Algorithmic Efficiency

Linear Search
• The simplest strategy for searching is to start at the beginning of the array and look at each element in turn. This algorithm is called linear search.
• Linear search is straightforward to implement, as illustrated in the following JavaScript function that returns the first index at which the value `key` appears in `array`, or `-1` if it does not appear at all:

  ```javascript
  function linearSearch(key, array) {
    for (var i = 0; i < array.length; i++) {
      if (key === array[i]) return i;
    }
    return -1;
  }
  ```

Searching for Area Codes
• To illustrate the efficiency of linear search, it is useful to work with a somewhat larger example.
• The example on the next slide works with an array containing many of the area codes assigned to the United States.
• The specific task in this example is to search this list to find the area code for the Silicon Valley area, which is 650.
• The linear search algorithm needs to examine each element in the array to find the matching value. As the array gets larger, the number of steps required for linear search grows in the same proportion.
• As you watch the slow process of searching for 650 on the next slide, try to think of a more efficient way in which you might search this particular array for a given area code.

Linear Search (Area Code Example)
Linear search needs to look at 166 elements to find 650.

The Idea of Binary Search
• The fact that the area code array is in ascending order makes it possible to find a particular value much more efficiently.
• The fundamental insight is that starting at the middle element gives you more information than starting at the beginning.
• When you look at the middle element in relation to the value you’re searching for, there are three possibilities:
  – If the key is greater than the middle element, you can discard every element in the first half.
  – If the key is less than the middle element, you can discard every element in the second half.
  – If the key is equal to the middle element, you can stop.
• You can repeat this process on the elements that remain after each cycle. Because this algorithm proceeds by dividing the list in half each time, it is called binary search.
**Binary Search (Area Code Example)**

Binary search needs to look at only eight elements to find 650.

1. Binary search works only on arrays in which the elements are arranged in order. The process of putting the elements of an array in order is called **sorting.**
2. There are many algorithms that one can use to sort an array. As with searching, these algorithms can vary substantially in their efficiency, particularly as the arrays become large.
3. Of all the algorithms presented in this text, sorting is by far the most important in terms of its practical applications. Alphabetizing a telephone directory, arranging library records by catalogue number, and organizing a bulk mailing by ZIP code are all examples of sorting that involve reasonably large collections of data.

**Efficiency of Binary Search**

- The running time of binary search also depends on the number of elements, but in a profoundly different way.
- On each step, the binary search algorithm rules out half of the remaining possibilities. In the worst case, the number of steps is given by how many times you can divide the original size of the array in half until there is only one element remaining. In other words, you need to find the value of $k$ for which:
  $\frac{1}{2}^k \leq N$
- You can simplify this formula using basic mathematics:
  $1 = N / 2^k$
  $2^k = N$
  $k = \log_2 N$

**Efficiency of Linear Search**

- As the area code example makes clear, the running time of the linear search algorithm depends on the size of the array.
- The idea that the time required to search a list of values depends on how many values there are is not at all surprising. The running time of most algorithms depends on the size of the problem to which that algorithm is applied.
- In many applications, it is easy to come up with a numeric value that specifies the problem size, which is generally denoted by the letter $N$. For most array applications, the problem size is simply the size of the array.
- In the worst case—which occurs when the value you're searching for comes at the end of the array or does not appear at all—linear search requires $N$ steps. On average, it takes approximately half that time.

**Comparing Search Efficiencies**

- The difference in the number of steps required for the two search algorithms is illustrated by the following table, which compares the values of $N$ and the closest integer to $\log_2 N$:

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\log_2 N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>30</td>
</tr>
</tbody>
</table>

- For large values of $N$, the difference in the number of steps required is enormous. If you had to search through a list of a million elements, binary search would run 50,000 times faster than linear search. If there were a billion elements, that factor would grow to 33,000,000.

**Sorting**

- Of the many sorting algorithms, the easiest one to describe is **selection sort**, which is implemented by the following code:

```javascript
function sort(array) {
  for (var lh = 0; lh < array.length - 1; lh++) {
    var rh = findSmallest(array, lh, array.length); // find element with the smallest value
    swapElements(array, lh, rh); // exchange elements in position
  }
}
```

The variables $lh$ and $rh$ indicate the positions of the left and right hands if you were to carry out this process manually. The left hand points to each position in turn; the right hand points to the smallest value in the rest of the array.

- The method `findSmallest(array, p0, p1)` returns the index of the smallest value in the array from position $p_0$ up to but not including $p_1$. The method `swapElements(array, p0, p1)` exchanges the elements at the specified positions.

**The Selection Sort Algorithm**
Simulating Selection Sort

```javascript
function sort(array) {
    for (var lh = 0; lh < array.length; lh++) {
        var rh = findSmallest(array, lh, array.length);
        exchange(array, rh, lh);
    }
}
```

The Efficiency of Selection Sort

- The computational complexity of an algorithm is typically typically proportional to the number of times the most frequent operation is executed. In selection sort, this operation is the body of loop in `findSmallest`. The number of cycles in this loop changes as the algorithm proceeds:
  - `N` values are considered on the first call to `findSmallest`.
  - `N-1` values are considered on the second call.
  - `N-2` values are considered on the third call, and so on.

- In mathematical notation, the number of values considered in `findSmallest` can be expressed as a summation, which can then be transformed into a simple formula:
  \[ 1 + 2 + 3 + \ldots + (N-1) + N = \sum_{i=1}^{N} i = \frac{N(N+1)}{2} \]

Quadratic Growth

- The following table shows the value of \( \frac{N(N+1)}{2} \) for various values of `N`:

<table>
<thead>
<tr>
<th>N</th>
<th>( N(N+1)/2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>55</td>
</tr>
<tr>
<td>100</td>
<td>5050</td>
</tr>
<tr>
<td>1000</td>
<td>500,500</td>
</tr>
<tr>
<td>10,000</td>
<td>5,000,500</td>
</tr>
</tbody>
</table>

- The growth pattern in the right column is similar to that of the measured running time of the selection sort algorithm. As the value of `N` increases by a factor of 10, the value of \( \frac{N(N+1)}{2} \) increases by a factor of around 100, which is \( 10^2 \). Algorithms whose running times increase in proportion to the square of the problem size are said to be quadratic.

Sorting Punched Cards

From the 1880 census onward, information was often stored on punched cards like the one shown at the right, in which the number 236 has been punched in the first three columns.

Computer companies built machines to sort stacks of punched cards, such as the IBM 083 sorter on the left. The stack of cards was fed into a large hopper at the right end of the machine, and the cards would then be distributed into the various bins on the front of the sorter according to what value was punched in a particular column.

The Radix Sort Algorithm

- The IBM 083 sorter outperforms selection sort by using an algorithm called radix sort that requires the following steps:
  1. Set the machine so that it sorts on the last digit of the number.
  2. Put the entire stack of cards in the hopper.
  3. Run the machine so that the cards are distributed into the bins.
  4. Put the cards from the bins back in the hopper, making sure that the cards from the 9 bin are on the bottom, the cards from the 0 bin come on top of those, and so on.
  5. Reset the machine so that it sorts on the preceding digit.
  6. Repeat steps 3 through 5 until all the digits are processed.
- The next slide illustrates this process for a set of three-digit numbers.

Comparing \( N^2 \) and \( N \log N \)

- Radix sort runs in time proportional to the number of values times the number of digits, which is bounded by \( N \log N \).
- The difference between \( N^2 \) and \( N \log N \) can be enormous for large values of `N`, as shown in this table:

<table>
<thead>
<tr>
<th>N</th>
<th>( N^2 )</th>
<th>( N \log N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>33</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000,000</td>
<td>9,966</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000,000</td>
<td>172,877</td>
</tr>
<tr>
<td>100,000</td>
<td>100,000,000</td>
<td>1,668,964</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000,000</td>
<td>19,931,569</td>
</tr>
</tbody>
</table>
Big-O Notation

- The most common way to express computational complexity is to use big-O notation, which was introduced by the German mathematician Paul Bachmann in 1892.
- Big-O notation consists of the letter $O$ followed by a formula that offers a qualitative assessment of running time as a function of the problem size, traditionally denoted as $N$. For example, the computational complexity of linear search is $O(N)$ and the computational complexity of selection sort is $O(N^2)$.
- If you read these formulas aloud, you would pronounce them as “big-O of $N$” and “big-O of $N^2$” respectively.

Standard Complexity Classes

- The complexity of a particular algorithm tends to fall into one of a small number of standard complexity classes:

<table>
<thead>
<tr>
<th>Complexity Class</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>logarithmic</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>linear</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>quadratic</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>cubic</td>
<td>$O(N^3)$</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^N)$</td>
</tr>
</tbody>
</table>

- In general, theoretical computer scientists regard any problem whose complexity cannot be expressed as a polynomial as intractable.

Graphs of the Complexity Classes