Computer Science 20

Second Research Project This assignment replaces the previous assignment. Your work is due **Friday, April 22** by 5 o'clock.

Choose one of the questions below about the performance of a divide-and-conquer algorithm. Implement the algorithm and run experiments to find out the answer. Write a 2-5 page paper describing your experiments and your results.

Strassen's Algorithm To answer one of the first two questions, you will have to implement a version of Strassen's algorithm that works on general matrix sizes $n \times m$, not just $n = 2^k$. Here are some ideas about how to modify it:

- Pad out A and B by adding zeros to the bottom rows and right-most columns, to get square matrices of size $n = 2^k$. Then run Strassen's algorithm on the larger arrays; then pull out the answer C from the large result matrix.
- For square matrices: At each recursive stage: if n is even, split the arrays into quarters in the usual way and recur. If n is odd, make it even by adding an extra row and column of zeros. Then split into same-sized quarters and recur. Strip off the row/column from the result C before returning.
- It is also possible to strip off the odd row and column, and just multiply them separately after recurring on the smaller matrices. This is faster, but you have to figure out how matrix multiplication works in this situation.
- For odd-sized matrices $n \times m$, you can add an extra row and/or column separately depending on dimensions n and m.

Here are the questions.

1. Theory tells us that there must be a smallest n_0 such that Strassen's algorithm costs less than the classic 3-loops algorithm. Where is that cutoff if you define cost to be the number of scalar arithmetic operations on matrix elements? Where is that cutoff if you define cost to be CPU time?

   It is ok to restrict to square matrices for this question. Warning: depending on your implementation, you may not be able to find a cutoff. If so, say so. Whether you find it or not, you should compare the cost of these two strategies for values of n that a range of at least three powers of 2: $2^k, 2^{k+1}, 2^{k+2}$. These cost functions are by no means polynomials, and you shouldn't use regression analysis to summarize the results.

2. What is the penalty paid by Strassen's algorithm for coping with odd-sized matrices? For example, if you use the "pad with zeros" strategy, then the cost will be a step function that steps up at powers of 2. How far does this function get from a straight line between the power-of-2 costs, for general n, m? If you use one of the other coping strategies you will get a weird sort-of-stepping function – again, how far does it get from the baseline cost as the matrix shapes vary from the ideal? I suggest keeping n constant and then varying m to adjust the aspect ratio of the two matrices. Try this for more than one n, though.

Closest pair of points. I have created a **Point** class that you can download from `www.cs.amherst.edu/~cs20/Point.java`. This class provides x,y coordinates for 2-d points. Feel free to use it, modify it, or write your own class.

1. Fix the incorrect algorithm in the book, so that the divide step works even when there are duplicate x coordinates. If you use a simple fix, it is no longer $O(n \log n)$ worst case. Find out the new worst case cost, using integer coordinates so you can control the number of duplicates in the set (for example if there are $n = 1000$ points you can use integers in the range 1..10, 1..100, 1..1000000, etc. Find out new cost as a function of n and of the number of duplicates.

2. Fix the algorithm so it really does have $O(n \log n)$ worst case, even if there are duplicate x coordinates. I suggest using L. McGeoch's Mergesort-style modification that takes the points sorted by X coordinates as a parameter. It returns the points sorted by Y coordinates, then applies a "merge" operation to build Y and S. Since the function must also return a distance (the smallest distance it finds), you will have to figure out how to return both the distance and the array (the distance could be in array index 0, for example) Then run experiments to find out how much you "pay" for this worst cost bound: what is the difference in costs of the divide step and the marry step, of the book-version and your correct version?

3. The whole point of pre-sorting X and Y, is to keep an $O(n \log n)$ worst case bound. But what if you implemente a randomized algorithm with an $O(n \log n)$ average case bound, that, like randomized quicksort, is robust with respect to worst-case inputs?

   That is, do not pre-sort the point array P to build X and Y. Instead, take P as a parameter, and partition it like quicksort: select a point at random, and

partition P around the x coordinate of that point. Run tests to compare this average-case strategy to the worst-case strategy – what is the cost difference?