Ch 8. Searching and Sorting Arrays
8.1 and 8.3 only

CS 2308
Fall 2012
Jill Seaman

Definitions of Search and Sort

- Search: find an 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 numElems, int value) {
    int index=0;          //index to process array
    int position = -1;    //position of value
    bool found = false;   //flag, true when value is found

    while (index < numElems && !found) {
        if (list[index] == value)  //found the value!
        {                         //set the flag
            found = true;          //record which item
            position = index;      //increment loop index
        }
        index++;                   //increment loop index
    }
    return position;
}
```
Program using Linear Search

```cpp
#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;
    }
    return 0;
}
```

Evaluating the Algorithm

- Is it efficient? Does it do any unnecessary work?
- 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

Efficiency of Linear Search

<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>

*N is the number of elements in the array

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

Binary Search

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

The algorithm described in pseudocode:

while (number of items in list >= 1) and (target not found)
    if (item at middle position is equal to target)
        target is found!
        location = middle
    else
        if (target < middle item)
            list = lower half of list
        else
            list = upper half of list
    end while

if target not found, location = -1

Binary Search in C++

int binarySearch (int array[], int numElems, int value) {
    int first = 0, last = numElems - 1, middle, position = -1;
    bool found = false;
    while (first <= last && !found) {
        middle = (first + last) / 2;
        if (array[middle] == value) {
            found = true;
            position = middle;
        } else if (array[middle] > value) {
            last = middle - 1; //search lower half
        } else { //search upper half
            first = middle + 1;
        }
    }
    return position;
}

What if first + last is odd?
What if first == last?

Binary Search

Example

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

1 7 8 14 20 42 55 67 78 101 112 122 170 179 190

Repeat the exercise with a target of 82

| first | 0 0 4 |
| last  | 14 6 6 |
| middle| 7 3 5 |

| first | 0 8 8 8 9 |
| last  | 14 14 10 8 8 |
| middle| 7 11 9 8 |

Program using Binary Search

#include <iostream>
using namespace std;

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

const int SIZE=5;

int main() {
    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\n";
    } else {
        cout << "That id number is found at location ";
        cout << results+1 << endl;
    }
    return 0;
}
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>

1024 = 2^10 How could we calculate 10 from 1024?

Efficiency of Binary Search

<table>
<thead>
<tr>
<th>N=50,000</th>
<th>In terms of N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case:</td>
<td>1</td>
</tr>
<tr>
<td>Worst Case:</td>
<td>15.6</td>
</tr>
</tbody>
</table>

*N is the number of elements in the array

Is log₂ N (binary search) better than N (linear search)?

[Is it really fair to compare these two algorithms?]

Is Log₂N better than N?

Compare values of N/2, N, and Log₂ N as N increases:

<table>
<thead>
<tr>
<th>N</th>
<th>N/2</th>
<th>Log₂N</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>

observation: n/2 is growing much faster than log n!
slower growing is more efficient (fewer steps).

Classifications of (math) functions

<table>
<thead>
<tr>
<th>Classification</th>
<th>Function</th>
<th>Big O notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>y=b</td>
<td>O(1)</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>y=logₐ(x)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Linear</td>
<td>y=ax+b</td>
<td>O(n)</td>
</tr>
<tr>
<td>Linearithmic</td>
<td>y=x logₐ(x)</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>Quadratic</td>
<td>y=ax²+bx+c</td>
<td>O(n²)</td>
</tr>
<tr>
<td>Exponential</td>
<td>y=b^n</td>
<td>O(2^n)</td>
</tr>
</tbody>
</table>

* Last column is “big Oh notation”
* It ignores all but dominant term, constant factors
Comparing growth of functions

Efficiency of Algorithms
- To classify efficiency of an algorithm:
  - Express "time" using number of main steps, as a function of input
  - 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
  - phone book
  - card catalog in library
  - 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 (already processed) is always sorted
- Each pass increases the size of the sorted portion.

Selection Sort: Pass One

Selection Sort: End Pass One

Selection Sort: Pass Two
Selection Sort: End Pass Four

Program using Selection Sort

```
#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;
}
```

Efficiency of Selection Sort

- \( N \) is the number of elements in the list
- Outer loop (in selectionSort) executes \( N \) times
- Inner loop (in minIndex) executes \( N-1 \), then \( N-2 \), then \( N-3 \), ... then once.
- Total number of comparisons (in inner loop):
  \[
  (N-1) + (N-2) + \ldots + 2 + 1 = \text{sum of 1 to } N-1
  \]
  Note: \( N + (N-1) + (N-2) + \ldots + 2 + 1 = N(N+1)/2 \)
  Subtract \( N \) from each side:
  \[
  (N-1) + (N-2) + \ldots + 2 + 1 = N(N+1)/2 - N
  = (N^2+N-2N)/2
  = N^2/2 - N/2
  \]
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

Bubble sort 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 1 9  9 > 1, swap
- finished pass 1, did 3 swaps

Note: largest element is in last position

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 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

2 largest elements in last 2 positions

Bubble sort Example

- 2 3 7 8 9 2<3, !(3<1) swap, 3<7<8<9
- 2 1 3 7 8 9  finished pass 4, did one swap
- 2 1 3 7 8 9  !(2<1) swap, 2<3<7<8<9
- 1 2 3 7 8 9  finished pass 5, did one swap
- 1 2 3 7 8 9  1<2<3<7<8<9, no swaps
- finished pass 6, no swaps, list is sorted!
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).

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);
}
```

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;
}
```

Efficiency of Bubble Sort

- Each pass makes N-1 comparisons
- There will be at most N passes
- So worst case it's: \((N-1)N = N^2 - N\) \(O(N^2)\)
- 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:
  \[(N-1) + (N-2) + \ldots + 2 + 1 = \text{sum of 1 to N-1}\]