Name:_____ UW NetID:_____

CSE332 Summer 2012 Final Exam, August 15, 2012

SOLUTIONS

Please do not turn the page until the bell rings.

Rules:

- The exam is closed-book and limited-note. You are permitted a single, handwritten 3x5 index card of notes. You must turn in this card with your exam.
- Calculators are also permitted but not necessary.
- Please stop promptly at 12:20.
- You can rip apart the pages, but please staple them back together before you leave.
- Blank paper for extra room are available upon request.
- This exam contains 9 questions (many with multiple parts). There are **110 points** total, but the exam is worth **100 points**, meaning that you may earn some extra points.

Advice:

- The questions are not necessarily in order of difficulty. Read through the entire exam first and then **skip around** as you see fit. Make sure you get to all the problems.
- Read questions carefully. Understand a question before you start writing.
- Write down thoughts and intermediate steps to earn partial credit. **Circle your final answer**.
- If you have questions, ask.
- Relax. You are here to learn.

EXAM SCORE	/ 100

1. (18 pts) General Knowledge

The following questions all refer to ideas involving algorithms, complexity, parallelism, and concurrency. Provide your answer as indicated.

(a) Which of the following comparison sorts are in-place sorts? Circle all that apply.

[SelectionSort]	[InsertionSort]	[HeapSort]	BucketSort
[SequentialQuickSort]	SequentialMergeSort	ParallelQuickSort	ParallelMergeSort

(b) A lock that allows a single thread to acquire the lock multiple times is called a(n) re-entrant lock

(c) Assume a graph algorithm is $\Theta(|V|\log|V| + |E|)$. If a graph is **dense**, what will the algorithm's actual performance be? $\Theta(|E|)$ or $\Theta(|V|^2)$

(d) What are the **average case** performances (big-O) for the following sequential sorting algorithms?

SelectionSort	$O(n^2)$
InsertionSort	$O(n^2)$
HeapSort)	$O(n\log n)$
MergeSort	$O(n\log n)$
QuickSort	$O(n\log n)$

(e) For a parallel algorithm, its *parallelism* is $O(n)$ and its *work* is $O(n^2\log n)$. What is the algorithm's *span*? $O(n\log n)$

(f) The time complexity for calculating the *degree* of a node in an undirected graph represented by an adjacency matrix is? $O(|V|)$

(g) In regards to performance, Moore's 's law is an observation and Amdahl's 's law is a mathematical theorem.

(h) Using one lock per bucket in a separate chaining hashtable is an example of fine-grained lock granularity.

(i) A parallel pack has $O(n)$ *work* and $O(\log n)$ *span*.

(j) To optimize the amortized efficiency for `find` in Disjoint Set Union-Find, one needs to implement both weighted union and path compression

(k) No comparison-based sorting algorithm can do better than $\Omega(n\log n)$ in the worst-case.

2. (10 pts) Which Sort is Which?

Each of the following arrays shows a comparison sort in progress. There are five different algorithms: `SelectionSort`, `InsertionSort`, `HeapSort`, `QuickSort`, and `MergeSort`. Your task is to match each array to the algorithm that would produce such an array during its execution. You must also provide a short justification for your answer.

(a) **02 04 01 07 09 08 12 19 13 27 25 33 44 35 51 85 98 77 64 56**

Sorting Algorithm: QuickSort

Reason: The values show signs of having been partioned (and by process of elimination)

(b) **12 25 51 64 77 08 35 09 01 07 04 33 44 19 02 85 98 13 27 56**

Sorting Algorithm: InsertionSort

Reason: The first five elements are sorted but are not the minimum values

(c) **56 51 44 27 13 33 35 25 09 12 04 01 08 19 02 07 64 77 85 98**

Sorting Algorithm: HeapSort

Reason: Maximum values are at the end and the rest of the array is a max-heap

(d) **01 02 04 64 12 08 35 09 51 07 77 33 44 19 25 85 98 13 27 56**

Sorting Algorithm: SelectionSort

Reason: The first few elements are the minimum values in the array

(e) **12 25 51 64 77 01 07 08 09 35 04 19 33 44 02 85 98 13 27 56**

Sorting Algorithm: MergeSort

Reason: The array is divided into fourths of which some are in sorted order

3. (12 pts) Radix Sort

Perform a radix sort of the following list of numbers, using a radix of 10, into ascending order:

329 595 408 15 291 466 7 290 141 53 210 883 107 395 663

Show the bin/bucket sort conducted in each pass of the radix sort using the provided tables. You must also write down the order of the numbers after each pass.

First Pass

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
290 210	291 141		53 883 663		595 15 395	466	7 107	408	329

Order after first pass: 290 210 291 141 53 883 663 595 15 395 466 7 107 408 329

Second Pass

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
7 107 408	210 15	329		141	53	663 466		883	290 291 595 395

Order after second pass: 7 107 408 210 15 329 141 53 663 466 883 290 291 595 395

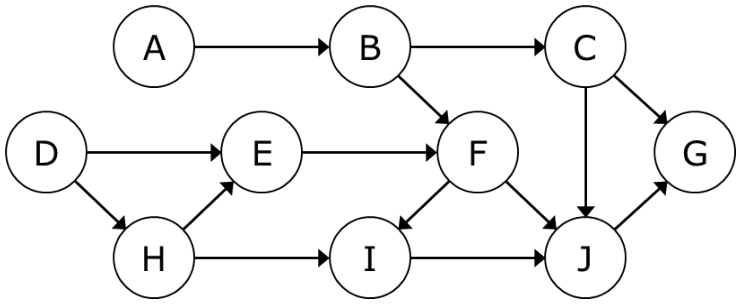
Third Pass

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
7 15 53	107 141	210 290 291	329 395	408 466	595	663		883	

Order after third pass: 7 15 53 107 141 210 290 291 329 395 408 466 595 663 883

4. (12 pts) Topological Sorting

You will perform two topological sorts on the directed graph to the right:



When the processing of a vertex creates more than one new pending vertex, add the new pending vertices to your set of pending vertices in alphabetical order (e.g., `push(X)`, `push(Y)`, `push(Z)`).

For each topological sort, use the provided tables to compute the topological sort and your final solution. Show your work to allow partial credit (e.g., show adding and removing from the set).

(a) Perform a topological sort using a **queue** to maintain the set of pending vertices.

	A	B	C	D	E	F	G	H	I	J
In-degree	~~0~~	~~1~~ ~~0~~	~~1~~ ~~0~~	~~0~~	~~2~~ ~~1~~ ~~0~~	~~2~~ ~~1~~ ~~0~~	~~2~~ ~~1~~ ~~0~~	~~1~~ ~~0~~	~~2~~ ~~1~~ ~~0~~	~~3~~ ~~2~~ ~~1~~ ~~0~~

Queue	~~A~~ ~~D~~ ~~B~~ ~~H~~ ~~C~~ ~~E~~ ~~F~~ ~~I~~ ~~J~~ ~~G~~

Final	A	D	B	H	C	E	F	I	J	G

(b) Perform a topological sort using a **stack** to maintain the set of pending vertices:

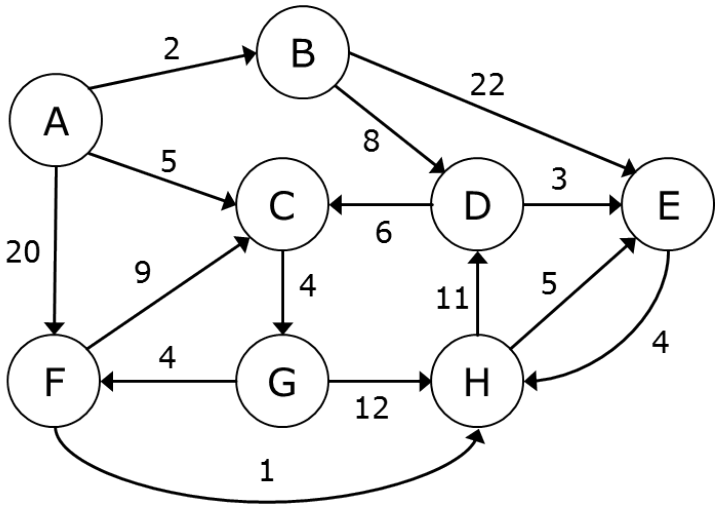
	A	B	C	D	E	F	G	H	I	J
In-degree	~~0~~	~~1~~ ~~0~~	~~1~~ ~~0~~	~~0~~	~~2~~ ~~1~~ ~~0~~	~~2~~ ~~1~~ ~~0~~	~~2~~ ~~1~~ ~~0~~	~~1~~ ~~0~~	~~2~~ ~~1~~ ~~0~~	~~3~~ ~~2~~ ~~1~~ ~~0~~

Stack	~~G~~ ~~J~~ ~~I~~ ~~F~~ ~~C~~ ~~B~~ ~~E~~ ~~H~~ ~~D~~ ~~A~~

Final	D	H	E	A	B	F	I	C	J	G

5. (10 pts) Dijkstra's Shortest Path

Consider the following directed, weighted graph:



(a) Use Dijkstra's algorithm to calculate the single-source shortest paths from vertex *A* to every other vertex. Show your steps in the table below. As the algorithm proceeds, cross out old values and write in new ones, from left to right in each cell. If during your algorithm two unvisited vertices have the same distance, use alphabetical order to determine which one is selected first. Also list the vertices in the order which Dijkstras algorithm marks them known:

Order vertices marked as known: A B C G D E F H

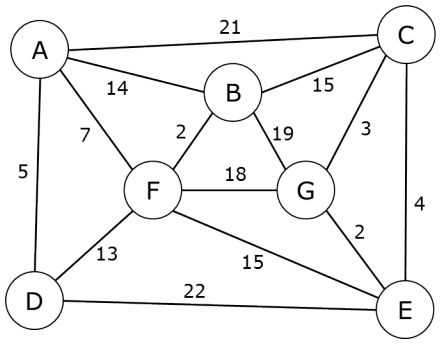
Vertex	Known	Distance	Path
A	Y	0	–
B	Y	2	A
C	Y	5	A
D	Y	10	B
E	Y	~~24~~ 13	~~B~~ D
F	Y	~~20~~ 13	~~A~~ G
G	Y	9	C
H	Y	~~17~~ 14	~~E~~ F

(b) What is the lowest-cost path from *A* to *H* in the graph, as computed above?

A—C—G—F—H

6. (14 pts) Minimum Spanning Tree

You will be computing two minimum spanning trees for the following undirected, weighted graph.



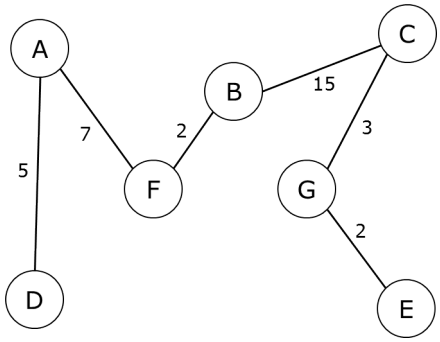
(a) Step through **Prim's algorithm** to calculate a minimum spanning tree starting from vertex *A*. Show your steps in the table below. As the algorithm proceeds, cross out old values and write in new ones, from left to right in each cell. If during your algorithm two unvisited vertices have the same distance, use alphabetical order to determine which one is selected first. Also list the vertices in the order which Prims algorithm marks them known:

Order vertices marked as known: A D F B C G C

Vertex	Known	Distance	Path
A	Y		–
B	Y	~~14~~ 2	~~A~~ F
C	Y	~~21~~ 15	~~A~~ B
D	Y	5	A
E		~~22~~ ~~15~~ ~~4~~ 2	~~D~~ ~~E~~ ~~C~~ G
F	Y	7	A
G	Y	~~18~~ 3	~~F~~ C

(b) List the edges in the minimum spanning tree as computed above. Please list vertices in edges by alphabetical order (e.g., A—B and not B—A).

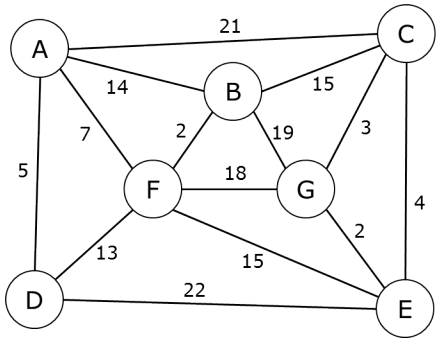
A—D, A—F, B—C, B—F, C—G, E—G



Question 6 continues on back ⟹

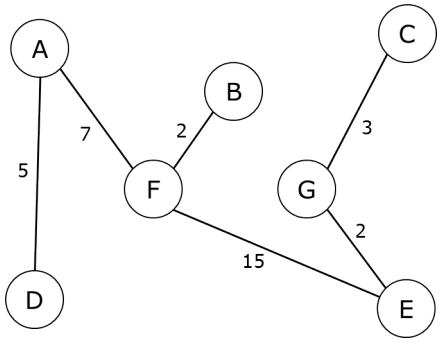
6. (continued)

For your convenience, here is the graph again.



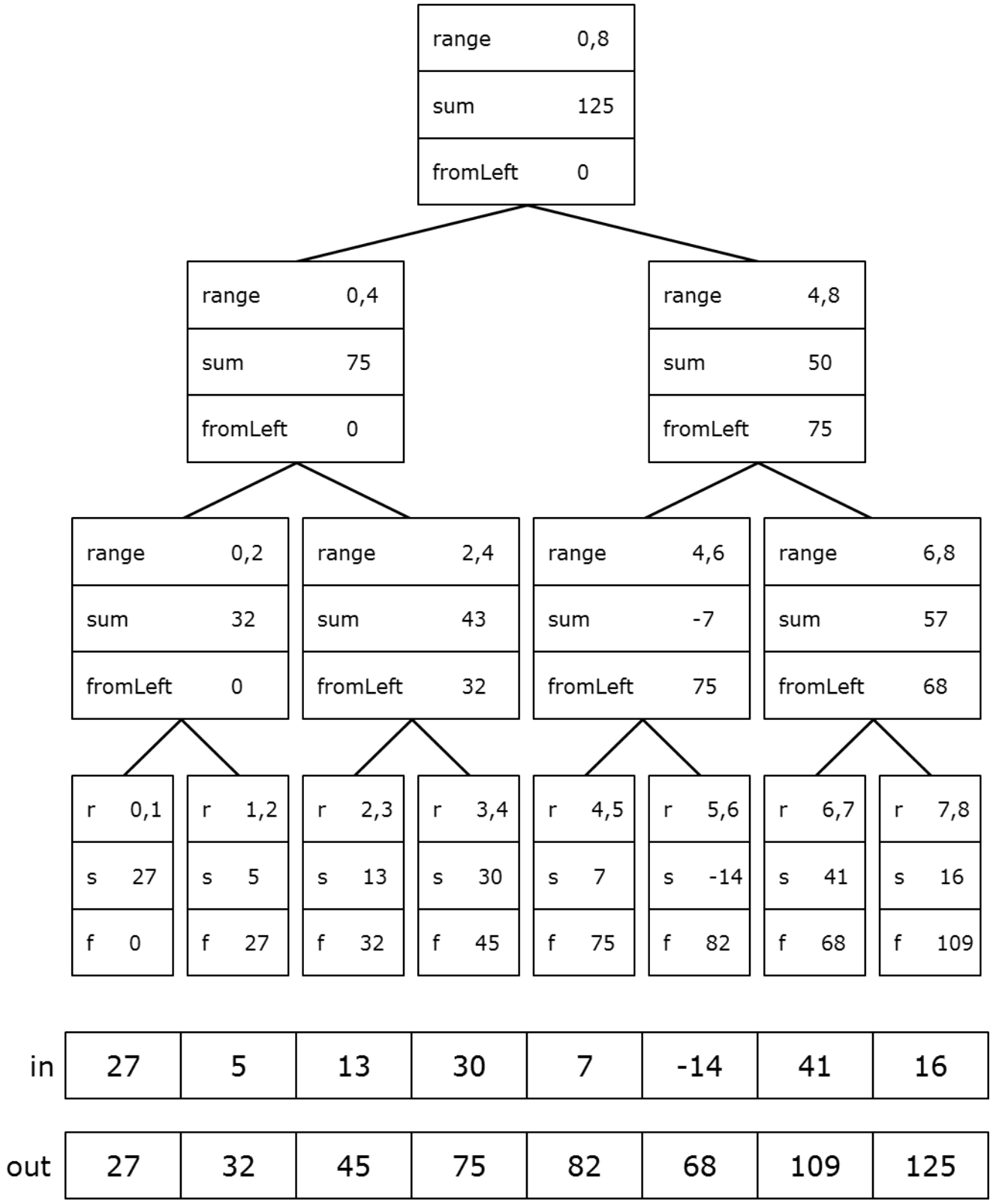
(c) Step through **Kruskal's algorithm** to calculate a minimum spanning tree of the graph. Show your steps in the table below, including the disjoint sets at each iteration. If you can select two edges with the same weight, select the edge that would come alphabetically **last** (e.g., select E—F before B—C. Also, select A—F before A—B).

Edge Added	Edge Cost	Running Cost	Disjoint Sets
–	–	0	(A) (B) (C) (D) (E) (F) (G)
E—G	2	2	(A) (B) (C) (D) (E G) (F)
B—F	2	4	(A) (B F) (C) (D) (E G)
C—G	3	7	(A) (B F) (D) (C E G)
A—D	5	12	(A D) (B F) (C E G)
A—F	7	19	(A B D F) (C E G)
E—F	15	34	(A B C D E F G)



7. (12 pts) Parallel Prefix

Simulate the parallel prefix algorithm by filling in the appropriate values in the prefix tree below. The input array is provided. You will need to determine the output array and the values of `range`, `sum`, and `fromLeft` in the tree's nodes.



8. (12 pts) Another Parallel Sort

In addition to QuickSort and MergeSort, SelectionSort can also be made parallel.

(a) Provide a pseudocode description of how you would implement *ParallelSelectionSort*. Hint: Your solution need not be complex and should take advantage of parallel algorithms covered in class.

(b) What is the *work* for your algorithm?

(c) What is the *span* for your algorithm?

(d) What is the *parallelism* for your algorithm?

(e) Is this a significant speed-up? Why or why not?

(f) Does this suggest that ParallelSelectionSort is a good algorithm to implement? Why or why not?

Solution

(a) We will just call ParallelFindMin multiple times but will assume that ParallelFindMin returns the index of the minimum.

```
ParallelSelectionSort(int[] array) {
   for(int i=0; i<array.length-1; i++) {
      int j = ParallelFindIndexOfMin(array,i,array.length);
      swap array[i] and array[j]
   }
}
```

(b) The work is the same as for sequential SelectionSort: $O(n^2)$

(c) Since the parallel min method takes $O(\log n)$ time and we run it n times, the span is $O(n \log n)$

(d) The parallelism is $O(\frac{n}{\log n})$

(e) This is a significant speed-up as it is an exponential improvement.

(f) This is not a great algorithm as we are using parallelism to get as fast a sort as sequential sorts like MergeSort and HeapSort. This is like using a nuclear bomb to swat a fly.

9. (10 pts) Completing Fork/Join Code

The following is partial ForkJoin code. Take a moment to briefly read through it.

```
class TwoSmallestDistinct {
   static final ForkJoinPool fjPool = new ForkJoinPool();

   TwoSmallestDistinctResult doParallel(int[] array) {
      TwoSmallestDistinctTask task =
         new TwoSmallestDistinctTask(array, 0, array.length);

      return fjPool.invoke(task);
   }
}

class TwoSmallestDistinctResult {
   //Three possible pairings for minOne and minTwo:
   //i.  minOne == minTwo == -1, or
   //ii.  minOne > 0 and minTwo == -1, or
   //iii.  0 < minOne < minTwo (note that they are distinct)
   int minOne, minTwo;

   TwoSmallestDistinctResult() {
      minOne = -1; minTwo = -1;
   }
}

class TwoSmallestDistinctTask extends RecursiveTask<TwoSmallestDistinctResult> {
   int[] array;
   int low, high; // array indices

   TwoSmallestDistinctTask(...)  { // Constructor that stores these three fields }

   TwoSmallestDistinctResult compute() { // FOR YOU TO WRITE }

   private int[] FastFourSort(int a, int b, int c, int d)
   { // O(1) helper method that returns array of four values in ascending order}
}
```

This code takes an array of *positive* numbers and determines the two smallest *DISTINCT* values in the array. If the array contains all duplicates, then the second value is stored as -1.

Examples: if *arr* is {12, 37, 64, 29, 18, 27, 8, 17, 13}, then *minOne* = 8 and *minTwo* = 13
if *arr* is {12, 12, 12, 12, 12, ..., 12}, then *minOne* = 12 and *minTwo* = -1

You need to complete the `TwoSmallestDistinctTask` class using the Java ForkJoin library. We have provided the member declarations and the constructor, so you only need to implement `compute()`. Do not use a sequential cutoff; your code should be able to handle base cases: the sub-array contains either 1 or 2 elements. Your implementation should also minimize the number of threads produced and perform with O(n) work and O($\log n$) span. You may find the provided `FastFourSort` method helpful in simplifying your code. Be sure that your code ensures that `minOne` and `minTwo` are distinct.

Answer question 9 on back or on a separate sheet of paper $\Longrightarrow$

9. (continued)

Solution

```
TwoSmallestDistinctResult compute() {
   TwoSmallestDistinctResult ans = new TwoSmallestDistinctResult();
   if(high - low == 1) { // sub-array is only one element
      ans.minOne = array[low];
      return ans;
   }
   else if(high - low == 2) { // sub-array contains only two elements
      if(array[low] == array[high - 1])
      { ans.minOne = array[low]; ans.minTwo = -1; }
      else if(array[low] < array[high - 1])
      { ans.minOne = array[low]; ans.minTwo = array[high - 1]; }
      else
      { ans.minOne = array[high - 1]; ans.minTwo = array[low]; }
      return ans;
   }
   else {
      TwoSmallestDistinctTask left
        = new TwoSmallestDistinctTask(array, low, (low+high)/2);
      TwoSmallestDistinctTask right
        = new TwoSmallestDistinctTask(array, (low+high)/2, high);
      left.fork();
      TwoSmallestDistinctResult rightAns = right.compute();
      TwoSmallestDistinctResult leftAns = left.join()
      int[] mins = FastFourSort(left.minOne, left.minTwo, right.minOne, right.minTwo);
      // we need to find the smallest distinct values in the array mins
      if(mins[3] == -1) // all -1s in the array (should be impossible but just in case)
        return ans;
      ans.minOne = mins[3];
      int i = 2;
      while(mins[i] != -1 && i >= 0) {
        if(mins[i] < ans.minOne) {
          ans.minTwo = ans.minOne;
          ans.minOne = mins[i];
        }
        i = i - 1;
      }
      return ans;
   }
}
```