Searching and Sorting II

“Divide and Conquer” strategies for searching and sorting
Binary Search
QuickSort
MergeSort

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

Simple Searches

- The previous slides on sorting introduced algorithms that did a “linear scan” through a list
  - to find a particular item: search(a,x)
    - start at the front of a, scan right until x found
  - to find the largest item: max(a)
    - set place-holder to a[0], scan from a[1] to a[n-1], updating place-holder

Simple Sorts

- Those slides also introduced a sorting algorithm that used a similar strategy
- Scan the list from left to right, and for each item x:
  - remove x from the list
  - scan left to find a place for x
  - re-insert x into the list
- This “insertion sort” algorithm has nested loops
  - outer loop is a linear progression left to right
  - inner loop scans back to find a place for x
- The number of comparisons made when sorting a list of n items is as high as
  \((n \times (n - 1))/2 \approx n^2/2\)

Divide and Conquer

- The common theme for the previous slides: linear scan through every item in the list
- There is also a common theme for this set of slides: divide and conquer
  - breaks a problem into smaller pieces and solve the smaller sub-problems
- It may not seem like that big a deal, but the improvement can be dramatic
  - approximate number of comparisons (worst case):

<table>
<thead>
<tr>
<th></th>
<th>search</th>
<th>sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>linear</td>
<td>100</td>
<td>5000</td>
</tr>
<tr>
<td>divide and conquer</td>
<td>7</td>
<td>700</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>10,000</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>500,000</td>
</tr>
</tbody>
</table>
Searching a Dictionary

- To get a general sense of how the new search algorithm works, consider how people find information in a (physical) phone book or dictionary
  - suppose you want to find “janissary” in a dictionary
  - open the book near the middle
  - the heading on the top left page is “kiwi”, so move back a small number of pages
  - here you find “hypotenuse”, so move forward
  - find “ichthyology”, move forward again
- The number of pages you move gets smaller (or at least adjusts in response to the words you find)

Binary Search

- An algorithm for that use this divide and conquer strategy is known as binary search
  - “binary” comes from the fact that the region to search is divided in two on each step
- But: this strategy only works if the list is sorted
  - recall the point raised in the previous set of slides
  - the methods used to access data depend on how the information is organized

Searching a Dictionary

- A detailed specification of this process:
  - the goal is to search for a word \( w \) in region of the book
  - the initial region is the entire book
  - at each step pick a word \( x \) in the middle of the current region
  - there are now two smaller regions: the part before \( x \) and the part after \( x \)
  - if \( w \) comes before \( x \), repeat the search on the region before \( x \), otherwise search the region following \( x \)
- At first your region consists of a group of pages, but eventually a region is a set of words on a single page

Binary Search

- To search a list of \( n \) items, first look at the item in location \( n/2 \)
  - then search the region from 0 to \( n/2-1 \) or from \( n/2+1 \) to \( n-1 \)
- Example: searching for 36 in a sorted list of 15 numbers

<table>
<thead>
<tr>
<th>4</th>
<th>10</th>
<th>22</th>
<th>35</th>
<th>38</th>
<th>57</th>
<th>65</th>
<th>71</th>
<th>79</th>
<th>83</th>
<th>87</th>
<th>89</th>
<th>90</th>
<th>91</th>
<th>93</th>
</tr>
</thead>
<tbody>
<tr>
<td>①</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>②</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>③</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The algorithm uses two variables to keep track of the boundaries of the region to search:
- `lower` - the index **one below** the leftmost item in the region
- `upper` - the index **one above** the rightmost region

| 4 10 22 35 36 57 65 71 79 83 87 89 90 91 93 |

Initial values when searching an array of `n` items,
- `lower = -1`
- `upper = n`

The body of the loop contains these statements:

```python
mid = (lower + upper) / 2
return mid if k == a[mid]
upper = mid if k < a[mid]
lower = mid if k > a[mid]
```

The first iteration when searching for 57 in a list of size 15:

| 4 10 22 35 36 57 65 71 79 83 87 89 90 91 93 |

- `lower = -1`
- `upper = 15`
- `mid = 15 / 2 = 7`
- `upper for next iteration = 7`

This search required only 3 comparisons:

What happens in this loop if the item we're looking for is not in the array?

**Example:** search for 58

- `lower = 3`
- `upper = 7`
- `mid = 5`
- `lower = 5`
- `upper = 7`
- `mid = 6`
- `upper = 6`

This search required only 3 comparisons:

**Unsuccessful Searches**
Unsuccessful Searches

- To fix this problem we have to add another condition to the loop
  - we want the result to be nil if the region shrinks to 0 items
  - this happens when upper equals lower + 1

```ruby
mid = (lower + upper) / 2
return nil if upper == lower + 1
return mid if k == a[mid]
upper = mid if k < a[mid]
lower = mid if k > a[mid]
```

if Statements

- Usually when a program has tests for opposite conditions the test is written in the form of an if statement
  - Instead of
    ```ruby
    upper = mid if k < a[mid]
    lower = mid if k > a[mid]
    ```
    we normally write
    ```ruby
    if k < a[mid]
      upper = mid
    else
      lower = mid
    end
    ```

if Statements

- If there are three conditions we can use elsif (a combination of if and else):
  ```ruby
  if k == a[mid]
    return mid
  elsif k < a[mid]
    upper = mid
  else
    lower = mid
  end
  ```

Binary Search Method

- The full definition of a method that does a binary search of an array `a` to look for an item `x` is shown at right
  - the name is `bsearch` to distinguish it from the search method shown in the previous slides

```ruby
def bsearch(a, k)
  lower = -1
  upper = a.length
  while true
    mid = (lower + upper) / 2
    return nil if upper == lower + 1
    if k == a[mid]
      return mid
    elsif k < a[mid]
      upper = mid
    else
      lower = mid
    end
  end
end
```

Extra credit: Is this an infinite loop??

See `bsearch.rb`
Examples with `bsearch`

- The search methods from the previous lecture and the binary search method are in a file named `searchlab.rb`

```ruby
>> load "searchlab.rb"
=> true
>> a = randoms(15).sort
=> [8, 15, 18, 30, 39, 43, 46, 47, 48, 51, 61, 73, 80, 82, 89]
>> bsearch(a, 30)
=> 3
>> bsearch(a, 31)
=> nil
```

Call `randoms(15)` to make an array of 15 random integers, then call the sort method of that array.

Include `:trace` with the call to have the method print a detailed output showing the progress of the search.

```ruby
>> bsearch(a, 51, :trace)
8 15 18 30 39 43 46 47 48 51 61 73 80 82 89
8 15 18 30 39 43 46 47 [ 48 51 61 +73 80 82 89 ]
8 15 18 30 39 43 46 47 [ 48 *51 61 ] 73 80 82 89
=> 9
>> bsearch(a, 26, :trace)
8 15 18 30 39 43 46 47 48 51 61 73 80 82 89
8 15 18 30 39 43 46 47 [ 48 51 61 ] 73 80 82 89
8 15 18 *[ ] 39 43 46 47 48 51 61 73 80 82 89
8 15 18 *[ ] 39 43 46 47 48 51 61 73 80 82 89
=> nil
```

Summary

- It should be clear why we say the binary search uses a divide and conquer strategy
  - the problem is to find an item within a given range
    - initial range: entire array
    - at each step the problem is split into two equal sub-problems
    - focus turns to one sub-problem for the next step

```
  4 10 22 36 38 57 65 71 79 83 87 90 91 93
  *
  4 10 22 36 38 57 65
  *
  36 57 65
```

* = value of mid on each iteration in a search for 57

Number of Comparisons

- The number of comparisons made by this algorithm when it searches an array of \( n \) items is roughly \( \log_2 n \)
- To see why, consider the question from the other direction
  - suppose we have an array that starts out with 1 item
  - on each step of an iteration we double the size of the array
  - after \( n \) steps we will have \( 2^n \) items in the array

```
10
11
20
21
40
41
80
81
```

\[ 1 = 2^0 \]

\[ 2 = 2^1 \]

\[ 4 = 2^2 \]

\[ 8 = 2^3 \]
Number of Comparisons

- When we’re searching we’re reducing an area of size \( n \) down to an area of size 1
  - e.g. \( n = 8 \) in this diagram

- There is one comparison for each line in the diagram
  - a successful search might return after the first comparison
  - an unsuccessful search does one comparison for each line

\[
\begin{align*}
1 &= 2^0 \\
2 &= 2^1 \\
4 &= 2^2 \\
8 &= 2^3
\end{align*}
\]

\( \# \text{steps} = \log_2 n + 1 \)

Counting the Number of Steps

- The `bsearch` method in `sortlab.rb` has an option to count the number of comparisons

```ruby
>> a = randoms(8).sort
[12, 19, 29, 58, 68, 72, 96, 98]
>> bsearch(a, 98, :trace)
[ 12 19 29 *58 68 72 96 98 ]
12 19 29 58 [ 68 *72 96 98 ]
12 19 29 58 68 72 [ *96 98 ]
12 19 29 58 68 72 96 [ *98 ]
=> 7
>> bsearch(a, 98, :count)
=> 4
>> bsearch(a, 19, :count)
=> 2
```

Recursion

- In computer science a **recursive** description of a problem is one where
  - a problem can be broken into smaller parts
  - each part is a **smaller version of the original problem**
  - there is a “base case” that can be solved immediately (i.e. it has no sub-problems)

- Binary search can be described recursively:

```ruby
def bsearch(a, k, lower = -1, upper = a.length)
    mid = (lower + upper) / 2
    if mid == lower
        return nil
    elsif a[mid] == k
        return mid
    elsif k < a[mid]
        return bsearch(a, k, lower, mid)
    else
        return bsearch(a, k, mid, upper)
    end
end
```

Recursion

- We can write recursive methods in Ruby
  - the body of a method might have a call to itself
  - see `rsearch` in `searchlab.rb`

```ruby
def rsearch(a, k, lower = -1, upper = a.length)
    mid = (lower + upper) / 2
    if mid == lower
        return nil
    elsif a[mid] == k
        return mid
    elsif k < a[mid]
        return rsearch(a, k, lower, mid)
    else
        return rsearch(a, k, mid, upper)
    end
end
```
Recursion

Understanding recursive methods takes some getting used to:
- It's easy to get lost, especially if you mentally trace what the system is doing
- It's a powerful tool as part of a programmer's workbench
- Many complex problems are much easier to solve when one realizes there is a recursive description

Key points to remember:
- A recursive call is like any other call -- parameters are passed, expressions are evaluated, results are returned to the point of the call
- There must be a base case, otherwise the result is an infinite recursion

We will see other examples of recursive methods later in the term
- You don't need to follow the code -- just understand the "big picture", in this case how recursion implements divide and conquer

Recursive call, lower = -1, upper = 3

Recursive call, lower = 1, upper = 3

Location were 29 was found

Divide and Conquer Sorting Algorithms

The divide and conquer strategy used to make a more efficient search algorithm can also be applied to sorting.

Two well-known sorting algorithms:
- QuickSort
  - Divide a list into big values and small values, then sort each part
- Merge Sort
  - Sort subgroups of size 2, merge them into sorted groups of size 4, merge those into sorted groups of size 8, ...
- The remaining slides will have an overview of each algorithm, and a look at how Merge Sort can be implemented in Ruby
  - See the msort method in sortlab.rb

QuickSort

The main idea in QuickSort is to partition the array into two regions:
- Small items are moved to the left half of the array
- Large items are moved to the right half

After partitioning repeat the sort on the left and right regions
- Each region is a sub-problem, a smaller version of the original problem

Main question: how do we decide what is "small" and what is "large"?
- A common technique: use the first item in the region as a "pivot"
  - Everything less than the pivot ends up in the left region
  - Items greater than or equal to the pivot go in the right region

[See example on next slide]
Partition Example

The partitioning algorithm works from both ends

When it finds a large item on the left and a small item on the right it swaps them

When there are no more exchanges to make the two regions are complete

QuickSort Algorithm

Since the partition step does all the hard work the QuickSort algorithm is straightforward

Here is the pseudo-code:

```java
qsort(a, lower, upper):
    if lower < upper
        mid = partition(a, lower, upper)
        qsort(a, lower, mid)
        qsort(a, mid+1, upper)
```

QuickSort Demo

The animation applet uses a slightly different implementation
- it also uses the name “temp” instead of “pivot”
- but it clearly shows this divide and conquer process at work
- Look for green boxes around the subproblems as the algorithm starts sorting each new region

QuickSort Recursion

The animation illustrated an important point about recursive methods
- the computer needs to keep an internal “to do” list
- when it makes the first recursive call, it needs to record the fact that it needs to come back and eventually make the second call
- Notice how, after making recursive calls to sort the left region, the system seems to jump back and sort the right region of the top level problem

The to-do list is kept in a data structure known as a stack
More on these later in the term....

http://math.hws.edu/TMCM/java/xSortLab
QuickSort Performance

- QuickSort is not guaranteed to be more efficient than Insertion Sort
  - if it makes an unlucky choice for the pivot the array will not be divided equally
- worst case: sorting an array that is already in order
- The analysis of the average number of steps for random lists is fairly complex
- Bottom line: to sort a list of \( n \) items requires approximately
  \[ \# \text{steps} = n \times \log_2 n \]

- Many tests on real-world data show that QuickSort is very effective in practice
  and it is a popular choice in many applications

The qsort Method

- The file sortlab.rb has a method named qsort
- You can pass optional parameters that will be useful when running experiments
  \[ \text{qsort}(a) \quad \text{return a sorted version of array a} \]
  \[ \text{qsort}(a, :trace) \quad \text{same as above, printing a detailed trace} \]
  \[ \text{qsort}(a, :count) \quad \text{return the number of comparisons} \]
  \[ \text{qsort}(a, :timer) \quad \text{return the execution time (in seconds)} \]

[see examples next slide]

The qsort Method

```ruby
>> a = randoms(10)
=> [92, 0, 51, 69, 3, 89, 51, 47, 85, 75]
>> qsort(a)
=> [0, 3, 47, 51, 51, 69, 75, 85, 89, 92]
>> qsort(a, :count)
=> 58
```

```ruby
pivot = 36
subproblems: numbers less than 36, numbers greater than or equal to 36
solving the first subproblem
retrieving the second subproblem from the “to do list”
```

See also: qsort.rb

```ruby
>> a = randoms(100)
>> qsort(a, :trace)
```

```ruby
pivot = 36
```

```ruby
subproblems: numbers less than 36, numbers greater than or equal to 36
```

```ruby
solving the first subproblem
```

```ruby
retrieving the second subproblem from the “to do list”
```
What do you think `qsort` will do if we pass it a sorted array?

```ruby
> a = randoms(10)
=> [14, 49, 26, 1, 33, 50, 91, 70, 25, 77]
> qsort(a.sort, :trace)
[ 1 14 25 26 33 49 50 70 77 91 ]
[ 1 ] 14 25 26 33 49 50 70 77 91
1 [ 14 ] 25 26 33 49 50 70 77 91
1 14 [ 25 ] 26 33 49 50 70 77 91
1 14 25 [ 33 ] 49 50 70 77 91
1 14 25 33 [ 49 ] 50 70 77 91
1 14 25 33 49 [ 50 ] 70 77 91
1 14 25 33 49 50 [ 70 ] 77 91
1 14 25 33 49 50 70 [ 77 91 ]
[ 97 ] 70 77 91
```

Pass a sorted copy of `a` to `qsort`.

Merge Sort

The merge sort algorithm works from “the bottom up”
- Start by breaking the original problem into the smallest subproblems
- Keep solving small problems and combining their results into larger solutions
- Eventually the original problem will be solved

Example: sorting playing cards
- Divide the cards into groups of two
- Sort each group -- put the smaller of the two on the top
- Merge pairs of groups into groups of four
- Merge pairs of these into groups of eight
- ...

Example with a hand of seven cards
- Initial hand
- Sorted piles of size two
- Sorted piles of size four
- Final sorted pile of all cards

What makes this method more effective than simple insertion sort?
- Merging two piles is a very simple operation
- Only need to look at the two cards currently on the top of each pile
- No need to look deeper into either group

In this example:
- Compare 2 with 5, pick up the 2
- Compare 5 with 7, pick up the 5
- Compare 7 with 10, pick up the 7
- ....
Merge Sort

- Another example, using an array of numbers
- Sorted blocks are indicated by adjacent cells with the same color

Merger Sort Demo

- The merge sort algorithm has been implemented in a method named `msort`
  - Has the same parameters and options as `isort` and `qsort`

```ruby
>> a = randoms(16,100)
=> [1, 87, 0, 52, 12, 32, 44, 32, 35, 94, 55, 63, 17, 38, 86, 33]
>> msort(a, :trace)
=> [1 87] [0 52] [12 32] [32 44] [35 94] [55 63] [17 38] [86 33]
>> msort(a, :count)
=> 38
```

Merge Sort Algorithm

- The key operation in `msort` is done by a “helper” method named `merge`
  - A call to `merge(a, i, n)` means “merge the two groups in array `a` that start at index `i` and have size `n`”
    - The array is modified “in place”
    - After the call the region from `a[i]` to `a[i+2*n-1]` will be sorted
  - Examples:
    - `merge(a, 0, 1)` means “merge `a[0]` and `a[1]`”
    - `merge(a, 4, 2)` means “merge `a[4..5]` and `a[6..7]`”

```ruby
def msort(a, mode = nil)
  n = 1
  while n < a.length
    i = 0
    while i < a.length
      merge(a, i, n)
      i += 2*n
    end
    n *= 2
  end
  return a
end
```

```
def merge(a, i, n)
  left = a[i..i+n-1]
  right = a[i+n..i+2*n-1]
  # Merge logic here...
end
```
Comparisons in Merge Sort

- To completely sort an array with \( n \) items requires \( \log_2 n \) iterations
  - the group size starts at 1 and doubles on each iteration
- During each iteration there are at most \( n \) comparisons
  - comparisons occur in the merge method
  - compare values at the front of each group
  - may have to work all the way to the end of each group, but might stop early (e.g. with cards one pile is emptied but more than one left in the other pile)

\[
\begin{align*}
1 &= 2^0 \\
2 &= 2^1 \\
4 &= 2^2 \\
8 &= 2^3
\end{align*}
\]

Total comparisons \( = n \times \log_2 n \)

Comparisons in Merge Sort

- Is this new formula that much better than the \( n^2/2 \) comparisons made by isort?
  - not that big of a difference for small arrays
  - huge difference for larger arrays

\[
\begin{align*}
\gg & \ a = \text{randoms}(1000) \\
\gg & \ \text{isort}(a, \ :\text{count}) \\
\gg & \ 243995 \\
\gg & \ \text{msort}(a, \ :\text{count}) \\
\gg & \ 8720
\end{align*}
\]

Sort Algorithms in Real Life

- These algorithms can be used in the real world
  - it might be fun to try QuickSort or merge sort on a deck of cards
- For QuickSort you'll need a lot of room to lay out all the cards
- Merge sort can be done in a very small space
  - pick up the smaller of the two top cards
  - lay it face down in a new pile
  - when merging the next two groups the new pile should be at right angles so you know where the group starts
  - turn the deck over and repeat

![Card deck diagram]

The way most people sort a full deck (make one pile for aces, one for kings, ...) is known as radix sort

Probably more efficient than merge sort for this problem...
Sort Algorithms in Real Life

- A place where merge sort might be the best method is sorting stacks of papers
- Example: sorting a set of exams from a class with 45 students

Use the method sketched on the previous slide for cards
- new groups are formed face down below existing groups
- alternate the orientation of each new group
- turn the stack over and repeat

Summary

- These slides introduced the **divide and conquer** strategy
  - for searching: **binary search**
    - requires list to be sorted
  - for sorting: QuickSort and **merge sort**
- Binary search will find an item using at most \( \log_2 n \) comparisons
- QuickSort and merge sort do at most \( n \times \log_2 n \) comparisons
- An algorithm that uses divide and conquer can be written using iteration or **recursion**
  - recursive = “self-similar”
  - a problem that can be divided into smaller subproblems of the same type
  - a recursive method calls itself

Extra Credit Challenge

- iTunes allows you to sort your music catalog by many different attributes
  - suppose you first sort by album title, then sort again by artist
  - are the results of the first sort still valid? i.e. for any artist, are their albums all listed in alphabetical order?
  - do any of the sorting algorithms presented in class have this property, that multiple sorts will preserve the order of previous sorts? or would a subsequent sort scramble the array so previous results are lost?