Oblig 1 in IN3030/IN4330 – v2022
Find the *k* largest numbers in a large array

January 27th, 2022

One problem for web search programs such as Google, Bing and DuckDuckGo is that a search can generate millions or even billions of answers (if you searched for "football" in Google using Chrome on 25/1-2022 you get 3.18 billion answers!) – more or less relevant to the one who requested the search. She / he will never be able to look at all the answers, but will merely like to look at the most relevant.
Each page (of the many millions of hits) has a relevance score, which we can assume is an integer so that the the larger this number is, the more relevant the page is. The problem then is to select the most relevant answers.

Let us help DuckDuckGo parallelize the solution to this problem. Suppose you have n answers and that the relevance score is stored in the integer array a[0..n-1]. A simple sequential solution A1 in Java is to use Javas built-in sort algorithm (Arrays.sort (int [] a)) to sort the entire array and then pick out the *k* largest numbers as a[0..k-1]. But this takes way too long because the sorting is typically $\mathcal{O}(n \cdot \log n)$.

A faster algorithm A2 is the following:

1. We first note that the first *k* numbers in a[0..k-1] are, obviously, the largest of the first k numbers in a[]. We then insert-sort a[0..k-1] in descending order (you must trivially rewrite the normal insert-sort (given below) to sort in descending order – i.e., with the largest first).
2. Then we know that the smallest number of the first *k* numbers in a[] is in a[*k-1*]. Then we compare a[*k-1*] in turn with each element in the rest of the array a[*k*..n-1]. If we find an element a[j] where a [j]> a [*k-1*], then we do the following:
   a. Replace a [*k-1*] with a[j].
   b. Insert-sort the new element into a[0 .. *k-2*] in descending order (remember that the code to sorting only one item is easier than full insertion sorting).
3. When step 2 is done, the *k* largest numbers are in [0.. *k-1*] and none of the other numbers has been overwritten or broken.

Task 1 - Sequential Algorithm

Implement the sequential algorithm A2 above. Test it for these different values of n = 1000, 10,000, …, 100 million by creating an array of pseudo-random numbers (java.util.random) and for each of these values, you test for two values of *k* = 20 and *k* = 100. Furthermore, test that you get the correct answer by sorting the same numbers Arrays.sort (int [] a) and compare your answers (descending order) with the corresponding *k* places in a[] after you have use Arrays.sort () to check that it is correct.
(Note to get the same 'random' numbers in the array when the Random class if you want to redo the run several times, the constructor of the Random class must get a starting number that is the same as previous runs – e.g., **Random r = new Random (7363);** Then we will get the same number sequence when we say: **r.nextInt (n)** in the loop that gets the next number between 0 and n-1**).** The numbers n and *k* are included as parameters when you start your program

```
java myprog <n> <k>
```

Write two tables, one for *k* = 20, one for 100, showing the time the two different methods A1: Arrays.sort and A2: Insert method uses for different values of n (n = 1000, 10,000, ..., 100 million). You should also produce two curves, one for k = 20 and one for k = 100 showing the results. The times reported must be the median of 7 calls on both methods as shown in the lecture week2. Submit your code to A2 and the table with your comments.

Optionally, you are welcome to explain why A2 is faster than A1 – even to give the asymptotic behavior of A1 and A2.

Task 2 - Parallel Algorithm

You should parallelize A2 as best you can with the *p* cores you have on your machine. Use an IfI machine, if necessary to have at least 4 cores. Take, for example, the starting point the parallelization of the FinnMax problem. You may then be left with a small sequential phase after that most of the calculations are done (as in FinnMax).

Submission

Write a report with two tables, one for $k = 20$ and one for $k = 100$, showing the times that Arrays.sort (), sequential and parallel insertion method A2 uses for different values of n (n = 1,000, 10,000, …, 100 million).
In addition, a graph with two curves is also preferred, one for $k = 20$ and one for $k = 100$, showing speedup as a function of n. The times reported here should be the median of 7 calls on both methods as shown in one of the lectures. Submit your code to both the sequential and parallel solution and the report.

The report should include what type of CPU (name and speed in GHz and the number of cores that it has, and, if available, the make and model of the CPU chip) you used; as well as comments on what value of n you observe that the parallel code gives speedup > 1, or why it does not achieve a speedup > 1 for that value of k. Also comment on how the execution times change for the two choices of k.

Submissions in IN3030/IN4330 are done thru Devilry.

Oblig1 must be done individually and submitted no later than

Wednesday 9th February 2022 at 23.59.00 (*NOT* 23.59.**59**)

NOTE: you can submit *multiple* times – only the last submission will be considered, so submit a version early so that you are sure that you do *not* miss the deadline.

Tips

1) You might get an error message when you try to run your program for n = 100 million saying that you have too little memory. If so, you can start the program with an option for increased memory. To request 6 GB for the program use:

```
java myprog -Xmx6000m 100000000 <+ other parameters>
```

If you do not have 64-bit Java, this does not work, max is then 1000m that you can ask for. If necessary, use a machine at IfI – or download 64-bit Java 8 to your machine from:
http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Appendix: Code for insert-sort a[0.. *k-1*];

a) Note: This sort is in **ascending** order – you have to rewrite it yourself and have it sorted in **descending order**:

```
/** This sorts a [v..h] in ascending order with the insertion algorithm */
void insertSort (int [] a, int v, int h) {
  int i, t;
  for (int k = v; k < h; k++) {
    // invariant: a [v..k] is now sorted ascending (smallest first)
    t = a[k + 1];
    i = k;
    while (i> = v && a[i]> t) {
      a[i + 1] = a[i];
      i--;
    }
    a[i + 1] = t;
  } // than for k
} // end insertSort
```

b) Note: To insert a new item at place a[*k-1*] you need a simpler version of the code above because the first k -1 elements in a[0 .. *k-2*] have already been sorted into descending order.