Ch 8. Searching and Sorting Arrays
8.1 and 8.3 only

CS 2308
Fall 2013

Jill Seaman

Covers objectives 1-4 in the syllabus

Definitions of Search and Sort

- **Search**: find a given item in an array, return the index to the item, or -1 if not found.
- **Sort**: rearrange the items in an array into some order (smallest to biggest, alphabetical order, etc.).
- There are various methods (algorithms) for carrying out these common tasks.
- Which ones are better? Why?

Linear Search

- Very simple method.
- Compare first element to target value, if not found then compare second element to target value . . .
- Repeat until:
  target value is found (return its index) or we run out of items (return -1).

Linear Search in C++

```cpp
int searchList (int list[], int size, int target) {
    int position = -1;       //position of target
    for (int i=0; i<size; i++)
        if (list[i] == target) //found the target!
            position = i;       //record which item
    return position;
}
```

Is this algorithm correct?

Is this algorithm efficient (or does it do unnecessary work)?
Program that uses linear search

```
#include <iostream>
using namespace std;

int searchList(int[], int, int);

int main() {
    const int SIZE=5;
    int idNums[SIZE] = {871, 750, 988, 100, 822};
    int results, id;
    cout << "Enter the employee ID to search for: ";
    cin >> id;
    results = searchList(idNums, SIZE, id);
    if (results == -1) {
        cout << "That id number is not registered\n";
    } else {
        cout << "That id number is found at location ";
        cout << results+1 << endl;
    }
}
```

Efficiency of Linear Search

```
N is the number of elements in the array

<table>
<thead>
<tr>
<th></th>
<th>N=50,000</th>
<th>In terms of N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case:</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Average Case:</td>
<td>25,000</td>
<td>N/2</td>
</tr>
<tr>
<td>Worst Case:</td>
<td>50,000</td>
<td>N</td>
</tr>
</tbody>
</table>
```

Note: if we search for many items that are not in the array, the average case result will increase.

Evaluating the Algorithm

- Does it do any unnecessary work?
- Is it efficient? How would we know?
- We measure efficiency of algorithms in terms of number of main steps required to finish.
- For search algorithms, the main step is comparing an array element to the target value.
- Number of steps depends on:
  - size of input array
  - whether or not value is in array
  - where the value is in the array

Linear Search in C++

```
int searchList (int list[], int size, int value) {
    int index=0;          //index to process the array
    int position = -1;    //position of target
    bool found = false;   //flag, true when target is found
    while (index < size && !found) {
        if (list[index] == value)  //found the target!
            found = true;            //set the flag
            position = index;        //record which item
        index++;                   //increment loop index
    }
    return position;
}
```

Is this algorithm correct?
Is this algorithm efficient (or does it do unnecessary work)?

int list[]; int size, int value) {
int index=0;          //index to process the array
int position = -1;    //position of target
bool found = false;   //flag, true when target is found
while (index < size && !found) {
    if (list[index] == value)  //found the target!
        found = true;            //set the flag
        position = index;        //record which item
    index++;                   //increment loop index
}  return position;
}
Binary Search

- Works only for SORTED arrays
- Divide and conquer style algorithm
- Compare target value to middle element in list.
  - if equal, then return its index
  - if less than middle element, repeat the search in the first half of list
  - if greater than middle element, repeat the search in last half of list
- If current search list is narrowed down to 0 elements, return -1

Binary Search Algorithm

- The algorithm described in pseudocode:
  ```
  while (number of items in list >= 1)
      if (item at middle position is equal to target)
          target is found!  End of algorithm
      else
          if (target < middle item)         (narrow search list)
              list = lower half of list
          else
              list = upper half of list
      end while
  if we reach this point: target not found
  ```

Binary Search Algorithm example

<table>
<thead>
<tr>
<th>target is 11</th>
<th>first</th>
<th>mid</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 4 7 10 11 45</td>
<td></td>
<td>50 59 60 66 69 70 79</td>
</tr>
<tr>
<td></td>
<td>first mid last</td>
<td></td>
<td></td>
</tr>
<tr>
<td>target &gt; 7</td>
<td>[0] [1] [2] [3] [4] [5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 4 7 10 11 45</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>first mid last</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10 11 45</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Binary Search in C++

```cpp
int binarySearch (int array[], int size, int target) {
    int first = 0,          //index to (current) first elem
        last = size - 1,    //index to (current) last elem
        middle,             //index of (current) middle elem
        position = -1;      //index of target value
    bool found = false;     //flag
    while (first <= last && !found) {
        middle = (first + last) / 2;    //calculate midpoint
        if (array[middle] == target) {
            found = true;
            position = middle;
        } else if (target < array[middle]) {
            last = middle - 1;            //search lower half
        } else {
            first = middle + 1;          //search upper half
        }
    }
    return position;
}
```
Binary Search
Example Exam Question!

The target of your search is 42. Given the following list of integers, record the values stored in the variables named first, last, and middle during a binary search. Assume the following numbers are in an array.

values:  1 7 8 14 20 42 55 67 78 101 112 122 170 179 190
indexes: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Repeat the exercise with a target of 82

values:  first 0 0 4
last   14  6  6
middle 7  3  5

Program using Binary Search

#include <iostream>
using namespace std;

int binarySearch(int[], int, int);

int main() {
    const int SIZE = 5;
    int idNums[SIZE] = {100, 750, 822, 871, 988};
    int results, id;
    cout << “Enter the employee ID to search for: “; cin >> id;
    results = binarySearch(idNums, SIZE, id);
    if (results == -1) {
        cout << “That id number is not registered” << endl;
    } else {
        cout << “That id number is found at location “ << results + 1 << endl;
    }
}

How is this program different from the one on slide 6?

Efficiency of Binary Search

Calculate worst case for N=1024

<table>
<thead>
<tr>
<th>Items left to search</th>
<th>Comparisons so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>2</td>
</tr>
<tr>
<td>128</td>
<td>3</td>
</tr>
<tr>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

Goal: calculate this value from N

1024 = 2^{10}  \iff \log_2 1024 = 10

Efficiency of Binary Search

If N is the number of elements in the array, how many comparisons (steps)?

1024 = 2^{10}  \iff \log_2 1024 = 10

N = 2^{\text{steps}}  \iff \log_2 N = \text{steps}

To what power do I raise 2 to get N?

N=50,000  In terms of N
| Best Case: | 1 | 1 |
| Worst Case: | 16 | \log_2 N |

Rounded up to next whole number
Is $\log_2 N$ better than $N$?

Is binary search better than linear search? Is this really a fair comparison?

Compare values of $N/2$, $N$, and $\log_2 N$ as $N$ increases:

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N/2$</th>
<th>$\log_2 N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2.5</td>
<td>2.3</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>5.6</td>
</tr>
<tr>
<td>500</td>
<td>250</td>
<td>9.0</td>
</tr>
<tr>
<td>5,000</td>
<td>2,500</td>
<td>12.3</td>
</tr>
<tr>
<td>50,000</td>
<td>25,000</td>
<td>15.6</td>
</tr>
</tbody>
</table>

$N$ and $N/2$ are growing much faster than $\log N$!

Slower growing is more efficient (fewer steps).

Classifications of (math) functions

- Constant: $f(x)=b$, $O(1)$
- Logarithmic: $f(x)=\log_b(x)$, $O(\log n)$
- Linear: $f(x)=ax+b$, $O(n)$
- Linearithmic: $f(x)=x \log_b(x)$, $O(n \log n)$
- Quadratic: $f(x)=ax^2+bx+c$, $O(n^2)$
- Exponential: $f(x)=b^x$, $O(2^n)$

- Last column is “big Oh notation”, used in CS.
- It ignores all but dominant term, constant factors.

Comparing growth of functions

Efficiency of Algorithms

- To classify the efficiency of an algorithm:
  - Express “time” (using number of main steps or comparisons), as a function of input size
  - Determine which classification the function fits into.

- Nearer to the top of the chart is slower growth, and more efficient (constant is better than logarithmic, etc.)
8.3 Sorting Algorithms

- Sort: rearrange the items in an array into ascending or descending order.
- Selection Sort
- Bubble Sort

Why is sorting important?

- Searching in a sorted list is much easier than searching in an unsorted list.
- Especially for people:
  - dictionary entries (in a dictionary book)
  - phone book (remember these?)
  - card catalog in library (it used to be drawers of index cards)
  - bank statement: transactions in date order
- Most of the data displayed by computers is sorted.

Selection Sort

- There is a pass for each position (0..size-1)
- On each pass, the smallest (minimum) element in the rest of the list is exchanged (swapped) with element at the current position.
- The first part of the list (the part that is already processed) is always sorted
- Each pass increases the size of the sorted portion.
**Selection Sort: End Pass One**

values [0]

[1] 6
[2] 24
[3] 10
[4] 36
[5] 12

**Selection Sort: Pass Two**

values [0]

[1] 6
[2] 24
[3] 10
[4] 36
[5] 12

**Selection Sort: End Pass Two**

values [0]

[1] 6
[2] 10
[3] 24
[4] 36
[5] 12

**Selection Sort: Pass Three**

values [0]

[1] 6
[2] 10
[3] 24
[4] 36
[5] 12
Selection Sort in C++

```cpp
// Returns the index of the smallest element, starting at start
int findIndexOfMin (int array[], int size, int start) {
    int minIndex = start;
    for (int i = start+1; i < size; i++) {
        if (array[i] < array[minIndex]) {
            minIndex = i;
        }
    }
    return minIndex;
}

// Sorts an array, using findIndexOfMin
void selectionSort (int array[], int size) {
    int temp;
    int minIndex;
    for (int index = 0; index < (size -1); index++) {
        minIndex = findIndexOfMin(array, size, index);
        // swap
        temp = array[minIndex];
        array[minIndex] = array[index];
        array[index] = temp;
    }
}
```

Note: saving the index

We need to find the index of the minimum value so that we can do the swap
Program using Selection Sort

```c++
#include <iostream>
using namespace std;

int findIndexOfMin (int [], int, int);
void selectionSort(int [], int);
void showArray(int [], int);

int main() {
    int values[6] = {7, 2, 3, 8, 9, 1};
    cout << "The unsorted values are: \n";
    showArray (values, 6);
    selectionSort (values, 6);
    cout << "The sorted values are: \n";
    showArray(values, 6);
}

void showArray (int array[], int size) {
    for (int i=0; i<size; i++)
        cout << array[i] << " " ;
    cout << endl;
}
```

Output:
The unsorted values are: 7 2 3 8 9 1
The sorted values are: 1 2 3 7 8 9

Efficiency of Selection Sort

- **N** is the number of elements in the list
- Outer loop (in selectionSort) executes N-1 times
- Inner loop (in minIndex) executes N-1, then N-2, then N-3, ... then once.
- Total number of comparisons (in inner loop):
  
  \[
  \sum_{i=1}^{N-1} i = \frac{N(N+1)}{2} - N
  \]

The Bubble Sort

- On each pass:
  - Compare first two elements. If the first is bigger, they exchange places (swap).
  - Compare second and third elements. If second is bigger, exchange them.
  - Repeat until last two elements of the list are compared.
- Repeat this process until a pass completes with no exchanges

Example

```
7 2 3 8 9 1
7 > 2, swap
2 7 3 8 9 1
7 > 3, swap
2 3 7 8 9 1
!(7 > 8), no swap
2 3 7 8 9 1
!(8 > 9), no swap
2 3 7 8 9 1
9 > 1, swap
2 3 7 8 1 9
finished pass 1, did 3 swaps
```

Note: largest element is now in last position

Note: This is one complete pass!
**Bubble sort Example**

- **2 3 7 8 1 9**: 2<3<7<8, no swap, !(8<1), swap
- **2 3 7 1 8 9**: (8<9) no swap
- finished pass 2, did one swap
  - 2 largest elements in last 2 positions
- **2 3 7 1 8 9**: 2<3<7, no swap, !(7<1), swap
- **2 3 1 7 8 9**: 7<8<9, no swap
- finished pass 3, did one swap
  - 3 largest elements in last 3 positions

**Bubble Sort in C++**

```cpp
void bubbleSort (int array[], int size) {
    bool swap;
    int temp;
    do {
        swap = false;
        for (int i = 0; i < (size-1); i++) {
            if (array[i] > array[i+1]) {
                temp = array[i];
                array[i] = array[i+1];
                array[i+1] = temp;
                swap = true;
            }
        }
    } while (swap);
}
```

**Bubble sort how does it work?**

- At the end of the first pass, the largest element is moved to the end (it’s bigger than all its neighbors)
- At the end of the second pass, the second largest element is moved to just before the last element.
- The back end (tail) of the list remains sorted.
- Each pass increases the size of the sorted portion.
- No exchanges implies each element is smaller than its next neighbor (so the list is sorted).
Program using bubble sort

```cpp
#include <iostream>
using namespace std;

void bubbleSort(int[], int);
void showArray(int[], int);

int main() {
    int values[6] = {7, 2, 3, 8, 9, 1};
    cout << "The unsorted values are: \n";
    showArray(values, 6);
    bubbleSort(values, 6);
    cout << "The sorted values are: \n";
    showArray(values, 6);
}

void showArray(int array[], int size) {
    for (int i = 0; i < size; i++)
        cout << array[i] << " ";
    cout << endl;
}
```

Output:
The unsorted values are: 7 2 3 8 9 1
The sorted values are: 1 2 3 7 8 9

Efficiency of Bubble Sort

- Each pass makes N-1 comparisons
- There will be at most N passes

So worst case it’s: \( (N-1) \times N = N^2 - N \) \( \text{O}(N^2) \)

If you change the algorithm to look at only the unsorted part of the array in each pass, it’s exactly like the selection sort:

\( (N-1) + (N-2) + \ldots + 2 + 1 = N^2/2 - N/2 \) \( \text{still O}(N^2) \)