Searching and Sorting

Eric Roberts
CSCI 121
October 16, 2019

Pioneers of Computer-Based Sorting

• One of the earliest computers was the ENIAC, completed in 1945 at the Moore School in Philadelphia.
• Betty Holberton (née Foster) was one of six women who worked as programmers for the machine.
• After the war, Holberton worked for Remington Rand, where she developed SORT/MERGE, which translated specifications of sorting problems into machine code.
• In 1997, Holberton received both the Ada Lovelace Award and the IEEE Computer Pioneer Award.

Searching

• Chapter 8 looks at two operations on arrays—searching and sorting—both of which turn out to be important in a wide range of practical applications.

• The simpler of these two operations is searching, which is the process of finding a particular element in an array or some other kind of sequence. Typically, a method that implements searching will return the index at which a particular element appears, or -1 if that element does not appear at all. The element you’re searching for is called the key.

• The goal of Chapter 8, however, is not simply to introduce searching and sorting but rather to use these operations to talk about algorithms and efficiency. Many different algorithms exist for both searching and sorting; choosing the right algorithm for a particular application can have a profound effect on how efficiently that application runs.

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 method that returns the first index at which the value key appears in array, or -1 if it does not appear at all:

```python
def linearSearch(key, array):
    for i in range(len(array)):
        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 primary area code for Portland, which is 503.

• 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 503 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.

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 you get more information by starting in the middle than by starting at the beginning.

• When you look at the middle element in relation to the key value for which you’re searching, there are three possibilities:
  - If the key value is greater than the middle element, you can discount every element in the first half of the array.
  - If the key value is less than the middle element, you can discount every element in the second half of the array.
  - If the key value is equal to the middle element, you’ve found it.

• 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.
Tracing Binary Search

Implementing Binary Search

• The following method implements binary search for an array:

```python
def binarySearch(key, array):
    min = 0
    max = len(array) - 1
    while min <= max:
        mid = (min + max) // 2
        if key == array[mid]:
            return mid
        elif key < array[mid]:
            max = mid - 1
        else:
            min = mid + 1
    return -1
```

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.

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 in the process, the binary search algorithm rules out half of the remaining possibilities. In the worst case, the number of steps required is equal to the number of times you can divide the original size of the array in half until there is only one element remaining. In other words, what you need to find is the value of $k$ that satisfies the following equation:

$$1 = \frac{N}{2} / 2 / 2 / 2 \cdots / 2$$

• You can simplify this formula using basic mathematics:

$$2^k = N$$

$$k = \log_2 N$$

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>20</td>
<td>5</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>20</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

• 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.
• 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.
• Of the algorithms presented in this course, sorting is probably 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 large collections of data.
The Selection Sort Algorithm

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

```
def sort(array):
    for lh in range(len(array)):  
        rh = lh
        for i in range(lh + 1, len(array)):
            if array[i] < array[rh]:
                rh = i
        array[lh], array[rh] = array[rh], array[lh]
```

• 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 Efficiency of Selection Sort

• Chapter 8 includes a table of actual running times for the selection algorithm for arrays of varying sizes.

• Another way to estimate efficiency is to count how many times the most frequent operation is executed. In selection sort, this operation is the comparison in the inner loop. The number of cycles changes as the algorithm proceeds:
  - `N` values are considered on the first cycle of the loop.
  - `N - 2` values are considered on the second cycle, and so on.

• In mathematical notation, the total number of comparisons can be expressed as a summation, which can then be written as a simple formula:

\[
1 + 2 + 3 + \cdots + (N - 1) + N = \sum_{i=1}^{N} i = \frac{N(N + 1)}{2}
\]

The Radix Sort Algorithm

• The IBM 083 sorter sorts much more quickly that selection sort by using an algorithm called radix sort, which 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 0 bin are on the bottom, the cards from the 1 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.

Quadratic Growth

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

<table>
<thead>
<tr>
<th><code>N</code></th>
<th><code>N(N+1)/2</code></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>5,005,000</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 `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 loaded in 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.
Simulating Radix Sort

Step 1a. Sort the cards using the last digit.
Step 1b. Refill the hopper by emptying the bins in order.
            Note that the list is now sorted by the last digit.
Step 2a. Sort the cards using the middle digit.
Step 2b. Again refill the hopper by emptying the bins.
            The list is now sorted by the last two digits.
Step 3a. Sort the cards using the first digit.
Step 3b. Refill the hopper one last time.
            The list is now completely sorted.