will be different).

9. Something not shown in the lecture slides is that Ruby allows us to access a range of items in a single expression. If you want to see the first 10 elements in a type this:

   >>> a[0..9]
   => [5935, 7343, 7296, 3281, 8965, 1658, 4293, 9451, 6144, 4520]

   An expression like a[i..j] means “make a new array that contains items i through j of array a.”

   The `randoms` method is just what we want for testing sort algorithms, but for experiments with searches we need to know what’s in an array so we can know when the search should succeed and when it should fail. An operation that will come in handy for testing searches is known as permutation. A random permutation basically “shuffles” an array to put items in a new order. The Ruby version is a method named `permute` (defined in `searchlab.rb`). A call to `permute(a)` will return a copy of a with the elements in random order.

10. We’ve seen “range operators” several times now. The following expression shows that ranges are actually objects in Ruby, and that we can call a method named `to_a` to turn a range into an array. To create a list of numbers from 1 to 100:

   >>> (1..100).to_a
   => [1, 2, 3, ... ]

11. Type this expression to make a list of numbers from 1 to 100 and then pass it to the `permute` method to scramble it:

   >>> permute((1..100).to_a)
   => [29, 43, 99, ... ]

   Type the expression above a few more times. Do you get a different ordering each time?

### Conditional Expressions

The slides on iterative algorithms introduced the idea of conditional expressions. The simplest type of conditional is made by adding a modifier to the end of any other expression. Ruby will first evaluate the condition, and if it is true the modified expression is evaluated.

12. Make a list of 10 random numbers:

   >>> a = randoms(10)
   => [68, 21, 63, 20, 22, 94, 32, 96, 50, 95]

13. Use the `each` iterator to print every item in a:

   >>> a.each { |n| puts n }
   68
   21
   63
   20
   ...

14. Repeat the previous expression, but this time add an `if` modifier to the print statement, so that Ruby prints an item only if it is an even number:
>> a.each { |n| puts n if n % 2 == 0}  
68  
20  
...  

Look carefully at the output from your last two expressions. Did the first one print all 10 items in your test array? Did the second one print only the even values?

### Breaking Out of Iterations

**Note:** This section is optional, for those who want to explore in more depth how Ruby iterators work and how the search algorithm breaks out of an iteration when it finds what it is looking for. You can skip this section and continue with the exercises on Linear Search.

15. For the tests in this section make an array that has some known values:  
   ```ruby  
   >> evens = [2,4,6,8,10]  
   => [2, 4, 6, 8, 10]  
   ```

16. One way to stop the iteration early is to put a “break” statement in the loop. If Ruby ever evaluates an expression of the form `break x` it stops the iteration and uses the value of `x` as the value of the call to `each`. This call to `each` should result in `true` since the number 6 is in the array:  
   ```ruby  
   >> evens.each { |n| break true if n == 6 }  
   => true  
   ```

17. If the iterator makes it all the way through the array without evaluating the `break` statement it means the number is not in the list:  
   ```ruby  
   >> evens.each { |n| break true if n == 7 }  
   => [2, 4, 6, 8, 10]  
   ```

Notice how Ruby printed the usual result of a call to `each`, which is just the contents of the array.

The loop shown above is used in the method named `search?` that was shown in the lectures slides (you can also see it if you look in the file `search.rb`). The only difference is that the method uses the word `return` to exit the loop early and return the specified value as the value of the method call.

### Linear Search

18. Make a small array with some known values:  
   ```ruby  
   >> colors = ["red","white","blue","green","yellow"]  
   => ["red", "white", "blue", "green", "yellow"]  
   ```

19. The method named `search?` (note the question mark at the end of the name) will return `true` or `false`, depending on whether the parameter it is passed is in the array:  
   ```ruby  
   >> search?(colors, "green")  
   => true  
   >> search?(colors, "orange")  
   => false  
   ```

Type a few more expressions with calls to `search?` to make sure you understand what this method does.

20. The method `search` (without the question mark) returns the location where an item can be found, or `nil` if the item is not there:

© 2008 John S. Conery. Please do not distribute without permission.
>> search(colors, "green")
=> 3
>> search(colors, "orange")
=> nil

21. Use the `permute` method to make a randomly ordered list of numbers between 1 and 100:
   ```ruby
   >> a = permute((0..99).to_a)
   => [10, 36, 86, ... ]
   ```

22. Type a few expressions that will call `search` to look for numbers in `a`. Test the method both by looking for numbers that are in the array and some that aren’t (e.g. numbers greater than 100).

   The `search` method accepts extra parameters that tell it what to do. If you pass it the word `:count` (note the colon at the front of this word) the method will return the number of comparisons it made.

   23. Find out how many comparisons are necessary to find the string "green" in the list of colors:
       ```ruby
       >> search(colors, "green", :count)
       => 4
       ```
       Is the result what you expected?

   24. How many comparisons do you think `search` will make in an unsuccessful search? Check your answer:
       ```ruby
       >> search(colors, "orange", :count)
       => 5
       ```

### Binary Search

The next set of experiments will use the binary search algorithm, implemented by the method `bsearch` (note the `b` at the front of the name).

25. Remember that binary search only works if the array being searched is sorted. Make a small sorted array of random numbers to test `bsearch`:
   ```ruby
   >> a = randoms(15,100).sort
   => [17, 22, 28, 32, 39, 41, 42, 48, 54, 67, 86, 89, 96, 97, 99]
   ```
   As before, the actual array you get will be different. Note the call to the `sort` method immediately after the call to `randoms`, with no space; this tells Ruby to call `randoms` to make a new array and then call that array's `sort` method before storing the result in `a`.

26. The `bsearch` method has an optional parameter named `:trace` that will print the state of the search at each iteration. Type this expression to see how `bsearch` searches for the number 41:
   ```ruby
   >> bsearch(a, 41, :trace)
   [ 17 22 28 32 39 41 42 *48 54 67 86 89 96 97 99 ]
   [ 17 22 28 *32 39 41 42 ] 48 54 67 86 89 96 97 99
   17 22 28 32 [ 39 *41 42 ] 48 54 67 86 89 96 97 99
   => 5
   ```
   At the start of each round, the algorithm prints the array, adding brackets around the region that has yet to be searched. The first line shows the initial region is the entire array. The asterisk marks the location of `mid`, a point halfway through the region. Since the number we’re looking for (41) is less than the number at `mid` the new region is the left half of the array, as shown on the second line. On the third step the new region is again half as big. This time the algorithm found what it was looking for in the middle of this region. The return value of 5 means 41 was found at `a[5]`.

© 2008 John S. Conery. Please do not distribute without permission.
27. Redo the previous test, but this time look for a number that is not in the array:

```ruby
>> bsearch(a, 26, :trace)
[ 17 22 28 32 39 41 42 48 54 67 86 89 96 97 99 ]
[ 17 22 28 *32 39 41 42 ] 48 54 67 86 89 96 97 99
[ 17 *22 28 ] 32 39 41 42 48 54 67 86 89 96 97 99
17 22 *[] 28 32 39 41 42 48 54 67 86 89 96 97 99
17 22 *[ ] 28 32 39 41 42 48 54 67 86 89 96 97 99
=> nil
```
Notice how the last line has brackets around nothing, meaning the search failed. If 26 was in this list it would be between 22 and 28.

28. Redo the previous two tests, this time passing the :count option instead of trace:

```ruby
>> bsearch(a, 41, :count)
=> 3

>> bsearch(a, 26, :count)
=> 4
```
In the trace above you can see that 41 was found on the third comparison, so the first result here looks right. The trace for the search for 26 has 5 lines, but `bsearch` doesn't count the last line (there is nothing there to compare) so that's why the result is 4 for this unsuccessful search.

All the tests and exercises to this point have been leading up to the following experiments, which are intended to emphasize the difference in performance between linear search and binary search.

29. Make the two arrays that will be used for the experiments. We'll want the sorted list of numbers from 0 to 99 for binary search, and a randomized set of the same numbers for linear search:

```ruby
>> sorted = (0..99).to_a
=> [0, 1, 2, 3, 4, ... ]

>> unsorted = permute(sorted)
=> [23, 21, 92, 5, 82, ... ]
```

30. This expression will tell us how many comparisons are made when looking for a random number (which we know will be in the list because `unsorted` was defined to be a permutation of all the numbers between 0 and 99):

```ruby
>> search(unsorted, rand(100), :count)
=> 25
```

31. We can enclose the above expression in the body of a call to `times` in order to run the experiment several times:

```ruby
>> 10.times { puts search(unsorted, rand(100), :count) }
27
73
81
12
...
```
Make sure you understand what these numbers mean: since the :count option was passed to `search`, the values printed here are the number of comparisons it took to find a bunch of random values.

32. The equivalent test for binary search is to again search for a random item, but to search in the sorted array:
Are these results more or less what you expected? The linear search should make, on average, \( n/2 \) comparisons to search an array of \( n \) items. Is the average of the 10 numbers printed for the call to \texttt{search} near 50? Binary search will make up to \( \log_2 n \) comparisons (or, more precisely, the smallest integer greater than \( \log_2 n \)). What is \( \log_2 100 \) rounded up to the next integer? Is this the most comparisons you saw in the call to \texttt{bsearch}?

\textbf{Note:} if your calculator does not have a function to compute logarithms to the base 2 you can use this formula to figure it out:

\[
\log_2 n = \frac{\log_{10} n}{\log_{10} 2}
\]

Or if you want, download the file \texttt{log2.rb} from the class web site and use Ruby to do the calculation:

\begin{verbatim}
>> load "log2.rb"
=> true
>> log2(100)
=> 6.64385618977473
\end{verbatim}

33. Repeat the previous two expressions, this time calling \texttt{search} and \texttt{bsearch} on arrays of 1,000 numbers:

\begin{verbatim}
>> sorted = (0..999).to_a; nil
=> nil
>> unsorted = permute(sorted); nil
=> nil
>> 10.times { puts search(unsorted, rand(1000), :count) }
583
978
...
>> 10.times { puts bsearch(sorted, rand(1000), :count) }
8
10
...
\end{verbatim}

Note carefully that the argument to \texttt{rand} is 1000, not 100 as it was in earlier experiments.

Did you get results you expected for this second set of runs? What is the average number of comparisons made by \texttt{search}? By \texttt{bsearch}? What is the largest number of comparisons? Is it around \( \log_2 1000 \)?

**Insertion Sort**

34. Load the file containing the definitions of the three methods that sort arrays:

\begin{verbatim}
>> load "sortlab.rb"
=> true
\end{verbatim}

35. Make a small array of numbers that will be used to illustrate the insertion sort algorithm:
6. Calling a sort method with no optional arguments will return a sorted copy of the input parameter:

```
>> isort(a)
=> [0, 7, 9, 10, 22, 33, 46, 63, 66, 86, 87, 88, 94]
```

37. The :timer option will return the number of seconds it took the method to sort the input:

```
>> isort(a, :timer)
=> 8.5e-05
```

The actual value you get will depend on several factors, not the least of which is the processor speed of the system you run the program on. A number like the one shown above is Ruby's shorthand for scientific notation. It stands for \(8.5 \times 10^{-5}\), or 0.000085. Ruby uses this form for very small or very large numbers.

38. Use the :timer method on a bigger array, so you can see what the “normal” output looks like:

```
>> isort(randoms(1000), :timer)
=> 0.139839
```

39. The :trace option tells the method to print the state of the array after each iteration:

```
>> isort(a, :trace)
9 *88 87 10 7 33 22 69 94 46 0 86 66 87 63
9 88 *87 10 7 33 22 69 94 46 0 86 66 87 63
9 87 88 *10 7 33 22 69 94 46 0 86 66 87 63
9 10 87 88 *7 33 22 69 94 46 0 86 66 87 63
7 9 10 87 88 *33 22 69 94 46 0 86 66 87 63
... ...
0 7 9 10 22 33 46 66 69 86 87 87 88 94 *63
=> [0, 7, 9, 10, 22, 33, 46, 63, 66, 69, 86, 87, 87, 88, 94]
```

The asterisk marks the boundary between the sorted and unsorted regions in the array. Notice how at each step the method takes the first unsorted number and inserts into its final position in the sorted region, and that the sorted region grows by one on each iteration.

40. The :count option will return a count of the number of comparisons made when the array was sorted:

```
>> isort(a, :count)
=> 45
```

Here note how even though only 14 iterations were required (one less than the size of the array) the overall number of comparisons is higher. That's because during the later iterations the method may have to do several comparisons as it scans left to find a home for the current item.

41. Let's run the isort method on arrays of size 100, 200, etc. up to 1000. The first step is to make sure we can write an iterator expression that generates these numbers:

```
>> 10.times { |n| puts 100*(n+1) }
100
200
... ...
1000
=> 10
```

Remember that the times method sets \(n\) to 0, 1, ... 9, so \(100*(n+1)\) is the expression we need to turn \(n\) into the numbers 100, 200, ...

42. Now that we know how to create the arguments, just go back and change the body of the iterator so it passes this value as the array size in a call to isort:
The output is the number of comparisons made by isort as it sorts arrays of size 100, 200, etc.

43. Repeat the previous expression, except this time measure the execution time (i.e. change :count to :timer):
   >> 10.times { |n| puts isort(randoms(100*(n+1)), :count) }
   2524
   10619
   ...

   >> 10.times { |n| puts isort(randoms(100*(n+1)), :timer) }
   0.00191
   0.006079
   ...

Save the results of these expressions someplace so you can refer to them after running the same experiments with the other two methods. You can write them on a piece of paper, or cut-and-paste them into a document (e.g. use Microsoft Word to make a table and enter the numbers in the table). If you know how to use a spreadsheet, enter the numbers in separate rows in two separate columns, one for counts and one for rows; later when the other results have been entered you can generate a plot to show the differences.

**QuickSort**

**Note:** This section is optional, for those who want to explore in more depth how recursive methods work. You can skip this section and continue with the exercises on MergeSort.

44. Make a small array to test QuickSort:
   >> a = randoms(10)
   => [52, 26, 65, 12, 40, 74, 59, 13, 59, 55]

45. The method that implements the QuickSort algorithm is named qsort. Call the method with no options to get back a sorted copy of the input:
   >> qsort(a)
   => [12, 13, 26, 40, 52, 55, 59, 59, 65, 74]

46. This method also makes use of optional parameters:
   >> qsort(a, :count)
   => 52
   >> qsort(a, :timer)
   => 2.0e-06

47. The trace for the qsort method shows the region the algorithm is currently working on as a set of numbers surrounded by brackets:
   >> qsort(a, :trace)
   [ 52 26 65 12 40 74 59 13 59 55 ]
   [ 13 26 40 12 ] 65 74 59 52 59 55
   [ 12 ] 26 40 13 65 74 59 52 59 55
   12 [ 26 40 13 ] 65 74 59 52 59 55
   12 13 [ 40 26 ] 65 74 59 52 59 55
   12 13 [ 26 ] 40 65 74 59 52 59 55
   12 13 26 [ 40 ] 65 74 59 52 59 55
   12 13 26 40 [ 65 74 59 52 59 55 ]
   12 13 26 40 [ 55 59 59 52 ] 74 65
   ...

© 2008 John S. Conery. Please do not distribute without permission.
The first line shows that for the top level call the region is the entire array. The pivot is the first item in the array, in this case 52. The method sorts the array into two parts, putting numbers less than 52 on the left side and numbers greater than or equal to 52 on the right side. It then makes a recursive call to sort the left part of the array.

The second line shows how this process repeats: the pivot is now 13, and numbers in the new region less than 13 go to the left of the region and numbers greater than or equal to 13 go to the right side.

The most difficult thing to understand about recursion is that when a recursive call finishes, the method “pops up” to return to the most recent pending call. In this case, after sorting the region that has only the number 12, the program still has some unfinished business – it has to sort the three values from the right side of the second call, but it also has to eventually return to the 6 numbers placed on the right side by the top level call. The program eventually gets around to this item on its “to do list”, as shown on line 9 of the output.

48. Try tracing a few more examples, and experimenting on a slightly larger arrays.

49. Repeat the experiments from the previous section, where you call the method from inside a times iterator to build tables of numbers of comparisons and execution times:

```ruby
>> 10.times { |n| puts qsort(qrandoms(100*(n+1)), :count) }
1004
2288
...
>> 10.times { |n| puts qsort(qrandoms(100*(n+1)), :timer) }
2.0e-06
2.0e-06
...
```

How do these numbers compare to the values for insertion sort?

**MergeSort**

50. Make a small array to test MergeSort. It’s easier to follow the “divide and conquer” strategy if the size of the array is a power of 2, but MergeSort will work for any size array:

```ruby
>> a = qrandoms(16,100)
=> [90, 39, 29, 15, 91, 84, 41, 26, 18, 24, 81, 14, 23, 54, 83, 13]
```

51. The MergeSort method is named `msort`:

```ruby
>> msort(a)
=> [13, 14, 15, 18, 23, 24, 26, 29, 39, 41, 54, 81, 83, 84, 90, 91]
```

52. This method also uses the :count and :timer options:

```ruby
>> msort(a, :count)
=> 45
>> msort(a, :timer)
=> 0.000207
```

53. The trace for the `msort` method uses brackets to surround the groups waiting to be merged:

```ruby
>> msort(a, :trace)
[39 90] [15 29] [84 91] [26 41] [14 24] [14 81] [23 54] [13 83]
[15 29 39 90] [26 41 84 91] [14 18 24 81] [13 23 54 83]
[15 26 29 39 41 84 90 91] [13 14 18 23 24 54 81 83]
[13 14 15 18 23 24 26 29 39 41 54 81 83 84 90 91]
=> [13, 14, 15, 18, 23, 24, 26, 29, 39, 41, 54, 81, 83, 84, 90, 91]
```

The first line shows that the algorithm first forms groups of size 2. Note that each group is sorted – the smaller item has been placed on the left. Next groups of 4 are formed by merging two adjacent 2-item groups. Then two 4-item groups are merged to make groups of...
size 8, and finally the two 8-item groups are merged to make the complete group of 16. The reason this algorithm makes fewer comparisons is that merging is very efficient. At each step in the merge the program just needs to look at the first item in each list – it doesn’t have to look deeper in the list to compare any items there.

54. Try running the algorithm on larger groups. What do you think will happen if you sort an array that has one item longer than a power of 2, e.g. an array of size 17? What about an array of size 15?

55. Use the `times` iterator to make tables of numbers of comparisons and execution times for MergeSort:

```ruby
>> 10.times { |n| puts msort(qrandoms(100*(n+1)), :count) }
551
1309
...
>> 10.times { |n| puts msort(qrandoms(100*(n+1)), :timer) }
0.001541
0.00276
...
```

Save these counts and execution times in the table you started in the section on insertion sort. How do the two algorithms compare? Can you see that the number of comparisons in insertion sort grows in proportion to \( n \times n \), while the merge sort is more like \( n \times \log_2 n \)? If the trend is not apparent, extend your table by running the algorithms a few more times on larger inputs.