University of Waterloo
CS240 - Spring 2014
Assignment 2

Due Date: Wednesday June 4 at 09:15am

Please read `http://www.student.cs.uwaterloo.ca/~cs240/s14/guidelines.pdf` for guidelines on submission. Problem 1 is a written problem; submit your solutions electronically as a PDF with file name `a02wp.pdf` using MarkUs.

Problem 1 [6+6+6+6+6+6+6+6=48 marks]

Historically, users would normally query a database and obtain as a result a report containing each and every item satisfying the given query conditions. Such reports in the 1970s and 1980s would extend to at most a few hundred items which were then printed for convenience.

With the advent of databases containing millions of items in the late 1990s, such as a Web search engine, this usage quickly becomes impractical. For example a typical query to a search engine matches millions of documents. As you can imagine, this is not a good way to find a relevant page related to your query. To resolve this problem the operation "Top K" has been added to search engines. Under this framework each result is given a score or *rank* and the result of the query consists of the top k results by rank.

For very large databases such as Google, Facebook or LinkedIn, the data is distributed across thousands of computers $S_1, S_2, \ldots, S_n$ and then aggregated by a main server A.

Each computer S_i implements a max heap containing its share of all results. The central server A is then charged with obtaining the global top k results. Let $S_i.Heap$ denote the heap on server S_i. The server A initializes its own heap as follows

```
for i = 1 to n do
   A.Heap.Insert(S_i.Heap.DeleteMax());
```

Then it produces the top k results as follows:

```
for j = 1 to k do
   res=A.Heap.DeleteMax();
   Let i be server of origin of the res item above
   A.Heap.Insert(S_i.Heap.DeleteMax());
   print res;
```

(a) Argue that this method necessarily produces the top k results in order (hint: use induction on k).

Base Case:

For k = 1, we take the max element from A.Heap and print it. This is naturally the top result.
Induction Hypothesis:

For an arbitrary $1 < i < k$, this method produces the top i results.
Induction Step:

The ith iteration has produced the top i results. For the i+1 selection of the top result, there are 2 cases. Either the ith iteration inserted a new value that is greater than all the existing values in A.heap, or it did not.

In the first case, then the max of A.heap is the value inserted by the ith iteration, which bubbled to the top of A.heap using the heap insert. Deleting the max from A.heap would then return this value and the top i+1 results have now been produced. In the second case, the next top results was already in the heap, and was placed at the top of A.heap as a result of the bubble down that occurred when deletemax was executed on A.heap during the ith iteration. Then the deletemax for the i+1 iteration would return this value and again the top i+1 results would be produced.

Then by induction, the heap algorithm produces the top k results in order.

(b) Assume that each server S_i holds m values in its heap. Give the time taken by A in the code above to build the Heap. Do not forget to account for the time taken by the call to S_i.

Using the upper bound of size m for each delete from the S_i.heap, we know that the maximum operations would be $\log m$ by the definition provided in class. Each insert into A.heap would similarly have a maximum of $\log n$ operations, bounding up to an insertion into a size n heap each iteration. Let T(n) denote the time to run this algorithm for length n. Summing these over all values we get:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} \log n + \log m \\ &= \theta(n \log n) + \theta(n \log m) \\ &\therefore T(n) \in \theta(n \log(nm)) \end{aligned}$$

(c) Now give the time taken to produce the top k results in the loop above in terms of m, n and k.

There are 3 constant time operations in the loop, the assignment of res, the printing of res and the evaluation of i as the server of origin of the res item above. These are represented each as $\theta(1)$ time. The loop does an insert and a delete from A.heap, which has $\log n$ operations in each case as discussed in class. The delete from the S_i heap of size m would have $\log m$ operations, bounding up to ignore the fact that the size of the heap is decreasing with each delete. Let T(k) represent the time to assemble the top k results. This results in the following:

$$\begin{aligned} T(k) &= \sum_{j=1}^{k} \theta(1) + \theta(1) + \theta(1) + 2\log n + \log m \\ &= \theta(2k) + \theta(2k \log n) + \theta(k \log m) \\ &\therefore T(k) \in \theta(k \log(nm)) \end{aligned}$$

Marc missed the class on heaps and ended up implementing a simpler but less efficient algorithm.

Each server S_i holds a sorted linked list of values by decreasing rank. Then server A assembles the results in a list of k candidate results as follows:

```
for each i = 1 to n do
  res = S_i.SortedList.DeleteMax()
  A.SortedCandidateList.K_InsertionSort(k,res)
```

where K_InsertionSort maintains a doubly linked list sorted by decreasing rank with at most k values as follows

```
procedure K_InsertionSort(int k, data item)
  if (A.SortedCandidateList.Size == k) && (item.rank > A.SortedList.LastElem.rank )
              // only insert elements big enough for the current top k
    A.SortedCandidateList.LastElem = A.SortedCandidateList.LastElem.Prev
    A.SortedCandidateList.Size--;     // drop last element about to be displaced

  if (A.SortedCandidateList.Size < k) // if list doesn't yet have k elements
    A.SortedCandidateList.InsertionSort(item)   // call classical insertion sort
```

Producing the top k values is as follows:

```
for j = 1 to k do
  res=A.SortedCandidateList.DeleteMax();
  print res;
  Let i be server of origin of the res item above
  res = S_i.SortedList.DeleteMax()
  A.SortedCandidateList.K_InsertionSort(k,res)
```

(d) Argue that this method also produces the top k results in order.

Marc's solution first created a list of the top k candidates in a sorted linked list from all n servers.

Base case:

In the k = 1 case, the top element from the server linked lists has been sorted to the top of the candidate list and is printed after taking it from the candidate list using the delete max function. This clearly produces the top result.

Induction Hypothesis:

For an arbitrary $i < i < k$, the ith iteration produces the top i results in order.

Induction Step:

For the i+1 iteration, the top i results have been return and there are again two cases to be considered. The ith iteration necessarily removed the top element from the candidate list and added the next largest item from the server's linked list that the item originated.

In the first case, this added item was larger than the existing items in the linked list and was sorted to the top using classical insertion sort. Deleting the max from the candidate list would then result in the top i + 1 results. In the second case, the inserted item was not the largest item when added to the candidate list and was placed in its correct position when the classical insertion sort was performed, while the actual max was positioned as the first element. In this case the i+1 iteration again removes the max from the candidate list and the top i+1 results are produced.

Then by induction, Marc's algorithm produces the top k results in order.

(e) Assume as before that each server S_i holds m results. Give the time taken in the worst case by A in the code above to assemble its list of candidates in terms of n and k. You may assume that $n \geq k$ where needed.

In each loop iteration, there are 5 constant time operations. The assignment to res in combination with the delete from S_i, since it is a sorted linked list, are two constant operations. The two comparisons in the first if statement, as well as the single comparison in the last if statement raise the total to 5 constant time operations.

Furthermore, each iteration i up until the list is filled with size k would take up to i swaps using the classical insertion sort insertion of one element into an already sorted array.

Past the kth insertion, in the worst case, the first if statement will always be entered

and there are an additional two constant time operations for every iteration from k+1 to n. The time complexity can therefore be expressed as the following:

$$\begin{aligned}
T(k) &= \sum_{i=1}^{n}(\theta(1)+\theta(1)+\theta(1)+\theta(1))+\theta(1))+\sum_{i=1}^{k} i + \sum_{i=k+1}^{n}(\theta(1)+\theta(1)+k) \\
&= \theta(5n)+\theta(\frac{k(k+1)}{2})+\theta((n-k)(2+k)) \\
&= \theta(5n)+\theta(\frac{k^2}{2}+\frac{k}{2})+\theta(2n+nk-2k-k^2) \\
&= \theta(7n)+\theta(\frac{3k}{2}+nk-\frac{k^2}{2}) \\
&\therefore T(k) \in \theta(nk)
\end{aligned}$$

(f) Now give the time taken to produce the top k results in the loop above in terms of m, n and k.

There are 9 constant time operations in each loop. The assignment of res and use of the max delete of the candidate list are each constant time operations. Printing res is an additional operation, as well as assigning i to be the server of origin, and assigning res once more to be the max of the sorted list, which is also a linked list like the candidate list.

In addition to these operations, since the size of the sorted candidate list is always k-1 when the K_InsertionSort is called, then the cost would be the 3 comparisons in the K_InsertionSort followed by the max k operation classical insertion of the new element. This total can be expressed as the following:

$$\begin{aligned}
T(k) &= \sum_{j=1}^{k}\theta(9)+k \\
&= \theta(9k)+\theta(k^2) \\
&\therefore T(k) \in \theta(k^2)
\end{aligned}$$

(g) Let $n = 10,000$ and $m = 10,000,000$ for what values of k is Marc's solution faster, in big-Oh order terms, than the optimized heap solution above, in the worst case?

Comparing the algorithms and determining when Marc's algorithm performs better than the heap solution, we see that:

Heap solution:

$$\theta(n\log(nm)) + \theta(k\log(nm)$$
$$= \theta(n\log(nm))$$

Marc's solution:

$$\theta(k^2) + \theta(nk)$$
$$= \theta(nk)$$

Subbing in we get:

$$10000k + k^2 < 10000\log_2(10000 * 10000000) + k\log_2(10000 * 10000000)$$

(h) Google actually uses $k \approx 1000$ and then applies a post-processing stage to produce the top ten results. Which solution would you use in the worst case knowing again that $m \approx 10^7$, and $n \approx 10^4$ assuming that the actual running time is a small constant times the number of operations you computed?

Solving the inequality above results in the following:

$$10000k + k^2 < 10000\log_2(10000 * 10000000) + k\log_2(10000 * 10000000)$$
$$k \leq 36$$

I would use the heap solution as for any values of k greater than 36, since past 36 Marc's solution is going to perform worse than the heap solution. In Google's case, I would definitely use the heap solution.