Basic searching and sorting

Table of contents

Introduction

We sort because we want to search faster, and that's all there is to it. If you didn't need to find something quickly, why would you bother storing it in any order in the first place! We'll start by looking at the very basic search techniques, and look at the basic sorting techniques after that.

Searching

We'll consider two different types of search in the basic category: sequential search through a sequence, and binary search through a sorted random-access sequence.

Sequential search

The easiest way to search for an item in a sequence is to start at the beginning, look at each item to see if there's a match. We stop if we find the item, or if we go past the end of the sequence, which means that the item does not exist in the sequence. This is the so-called **sequential search**, which requires **`n` comparisons in the worst-case for an n-element sequence**, which is rather slow for real use. However, it does have advantages: (a) it does not require that the container support random access (so a list would do), and (b) it does not require that the data be sorted.

The following shows how to sequentially search for a `key` in an array `data`, where the first `size` elements are used (where `size ≤ data.length`). If all the slots in the array are used, then you can simply substitute `data.length` for `size` (because `size == data.length` in that case).

```
/**
 * Searches for the given key in the array of size elements.
 *
 * @param data the array with the keys
 * @param size the number of keys in the array (≤ data.length)
 * @param key  the key to search for in the array
 * @return the position of the key if found, or -1 otherwise.
 */
```

```
public static int seqSearch(Object[] data, int size, Object key) {
    for (int i = 0; i < size; i++)
        if (key.equals(data[i]))
            return i;           // return index of item if found

    return -1;                  // return sentinel if not found
}
```

Similarly, the following shows how to sequentially search for a `key` in a linked list, where `list` is a reference to the first or head node.

```
/**
 * Searches for the given key in the list.
 *
 * @param head reference to the head node in the list
 * @param key  the key to search for in the array
 * @return the position of the key if found, or -1 otherwise.
 */
public static int seqSearch(Node head, Object key) {
    Node n = head;
    for (int i = 0; n != null; i++, n = n.next)
        if (key.equals(n.element))
            return i;           // return index of item if found

    return -1;                  // return sentinel if not found
}
```

Back to Table of contents

Binary search

If the data in the sequence is already **sorted**, and the sequence supports **random access** (if it's an array that is), then we can significantly improve the search peformance by using **binary search**. We're already quite familiar with this search technique — we use something similar when we look up a word in a dictionary, or a person in a phonebook. We see if the item is in the middle of the array. If so, we've found it. If not, we check if the item is larger or smaller. If it's larger, then we look at the right half of the array (to the right of the middle element), or else we look at the left half (to the left of the middle element). We continue dividing up the array, until either we find the item, or we run out of data to search (ie., the item wasn't there in the first place).

The following shows how to search for a `key` in an array `data` of sorted elements, where the first `size` elements are used (where `size ≤ data.length`). If all the slots in the array are used, then you can simply substitute `data.length` for `size` (because `size == data.length` in that case).

```
/**
 * Searches for the given key in the array of size elements.
 * Pre-condition: data must be sorted in non-decreasing order.
 *
 * @param data the array with the keys
 * @param size the number of keys in the array (≤ data.length)
 * @param key  the key to search for in the array
 * @return the position of the key if found, or -1 otherwise.
 */
```

```
public static int binSearch(Object[] data, int size, Object key) {
    int l = 0;
    int r = size - 1;
    while (l <= r) {
        int mid = (l + r)/2;
        if (key.equals(data[mid]))
            return mid;             // return index of item if found
        else if (((Comparable) key).compareTo(data[mid]) > 0)
            l = mid + 1;
        else
            r = mid - 1;
    }
    return -1;                      // return sentinel if not found
}
```

The question now is how fast (or slow) is binary search? One way to answer this question is find out the maximum number of comparisons one must make to find an item in an sorted array of `n` elements. The maximum number of comparisons happen in the "worst-case" scenario, in which we either find the item when the array size is just 1, or if it's not there at all. The following table shows the number of comparisons at each step.

Step	Array size	# of comparisons
	$n = n/2^0$	1
1	$n/2 = n/2^1$	1
2	$n/4 = n/2^2$	1
3	$n/8 = n/2^3$	1
.	.	1
.	.	1
.	.	1
k	$= n/2^k$	1
.	.	1
.	.	1
.	.	1
?	1	1

We start with an array size of `n`, and make a single comparison (with the item in the middle of the array). Then we continue with either the left half or the the right half (actually it's slightly less than `n/2`, but we can ignore such details when `n` is very large), and at each step make a single comparison. In the worst-case, we eventually end up with an array of just one item (`n == 1`), which is either the item ("found"), or not ("not found"). To find the maximum number of comparisons, we simply need to add up the 3rd column, but for that we need to know tall the column is, that is the number of steps! So, if we can find out the number of steps it takes for `n` to go down to one when we divide by 2 at each step, we can find out the maximum number of comparisons.

After `k` steps, we have an array size of $n/2^k$. So, if $n/2^k == 1$, `k` is number of steps we're looking for.

$$
\begin{aligned}
n/2^k &= 1 \\
2^k &= n \\
k &= \log_2(n)
\end{aligned}
$$

So, we have $k = \log_2(n)$ steps, and at each step, we make 1 comparison. So the total number of comparisons is then $k * 1 = \log_2(n) * 1 = \log_2(n)$.

In summary, in the worst-case, the number of comparisons made by **sequential search is n**, and by **binary search is $\log_2(n)$**.

It's summarized in the table below.

```
Worst-case performance:

    search algorithm | max # of comparisons
    -------------------------------------
       sequential    |    n
       binary        |    log2(n)
    -------------------------------------
```

Back to Table of contents

Sorting

We will look at 3 very basic sorting algorithms - bubble, selection and insertion sorts. Bubble is simply too silly to bother about, so let's take a look at selection first.

Back to Table of contents

Bubble sort

We're skipping bubble ... but you can read up on it on Wikipedia.

Back to Table of contents

Selection sort

The idea is very simple — we select the minimum element in the sequence, and move it to the first position, then select the second minimum element and move it to the second position, and so on until we've placed all the elements in its right place. Selecting the minimum is trivial — just iterate through the elements picking out the minimum. Selecting the second minimum is also trivial — **after** placing the minimum in the first position, we select the minimum of the rest of the sequence (that is, from the second position onwards). After placing the second minimum in the second position, we find the third minimum by find the minimum in the rest of the sequence, and so on.

At any given time, the array is partitioned into two areas — the left part is sorted, and the right part is being processed. And at each step, we move the partition one step to the right, until the entire array has been processed.

The following shows the sequence of steps in sorting the sequence `< 17 3 9 21 2 7 5 >`. In the example below, the "|" symbol shows where the partition is at each step.

```
    Input: 17 3 9 21 2 7 5
    n = 7

    Step 1: | 17 3 9 21 2 7 5   << minimum is 2, exchange with 17
    Step 2: 2 | 3 9 21 17 7 5   << minimum is 3, no exchange needed
    Step 3: 2 3 | 9 21 17 7 5   << minimum is 5, exchange with 9
    Step 4: 2 3 5 | 21 17 7 9   << minimum is 7, exchange with 21
    Step 5: 2 3 5 7 | 17 21 9   << minimum is 9, exchange with 17
    Step 6: 2 3 5 7 9 | 21 17   << minimum is 17, exchange with 21
          : 2 3 5 7 9 17 | 21   << STOP
```

At each step, we only consider the array to the right of the partition, since the part to the left is already sorted, and does not need to be dealt with. Also, note that we iterate `n-1` times, because in the last step, there is just a single item left (21 in this case), which is obviously the minimum.

```
/**
 * Sort the specified array using selection sort.
 *
 * @param data the array containing the keys to sort
 */
public static void selectionSort(Object[] data) {
    for (int i = 0; i < data.length - 1; i++) {
        // Find the index of the minimum item, starting at `i'.
        int minIndex = i;
        for (int j = i + 1; j < data.length; j++) {
            if (((Comparable) data[j]).compareTo(data[minIndex]) < 0)
                minIndex = j;
        // Exchange with the first item (at `i'), but only if different
        if (i != minIndex) {
            Object tmp = data[i];
            data[i] = data[minIndex];
            data[minIndex] = tmp;
        }
    }
}
```

Few things to note:

1. If only the first `size` (where `size ≤ data.length`) elements are used in the `data` array, then the iteration should stop at `size` instead of `data.length`.
2. We only need to advance through the sequence, and do not need random access. This means that we can perform selection sort on a list just as easily.

The following algorithm shows how to sort a singly-linked list using selection sort.

```
/**
 * Sort the specified list using selection sort.
 *
 * @param head reference to the first node of the list to sort
 */
public static void selectionSort(Node head) {
    // Empty list or list with a single element is already sorted.
    if (head == null || head.next == null)
        return;
```

```
    for (Node n = head; n.next != null; n = n.next) {
        // Find the node with the minimum item, starting at `n'.
        Node minNode = n;
        for (Node p = n.next; p != null; p = p.next) {
            if (((Comparable) p.element).compareTo(minNode.element) < 0)
                minNode = p;
        // Exchange with the first item (within `n'), but only if different
        if (n != minNode) {
            Object tmp = n.element;
            n.element = minNode.element;
            minNode.element = tmp;
        }
    }
}
```

How many comparisons do we make in sorting a sequence of `n` items using this algorithm? At each step, we have to find the minimum item to the right the partition.

```
  Step   |  Array size |  Max # of comparisons
  ----------------------------------------
    1    |   n         |        n - 1
    2    |   n - 1     |        n - 2
    3    |   n - 2     |        n - 3
    .    |     .       |          .
    .    |     .       |          .
    .    |     .       |          .
 n - 1   |     2       |          1
  ----------------------------------------
```

The total number of comparisons in the worst case is simply the sum of the 3rd column, which is an arithmetic series: $1 + 2 + 3 + \ldots + (n - 1) = n(n-1)/2 \sim n^2$.

Does it help if the data is already sorted? No. Regardless of the input, selection sort will always make $\sim n^2$ comparisons, so the best- and worst- cases are the same. Selection is **not adaptive**, which is one of its weaknesses. See Discussion for more on this issue.

Back to Table of contents

Insertion sort

Insertion sort also works by partitioning the array into sorted and being processed parts, but works very differently than selection sort. At each step, it takes the next "key" in the array, and moves it left until it is in its rightful place, and moves the partition one step to the right until the whole array is sorted. Since an one-element array is already sorted, we start by moving the second element (at index 1).

The following shows the sequence of steps in sorting the sequence ‹ 17 3 9 21 2 7 5 ›. In the example below, the "|" symbol shows where the partition is at each step.

```
    Input: 17 3 9 21 2 7 5
    n = 7

    Step 1: 17 | 3 9 21 2 7 5   << move 3 to the left
    Step 2: 3 17 | 9 21 2 7 5   << move 9 to the left
```

```
    Step 3: 3 9 17 | 21 2 7 5   << move 21 to the left
    Step 4: 3 9 17 21 | 2 7 5   << move 2 to the left
    Step 5: 2 3 9 17 21 | 7 5   << move 7 to the left
    Step 6: 2 3 7 9 17 21 | 5   << move 5 to the left
          : 2 3 5 7 9 17 21 5 | << STOP
```

At each step, we essential insert the key to the right of the partition into the sorted partition by scanning left. In the best case, it stays in its place (see when we moved 21 to the left); in the worst-case, it moves all the way to the first position (see when we moved 2 to the left). Also, note that we iterate `n-1` times.

```
/**
 * Sort the specified array using insertion sort.
 *
 * @param data the array containing the keys to sort
 */
public static void insertionSort(Object[] data) {
    // i denotes where the partition is
    for (int i = 1; i < data.length; i++) {
        // the key is to the right of the partition
        Object key = data[i];
        int j = i - 1;      // use j to scan left to insert key
        while (j >= 0 && ((Comparable) key).compareTo(data[j]) < 0) {
            // shift item right to make room
            data[j + 1] = data[j];
            j--;
        }
        // Found the position where key can be inserted
        data[j + 1] = key;
    }
}
```

Few things to note:

1. If only the first `size` (where `size ≤ data.length`) elements are used in the `data` array, then the iteration should stop at `size` instead of `data.length`.
2. The outer loop advances through the sequence, so we don't need random access. The inner loop also advances, but in the reverse direction. While we don't need random access, we do need to move in both directions, which means we at least need a doubly-linked list.
3. Insertion sort does not require that all the keys are already in place, like selection sort does (since it needs to scan to find the minimum element). In fact, insertion sort is very good at sorting data that arrives one element at a time, which is a typical scenario on *network algorithms*. This is a significant advantage of insertion sort over other comparable sorting algorithms of similar performance.

How many comparisons do we make in sorting a sequence of `n` items using this algorithm? At each step, we have to find the rightful place for the key by comparing it with item to the left of the partition. In the worst case, each key moves all the to the first position (which would happen if the input sorted, but in the reverse order), shifting all the sorted elements by one position to the right.

```
Step   | Sorted part size |  Max # of comparisons | Max # of shifts
-------------------------------------------------------------------
  1    |   1              |        1              |       1
  2    |   2              |        2              |       2
```

```
  3    |   3             |        3             |       3
  .    |   .             |        .             |       .
  .    |   .             |        .             |       .
  .    |   .             |        .             |       .
n - 1  |  n-1            |      n - 1           |     n - 1
-------------------------------------------------------------------
```

The total number of comparisons is simply the sum of the 3rd column, which is an arithmetic series: `1 + 2 + 3 + ... + (n - 1) = n(n-1)/2 ~ n^2`. The total number of shifts is the same: `n(n - 1)/2 ~ n^2`.

Does it help if the data is already sorted? YES! We make a single comparison at each step, since the key does not move at all. The total number of comparisons then is just `n-1`, instead of `~n^2`. This is the **best-case** performance for insertion sort. What if the input is **nearly sorted**? It also significantly improves the performance of an insertion sort over the worst-case, which makes it an **adaptive** sort. See Discussion for more on this issue.

In the **average case** however, if the input is randomly distributed, each new key is likely to travel only about halfway, which leads to slightly better performance. It's still `~n^2` however, so it's the same order as the worst-case.

```
Worst-case performance:

    sorting algorithm | max # of comparisons
    --------------------------------------
       selection      |    n(n-1)/2 ~ n^2
       insertion      |    n(n-1)/2 ~ n^2
    --------------------------------------

    sorting algorithm | max # of exchanges/shifts
    -------------------------------------------
       selection      |    n(n-1)/2 ~ n^2
       insertion      |    n(n-1)/2 ~ n^2
    -------------------------------------------

Best-case performance:

    sorting algorithm | max # of comparisons
    --------------------------------------
       selection      |    n(n-1)/2 ~ n^2    << doesn't change
       insertion      |    n - 1             << when input is sorted
    --------------------------------------

    sorting algorithm | max # of exchanges/shifts
    -------------------------------------------
       selection      |        0
       insertion      |        0
    -------------------------------------------
```

Back to Table of contents

Discussion

Given that both selection and insertion sorts take $\sim n^2$ comparisons in the worst-case, which one would you use? Let's ignore the best-case scenario for now, since that's really not a useful metric in real-life, other than its use in (often false) advertising. Let's look at the relative strengths of these sorting algorithms.

- To use using selection sort, you must have **all** the data already available in a sequence, since we have to find the minimum value to put in the first slot, the 2^{nd} to put in the second slot, and so on. As the partition moves to the right, the left (sorted) part is never touched again. For insertion sort, each new key is inserted into the left part in its rightful place, so while the left part is always sorted, it gains new keys as sorting progresses. The significant advantage of this approach is that insertion sort can deal with single keys arriving one at a time, and maintains a sorted sequence at any time by inserting the new key in its rightful place. The analogy we used in class was the difference between download a mp4 to view (have **all** the data available before you are able to play the video), or to stream a video from the network and view it real-time (see each frame **as it comes**, provided you have the bandwidth of course). This is one reason for choosing insertion sort.
- You should have noticed that selection sort performance does not improve even if the input data is already sorted, but insertion sort does take advantage of that. This makes insertion sort **adaptive**, which is a desirable property in any algorithm. For a **nearly sorted** input, insertion sort performs significantly better than selection sort.
- In the algorithm outlined above for selection sort, we exchange the minimum with the current element, which may move the current element far away from its current position. This make **selection sort**, as it's done in these notes, **unstable**. One can make selection sort stable by inserting the minimum element in the first position (**before** the current element that is) instead of exchanging it with the current element. This is fairly expensive for arrays, but can be made cheap for lists. **Insertion sort** is **stable** if we use < relation to compare the new key with the elements in the left part.
- Neither sorts require **random access**, so can be done on a list. However, insertion sort does require **bi-directional** access, which means that the list must be **doubly-linked**.

Except for the case where the sequence is a singly-linked list, insertion sort tends to perform (often much) better than selection sort. Its adaptive nature also helps its performance when the input is already partially or nearly sorted.

Back to Table of contents