Searching in a sorted array in Java

Algorithms and data structures ID1021

Johan Montelius

Fall 2024

Introduction

Searching for a key in an unsorted array is as you probably now know quite expensive. If the elements in the array are not sorted, the only way to find an element is to go through the whole data structure. As you will learn things will become much easier if the array is sorted.

A first try

Let's start by setting up a benchmark where we search through an unsorted array. The search procedure is of course a simple loop through all the elements in the array, let's call this `unsorted_search()`.

```java
public static boolean unsorted_search(int[] array, int key) {
  for (int index = 0; index < array.length ; index++) {
    if (array[index] == key) {
      return true;
    }
  }
  return false;
}
```

Set up the rest of a benchmark and do some measurements for a growing number of elements in the array. Describe the relationship between the size of the array and the time it takes to do the search. How long time does it take to search through an array of a million elements?

Now, if we know that the array is sorted we can of course do a quick optimization - we can stop the search once the next element in the array is larger then the key that we are looking for. Take a wild guess, how much better is this compared to our unsorted solution?

Set up a benchmark where you experiment with different data sizes. This is one way of generating a sorted array (no duplicates):

```
private static int[] sorted(int n) {
  Random rnd = new Random();
  int[] array = new int[n];
  int nxt = 0;
  for (int i = 0; i < n ; i++) {
    nxt += rnd.nextInt(10) + 1;
    array[i] = nxt;
  }
  return array;
}
```

How long time does it take to search through an array of a million entries? It might not sound very much but if our program constantly does search operations it will add up. There are however smarter things we can do and this will allow us to handle much larger data sets in reasonable time.

Binary search

The trick that we will do is what you would do if you searched for a word in a dictionary, a name in a phone book or a chapter in a book. You would not start on page one and look at one page at a time, you would jump to a page where you think it is likely to find the item and then jump a bit forward or backward depending on what you find on that page. After some jumps you have found the item you're looking for.

If we want to describe a general algorithm we could say that one should jump to the middle of the book and then jump either one quarter forward or backwards. In each round we will examine the page we jump to and make smaller and smaller jumps.

When we implement this algorithm we need to keep track off the *first possible page* and the *last possible page*. If we know this we can find the index in the middle and examine that page. If we find what we're looking for all is well but if not, we determine if we should update the *first* or *last* possible page. It could of course be the case that we have have no pages to jump to and in this case we know that the item that we are looking for is not in the book.

This algorithm is called *binary search* and is very efficient. There are of course many ways to encode it but here is some skeleton code that will get you started:

```
public static boolean binary_search(int[] array, int key) {
    int first = 0;
    int last = array.length-1;

    while (true) {
      // jump to the middle
      int index = ....... ;

      if (array[index] == key) {
        // hmm what now?
      }
      if (array[index] < key && index < last) {
        //  what is the first possible page?
        first = ...... ;
        continue;
      }
      if (array[index] > key && index > first) {
        // what is the last possible page?
        last = ......  ;
        continue;
      }
      // Why do we land here? What should we do?

    }
  }
```

Re-run your benchmarks but now using the binary search. Report the execution time and describe a function that given the size of the array roughly describes the execution time. How long time does it take to search through an array of one million items? Without running an experiment - how long time do you estimate that it would take to search through an array of 64M items? Give it a try - how well did you estimate the execution time?

the cost of sorting

Working with sorted data is much more efficient than working with unsorted data. The problem is of course that not all data is sorted so the question is how hard it is to sort data. In the last experiment the question is if it is worth first sorting the two arrays before doing the search for duplicates. How expensive is it to sort an array and what algorithms do we have - this will be the question of the next assignment.

Recursive programming

When we're at it, we might as well introduce an alternative way of implementing the binary search procedure. We will define a procedure `recursive()` that will do a binary search in an array from a minimum index to a maximum index. The maximum index has to be at least as large as the minimum index. We could for example search for a key between index 10 and 30 but instead of just modifying our existing implementation we will do a trick; we will call the method recursively.

```
private static boolean recursive(int[] arr, int key, int min, int max) {

  int mid = min + ((max - min)/2);

  if (arr[mid] == key) {
    // simple
  }

  if ((arr[mid] > key) && (min < mid)) {
    // call recursive but narrow the search
  }

  if ((arr[mid] < key) && (mid < max)) {
    // call recursive but narrow the search
  }
  // as before
}
```

This way of implementing a procedure is called recursive programming. It might be that this is the first recursive procedure that you have defined if you have only been programming in for example Java. Other languages have recursive programming as their default way of implementing procedures so it could be nice to learn.

In this example there is no obvious benefit of using a recursive strategy but sometimes the implementation becomes easier. In the next assignment you will see that it makes more sense; if you only get use to thinking recursively.

The downside of using a recursive strategy is that procedure calls are expensive and that you're using the programming stack extensively. You will get away with doing a hundred or even a thousand recursive calls after each other but if you try a hundred thousand you're pushing the limit.

How many recursive calls are done when searching an array or length 1000? How many are done when seraching: 2000, 4000 ... a million? Is the number of recursive calls on the stack a problem?