CS311H: Problem Set 9

RULES: You must indicate who you worked with (at most two others). If you worked by yourself please indicate this. You MAY NOT USE THE INTERNET OR ANY OTHER SOURCE TO LOOK UP SOLUTIONS (one small exception noted below). CHEATING WILL NOT BE TOLERATED. You may use anything we discussed in class/modules and any statement *proved* in the book.

1. This problem has three parts (10 points each). You will write several algorithms that take as input a *strictly sorted* $n \times n$ matrix of integers and a specific integer x, and return `true` if x is in the matrix, and `false` otherwise. Matrix M is *strictly sorted* iff $\forall i, j, p, q \in \{1, \ldots, n\}$ where $i \leq j$ and $p \leq q$, but not $(i = j \wedge p = q)$, $M[i, p] < M[j, q]$ is true. In other words, all rows and columns are increasing.

   For each algorithm: (1) write pseudo-code, (2) show that the algorithm has the specified big-O time complexity, and (3) formally prove the correctness of the algorithm.

   For each algorithm, you are encouraged to work on the pseudo-code part of the assignment collaboratively on Piazza. You can actually post your pseudo-code and talk about the ideas behind it with other students in the class. However, we don't want one person doing all the work for everyone. No single person should post pseudo-code for more than one algorithm, but everyone should feel free to comment on whatever anyone else posts.

   (a) `quadSearch(M,n,x)` is a divide-and-conquer algorithm that looks for x by checking the center element of the $n \times n$ matrix M and recursively searching whatever quadrants of M are necessary (a quadrant is a roughly $n/2 \times n/2$ submatrix in one of the corners of the current matrix). For ease of notation, let $M[1 \ldots n/2, 1 \ldots n/2]$ be the quadrant starting in the lowest-indexed corner of the matrix and going towards the center (similarly, the other quadrants are $M[n/2 \ldots n, 1 \ldots n/2], M[1 \ldots n/2, n/2 \ldots n], M[n/2 \ldots n, n/2 \ldots n]$). Also, feel free (throughout this assignment) to ignore the issue of whether or not there are an even or odd number of rows/columns, and how this affects the sizes of the submatrices. For example, the center element of the matrix is simply $M[n/2, n/2]$. You can also ignore out-of-bounds calls to matrices, by assuming that if you search a submatrix $M[x...i, x...i]$ and i happens to be out of bounds, you can simply assume your code automatically searches $M[x...(i-1), x...(i-1)]$ instead (rather than crashing the way it would in a real program). This algorithm should run in super-linear time (worse than $O(n)$). (Hint: For the proof of correctness, first prove a lemma or two about which quadrants need to be searched on each recursive call.)

   (b) `itrSearch(M,n,x)` is NOT a divide-and-conquer algorithm. This algorithm takes a strictly sorted $n \times n$ matrix M and searches for x using a loop or loops. This algorithm must run in linear ($O(n)$) time, so a simple double-loop traversal of the entire matrix will

not work (that would be $O(n^2)$). You still need to provide an analysis of the runtime, even though it will not be possible to use the Master Theorem. (Hint: consider a zig-zag path through the matrix).

(c) `binDiagSearch(M,n,m,x)` is a divide-and-conquer algorithm that looks for x in a strictly sorted $n \times m$ matrix M using the standard (almost) binary search algorithm on the diagonal of the matrix, and then recursively checking some sub-portion(s) (which?) of the matrix to look for x. Even though the algorithm deals with rectangular matrices, the *initial* matrix will always be $n \times n$, and the overall efficiency of this algorithm should be $O(n)$. Here is the variant of plain binary search that you can call in your code:

```
// pre: A is a strictly sorted array of integers, x is an integer,
//      imin and imax are the lowest and highest indices (inclusive)
//      within which to search for x.
binSearch(A,x,imin,imax){
  if (x > imax)       return (imax+1);
  if (imin == imax)   return imin;
  else if (imin + 1 == imax) {
    if (x <= A[imin]) return imin;
    else              return imax;
  } else {
    mid = imin + floor((imax - imin)/2);
    if (A[mid] == x)     return mid;
    else if (A[mid] < x) return binSearch(A,x,mid+1,imax);
    else                 return binSearch(A,x,imin,mid-1);
  }
}
```

This algorithm runs in $O(\log n)$ time ($n = imax - imin + 1$, which should be the length of A whenver this code is called from your code). When using the Master Theorem, remember we only care about big-O, so you can replace a $\log n$ with something bigger, but be warned that replacing it with $O(n)$ is already too big. Notice that this algorithm always returns an index within the array, even if x is not found. In particular, you are allowed to assume the following fact about this algorithm:

Lemma: For strictly sorted integer array A of length n (indexed from 1 to n), and some integer x, the index i returned by `binSearch(A,x,1,n)` will be an index of A such that $A[i-1] < x \leq A[i]$ (if we define $A[0] = -\infty$ and define $A[n+1] = \infty$).

As a notational convenience, the call `binSearch([M[1,1], ... , M[n,n]],x,1,n)` would call the binary search algorithm with an array filled with all elements in the diagonal of a square matrix. However, since only the initial matrix is guaranteed to be square, the above call won't really work ... you need to tweak it. When measuring the size of the sub-problems you are recursively solving, account for the worst case in overall size.