Search Efficiency in Sorted Arrays in Java

Alexander Alvarez

Fall 2024

Introduction

The time efficiency of different search operations will be measured in a sorted array with varying sizes. Two variants of the search operation will be tested and compared to the search time on an unsorted array.

The general behaviour of these operations are to be described as well as how they were created and implemented.

Method

We are already familiar with the search algorithm from a previous assignment. This report will expand on the search operation but include the same test methods that was used in the previous assignment.

The values that the operations will search for (keys) will be randomly generated and will not be sorted in any tests. The search methods will be looped where the time will be taken in intervals right before and after the loop. The difference of the time intervals will be the time to execute the search operation a `loop` amount of times. The time difference divided by `loop` will give the median execution time for one search operation. Only the minimal time values will be used for comparisons between operations due to consistency.

The search methods will be run for a sufficient time before the actual benchmark to enable the just-in-time compiler for time optimizations.

Unsorted search

The unsorted search time will be the time that I will compare the sorted search operations to.

The unsorted search method creates two arrays (`array` and `keys`) with randomly generated elements in a random order. The pool of values that the random number generator (RNG) takes from is double the size of the `array`, i.e. if the `array` size is 1000 then the pool size is 2000. The difference between the two arrays is that the `array` array has the varying size `n` while the `keys` array will have the size `loop`.

The created arrays are then used in the `for` loop between the time intervals. The element that is to be searched for (`key`) is taken from the `keys` array for each loop iteration. The unsorted search operation is then used with the `array` and the `key` as arguments.

```
long t0 = System.nanoTime();
for(int i = 0; i < loop; i++){
    int key = keys[i];
    unsorted_srch(array, key);
}
long t1 = System.nanoTime();
return (t1 - t0);
```

The unsorted search operation is a simple `for` loop that compares an element at a certain `array` index to the `key`. The `for` loop stops when the same value is found, but will search through the whole `array` if no same value was found.

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

The `loop` variable is set to $1,000$ and the `array` sizes start from $1,000$ and increases by doubling the size to $1,024,000$ (`n` = {1000, 2000, ..., 512000, 1024000}). Running the code with these values yields the following graph:

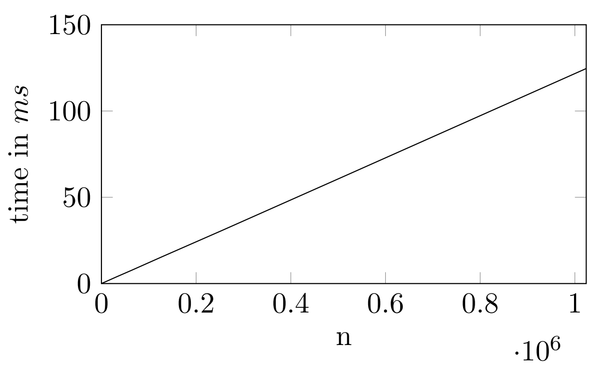

Figure 1: Unsorted search time

The search time for an unsorted array with a million elements takes about $125\mu s$.

Sorted search

The sorted search method is almost identical to the unsorted search method with the differences being:

- an added method to create a sorted array
- a change in the condition of the search operation
- an adjustment to the RNG pool

The elements of `array` were randomly generated in a random order in the unsorted search method. This time a method is created that still randomly generates the elements but every next element has a greater value than the previous.

The sort array method has a `for` loop that generates a random value from 0 to 10 and stores it in the `nxt` variable in each iteration. The `nxt` variable increases with the sum of the random value plus the value of the `nxt` variable from the previous iteration. This makes it so that the `nxt` variable has a greater value in every next iteration while maintaining the RNG. However, duplicates may still occur if the RNG generates the number 0. To solve this we add 1 to the sum which makes the `nxt` variable atleast 1 value greater in every next iteration.

```
public static int[] sort_array(int n, Random rnd){
  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;
}
```

The sorted search operation is almost the same as the `unsorted_srch` operation from earlier. The sorted search operation is however optimized by one change in the condition of the `if` statement. The `if` statement will stop the search loop if the value of an element is the same or greater than the `key`. This is because the `key` wont be found when every next element is greater than the `key` now that the `array` is sorted.

The RNG pool is the same for both arrays in the unsorted search method which is double the size of `n`. The sort array method complicates this in the sorted search method because it has an RNG pool of 10 times the size of `n` - i.e. the maximum value is $10 \times n$ if the RNG always generates the greatest value which is highly unlikely. For consistency I chose the RNG pool for the `keys` to be 10 times the size of `n`.

Running the code for the sorted search method with the same values as the unsorted search benchmark (`loop = 1000` and `n = {1000, 2000, ..., 512000, 1024000}`) yields the following graph:

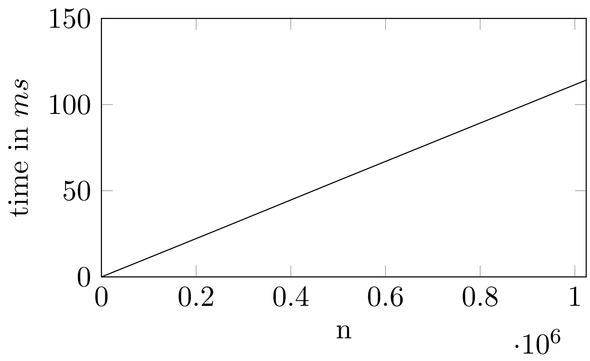

Figure 2: Sorted search time

The search time for a sorted array with a million elements takes about $114\mu s$.

Binary search

The binary search method is very similar to the sorted search method with the only difference being the usage of a whole different search algorithm.

The search algorithm in question is the binary search algorithm which checks the middle element and systematically divides the `array` in halves until the `key` is found. This is done by having a two variables (`first` and `last`) respectively pointing at the first and last index of the array. A `while(true)` loop is then made with four `if` statements and a `middle` variable that points to the element in the middle between the `first` and `last` index pointers.

The search loop stops if the value of the `middle` element is the same as the `key`. If the `middle` element is less than both the `key` and `last` pointer - i.e. if the `key` may be further to the `last` pointer - then the `first` pointer will point to where `middle` is pointing and `middle` will point to a new middle between `first` and `last`. Similarly, if `middle` is greater than `key` and `first` - i.e. if the `key` may be further to the `first` pointer - then `last` will point to the `middle` element. When the `middle` element is not the `key`, the `first` and `last` pointers will actually point to one index above and under `middle` respectively to not include that `middle` element that didn't have the `key`. The search loop is stopped if no `key` is found under the conditions of the `if` statements.

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

```
        while(true){
            int middle = (first + last)/2;
            if(array[middle] == key) return true;
            if(array[middle] < key && middle < last){
                first = middle + 1;
                continue;
            }
            else if(array[middle] > key && middle > first){
                last = middle - 1;
                continue;
            }
            else return false;
        }
    }
```

Running the code for the binary search method with the same values from the two previous benchmarks yields the following graph:

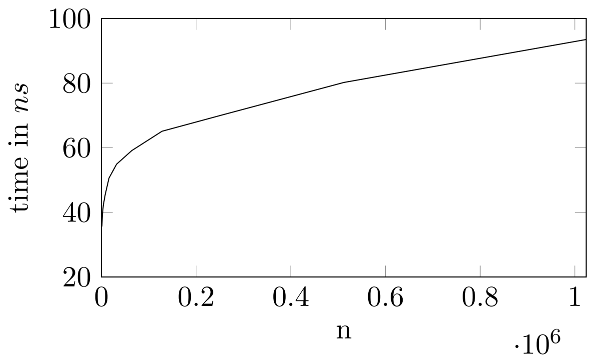

Figure 3: Binary search time

The binary search time for a sorted array with a million elements takes about $93ns$. The binary search algorithm divides the `array` in half every time it checks for the `key`. This means that n is divided by 2^k where k is the number of checks until no more subarrays can be created. A function that roughly describes the execution time on an array of size n:

$$O(log(n))$$

Looking at the graph, I estimated the time to search through an array with 64 million elements to take no more than $300ns$. My optimistic guess was undeniably crushed by the output time I got from the test which was more than double my guessed time at roughly $650ns$.

Conclusion

The time of the sorted search method compared to that of the unsorted search method shows a slight increase in performance. Decreasing the RNG pool for the `keys` array to the median max value of the last element in the `array` array improves the performance even more. However, those results may be skewed in a way that is inaccurate which is why they were not included to be compared.

The slight increase in performance of the sorted search method may likely prove useful in cases where an array is already sorted. The same can not be said if the array is unsorted for which the increased performance does not outweigh the cost of sorting the array.

The binary search method shows a significant increase in performance compared to both the sorted and unsorted search methods. The performance of the binary search method increases exponentially as the array size grows larger compared to the other two methods that has has a linear performance. This makes the binary searh method a highly viable option for searching through arrays with huge sizes. The significantly increased performance overall likely outweighs the cost of sorting an unsorted array.

There is an overall increase in searching through data that is already sorted.