Time performance of arrays in Java

Alexander Alvarez

Fall 2024

Introduction

The time performance of three different operations over an array of varying size are to be measured. The efficiency of these operations are to be measured in time and presented comprehensively.

The assignment document, of which this report is based upon, contains all the information used in this report.

The three different operations that are to be benchmarked are:

- Random access
- Search
- Duplicates

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

Random access

The time of a read operation depends on several factors; One of these factors includes the cache. A way for the cache to not have such an impact on the results is to access the array at random locations, i.e. `Random access`.

Another factor is a compiler in Java called "just-in-time" or JIT for short. The runtime system will make a more efficient compilation of a method if it detects that method being extensively used during execution. Although there is no expilcit way to controll when and if this phase kicks in, running the method that is to be benchmarked for a sufficient time before the actual benchmark will make it more likely to kick in.

As for measuring time, the minimum time value will be used due to it being the most predictable compared to both the median and maximum time value respectively.

the clock accuracy

The first task is to figure out the accuracy of the clock on the system. The system clock will be used to take the time at certain intervals in the program. The difference between these time intervals will be the time measurement for the operations. In Java the method `System.nanoTime()` measures the elapsed time and will be used to take the time at these intervals.

A code is provided to be tested to measure the accuracy of the clock. The code is a `for` loop containing two variables, with no other code between them, which takes the time using the system clock and then prints out the difference in nanoseconds. This will show how precise the clock is by measuring the minimal time between the two intervals.

Running the code most often gives a time difference of $100ns$, varying up and down with $100ns$. The output is therefore $0ns$, $100ns$ and $200ns$, showing that the clock has an accuracy of $100ns$, or in other words, the clock has a precision of $100ns$.

the method

The only thing we want to measure is the time to access an array at a random index. Any other code in between the time intervals may therefore impact the results.

The method will have two parameters: one for the size of the array (`n`) and the other for the amount of random indices (`loop`). The array to be accessed (`array`) and the array with random indices (`indx`) is created. The `array` array may have arbitrary elements, however we want the `indx` array to have random elements in the span of the size `n`.

A `for` loop is created with the time intervals being right before and after this `for` loop. The `for` loop will loop as many times as the size of the `indx` array, that is the amount or random indices we chose. Within the `for` loop we let a variable (`sum`) be incremented by the values of the `array` array at random indices using the `indx` array. The code is thus:

```java
public static long rndAccess(int n, int loop){
        int[] array = new int[n];
        for(int i = 0; i < n; i++) array[i] = i;

        Random rnd = new Random();
        int[] indx = new int[loop];
        for(int i = 0; i < loop; i++) indx[i] = rnd.nextInt(n);

        int sum = 0;
        long t0 = System.nanoTime();
        for(int i = 0; i < loop; i++) sum += array[indx[i]];
        long t1 = System.nanoTime();
        return (t1 - t0);
    }
```

In the `main` method i chose 10 sizes to be the first 10 powers of 2, starting from the exponent 0, multiplied by 100. Before running the benchmark we add our method with relatively high arguments as to warmup the JIT. From here we can choose the `loop` parameter and and how many times we want to run the benchmark to use the minimum time out of all runs.

Since the only thing we want to measure is the time to access an array at a random index once we need to divide the time by the amount of random accesses which is our variable `loop`.

results

Running the program with the `loop` variable being 1000 and taking the minimum time value out of 10 runs for every array size we get a table with the average minimum value:

size	runtime	ratio
100	1.4	1.0
200	1.4	1.0
400	1.4	1.0
800	1.4	1.0
1600	1.4	1.0
3200	1.4	1.0
6400	1.4	1.0
12800	1.4	1.0
25400	1.4	1.0
51200	1.4	1.0

Table 1: Random access time of different array sizes, runtime in nanoseconds.

conclusion

Running the code with higher values of `loop` or using more than 10 runs for the minimum time of every array size gives the same output as the time to increment a variable by 1. At the bigger array sizes the time will increase by only 100 or 200 nanoseconds depending on the size. This tells me that running the code too many times makes will make some optimization program, possibly cache/memory, kick in and make random accessing pointless. The output time would just be the time for the program to increment the sum of the whole array which scales with the array size and is not what we want to measure.

The values used for the output in the result section seem to have the desired balance to not trigger the optimization but be large enough to make use of as much data as possible. We then see that the time ratio of random access is the same regardless of array size. The performance of the operation is the same for different array sizes.

Search

The method for the search operation is similar to the method for the random access access operation. The difference being that instead of accessing an array at random indices, the program instead takes a value and compares it to every element in the target array until it finds a matching value. The program will access the array, beginning at index 0, and sequentially go through the array unlike random access which accesses the array at random indices.

In this case our pool of elements to search for is greater than the size of the target array. The elements are of random values to represent an unsorted list which, together with the greater size of the pool, enables the chance of elements not being found.

method

The search method is similar to the method for random access as mentioned previously. In this case the target array (`array`) and the array containing the elements we want to search for (`keys`) is created almost equally: as in both arrays is created to have random elements from the span of double the `array` size (`n*2`) with the exception being that the `keys` array is the length of `loop`.

The important part is the code between the time intervals. What we are measuring is the time to take an element (`key`) from the `keys` array and sequentially going through and comparing it to the elements of the `array` array to find an element of equal value. If the two elements are equal the program will break out of the comparison loop and move on to the next `key` to search for. If no element could be found that matches `key` the program will exit the comparison loop after going throgh the whole `array` array and move on to the next `key` to search for. The code is thus:

```
long t0 = System.nanoTime();
  for(int i = 0; i < loop; i++){
      int key = keys[i];
      for(int j = 0; j < n; j++){
          if(key == array[j]){
              sum++;
              break;
          }
      }
  }
long t1 = System.nanoTime();
```

The setup is the same in the `main` method as for how it was in random access.

results

The output of the code shows a time increase proportional to the size of the array beginning at size 400. As the size of the array doubles in size, so does the time as well. Running the program with the `loop` variable being 1000 and taking the minimum time value out of 10 runs for every array size we get a table with the average minimum value:

size	**runtime**	**ratio**
100	47	1.1
200	42	1.0
400	63	1.5
800	130	3.1
1600	220	5.2
3200	420	10
6400	800	19
12800	1500	36
25400	3100	74
51200	6200	150

Table 2: Search time of different array sizes, runtime in nanoseconds.

conclusion

The execution time for the search operation grows proportional to the size of the array as seen in the results section. Increasing the value of `loop` or the amount of runs makes the program take too long to execute.

The reason for the values used for these results is due to consistency with the other test. Changing `loop` or the amount of runs may complicate the result and may not be eligible for comparing efficiency.

As for finding a simple polynomial that roughly describes the execution time of the search operation, i found it a bit tricky. We have two sizes with independent values compared to the size `n` since the time does not begin to proportionally increase until the third size. We can describe a function as follows: Let the time of the first array be $C1 = 47$ and the time of the second array be $C2 = 42$. A value that roughly matches the proportional grow is 50 so we let $C3 = 50$. We then have

For $n = 1$

$$f(n) = C1,$$

for $n = 2$

$$f(n) = C2,$$

for $n > 2 > 8$

$$f(n) = C3 * 2^{n-3}.$$

Duplicates

The Duplicates operation is very similar to the search operation with one difference being that both arrays (`array_a` and `array_b`) will grow in size.

Another difference is how the `loop` variable works in the code. The `loop` variable is the amount of times that the operation is performed e.g. how many times a random access or search was performed. The `loop` variable serves the same function when searching for duplicates; However, the program is now searching through an exponentially growing array an exponentially growing amount of times for each loop.

method

For the operation method both arrays are created equally as target arrays. As for what we are measuring between the time intervals we see that the program is comparing one entire array with the other array for each loop.

```
long t0 = System.nanoTime();
for(int k = 0; k < loop; k++){
    for(int i = 0; i < n; i++){
        int key = array_a[i];
        for(int j = 0; j < n; j++){
            if(key == array_b[j]){
                sum++;
                break;
            }
        }
    }
}
long t1 = System.nanoTime();
```

This will take too long to execute as the size increase. I therefore added a bit of code in the `main` method that will decrease the amount of loops as the array size grows larger.

```
int[] loops = {1000, 1000, 1000, 1000, 10, 10, 10, 1, 1, 1};
int indx = 0;
int loop = loops[indx];
...
// inside the for loop
System.out.println(n + " " + ((double)min)/loop + " ns");
loop = loops[indx++];
```

As seen in the code above, we are still trying to have loops to get a more accurate result. The result for the largest array sizes may differ alot from run to run due to taking too much time and thus using fewer loops.

results

Running the program with the `loop` variable varying depending on array size and taking the minimum time value out of 10 runs for every array size we get a table with the average minimum value:

size	**runtime**	**ratio**
100	1.5μs	1.0
200	6.2μs	4.1
400	22μs	15
800	82μs	55
1600	320μs	210
3200	1.2ms	800
6400	5.0ms	3,300
12800	20ms	13,000
25400	79ms	53,000
51200	320ms	210,000

Table 3: Search time for duplicates in arrays of different array sizes.

conclusion

Looking at how the time ratio increases we see that the time is increasing exponentially as the size of the array grows. Although more loops would give a more accurate output, there was negligible differences between runs with the greater array sizes.

A simple polynomial that roughly describes the execution time for finding duplicates in two arrays of size n can be described as follows:

Let n be the different arrays where $n = 1$ is the first, smallest array, then for $0 < n < 10$ we have:

$$f(n) = 1.5 * (3.9^n).$$

Summary

The time performances of the three different operations are different for each operation. The time does not change depending on the array size for the random access operation. The time increases proportionally to the array size for the search operation. The time increases exponentially as the array size gets bigger for the duplicates operation. The graphs for the time performance of the different operations are shown below:

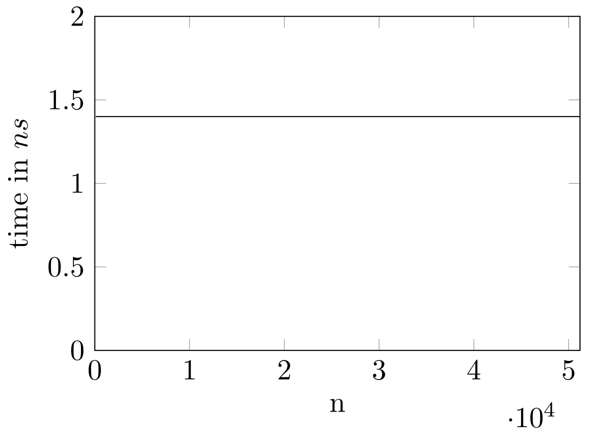

Figure 1: Random search

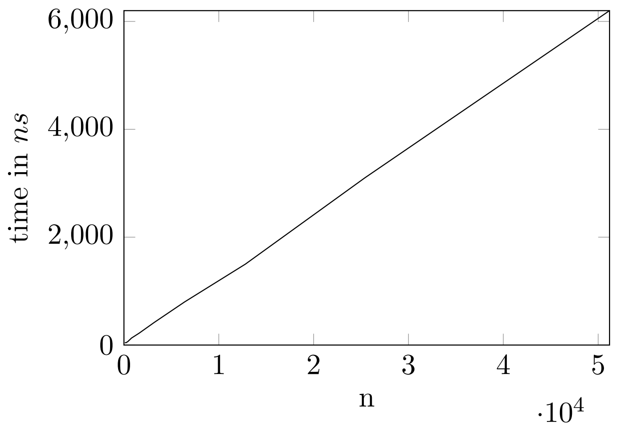

Figure 2: Search

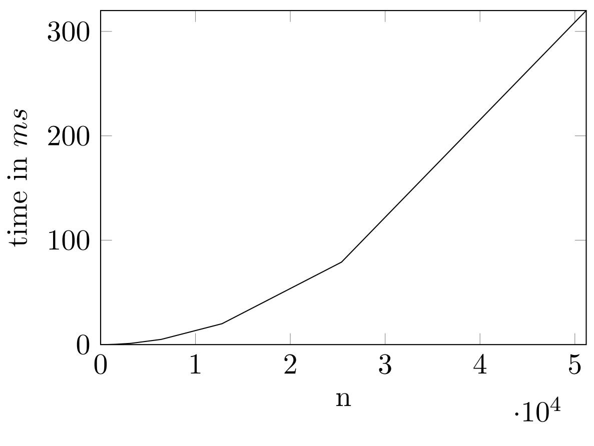

Figure 3: Duplicates