Searching and Sorting

Last updated: October 19th 2018, at 2.48pm

Contents

1 Searching

1.1 The Problem

Problem Find an entry in a collection of entries

Algorithm Search an array for a given key

> **Input:** An array of `Comparable`s and a `Comparable` key
>
> **Output:** Index into array or a `NotFound` exception

Aside 1

A `Comparable` type is a type with a total ordering — every element is less than, equal to, or greater than every other element.

This is determined by the `compareTo` method. If `a` and `b` are `Comparable`s of the same type:

- If `a` $<$ `b` then `a.compareTo(b) < 0`
- If `a` $=$ `b` then `a.compareTo(b) = 0`
- If `a` $>$ `b` then `a.compareTo(b) > 0`

See, for example, `String`s. Indeed, `String`s have a `compareTo` method because they *are* `Comparable`s.

1.2 Sequential Search

1.2.1 Description

Here we simply work through the array element by element until we find an occurrence of the key or we run out of array to search.

1.2.2 Pseudocode

```
for (int index = 0; index < array length; index++) {
   if (index^th element of array == our key) {
      return with the current index;
   }
}
throw a NotFound exception;
```

1.2.3 Java Code

```
public <T extends Comparable<?  super T>> int
    search (T key,T[] list) throws NotFound {
   for (int i = 0; i < list.length; i++) {
      if (list[i].compareTo(key) == 0) {
         return i;
      }
   }
   throw new NotFound();
}
```

Aside 2

Why

```
<T extends Comparable<?  super T>> int search
```

1. `T extends Comparable` is required to ensure `T` is comparable.
2. `Comparable` is a generic class. The generic type specifies the type of object that may be compared to. The following is permissible:

   ```
   class MyClass implements Comparable<Integer>
   ```

3. Usually (always?) want to implement a class that can compare to its own type, but `T extends Comparable<T> int search` is too restrictive — requires that `compareTo` is defined for `T`
4. `T extends Comparable<?  super T> int search` allows for `compareTo` to be defined for `T`, *or some superclass of* `T`

1.3 Binary Search

1.3.1 Description

Sometimes we can exploit properties of our data to be searched — e.g. it may be ordered in some way. Binary search works best with lists or arrays that may be sorted.

Idea is to first sort our array (see later) and then to perform our binary search. To do a binary search, we repeatably split our array, around a pivot or middle element, into two subarrays. Since our original array was sorted, we can easily determine what subarray our key might reside in by comparing our key against the pivot element.

Sooner or later, we either find our key or have no list left to search through.

1.3.2 Binary Search

```
if (element in middle of array is required element) {
    return element's index;
}  else if (middle element is bigger than required element) {
    search lower half of array;
}  else {
    search upper half of array;
}
```

1.3.3 Examples

Example 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13
−62	−27	−15	−13	−5	0	3	8	9	12	17	29	32	54

Find −5

0	1	2	3	4	5	6	7	8	9	10	11	12	13
−62	−27	−15	−13	−5	0	3	8	9	12	17	29	32	54

$-5 < 3$

The pivot is greater than the search key so discard the top half of the list (including the pivot).

0	1	2	3	4	5	6	7	8	9	10	11	12	13
−62	−27	−15	−13	−5	0	3	8	9	12	17	29	32	54

$-15 < -5$

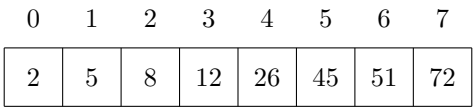
The pivot is less than the search key, so discard the bottom half of the list (again including the pivot).

0	1	2	3	4	5	6	7	8	9	10	11	12	13
−62	−27	−15	−13	−5	0	3	8	9	12	17	29	32	54

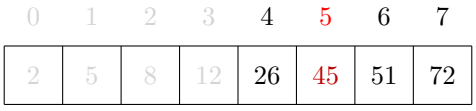
$-5 = -5$: **return** index 4

The pivot *is* the search key, so return the pivot's index.

Example 2



Key is greater, so discard lower half of list.



Key is smaller, so discard upper half of list.

Note that the list now consists of a single element.

Key is larger, so discard lower half of list.

The (active part of the) list is now empty, so the key could not be found.

Notice how we have used the ordering property of the list to discard a part of the list.

As we halve our list to be searched each time, we expect (on average) to have halved the time that a sequential search might have taken.

1.3.4 Java Code

```
/**
   * Top level call: starts recursive search
   */

public <T extends Comparable<?  super T>>
    int search(T key,T[] list) throws NotFound {
   return searchBetween(0,list.length-1,key,list);
}

public <T extends Comparable<?  super T>>
    int searchBetween(int bottom,int top, T Key,T[] list)
      throws notFound {
   int pivot = (bottom + top)/2;
   if (list[pivot].compareTo(key) == 0) {
      return pivot;
   } else if (top == bottom) {
      throw new NotFound();
   } else if (list[pivot].compareTo(key) > 0) {
      return searchBetween(bottom,pivot-1,key,list);
   } else {
      return searchBetween(pivot+1,top,key,list);
   }
}
```

Another example

2 Sorting

2.1 Introduction

2.1.1 The Problem

Imagine we have a pack of playing cards and (for simplicity) we will assume that we have only got the suit of diamonds.

The aim is to sort the diamond suit.

2.1.2 Solutions

- sort the pile we have been given (*in situ* sorting),
- or move cards from our unsorted pile to a new sorted pile ("copy" sorting).

In computing terms, we can either sort the array (or list) that we have been given, or we can create a new array (or list) that contains the same elements as the original array/list but now sorted. Generally *in situ* sorting is to be preferred. However it is often conceptually easier to think of copy sorting.

We will maintain two piles of cards:

- the one given to us,
- and a sorted pile (obviously, this is initially empty)

Generally we want to sort *in situ*. In in situ sorting the "two piles" are both parts of the original pile — e.g. the "sorted pile" might be the first n elements of the pile.

2.2 Bubble Sort

2.2.1 Description

The sorted part of the list is the last part of the list.

```
for (the whole list is unsorted;
   there is still part of the list unsorted;
   one more element has been sorted) {
   for (elements to be compared = first two elements;
      we haven't reached the end of the unsorted part;
      select the next pair of elements to be compared) {
      if (the first element > the second element) {
         swap them;
      }
   }
}
```

At the end of each "sweep" through the inner the largest element in the unsorted part will have swapped all the way up to the final position in the unsorted part — i.e. it will be sorted.

2.2.2 Algorithm

```
public <T extends Comparable<?  super T>>
    void bubbleSort(T[] array) {
   for (int unsorted = array.length-1; uns'd > 0; uns'd--) {
      for (int nextToCompare = 0; // start with first two elements
         nextToCompare < lastUnsorted; // stop at end of unsorted portion
         nextToCompare++) {
         if (array[nextToCompare].compareTo(array[nextToCompare+1]) > 0) {
            // the elements in wrong order so swap them
            swap(array,nextToCompare,nextToCompare+1);
```

```
        }
      }
   }
}
```

Example

2.3 Selection Sort

Similar to bubble sort, but reduce the number of swaps by simply finding the *largest* element in the unsorted part and swapping it with the *last* element in the unsorted part.

The sorted part of the list is the last part of the list

```
for (the whole list is unsorted;
   there is still part of the list unsorted;
   another element has been sorted) {
   find the (index of) largest unsorted element;
   swap it with the last unsorted element;
}
```

Find the largest unsorted element by searching through the unsorted part of the list an element at a time and keeping track of the largest element seen so far. When we reach the end we will have the largest element. Since we want to swap it we will need its index in the array.

2.3.1 Algorithm

The algorithm is left as an exercise.

Example

2.4 Insertion Sort

2.4.1 Description

- The sorted part of the list is the first part of the list
- ```
  for (the whole list is unsorted;
      part of the list is still unsorted;
      another element is sorted) {
      take the first unsorted element;
      insert it into the correct position in
        the sorted part;
   }
  ```
  ```

To insert the element into the correct position in the sorted part we need to find that position, shift all the sorted values that are larger along one to free up the correct position and hten insert the new element. We can do this simultaneously. The new element is the first unsorted element, so this position has been freed for use by the sorted part. We start by comparing the new element to the last element in the sorted part. If the new elment is larger than (or the same size as) the last element the new element can go in the free position. If, on the other hand, the new element is smaller we shift the last element along one into the free space, and then repeat the process with the next sorted element.

2.4.2 Algorithm

```
public <T extends Comparable<?  super T>>
    void insertionSort(T[] array) {
   for (sorted = 0; sorted < array.length-1; sorted++) {
      T newElement = array[sorted+1];
      int compareTo = sorted; // start by comparing last sorted
      while (compareTo >= 0 &&
           newElement.compareTo(array[compareTo]) < 0) {
         array[compareTo+1] = array[compareTo];
         compareTo--;
      }
      array[compareTo+1] = newElement;
   }
}
```

Example

2.5 Merge Sort

2.5.1 Description

Rather than splitting the list into a completely sorted and an unsorted part we can use a "divide and conquer" approach. The idea is to repeatedly split the list to be sorted into smaller arrays (which will be easier to sort). The two sublists then need to be glued back together in some way. Merge sort is the basic "divide and conquer" algorithm.

- If the length of the list is less than two it is sorted
- Otherwise split it into two halves and apply merge sort to these two sublists
- Now merge the two lists maintaining the ordering.

2.5.2 Algorithm

Merging

```
public <T extends Comparable<?  super T>>
    void merge(T[] target,T[] source1,T[] source2) {
   int index1=0, index2=0;
   for (index = 0; index < target.length; index++) {
      if (source1[index1].compareTo(source2[index2]) < 0) {
         target[index] = source1[index1++];
      }  else {
         target[index] = source2[index2++];
      }
   }
}
```

The two indices `index1` and `index2` indicate the elements currently under consideration in `source1` and `source2` respectively. If the element in `source1` is smaller it must be copied to the target array and `index1` must be increased to point to the next element in `source1`. Otherwise we do the same, but for `source2`.

Note that we should really first check that `target` is large enough to hold all the values in `source1` and `source2`.

Copying

```
public <T> T[] copy(T[] source,int from,int to) {
   T[] copy = new T[to-from]; // can't actually do this
   for (int i = 0; i < copy.length; i++) {
      copy[i] = source[from+i];
   }
   return copy;
}
```

Can, e.g., do this...

```
public <T> T[] copy(T[] list,int from,int to) {
   List<T> copyL = new ArrayList<T>(to-from);
   for (int i = 0; i < to-from; i++) {
      copyL.add(i,list[from+i]);
   }
   return copyL.toArray(list);
}
```

Merge Sort

```
public <T extends Comparable<?  super T>>
    void mergeSort(T[] array) {
   if (array.length < 2) {
      return;
   }
   T[] temp1 = copy(array,0,array.length/2-1);
   T[] temp2 = copy(array,array.length/2,array.length-1);
   mergeSort(temp1);
   mergeSort(temp2);
   merge(array,temp1,temp2);
}
```

Example

2.6 Quicksort

2.6.1 Description

Merge sort is time efficient, but it is space inefficient — it cannot be done in situ. Quicksort is another recursive algorithm that can be as time efficient as merge sort, but can be done in situ.

- Split the list into two smaller, "more sorted" lists.
- Choose an element (called the "pivot").
- Split the list into two sublists such that
  - all elements to the left of the pivot are smaller (or equal to) than the pivot
  - all elements to the right of the pivot are larger than the pivot

  Then the pivot will be in the right position!
- Repeat quicksort, separately, on the parts of the list to the left and right of the pivot.

2.6.2 Splitting the List

To split the list:

- Choose a pivot — e.g. the first element in the list.
- This leaves a gap in the list that will be to the left of the final position of the pivot.

loop: Find the last element in the list that is smaller than the pivot and appears after the gap.

- Put it in the gap.
- This leaves a gap towards the end of the list that will be to the right of the pivot.
- Find the first element in the list that is larger than the pivot and appears before the gap.
- Put it in the gap.
- Repeat from "**loop:**", continuing the search for small elements from where we left off. Stop when the "large element index" and the "small element index" meet. That's where the pivot goes.

Example

3 Complexity

"Complexity" is a measure of how (space/time — usually time) efficient an algorithm is. It has nothing to do with how difficult it is to understand — though simple to understand algorithms are often more complex in the efficiency sense, and more efficient (less complex) algorithms are often more complex to understand. We will be looking at complexity in more detail later in the term. Which sort/search algorithm to use?

3.1 Sort

- Bubble sort, selection sort and insertion sort are all, on average, slower than merge sort and quick sort for large enough sets of data. However they can all be written such that they are more efficient if the data is already (nearly) sorted — i.e. they can be efficient ways of checking if a list is already sorted.
- Merge sort and quicksort are both, on average, more efficient than the other algorithms for large enough sets of data. However merge sort requires copying the list — which can be space expensive.
- Quicksort is as bad as bubble sort, selection sort and insertion sort in the worst case — when the list is (nearly) sorted. Can try to avoid this by using a pivot picked from a random place in the list, rather than from the start of the list.

3.2 Search

Binary search is clearly more efficient than sequential search, but for large enough sets of data it will take longer to sort a random list even with the most efficient sorting algorithm than it will to do a sequential search of that list.

3.3 Conclusions

- If you know the list is sorted, use binary search
- If you know the list is unsorted
  - if you are only going to do one search, use sequential search
  - if you are going to do many searches, sort the list using, e.g., quicksort then use binary search
- If you think the list might be sorted, check with bubble, selection or insertion sort, then sort if necessary, then use binary search

E.g. for list size 100, with 10 searches:

	List sorted	List unsorted
sequential search;	1,000	1,000
quicksort; binary search;	10,000	731

if (!sorted) {
  quicksort;
}
binary search;
	166	831

The following table gives an indication of how these values can affect the efficiency of your code. Each entry shows the time taken (in arbitrary units) to perform a number of searches on potentially unsorted data lists of given sizes. For each combination of array size and number of searches a pair of values is given:

- the top values are for a simple sequential search,
- the middle values are for an algorithm that first sorts the list using quicksort, and then uses a binary search for the searches,
- and the bottom values are for an algorithm which first checks if the data is already sorted, then sorts it, using quicksort, *if necessary*, and thereafter uses binary search for the searches.

In each case the left hand entry shows the time taken if the list was originally (almost) sorted, the right hand entry the time taken if the list was not sorted.

The abbreviation "th" stands for thousands of time units, "m" stands for millions of time units, "b" for billions (with 1b = 10^9), and "t" for trillions (10^{12}). All of these values are approximate, and given to two significant figures.

	No. of searches									
	1		10		100		1th		1m	
10	10	10	100	100	1th	1th	10th	10th	10m	10m
	103	37	133	66	432	365	3.4th	3.4th	3.3m	3.3m
	13	47	43	76	342	375	3.3th	3.4th	3.3m	3.3m
100	100	100	1.0th	1.0th	10th	10th	100th	100th	100m	100m
	10th	671	10th	731	11th	1.3th	17th	7.3th	6.7m	6.6m
	107	771	166	831	764	1.4th	6.7th	7.4th	6.6m	6.6m
1th	1.0th	1.0th	10th	10th	100th	100th	1.0m	1.0m	1.0b	1.0b
	1.0m	10th	1.0m	10th	1.0m	11th	1.0m	20th	11m	10m
	1.0th	11th	1.1th	11th	2.0th	12th	11th	21th	10m	10m
1m	1.0m	1.0m	10m	10m	100m	100m	1.0b	1.0b	1.0t	1.0t
	1.0t	20m	1.0t	20m	1.0t	20m	1.0t	20m	1.0t	40m
	1.0m	21m	1.0m	21m	1m	21m	1.0m	21m	21m	41m

If the list is to be set up once, and then searched frequently, it is generally better to sort once and then use a binary search algorithm. It is advisable to check first to see if the list is not already sorted.

If the list is to be searched only a few times sorting will be too expensive and we should use sequential search.