Searching & Sorting

Gaddis 8.1 and 8.3

CS 2308 :: Spring 2016
Molly O'Neil
Search & Sort Defined

• **The Searching Problem:**
  - **Input:** A sequence of \( n \) numbers \( A = \{a[0], .. a[n-1]\} \) and a value \( v \)
  - **Output:** An index \( i \) such that \( v = a[i] \), or -1 if \( v \) does not appear in \( A \)
  - Informally: find a given item in an array, and return its array index (or -1 if the item isn't in the array)

• **The Sorting Problem:**
  - **Input:** A sequence of \( n \) numbers \( A = \{a[0], .. a[n-1]\} \)
  - **Output:** A permutation (re-ordering) \( B = \{b[0], .. b[n-1]\} \) of the input sequence such that \( b[0] \leq b[1] \leq .. \leq b[n-1] \)
  - Informally: rearrange the items in an array into some order (e.g., smallest to largest, alphabetical, etc.)
A Reminder About Algorithms...

• There are often numerous algorithms to solve the same problem
  • Searching & sorting are no exception

• How do we identify the best algorithm?
  • **Most efficient (computation time)**
  • Count the number of basic operations
  • Count the number of "main steps"

Big-Oh notation drops constant factors
Linear (Sequential) Search

- Very simple algorithm:
  - Step through array element by element, starting with the first element
  - Compare each element with the value being searched for
  - Stop when:
    - The value is found (return the index)
    - Or: the end of the array is encountered (return -1)

```python
found = false
position = -1
index = 0

while found == false and index < number of elements:
    if list[index] == search value:
        found = true
        position = index
        index = index + 1

return position
```
int searchList(int value, const int list[], int size) {
    int position = -1;  // position of value in list

    for(int i = 0; i < size; i++) {
        if(list[i] == value) {  // we found the target!
            position = i;
        }
    }

    return position;
}

Is the algorithm correct?

Does the algorithm do unnecessary work?
int searchList(int value, const int list[], int size) {
    int position = -1;  // position of value in list

    for(int i = 0; i < size; i++) {
        if(list[i] == value) {  // we found the target!
            position = i;
            break;  // no need to keep iterating
        }
    }

    return position;
}

Is the algorithm correct?

Does the algorithm do unnecessary work?
Linear Search in C++: while() Version

• If you prefer to match the pseudocode more closely...

```cpp
int searchList(int value, const int list[], int size) {
    int index = 0;
    int position = -1;
    bool found = false;

    while(!found && index < size) {
        if(list[index] == value) {  // we found the target!
            found = true;
            position = index;
        }
        index++;
    }

    return position;
}
```
Using Linear Search: An Example

```cpp
int searchList(int value, const int list[], int size);

int main() {
    const int SIZE = 5;
    int studentIDs[SIZE] = { 142, 803, 027, 739, 546 };
    int id, id_index;

    cout << "Enter the student ID to search for: ";
    cin >> id;

    id_index = searchList(id, studentIDs, SIZE);

    if(id_index >= 0) {
        cout << "That student ID is not registered" << endl;
    } else {
        cout << "That student ID is found at index ";
        cout << id_index << endl;
    }
}
```
Algorithm Analysis: Linear Search

• Does it do unnecessary work?

• How efficient is it?
  • We measure algorithm efficiency by counting the number of ***main steps***
    • Here, the main step is *comparing target value to an array element*
  • # of main steps is a function of...
    • # of elements in input array \( n \)
    • *whether* or not target value is in the array
    • *where* the target value is in the array
Algorithm Analysis: Linear Search

• Array has \( n \) elements

• How many main steps?
  
  • Best case: target value is in list[0]
    
    • 1 step \( \Rightarrow \) \( O(1) \)

  • Average case: target value is in list[\( n/2 \)]
    
    • \( n/2 \) steps \( \Rightarrow \) \( O(n) \)
      
      • (Note that if we search for a lot of targets that aren’t in the array, the average case will actually be above \( n/2 \))

  • **Worst case:** target value is in list[\( n-1 \)] or target value is not in array at all
    
    • \( n \) steps \( \Rightarrow \) \( O(n) \)

  
  **Upper bound on algorithm’s efficiency:** \( O(n) \)

Is this “good enough”? Probably not...
How Do We Make Search Faster?

• Sometimes we can design a more efficient algorithm by leveraging additional information about the nature of the input data

• Let's decree that the array to search must be in sorted order (i.e., $a[0] \leq a[1] \leq .. \leq a[n-1]$)

<table>
<thead>
<tr>
<th>2</th>
<th>4</th>
<th>9</th>
<th>10</th>
<th>16</th>
<th>21</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td>[5]</td>
<td>[6]</td>
</tr>
</tbody>
</table>

Say we're searching for target value == 21
How many compares to find it w/ linear search? 6

Can we do better?
(Leverage the guarantee that list is ORDERED)
Binary Search

• "Divide and conquer" algorithm
  • Key insight: Because the list is ordered, we now know which side of any particular element our target value must be on (if it's in the list at all)
  • Each check of an element reduces the size of the list of elements remaining to be checked

• Compare target value to the middle element of the list
  • If middle element IS the target value, return its index
  • If target < middle element, repeat search on 1st half of list
  • If target > middle element, repeat search on 2nd half of list

• If you end up with an empty list, return -1
Binary Search Algorithm

• In pseudocode...

```plaintext
first_index = 0                // beginning of search list
last_index = list_size - 1     // end of search list
found = false
position = -1

// while we haven't found our target and we still have
// some list left to search...
while found is false and first_index <= last_index:
    middle = index of element halfway between list[first_index] and list[last_index]
    if target == list[middle]:      // we found the target!
        found = true
        position = middle
    else if target < list[middle]:  // only need to keep searching
        last_index = middle - 1     // in first half of list
    else                            // target > list[middle]:
        first_index = middle + 1   // only need to keep searching
                                    // in second half of list

return position
```
Binary Search Example

target = 10

2 4 9 10 16 21 42 44 51 67 89
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]

2 4 9 10 16
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]

middle

10 16
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]

first

last

middle

first

middle

last

first

last

middle
# Binary Search in C++

```cpp
int binarySearch(int value, const int list[], int size) {
    int first = 0;               // first element of active list
    int last = size - 1;        // last element of active list
    int position = -1;
    bool found = false;

    while(!found && first <= last) {
        int middle = (first + last) / 2;  // calculate midpoint

        if(list[middle] == value) {        // we found target!
            found = true;
            position = middle;
        }
        else if(value < list[middle]) {   // search lower half
            last = middle - 1;
        }
        else {                            // value > list[middle] (search upper half)
            first = middle + 1;
        }
    }

    return position;
}
```

What if \((first + last)\) is odd?

What if \((first == last)\)?
Using Binary Search: An Example

• What are the differences between the linear search example and this program?

```cpp
int binarySearch(int value, const int list[], int size);

int main() {
    const int SIZE = 5;
    int studentIDs[SIZE] = { 142, 803, 027, 739, 546 };
    int id, id_index;

    cout << "Enter the student ID to search for: ";
    cin >> id;

    id_index = binarySearch(id, studentIDs, SIZE);

    if(id_index >= 0) {
        cout << "That student ID is not registered" << endl;
    } else {
        cout << "That student ID is found at index ";
        cout << id_index << endl;
    }
}
```
Sample Binary Search Question

Target value is 42.

Given the following array of ints, record the values stored in the variables first, last, and middle during each iteration of a binary search.

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

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

Now search for value 82...

```
first  0 8 8 8 9
last   14 14 10 8 8
middle 7 11 9 8 –
```

Careful: These are indices, not values!
Algorithm Analysis: Binary Search

- If $N$ is the # of elements in the array, how many comparisons (steps)?
  - Think about worst case: target is in last element checked, or target is not in the array at all

<table>
<thead>
<tr>
<th>$N = 16$:</th>
<th>items left to search:</th>
<th>comparisons so far:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

(...and then one last comparison for last item)

- How many times can we divide $N$ in half without going below 1? $\log_2 N$
  
  $16 = 2^4 \iff \log_2 16 = 4$

  $N = 2^{\text{steps}} \iff \log_2 N = \text{steps}$
So Is This Better? **MUCH!**

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N/2$</th>
<th>$\log_2 N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2.5</td>
<td>2.3</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>3.3</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>
<tr>
<td>500,000</td>
<td>250,000</td>
<td>18.9</td>
</tr>
<tr>
<td>5,000,000</td>
<td>2,500,000</td>
<td>22.3</td>
</tr>
<tr>
<td>50,000,000</td>
<td>25,000,000</td>
<td>25.6</td>
</tr>
</tbody>
</table>

Is this really a fair comparison?

Remember the search-for-NetID example from the Algorithms lecture?

18 iterations \( \times \frac{1 \text{ second}}{50,000 \text{ iterations}} < 0.0004 \text{ seconds} \)
Sorting

• **The sorting problem**: re-arrange the items in an array into ascending or descending order

• Why is sorting important?
  • Often need to display data in sorted order, e.g.
    • Phone book
    • Dictionary entries
    • Bank transactions in order by date
  • **Searching in a sorted list is much more efficient than searching in an unsorted list!**

• There are many sorting algorithms and you'll learn more in later courses. For now, only 2 (not terribly good ones)...

Bubble Sort

- On each pass...
  - Compare the first two elements. If the first is bigger, swap them (exchange places)
  - Compare the second and third elements. If the second is bigger than the third, swap them
  - Repeat until you've compared the last two elements
  - Repeat this process, re-starting from front of array each time, until an entire pass completes with no swaps
Bubble Sort, Illustrated

7 2 3 8 9 1
Bubble Sort, Illustrated

2 7 3 8 9 1

2 7 3 8 9 1
Bubble Sort, Illustrated

<table>
<thead>
<tr>
<th>2</th>
<th>7</th>
<th>3</th>
<th>8</th>
<th>9</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>
Bubble Sort, Illustrated

1st Pass Complete:
5 compares, 3 swaps
Largest element is now in last position
Bubble Sort, Illustrated

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>1</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>
Bubble Sort, Illustrated

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>1</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

2nd Pass Complete:
5 compares, 1 swap
Largest 2 elements now in last 2 positions
Bubble Sort, Illustrated

2  3  7  1  8  9

2  3  7  1  8  9

2  3  7  1  8  9
Bubble Sort, Illustrated

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>7</th>
<th>1</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

3rd Pass Complete:
- 5 compares, 1 swap
- Largest 3 elements now in last 3 positions
Bubble Sort, Illustrated

4th Pass:

```
2 3 1 7 8 9
```
Bubble Sort, Illustrated

4th Pass:

```
 2 1 3 7 8 9
```

5th Pass:

```
 2 1 3 7 8 9
```
Bubble Sort, Illustrated

4th Pass:

```
2 1 3 7 8 9
```

5th Pass:

```
1 2 3 7 8 9
```

6th Pass:

```
1 2 3 7 8 9
```

No swaps
List is now sorted!
Why Does This Work?

• At the end of the first pass, the largest element in the array has "bubbled" to the end
  • It will always be larger than its neighbors, always swapped towards back of array

• At the end of second pass, the second-largest element has "bubbled" to the second-to-last slot, etc.

• The back end (tail) of the list remains sorted, and each pass increases the size of the sorted tail portion

• If a pass completes with no exchanges, it means each element is smaller than its next neighbor
  • I.e., the list is sorted
void bubbleSort(int array[], int size) {

    bool swapped;
    int temp;

    do { // each iteration of do/while loop is a pass
        swapped = false;

        // for loop checks each pair of elements
        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;
                swapped = true;
            }
        }
    }
    while(swapped); // terminate when full pass doesn't
                    // make any swaps
}
Algorithm Analysis: Bubble Sort

• Each pass makes \( N - 1 \) comparisons

• How many passes are there?
  • It depends…
  • Each pass moves the next largest element to its spot in the sorted tail portion
  • Worst case: the smallest list element is at the very end. (Each iteration moves largest unsorted element to tail, shifting smallest element left by 1. It must shift \( N \) spaces).
    • \( N \) passes

• Worst case: \((N - 1) \times N = N^2 - N = O(N^2)\)
Selection Sort

- Issues with bubble sort:
  - Bad when small element starts near end of list
  - Inefficient for large arrays because items only move one array element at a time

- **Selection sort** has a fixed # of passes: one for each position in the array

- Sorted portion of list grows at the front (head) of array

- On each pass, smallest element in unsorted portion of list is swapped with element at current position
  - Each pass increases the size of the sorted portion
  - *Moves items immediately into their final position in sorted array*
Selection Sort, Illustrated

5 7 2 9 8 1

unsorted
Selection Sort, Illustrated

5 7 2 9 8 1

unsorted
Selection Sort, Illustrated

1 7 2 9 8 5
sorted unsorted

1 7 2 9 8 5
sorted unsorted
Selection Sort, Illustrated

1 7 2 9 8 5
sorted  unsorted

1 2 7 9 8 5
sorted  unsorted

1 2 7 9 8 5
sorted  unsorted
Selection Sort, Illustrated

1 7 2 9 8 5
sorted unsorted

1 2 7 9 8 5
sorted unsorted

1 2 5 9 8 7
sorted unsorted
Selection Sort, Illustrated

1 2 5 9 8 7

sorted  unsorted
Selection Sort, Illustrated

![Selection Sort Diagram](image-url)
Selection Sort, Illustrated

Sort Complete:
Last element doesn't need to be considered (guaranteed to be maximum element)
Selection Sort in C++

```cpp
int findIndexOfMin(int array[], int size, int start) {
    int indexOfMin = start;
    for(int i = start + 1; i < size; i++) {
        if(array[i] < array[indexOfMin]) {
            indexOfMin = i;
        }
    }
    return indexOfMin;
}

void selectionSort(int array[], int size) {
    int temp;
    int indexOfMin;
    for(int i = 0; i < (size - 1); i++) {
        indexOfMin = findIndexOfMin(array, size, i);
        // swap minimum element into current position
        temp = array[indexOfMin];
        array[indexOfMin] = array[i];
        array[i] = temp;
    }
}
```
Algorithm Analysis: Selection Sort

- Outer loop executes N-1 times
  - *One pass for each position* in the array (except the last)
- Inner loop (in findIndexOfMin) executes N-1 times the 1st time it's called, then N-2 times the next, then N-3 times, etc. as unsorted portion grows smaller
- Total number of *comparisons* (from inner loop):
  \[
  (N-1) + (N-2) + ... + 2 + 1 = \frac{N(N+1)}{2} - N
  \]
  \[
  = \frac{N^2 + N}{2} - \frac{2N}{2}
  \]
  \[
  = \frac{N^2 + N - 2N}{2}
  \]
  \[
  = \frac{N^2 - N}{2}
  \]
  \[
  = O(N^2)
  \]

What is Selection Sort's best-case efficiency?

What about Bubble Sort?

So which is "better"???