1.204 Computer Algorithms in Systems Engineering

Spring 2010

Problem Set 4: Satellite data sets

Due: 12 noon, Monday, March 29, 2010

1. Problem statement

You receive data from a series of satellites on cloud cover and atmospheric temperatures on a near-real-time basis. Each satellite transmits a series of data points, represented as an array, for the areas it can observe. There are multiple satellites; sometimes their measurement areas overlap, and sometimes there are gaps. To simplify the problem, we will assume that the data can be sorted by one dimension (longitude) instead of two (latitude and longitude).

The satellites report their data at approximately the same time, and you must merge all the data you receive into a single array, sorted by longitude. The order in which the satellites report appears random. They all report within a few seconds or even milliseconds of each other, but the order is essentially unpredictable since it's based on their distance from the ground station, the processing time on the satellite to compute the measurements and the number of measurements to transmit.

This is a common problem in monitoring and analyzing sensor data. To solve it, you will implement an optimal merge algorithm as outlined in the Horowitz text in section 4.7.

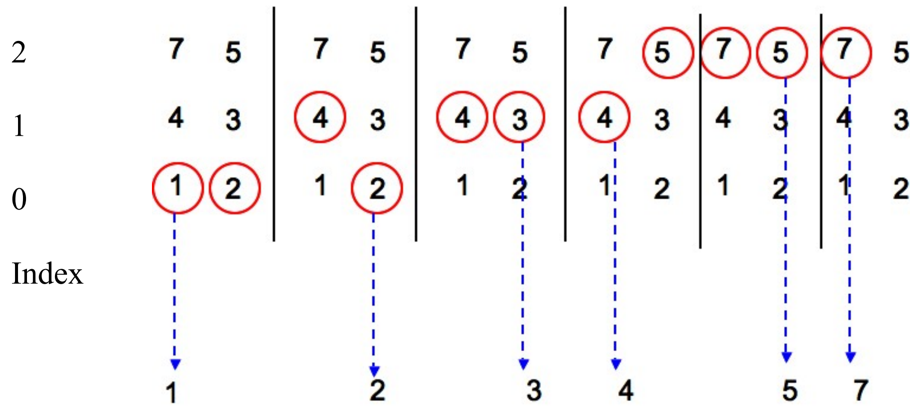
2. Algorithm

a. Merging two arrays

Section 3.4 in the textbook describes the mergesort algorithm, which recursively sorts an array. The mergesort algorithm uses a helper algorithm, called merge, which is used for merging two sorted subsets of an array. This is similar to how quicksort uses partition.

The merge algorithm works as follows. Initially, you have two indexes pointing to the 0th elements of both arrays. These are the smallest elements in both the arrays as they are sorted. You then choose the smaller of these two elements and this is the 0th element in the new (result) array. You then update the index in the array from which you have chosen the 0th element to the next element (1st element) and again compare the elements to fill the 1st position in the sorted array. At each step, you compare the smallest element in the two arrays and choose the smaller of them to fill the new array and update the indexes accordingly until you fill the new array completely.

As an example, consider the two arrays: {1,4,7} and {2,3,5}, which are sorted and have to be merged into a single sorted array. The size of this sorted array is 6 (3+3). The figure below shows how the merge operation works for these two arrays. The two arrays are represented vertically. The vertical lines separate the successive steps in the merge algorithm. The circles represent the indexes. At each step of the algorithm, you compare two elements (pointed to by the indexes) and choose the smaller of them to fill in the new array.



<u>Pseudocode:</u>

```
Algorithm: merge(a[ ], b[ ]) returns c[ ] {
  d= size of a + size of b;
  c[ ] = new array of size d;
  h= i= j =0;
  while (( h < a.length) and (j < b.length) {
        if (a[h] <= b[j]) then
            c[i] = a[h]; h= h+1;
         else
           c[i] = b[j]; j= j+1;
    i =i+1;
   }
  if (h >= a.length) then
     for (k= j to b.length) do
        { c[i] = b[k]; i=i+1;}
   else
     for (k= h to a.length) do
       { c[i] =a[k]; i=i+1;}
  return c[];
}
```

b. Merging many arrays

Consider two arrays of sizes x1 and x2, as sent by two satellites. A merge operation of these two arrays produces an array of size (x1+x2) and this operation involves x1+x2 moves, as described above. The greedy algorithm described in the text is an algorithm for merging arrays x1, x2, …, xn. It minimizes the total number of moves to merge all the arrays. Consider merging 3 arrays of sizes 5, 10, and 15. Merging arrays of sizes 10 and 15 would require 25 moves and merging this array with a size of 5 would require 30 moves. Thus, the total number of moves required is 25+30 = 55 moves. However, the optimal merge only requires 5+10 = 15 moves for merging the two smallest arrays and 15+15 =30 moves for merging with the largest array. Thus, the optimal merge requires 45 moves.

The greedy algorithm works in the following manner. A list of the different array sizes is maintained and at each step the two smallest arrays are merged. Once two arrays are merged, the merged array becomes a new array, whose size is the sum of the sizes of the two arrays. After this merge operation, the two arrays are removed and replaced with this new array. At the end of this whole operation, a single array is obtained whose size is the sum of the sizes of all the arrays.

The problem requires a tree structure for the merge operations and a second data structure for getting the two smallest array sizes at any given step in the algorithm. A heap is, of course, the obvious candidate for the second data structure. The pseudocode for the algorithm is described in page 236 of the text.

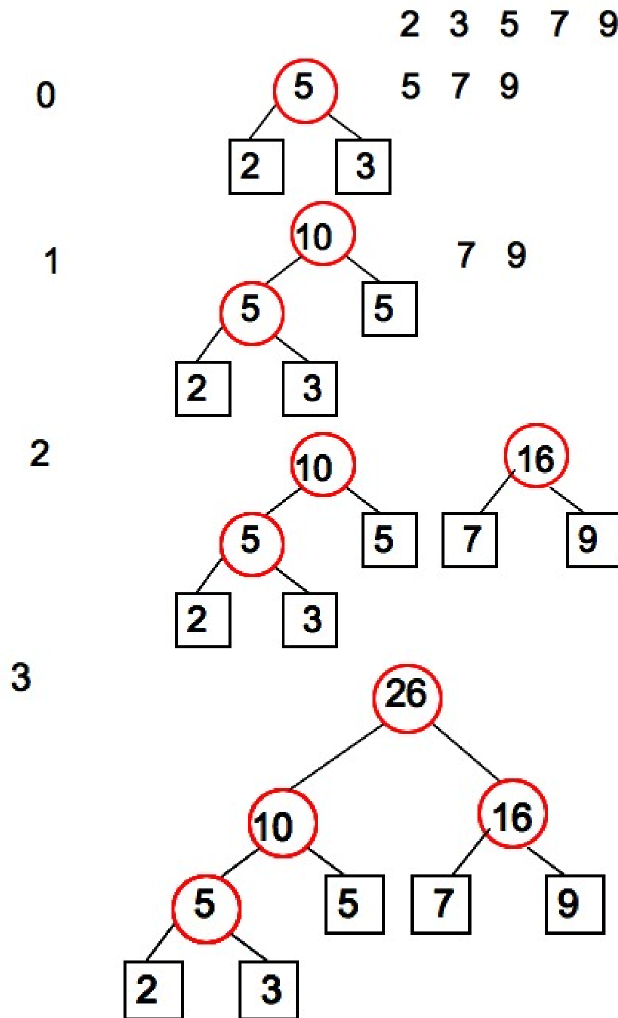
Pseudocode:

```
Algorithm mergeArrays(n) {            // n is number of arrays
    for i:=0 to n-2 do {
        p = new node
        p.lchild = min node from heap;
        p.rchild = min node from heap;
        p.size = p.lchild.size  + p.rchild.size
        insert p into heap;
      }
  return min node from heap with final result;
}
```

This pseudocode uses a node data structure, similar to the one covered in lecture:

```
Data Structure node {
  node lchild;
  node rchild;
  the array to be merged;
}
```

Consider an example where the sizes of arrays are: 2, 3, 5, 7, and 9. The figure below shows how the optimal merge algorithm works and the resulting tree structure. The root element of the tree has the value of the optimal number of moves. We also show the iteration number in the leftmost part of the figure. The numbers along the top line of each iteration are the arrays still to be merged; they may be circled or not. When only one array is left, at the last step, the algorithm is done.



Step	Arrays	Action
0	2, 3, 5, 7, 9	Combine the two smallest arrays, 2 and 3, into array of size 5
1	5, 5, 7, 9	Combine the two smallest arrays, 5 (just created) and 5 (original) to create an array of size 10
2	7, 9, 10	Combine the two smallest arrays, 7 and 9, into array of size 16
3	10, 16	Combine two smallest arrays, 10 and 16, into array of size 26
Done	26	Single merged array of size 26

Keep the arrays that exist at each iteration in a heap. Delete the two smallest from the heap, merge them, create a new node, and insert it into the heap.

c. Assignment details

You are completely free to implement the array merge algorithm in any way that you wish. The following hints are provided.

1. You will need a node class; it will contain an array to be merged. Node objects will be put into a heap; the size of the array will determine where the node is stored in the heap. Your node class must implement Comparable; you must then write a compareTo() method that the heap will use to determine where the node object is placed.

While your nodes will conceptually form a tree, you may or may not need to create and manage an actual tree in your implementation.

2. You will need a simple class to store a single satellite data observation. Each observation has a longitude (0-360), cloud cover (0-100%) and temperature (0-400 K). All three are doubles. This class must also implement Comparable so that the merge() method knows how to put satellite data objects in order. The compareTo() method in this class should result in the objects being sorted by longitude.

3. You will need to write the merge method (described in part a above). Implement the pseudocode shown above. Your merge method should take two arrays of type Comparable as its arguments, and return an array of type Comparable as its result.

4. Use the Heap class provided in lecture to manage the nodes. Heap.java, provided in lecture, is a max heap, and for algorithm you need a min heap, in which the value of the parent element is smaller than its children. In a min heap, the root element is the smallest element in the heap. It's probably easiest to modify the code from class to create a new class MinHeap; it requires reversing a few inequalities. You can also use the existing Heap.java code, with appropriate design of the compareTo() method in the objects that you keep in the heap.

5. We give you part of Java class MergeTest. It contains a method randomGenerator(), which generates an array of random satellite data. The size of this array is between 1 and 200. The array of satellite data generated by this function is already sorted on the basis of the longitude. We also give you a fragment of the main() method. The first few lines of main() generate a random number of satellite data arrays, between 1 and 20. You are free to modify, use or not use the code provided to you.

You must write a complete implementation of the optimal algorithm for merging multiple arrays. We suggest that, after implementing the classes mentioned above, you complete the main() method in MergeTest to:

- Generate a number of satellite data arrays
- Invoke the optimal merge algorithm to find the single merged array
- Report the total number of steps (the sum of the steps in merging each pair of arrays) required
- Compare this to the number of elements in the arrays (Show the ratio)

d. Analysis

Include a text file (MS Word or similar) in the zip file you submit with your answers to the following questions:

1. Show that the binary tree of nodes is a full binary tree. In a full binary tree, each node has either zero or two children, and each level of the tree is filled before a lower level is created. A very brief explanation is fine.

2. If n is the number of arrays, what is the running time of this algorithm? Assume that you have a heap implementation for finding the two smallest elements. Use O() or Ω() or Θ() notation, whichever is most descriptive. Hint:

$$\Sigma \, [\, \lg(n) + \lg(n-1) + \lg(n-2) \,] <= n \lg(n)$$

3. Perform a series of runs of your algorithm to see if its performance is consistent with the result you obtained in question 2 above. Choose a simple strategy for the series of runs to trace the time your algorithm takes as n increases. You will need to control the number of data elements in each array and the number of arrays, so those will need to be deterministic. Continue to generate the data within each array randomly. You can either time the runs or count the number of steps. You may assume the number of steps to merge two arrays of size m and n is m+n. Discuss briefly.

Turn In

1. Place a comment with your full name, Stellar username, and assignment number at the beginning of all `.java` files in your solution.
2. Place all of the files in your solution in a single zip file.
    a. Do not turn in electronic or printed copies of compiled byte code (`.class` files) or backup source code (`.java~` files)
    b. Do not turn in printed copies of your solution.
3. Submit this single zip file on the 1.204 Web site under the appropriate problem set number. For directions see **How To: Submit Homework** on the 1.204 Web site.
4. Your solution is due at noon. Your uploaded files should have a timestamp of no later than noon on the due date.
5. After you submit your solution, please recheck that you submitted your .java file. If you submitted your .class file, you will receive **zero credit.**

Penalties

- 30 points off if you turn in your problem set after Monday (March 29) noon but before noon on Wednesday (March 31). You have two no-penalty two-day (48 hrs) late submissions per term.
- No credit if you turn in your problem set after noon on the following Wednesday.

MIT OpenCourseWare
http://ocw.mit.edu

1.204 Computer Algorithms in Systems Engineering
Spring 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.