CSc 110, Spring 2017

Lecture 39: searching

When a user takes a photo, the app should check whether they're in a national park...

Sure, easy GIS lookup. Gimme a few hours...

...and check whether the photo is of a bird.

I'll need a research team and five years.

In CS, it can be hard to explain the difference between the easy and the virtually impossible.
Sequential search

• **sequential search**: Locates a target value in a list (may not be sorted) by examining each element from start to finish. Also known as *linear* search.

  • How many elements will it need to examine?

  • Example: Searching the list below for the value **42**:

    | index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
    |-------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|
    | value | 2 | 7 | 10| 30| 56| 20| 68| 36| -4| 25| 42 | 50 | 22 | 92 | 15 | 85 | 103|

    i
Sequential (linear) search

- **sequential search**: Even if the list is sorted, elements are examined in the way (one after the other).

  - Example: Searching the list below for the value 42:

```
index 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
value -4 2 7 10 15 20 22 25 30 36 42 50 56 68 85 92 103
```

i
Sequential (linear) search

• Sequential search code:

```python
def sequential_search(my_list, value):
    for i in range(0, len(my_list)):
        if my_list[i] == value:
            return i
    return -1  # not found
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>

• Note that -1 is returned if the element is not found.
Sequential (linear) search

• For a list of size N, how many elements will be checked worst case?

• On average how many elements will be checked?

• A list of 1,000,000 elements may require 1,000,000 elements to be examined.

• The number of elements to check grows in proportion to the size of the list, i.e., it grows linearly.
Binary Search

• **Binary search**: a method of searching that takes advantage of sorted data.

• Consider a guessing game:

  Someone thinks of a number between 1 and 100. You must guess the number. On each round, you are told whether your number is low, high, or correct.

• Best strategy: use a first guess of 50
  Eliminates half of the numbers immediately
  On each round, half the numbers are eliminated:
  
  100
  50
  25
  ...

Binary search

- **binary search**: Locates a target value in a *sorted* list by successively eliminating half of the list from consideration.

- How many elements will it need to examine?

- Example: Searching the list below for the value **42**:

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>

Keep track of indices for a min, mid and max.
• Search for 42: Round 1.

list[mid] < 42
eliminate from min to mid (left half)
**Search for 42**: Round 2.

list[mid] > 42

eliminate from mid to max (right half of what's left)

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>

min  | mid | max
**Search for 42**: Round 3.

\[
\text{list[mid]} == 42
\]

found!

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>

min  mid  max
Binary search runtime

• For a list of size N, it eliminates ½ until 1 element remains.
  N, N/2, N/4, N/8, ..., 4, 2, 1
  • How many divisions does it take?
  • Suppose N = 1024
    1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1   (10 divisions)
  • 10 = \log_2 (1024)

• Suppose we double the number the number of elements.
  • How many divisions does it take?
  • Suppose N = 2048
    2048, 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1   (11 divisions)
  • 11 = \log_2 (2048)
Binary search runtime

- For a list of size N, it eliminates ½ until 1 element remains.
  \[ N, N/2, N/4, N/8, \ldots, 4, 2, 1 \]
  - How many divisions does it take?
  - Suppose \( N = 1024 \)
    \[ 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1 \] (10 divisions)
  - Binary search examines a number of elements proportional to the number of divisions

- Think of it from the other direction:
  - How many times do I have to multiply by 2 to reach N?
    \[ 1, 2, 4, 8, \ldots, N/4, N/2, N \]
  - Call this number of multiplications "x".
    \[ 2^x = N \]
    \[ x = \log_2 N \]

- Binary search examines a number of elements proportional to \( \log \text{ of } N \).
# Returns the index of an occurrence of target in a, 
# or a negative number if the target is not found. 
# Precondition: elements of a are in sorted order 

def binary_search(a, target):
    min = 0
    max = len(a) - 1

    while (min <= max):
        mid = (min + max) // 2
        if (a[mid] < target):
            min = mid + 1
        elif (a[mid] > target):
            max = mid - 1
        else:
            return mid       # target found

    return -(min + 1)    # target not found
Binary search

| index | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| value | -4 | 2  | 7  | 12 | 18 | 25 | 27 | 30 | 36 | 42 | 56 | 68 | 85 | 91 | 92 | 98 | 102|

What do the following calls return when passed the above list?

`binary_search(a, 2)`
`binary_search(a, 68)`
`binary_search(a, 12)`

How many comparisons does each call do?
Comparing Binary vs. Sequential search

- **Binary search vs Sequential search**: number of items examined

<table>
<thead>
<tr>
<th>List size</th>
<th>Binary search</th>
<th>Sequential search</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>1,000</td>
<td>11</td>
<td>1,000</td>
</tr>
<tr>
<td>5,000</td>
<td>14</td>
<td>5,000</td>
</tr>
<tr>
<td>100,000</td>
<td>18</td>
<td>100,000</td>
</tr>
<tr>
<td>1,000,000</td>
<td>21</td>
<td>1,000,000</td>
</tr>
</tbody>
</table>
from bisect import *

# searches an entire sorted list for a given value
# returns the index the value should be inserted at to maintain sorted order
# Precondition: list is sorted
bisect(list, value)

# searches given portion of a sorted list for a given value
# examines min_index (inclusive) through max_index (exclusive)
# returns the index the value should be inserted at to maintain sorted order
# Precondition: list is sorted
bisect(list, value, min_index, max_index)
Using `bisect`

```python
# index 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
a = {-4, 2, 7, 9, 15, 19, 25, 28, 30, 36, 42, 50, 56, 68, 85, 92}

index1 = bisect(a, 42, 0, 16)  # index1 is 11
index2 = bisect(a, 21, 0, 16)  # index2 is 6

• `bisect` returns the index where the value could be inserted while maintaining sorted order

• if the value is already in the list the next index is returned
Sorting

• **sorting**: Rearranging the values in a list into a specific order (usually into their "natural ordering").
  
  • one of the fundamental problems in computer science
  • can be solved in many ways:
    • there are many sorting algorithms
    • some are faster/slower than others
    • some use more/less memory than others
    • some work better with specific kinds of data
    • some can utilize multiple computers / processors, ...

• **comparison-based sorting**: determining order by comparing pairs of elements:
  • <, >, ...
Sorting algorithms

- **bogo sort**: shuffle and pray
- **bubble sort**: swap adjacent pairs that are out of order
- **selection sort**: look for the smallest element, move to front
- **insertion sort**: build an increasingly large sorted front portion
- **merge sort**: recursively divide the list in half and sort it
- **heap sort**: place the values into a sorted tree structure
- **quick sort**: recursively partition list based on a middle value

Other specialized sorting algorithms:

- **bucket sort**: cluster elements into smaller groups, sort them
- **radix sort**: sort integers by last digit, then 2nd to last, then ...
- ...
Bogo sort

- **bogo sort**: Orders a list of values by repetitively shuffling them and checking if they are sorted.
  - name comes from the word "bogus"

The algorithm:
- Scan the list, seeing if it is sorted. If so, stop.
- Else, shuffle the values in the list and repeat.

- This sorting algorithm (obviously) has terrible performance!
Bogo sort code

# Places the elements of a into sorted order.
def bogo_sort(a):
    while (not is_sorted(a)):
        shuffle(a)

# Returns true if a's elements
# are in sorted order.
def is_sorted(a):
    for i in range(0, len(a) - 1):
        if (a[i] > a[i + 1]):
            return False
    return True