Arrays and performance

Algorithms and data structures ID1021

Johan Montelius

Fall term 2022

Introduction

In this assignment you should explore the efficiency of different operations over an array of elements. We take for granted that you have been working with arrays before so this should not be a problem, focus will be om performance measurement, presenting numbers and realizing that there is a fundamental difference between what is called $O(1)$, $O(n)$ and $O(n^2)$ (more on this later).

You should benchmark three different operations and determine how the execution time differs with the size of the array. The three operations are:

- Random access : reading or writing a value at a random location in an array.
- Search : searching through an array looking for an item.
- Duplicates : finding all common values in two arrays.

The implementation of these operations are quite straight forward, this is not problem. The problem is to do the benchmark, present the results and describe the general behavior for these operations.

Random access

When trying to measure how long time an operation takes one is limited by the resolution of the clock. If we have a clock that measures microseconds then obviously it will be very hard to measure an operation that only takes a few nanoseconds.

nanoTime

Fortunately we have a clock in the Java runtime system that will give us the execution time in nanoseconds - or at least the method is canned `System.nanoTime`. You first task is to figure out the accuracy of this clock. Run the following code and present your conclusion in the report.

```
for (int i = 0; i < 10; i++) {
  long n0 = System.nanoTime();
  long  n1 = System.nanoTime();
  System.out.println(" resolution " + (n1 - n0) + " nanoseconds");
}
```

Will the clock be accurate enough to measure how long time it takes to perform a single array access? We could try the following which will give us an idea of the problem we have:

```
int[] given = {1,2,3,4,5,6,7,8,9,0};

int sum = 0;
for (int i = 0; i < 10; i++) {
  long  t0 = System.nanoTime();
  sum += given[i];
  long t1 = System.nanoTime();
  System.out.println(" resolution " + (t1 - t0) + " nanoseconds");
}
```

The summing of all the values is there to make sure that the compiled code actually does something. If we omitted the summing the compiler could skip doing the access altogether. Note - a word of warning. A clever compiler could optimize this code if it realizes that the array is given and will not change during execution. It could (the Java compiler will probably not) do an optimization and compile the following code:

```
int[] given = {1,2,3,4,5,6,7,8,9,0};

int sum = 45;
for (int i = 0; i < 10; i++) {
  long  t0 = System.nanoTime();
  long t1 = System.nanoTime();
  System.out.println(" resolution " + (t1 - t0) + " nanoseconds");
}
```

Another problem with using the code as our benchmark is that we access the array in sequential order. In the above example it does probably not

matter but if we have a large array, sequential access could be faster then random access due to caching. In our benchmark we need to make sure that the access is random.

more operations

If you have done the above experiments you now realize that we need to do hundreds if not thousands or a hundred-thousand array operations in order to say something about the performance. We preferably also want to make the access random to prevent caching from playing a role.

To do random read operations we could of course make use of the `java.util.Random` method. Using this we can generate a random number and read a value from an array.

We first create an array that we will use as our target. We then try to measure the time it takes to do the random access operations.

```
int[] array = new int[n];
for (int i = 0; i < n ; i++) {
  array[i] =  i;
}

Random rnd = new Random();

sum = 0;
long  t0 = System.nanoTime();
for (int i = 0; i < n; i++) {
  sum += array[rnd.nextInt(n)];
}
long t1 = System.nanoTime();
```

What are we actually measuring? How do we avoid this - can we generate the random sequence before doing the array access? Could this piece of code help:

```
int[] indx = new int[l];
for (int i = 0; i < l ; i++) {
  indx[i] =  rnd.nextInt(n);
}
```

Now you could be ready to implement a function that does random access in an array of size n and returns the time it took in nanoseconds to do one operation (and one operation will be two read).

```
private static double access(int n) {
```

```
    int k = 1_000_000;
    int l = n;

    Random rnd = new Random();

    int[] indx = new int[l];
    // fill the indx array with random number from 0 to n (not including n)

    int[] array = new int[n];
    // fill the array with dummy values (why not 1)

    int sum = 0;
    long  t0 = System.nanoTime();
    for (int j = 0; j < k; j++) {
      for (int i = 0; i < l; i++) {
        // access the array with the index given by indx[i]
        // sum up the result
      }
    }
    long t_access = (System.nanoTime() - t0);

    t0 = System.nanoTime();
    // do the same loop iteration but only do a dummy add operation
    long t_dummy =  (System.nanoTime() - t0);

    return ((double)(t_access - t_dummy))/((double)k*(double)l);
  }
```

A last word of warning - when you are doing benchmark of methods like this the Java *Just-In-Time* compiler could play you some tricks. The first time the java engine executes a method it is probably not compiled into machine code. The run time system can however decide to compile the method into machine code and this compilation will of course take time. If you see that your first run of a method takes longer time than expected this is probably what is going on. One way to make things a bit more stable is to in your program run the method being benchmarked before the actual time measurements begin.

Task 1: Now for the experiment, set up a benchmark where you call the access function with a larger and larger n. Present your conclusions in a nice table and pay attention to the number of significant figures that you use.

presenting execution time

When you report execution times it is quite ridiculus to report somthing like $123456789ns$ - even if this is actually the result that you got. The reason that it is ridiculus is that the execution time will most likely change very much if you run the benchmark again. If the next run results in $124356789us$ then it is rather pointless to present the runtime using nine significant figures.

Your choice of the number of significant figures should reflect how stable your messurements are. If you run your benchmarks on a regula rlaptop there is a lot of things going on in the background that will effect the execution time. My guess is that you will not be able measure anything with more than three figures and in most cases two will do fine.

Even if you can make a stable measurment with three significant figures it might not be how you should report it. If you want to show that a sequence of measurments increase with a factor 2 for each measurment, large numbers will only be in the way.

Take a look at the table 1. The conclusion might be correct but it is not "clearly seen" at all.

n	100	200
prgm1	$12345678\mu s$	$22345678\mu s$
prgm2	$14325678\mu s$	$56213478\mu s$

Table 1: As clearly seen prgm1 double the execution time whereas almost incease with a factor four when n doubles.

Always think about the message that you want to deliver and then use as few numbers as possible to convey that message. If the doubling of execution time was the message then table 2 might be a better idea.

n	100	200
prgm1	$12ms$	$22ms$
prgm2	$14ms$	$56ms$

Table 2: Much better

When you present numbers thinks about readability. To say $23000000\mu s$ is of course only using two significant figures but is bery hard to compare this measument with $1200000\mu s$ ... is that half the exeution time or ...? Also refrain from using mathematical notations such as $1.23 \times 10^7 \mu s$, especially when you want someone to quickly see the relationship with 2.4×10^6, the information is of course all there but you don't make the connection without thinking - how about $12ms$ compared to $2.4ms$.

search for an item

Once we know how to set up a nice benchmark we can explore a simple search algorithm and see how the execution time varies with the size of the array.

When we setup the benchmark we want to capture the estimated time it would take to search for a random key in an array of unsorted keys. We assume that the number of keys is larger the size of the array - if we only had the keys 1, 2 and 3 we would probably find one very quickly even of the array was very large. So when we search for a given key it mighty be that we search through the whole array without finding it.

If we follow the pattern in the previous section we could try the following:

```
int[] keys = new int[m];
// fill the keys array with random number from 0 to 10*n

int[] array = new int[n];
// fill the array with with random number from 0 to 10*n

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

This works but we could be lucky/unlucky when we construct the array of keys or the array in which we do the search. Even if we do the search operation k times we are making an exact measurement of something that might be unlikely. We might want to make several different benchmarks and then add the execution time of all. How is this:

```
int[] keys = new int[m];
int[] array = new int[n];

for (int j = 0; j < k; j++) {
```

```
    // fill the keys array with random number from 0 to 10*n

    // fill the array with with random number from 0 to 10*n

    long  t0 = System.nanoTime();
    for (int ki = 0; ki < m; ki++) {
      int key = keys[ki];
      for (int i = 0; i < n ; i++) {
        if (array[i] == key) {
          sum++;
          break;
        }
      }
    }
    t_total += (System.nanoTime() - t0);
  }
```

Task 2: You now have to choose k (number of rounds) and m (number of search operation in each round) to something that gives you predictable results. You then do the benchmark for a growing size of the array n and examine the result. Find a polynomial that roughly describes the execution time as a function of n.

search for duplicates

So now for the final task, finding duplicates in two arrays of length n. This task is very similar to the search exercise but now we have one array given that will work as the keys that we search for in the second array.

Do as before an run the benchmark for a growing size of the arrays, n. Find a simple polynomial that roughly describes the execution time. Estimate how large n you would be able to handle if you had an hour of computation time.