Searching & Sorting

Topics

- Searching
  - Linear Search
  - Binary Search
- Sorting
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
- Efficiency
  - Big O notation
Searching

- **Typical Problem**: Find an occurrence of a given value (the search key) in a collection of n elements.
- As examples we will consider searching for a particular integer in an array of int’s, and return the index where it is found.
  - The same ideas can be used to search for other types of information, e.g. find a contact with a given last name.
  - We can also use the same ideas for searching other structures (besides arrays) such as ArrayLists, etc.
- If the value we are searching for is not in the array we will return the value -1.

Linear Search

- For each element of the array, beginning with the first:
  - Compare the array element with the search key.
  - If they are equal, stop and return the index, otherwise continue with the next element.

```java
public int linearSearch(int[] a, int key) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] == key) {
            return i;
        }
    }
    return -1;
}
```
Efficiency of Linear Search

- Observations
  - We are most interested in efficiency when the size of the array is large.
  - The initialization step, \( i = 0 \), is performed just once.
  - A \textit{return} statement is executed just once.
  - Most of the work is performed by the operations that are repeated over and over in the loop.
    - The loop test comparison: \( i < a.\text{length} \)
    - The search key comparison: \( a[i] == \text{key} \)
    - The re-initialization: \( i++ \)

Efficiency of Linear Search (2)

- How many operations are performed each time through the loop? 3
- Assuming each operation has the same cost, \( k \), what is the cost of each iteration of the loop? \( 3k \)
- How many times do we go into the loop?
  - Best Case: 1
  - Worst Case: \( n \)
  - Average Case: \( n/2 \)
- What is the total cost of the loop?
  - Best Case: \( 3k \)
  - Worst Case: \( 3k(n) \)
  - Average Case: \( 3k(n/2) \)
**Efficiency of Linear Search (3)**

- Assuming the initialization step and return statement also have cost $k$, what is the average case total cost of linear search?
  \[ \text{cost} = 2k + 3k(n/2) \]

- If the size of the array, $n$, is very large, how significant is $2k$?
  \[ \text{cost} = 3k(n/2) \]

- Since we don’t know $k$ (it depends on processor speed, etc.) we might as well use a different constant $k' = 3k/2$.
  \[ \text{cost} = k'n \]

- For large values of $n$, the cost grows in proportion to $n$. In Big-Oh notation we write:
  \[ \text{cost} = O(n) \]

---

**Binary Search**

- With linear search, each time we test an element of the array, we eliminate one possibility of where the key could be found.
  - The search space is reduced by one element.

- If the array is sorted, we can do much better.
  - Start by looking in the middle of the array.
  - If the element there is too big, we only need to continue the search in the first half of the array.
  - If the element in the middle is too small, we continue the search in the second half of the array.
  - Either way, half of the array has been eliminated, e.g. the search space is cut in half.
  - We continue the search by looking in the middle of the search space (the portion of the array we have not yet eliminated).
**Binary Search Example**

- Search for 42 in the array:

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>6</th>
<th>8</th>
<th>9</th>
<th>12</th>
<th>23</th>
<th>31</th>
<th>36</th>
<th>40</th>
<th>42</th>
<th>64</th>
<th>67</th>
<th>72</th>
<th>88</th>
<th>95</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>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

  - Look in position \( (0 + 15)/2 = 7 \):

    | 1 | 3 | 6 | 8 | 9 | 12 | 23 | 31 | 36 | 40 | 42 | 64 | 67 | 72 | 88 | 95 |
    |---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|
    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

  - Look in position \( (8 + 15)/2 = 11 \):

    | 1 | 3 | 6 | 8 | 9 | 12 | 23 | 31 | 36 | 40 | 42 | 64 | 67 | 72 | 88 | 95 |
    |---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|
    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

  - Look in position \( (8 + 10)/2 = 9 \):

    | 1 | 3 | 6 | 8 | 9 | 12 | 23 | 31 | 36 | 40 | 42 | 64 | 67 | 72 | 88 | 95 |
    |---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|
    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

**Binary Search Code**

```java
public int binarySearch(int[] a, int key) {
    int start = 0;
    int end = a.length - 1;
    while (start <= end) {
        int middle = (start + end) / 2;
        if (a[middle] == key)
            return middle;
        else if (a[middle] < key)
            start = middle + 1;
        else
            end = middle - 1;
    }
    return -1;
}
```
Efficiency of Binary Search

Again, most of the work is done in the operations that are performed over and over in the loop:
- The loop test comparison (1 operation):
  - `start <= end`
- The addition, division, and assignment (3 operations):
  - `middle = (start + end)/2`
- The search key comparisons (2 operations):
  - `a[i] == key`
  - `a[i] < key`  (every time except the last)
- The re-initialization (2 operations). Either addition followed by an assignment, or subtraction followed by assignment:
  - `start = middle + 1 or`
  - `end = middle - 1`

Efficiency of Binary Search (2)

- How many operations are performed each time through the loop? 8
- Assuming each operation has the same cost, k, what is the cost of each iteration of the loop? 8k
- How many times do we go into the loop?
  - Best Case: 1
  - Worst Case: `[log₂n]`
  - Average Case: About the same as worst case.
- What is the total cost of the loop?
  - Best Case: 8k
  - Worst Case: `8k[log₂n]`
  - Average Case: About the same as worst case.
Efficiency of Binary Search (3)

- For large values of n, the cost grows in proportion to $\log_2 n$ (which is also proportional to $\log_{10} n$). In Big-Oh notation we write:

\[
\text{cost} = O(\log n)
\]

Comparison of Linear and Binary Search

- Let’s consider the looping cost for each algorithm for the average case and worst case.

<table>
<thead>
<tr>
<th>Linear Search</th>
<th>n</th>
<th>iterations (worst case)</th>
<th>iterations (avg case)</th>
<th>cost (worst case)</th>
<th>cost (avg case)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>24</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>1,000</td>
<td>500</td>
<td>3,000</td>
<td>1,500</td>
</tr>
<tr>
<td></td>
<td>1,000,000</td>
<td>1,000,000</td>
<td>500,000</td>
<td>3,000,000</td>
<td>1,500,000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary Search</th>
<th>n</th>
<th>iterations (worst case)</th>
<th>iterations (avg case)</th>
<th>cost (worst case)</th>
<th>cost (avg case)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>10</td>
<td>10</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>1,000,000</td>
<td>20</td>
<td>20</td>
<td>160</td>
<td>160</td>
</tr>
</tbody>
</table>
Selection Sort

- One simple sorting algorithm is *selection sort*:
  - **step:** 0) Find the smallest element and put it in position 0.

  What do you do with the value that was already in position 0?
  1) Find the 2nd smallest element and put it in position 1.
  2) Find the 3rd smallest element and put it in position 2.

  ⋮

  n-2) Find the (n-1) smallest and put it in position n-2

- Since elements 0 through n-2 are in the correct place, element n-1 must also be correct.

Selection Sort Example

At step i we find the smallest element from the unsorted portion of the array and swap it into position i.

At the beginning of step i, the unsorted portion of the array starts at index i.
Selection Sort Code

```java
public void selectionSort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        // Find the smallest remaining element
        int minIndex = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[minIndex])
                minIndex = j;
        }
        // Swap it into position i
        int tmp = a[i];
        a[i] = a[minIndex];
        a[minIndex] = tmp;
    }
}
```

What if the correct element was already at position i?

Selection Sort Efficiency

- Should we swap only when necessary?

```java
if (a[i] != a[minIndex]) {
    int tmp = a[i];
    a[i] = a[minIndex];
    a[minIndex] = tmp;
}
```

- Without the if statement, we always perform 3 operations to complete the swap.
- With the if statement, we perform only 1 operation (the test) when the swap is unnecessary (saving 2), but 4 operations when the swap is necessary (costing 1).
- We break even when the odds are 1/3 that the swap is unnecessary. This is the next to last iteration of the loop. On all earlier iterations we lose by including the if statement!
Selection Sort Efficiency (2)

- Consider how many times each operation is performed when sorting an array of size n.
  - The outer loop initialization is performed only once.
  - The body of the outer loop is executed \( n - 1 \) times, so:
    - The loop test, 3 swap operations, and re-initialization are each performed \( n - 1 \) times.
    - The initialization of the inner loop is performed \( n - 1 \) times.
  - How many times is the body of the inner loop executed?

Selection Sort Efficiency (3)

<table>
<thead>
<tr>
<th>( i )</th>
<th>inner loop iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( n - 1 )</td>
</tr>
<tr>
<td>1</td>
<td>( n - 2 )</td>
</tr>
<tr>
<td>2</td>
<td>( n - 3 )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( n - 4 )</td>
<td>3</td>
</tr>
<tr>
<td>( n - 3 )</td>
<td>2</td>
</tr>
<tr>
<td>( n - 2 )</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \text{iterations} = n \cdot \frac{n-1}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \]

\[ \text{cost} = k_1n^2 + k_2n + k_3 = O(n^2) \]

- The best known sorting algorithms have a running time of: \( O(n \log n) \)
Bubble Sort

- Bubble sort works by making repeated passes over the array.
- In each pass, adjacent elements are compared as we loop over the array.
- If a pair of elements are in the wrong order, we swap them.
- During the first pass, the largest element will “bubble up” to the end of the array.
- In each remaining pass, another element bubbles up into the correct position.

Bubble Sort Code

```java
public int bubbleSort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        for (int j = 1; j < a.length - i; j++) {
            if (a[j-1] > a[j]) {
                int tmp = a[j];
                a[j] = a[j-1];
                a[j-1] = tmp;
            }
        }
    }
}
```

- The nested loops result in the same kind of efficiency as for selection sort:

  \[ \text{cost} = O(n^2) \]