Search Algorithms

Overview

- The Search Problem
- Performance of Search Algorithms
- Types of Search Algorithms
- Linear Search (& how to beat it)
- Binary Search

The Search Problem

- Searching is a fundamental operation in computing.
- E.g. finding information on the web, looking up a bank account balance, finding a file or application on a PC, querying a database…
- The Search Problem relates to retrieving information from a data structure (e.g. Arrays, Linked Lists, Search Trees, Hash Tables etc.).
- Algorithms which solve the Search Problem are called Search Algorithms.
- Includes algorithms which query a data structure, such as the SQL SELECT command.

The Search Problem

- Two fundamental queries about any collection C:
  - **Existence**: Does C contain a target element? Given a collection C, we often simply want to know whether the collection already contains a given element t. The response to such a query is true if an element exists in the collection that matches the desired target t, or false if this is not the case.
  - **Associative lookup:** Return information associated in collection *C* with a target key value *k*. A key is usually associated with a complex structure called a value. The lookup retrieves or replaces this value.
- A correct Search Algorithm should return true or the index of the requested key if it exists in the collection, or -1/null/false if it does not exist in the collection.

Reference: Pollice G., Selkow S. and Heineman G. (2016). Algorithms in a Nutshell, 2nd Edition. O' Reilly.

Performance of Search Algorithms

- Ultimately the performance of a Search Algorithm is based on how many operations performed (elements the algorithm inspects) as it processes a query (as we saw with Sorting Algorithms).
- Constant O(n) time is the worst case for any (reasonable) searching algorithm – corresponds to inspecting each element in a collection exactly once.
- As we will see, sorting a collection of items according to a comparator function (some definition of "less than") can improve the performance of search queries.
- However, there are costs associated with maintaining a sorted collection, especially for use cases where frequent insertion or removal of keys/elements is expected.
- Trade-off between cost of maintaining a sorted collection, and the increased performance which is possible because of maintaining sorted data.
- Worth pre-sorting the collection if it will be searched often.

Pre-sorted data

- Several search operations become trivial when we can assume that a collection of items is sorted.
- E.g. Consider a set of economic data, such as the salary paid to all employees of a company. Values such as the minimum, maximum, median and quartile salaries may be retrieved from the data set in constant $O(1)$ time if the data is sorted.
- Can also apply more advanced Search Algorithms when data is pre-sorted.

Types of Search Algorithms

- **Linear**: simple (naïve) search algorithm
- **Binary**: better performance than Linear
- **Comparison**: eliminate records based on comparisons of record keys
- **Digital**: based on the properties of digits in record keys
- **Hash-based**: maps keys to records based on a hashing function

As we saw with Sorting Algorithms, there is no universal best algorithm for every possible use case. The most appropriate search algorithm often depends on the data structure being searched, but also on any a priori knowledge about the data.

Linear Search

- Linear Search (also known as Sequential Search) is the simplest Searching Algorithm; it is trivial to implement.
- Brute-force approach to locate a single target value in a collection.
- Begins at the first index, and inspects each element in turn until it finds the target value, or it has inspected all elements in the collection.
- It is an appropriate algorithm to use when we do not know whether input data is sorted beforehand, or when the collection to be searched is small.
- Linear Search does not make any assumptions about the contents of a collection; it places the fewest restrictions on the type of elements which may be searched.
- The only requirement is the presence of a match function (some definition of “equals”) to determine whether the target element being searched for matches an element in the collection.

Linear Search

- Perhaps you frequently need to locate an element in a collection that may or may not be ordered.
- With no further knowledge about the information that might be in the collection, Linear Search gets the job done in a brute-force manner.
- It is the only search algorithm which may be used if the collection is accessible only through an iterator.
- Constant $\mathrm{O}(1)$ time in the best case, linear $\mathrm{O}(n)$ time in the average and worst cases. Assuming that it is equally likely that the target value can be found at each array position, if the size of the input collection is doubled, the running time should also approximately double.
- Best case occurs when the element sought is at the first index, worst case occur when the element sought is at the last index, average case occurs when the element sought is somewhere in the middle.
- Constant space complexity.

Linear Search example 1

Search the array a for the target value 7:

0	1	2	3	4	5	6
2	1	5	4	8	7	9

1. a[0] = 2; this is not the target value, so continue
2. a[1] = 1; this is not the target value, so continue
3. a[2] = 5; this is not the target value, so continue
4. a[3] = 4; this is not the target value, so continue
5. a[4] = 8; this is not the target value, so continue
6. a[5] = 7; this is the target value, so return index 5

Linear Search example 2

Search the array a for the target value 3:

0	1	2	3	4	5	6
2	1	5	4	8	7	9

1. a[0] = 2; this is not the target value, so continue
2. a[1] = 1; this is not the target value, so continue
3. a[2] = 5; this is not the target value, so continue
4. a[3] = 4; this is not the target value, so continue
5. a[4] = 8; this is not the target value, so continue
6. a[5] = 7; this is not the target value, so continue
7. a[6] = 9; this is not the target value, so continue
8. Iteration is complete. Target value was not found so return -1

Linear Search in Java

```java
static int linearSearch(int[] a, int value) {
    for(int i=0; i<a.length; i++) {
        if(a[i] == value) {
            //return index if the value exists
            return i;
        }
    }
    //return -1 if the value does not exist
    return -1;
}
```

Binary Search

- Linear Search has linear time complexity:
  - Time n if the item is not found
  - Time $n/2$, on average, if the item is found
- If the array is sorted, faster searching is possible
- How do we look up a name in a phone book, or a word in a dictionary?
  - Look somewhere in the middle
  - Compare what’s there with the target value that you are looking for
  - Decide which half of the remaining entries to look at
  - Repeat until you find the correct place
  - This is the Binary Search Algorithm

Binary Search

- Better than linear running time is achievable only if the data is sorted in some way (i.e. given two index positions, i and j, a[i] < a[j] if and only if i < j).
- Binary Search delivers better performance than Linear Search because it starts with a collection whose elements are already sorted.
- Binary Search divides the sorted collection in half until the target value is found, or until it determines that the target value does not exist in the collection.
- Binary Search divides the collection approximately in half using whole integer division (i.e. 7/4 = 1, where the remainder is disregarded).
- Binary Search has logarithmic time complexity and constant space complexity.

Binary Search example 1

Search the array a for the target value 7:

0	1	2	3	4	5	6
2	4	5	6	7	8	9

1. Middle index =(0+6)/2 = 3. a[3] = 6 < 7, so search in the right subarray (indices left=4 to right=6)
2. Middle index =(4+6)/2 = 5. a[5] = 8 > 7, so search in the left subarray (indices left=4 to right=4)
3. Middle index =(4+4)/2 = 4. a[4] = 7 = 7, so return index 4

Binary Search example 2

Search the array a for the target value 3:

0	1	2	3	4	5	6
2	4	5	6	7	8	9

1. Middle index =(0+6)/2 = 3. a[3] = 6 > 3, so search next in the left subarray (indices left=0 to right=2)
2. Middle index =(0+2)/2 = 1. a[1] = 4 > 3, so search next in the left subarray (indices left=0 to right=1)
3. Middle index =(0+1)/2 = 0. a[0] = 2 < 3, so search next in the right subarray (indices left=0 to right=0)
4. Middle index =(0+0)/2 = 0. a[0] = 2 < 3. Now there is just one element left in the partition (index 0) – this corresponds to one of two possible base cases. Return -1 as this element is not the target value.

Iterative Binary Search in Java

```java
static int binarySearch(int[]a, int target) {
    int left = 0;
    int right = a.length - 1;

    while(right >= left) {
        int middle = (left + right) / 2;
        if(a[middle] == target) {
            return middle; //return index if the value exists
        }
        if(a[middle] < target) {
            left = middle + 1;
        }
        if(a[middle] > target) {
            right = middle - 1;
        }
    }
    return -1; //return -1 if the value does not exist
}
```

Recursive Binary Search in Java

```java
static int binarySearch(int a[], int left, int right, int target) {
    if (right>=left) {
        int middle = (left+right)/2; // determine mid point

        if (a[middle] == target) {
            return middle; //return index if the value exists
        }
        // If target is < a[mid], it can only be present in the left subarray
        else if (a[middle] > target) {
            return binarySearch(a, left, middle-1, target);
        }
        // Else target can only be present in the right subarray
        else {
            return binarySearch(a, middle+1, right, target);
        }
    }
    return -1; //return -1 if the value does not exist
}
```

Analysis of Binary Search

- In Binary Search, we choose an index that cuts the remaining portion of the array in half
- We repeat this until we either find the value we are looking for, or we reach a subarray of size 1
- If we start with an array of size n, we can cut it in half $log_2 n$ times
- Hence, Binary Search has logarithmic ($\log n$) time complexity in the worst and average cases, and constant time complexity in the best case
- For an array of size 1000, this is approx. 100 times faster than linear search ($2^{10} \approx 1000$ when neglecting constant factors)

Conclusion

- Linear Search has linear time complexity
- Binary Search has logarithmic time complexity
- For large arrays, Binary Search is far more efficient than Linear Search
- However, Binary Search requires that the array is sorted
- If the array is sorted, Binary Search is
  - Approx. 100 times faster for an array of size 1000 (neglecting constants)
  - Approx. 50 000 times faster for an array of size 1 000 000 (neglecting constants)
- Significant improvements like these are what make studying and analysing algorithms worthwhile