Arrays and Performance

Isak Wilkens TIDAB

Fall 2022

Introduction

In this assignment performance measurements will be done in the programming language java regarding the efficiency of three different array operations. The efficiency will be measured with the built in clock "nanoTime" from the programming language. The three operations that will be tested are: random access, search for a key and duplicates search.

nanoTime

To measure the accuracy of the built in clock "nanoTime" i run a loop 10 times where i check the clock two times and then report the difference between the two measurements. The mean value for these 10 time differences is 140 ns with the lowest at 100 and highest of 200. This tells me that the clock is not accurate within these intervals and I will have to make large samples to minimize the error from the built in clock function.

These errors could be caused by software and hardware related features from the java programming language and the computer that runs the tests.

Performance tasks

task 1

The first task is to measure the performance of random access. The code here measures the time it takes to go through an array but which array value is chosen at random and the test is repeated one million times. The code also includes a dummy addition function to try to combat the built in errors from the clock. The time this dummy add operation takes is then deducted from the time it took to access the array. It can be questioned whether this is necessary since the sample size is so large. What is varied between these measurements is the size of the array which is represented by the variable "n".The results from the performance tests is shown in table

1 and the values presented are the average of 5 measurements on the same ”n”.

Filling the index array with a random number from 0 to n:

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

Accessing the array with random idex:

```
for (int j = 0; j < k; j++) {
    for (int i = 0; i < l; i++) {
        sum += array[indx[i]];
    }
}
```

Dummy add operation:

```
for (int j = 0; j < k; j++){
    sum=sum+1;
}
```

Size(n)	**10**	**100**	**1000**	**10000**	**100000**
time(ns)	1,7	0,54	0,40	0,41	1,0

Table 1: Mesurements of random access. Run-time in nanoseconds

What we can see is that the run-time decreases very little with larger array sizes up until around 10.000 where it instead increases. The reason for the decrease in run-time could be that the hardware and software uses temporal and spatial locality to speed up the process. The reason for the decrease in time could be software and hardware limitations on the other side such having to allocate new memory slots and using a large amount of resources. Ultimately the performance we can see is that the time varies very little and the algorithm has a complexity of O(1) meaning unchanged with regards to array size.

task 2

The second task is a search task where firstly an array of keys is randomly generated. The size was set to 100 so the keys would not always be found and not repeated. Secondly an array is generated which is searched until a key is found, it is the size of this array that is varied in the performance test. A polynomial is estimated for the results. This test is also repeated

one million times per run, the results are shown in table 2 and what is seen is an average of three runs.

Key array filled with random numbers, m is constant and set to 100:

```
 for (int i = 0; i < m; i++){
    keys[i]= rnd.nextInt((n*10));
}
```

Searched array filled with random numbers, n represents array size and l has the same value as n:

```
 for (int i = 0; i < l; i++){
    array[i]= rnd.nextInt((n*10));
}
```

Search loop that returns the run-time:

```
  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);
        }
        return t_total/(double)k;
```

Size(n)	**5**	**10**	**20**	**40**	**60**
time(ns)	790	2000	5400	7200	14000

Table 2: Mesurements of search algorithm. Runtime in nanoseconds

The result in table 2 shows that the run-time is increased linearly with the size of the array. Adapting a linear trend line of the results in Microsoft Excel gives the polynomial y=224x-164 representing the increase in time per additional size of the array. Above conclusion gives this algorithm the linearly increasing time complexity of O(n).

task 3

The third task is to create two randomly generated arrays where one array will serve as the key and then the second one is then searched for matching

elements. This way we can search for duplicates. The iterations per run is still kept at one million to have a large sample. A polynomial will then be derived from these results to analyze what array size would be required for one hour of run-time. The results are shown in table 3 and the results are an average of three runs. The code here is very similar apart from that the break function is removed and both arrays are dependent on the input value of "n".

search loop that returns the run-time:

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

Size(n)	**5**	**10**	**20**	**40**	**100**
time(ns)	57	130	340	990	6400

Table 3: Mesurements of search algorithm for duplicates. Runtime in nanoseconds

The results here show an exponential growth in run-time from increase in array size. Since both the key-array and the searched array grown with n we have a n^2 situation which results in the time complexity O(n^2), meaning exponential growth. The polynomial derived from these results through Microsoft Excel is: $y = 0,21x^2 + 15 + 153$, this means that to get one hour of run time on the same machine it would require an array size of approximately four million.