CSI3105 Fall 2009
Assignment 2 Solutions

1.
Input size: n, the number of keys in S.
Basic operation counted: Comparisons of x with keys in S.
Average case analysis:
Let the 9 input classes I_i for i = 1, 2, ..., 9 be as described in the question. Since all the input classes are equally likely, we have that $p(I_i) = 1/9$ for i = 1, 2, ..., 9. By tracing the algorithm we can also obtain the values for $t(I_i)$, as shown in the chart below:

i	1	2	3	4	5	6	7	8	9
$t(I_i)$	3	1	3	5	4	4	6	4	6

Thus we have the following average case analysis:

$$A(n) = \sum(p(I_i)t(I_i): i = 1, 2, ..., 9)$$
$$= 1/9\ (3 + 1 + 3 + 5 + 4 + 4 + 6 + 4 + 6)$$
$$= 1/9*36$$
$$= 4.$$

2. We will use the log relation $\lg n = (\lg e)* (\ln n)$ ....(*)
and $\log ab = \log a + \log b$ (**)
and L'Hôpital's Rule ....(LH)

a) Let g(n) = nlgn, let f(n) = knlg(kn), k>0. (NOTE: k>=0 in the assignment was a typo, as mentioned in class)

$$\lim_{n\to\infty} \frac{g(n)}{f(n)}$$

$$= \lim_{n\to\infty} \frac{n\lg n}{kn\lg kn}$$

$$= \lim_{n\to\infty} \frac{\lg n}{k\lg kn} \quad \text{.................simplifying}$$

$$= \lim_{n\to\infty} \frac{(\lg e)(\ln n)}{(k)(\lg k + \lg n)} \quad \text{...........using (**)}$$

$$= \lim_{n\to\infty} \frac{(\lg e)(\ln n)}{(k\lg k) + k\lg e(\ln n)} \quad \text{...........using (*)}$$

$$= \lim_{n\to\infty} \frac{(\lg e)/n}{(k\lg e)/n} \quad \text{...........using (LH)}$$

$$= \lim_{n\to\infty} \frac{1}{k} . \quad \text{..........simplifying}$$

$= 1/k$. Since $1/k$ is a constant which is >0, we have $g(n)$ is $\Theta(f(n))$ as required.

b) Let $g(n) = 6n^2 + 20n$, let $f(n) = n + 5$.

$$\lim_{n\to\infty} \frac{g(n)}{f(n)}$$

$$= \lim_{n\to\infty} \frac{6n^2 + 20n}{n + 5}$$

$$= \lim_{n\to\infty} \frac{12n + 20}{1} \quad \text{.................using (LH)}$$

$= \infty$. Thus $g(n)$ is $\Omega(f(n))$ as required.

3.a) Worst case analysis (assuming n is a power of 2, i.e. $n = 2^k$):

Input size: n, the number of integers in X
Basic operations counted: Additions involving X and using the max function
Input of size n which gives worst case: All cases the same

Analysis:
If $n > 1$: In this case we have two for loops, each with two basic operations per iteration, and each for loop executes $n/2$ times, for a total of $2n$ basic operations. We also have two other basic operations to count (at the line Maxcrossing ..., and the last line return...) which occur outside the for loops, plus the basic operations that occur inside the two recursive calls to the algorithm MaxSum. Each recursive call to MaxSum involves looking at a range of $n/2$ numbers in X, and so each requires $W(n/2)$ basic operations. So in total we have:

$$W(n) = 2n + 2 + 2W(n/2).$$

If $n = 1$: In this case (which is the base case) we have just one max operation, and so $W(1) = 1$.

So the recurrence relation for $W(n)$ is:

$$W(n) = 2W(n/2) + 2n + 2 \quad \text{for } n > 1$$
$$W(1) = 1.$$

(Note: The algorithm never reaches the case $n = 0$ recursively, and 0 is not a power of 2, hence we do not consider $n=0$)

Solving the recurrence relation using back substitution (recall $n = 2^k$):

$W(2^k) = 2W(2^{k-1}) + 2*2^k + 2$
$= 2(2W(2^{k-2}) + 2*2^{k-1} + 2) + 2*2^k + 2$
(since $W(2^{k-1}) = 2W(2^{k-2}) + 2*2^{k-1} + 2$
by the recurrence relation)

$= 2^2*W(2^{k-2}) + 2*2^{k+1} + 2^2 + 2$ (simplifying and regrouping)
.
. (note that you could do one more substitution here, if you weren't sure of the pattern yet)
.
$= 2^i *W(2^{k-i}) + i*2^{k+1} + 2^i + 2^{i-1} + ... + 2$
.
.
.

$= 2^k *W(2^{k-k}) + k*2^{k+1} + 2^k + 2^{k-1} + ... + 2$ (when i=k)

$= 2^k + k*2^{k+1} + \Sigma(2^j$: for j = 1 to k) (using W(1) = 1)

$= 2^k + k*2^{k+1} +((2^{k+1} - 1) – 2^0)$ (using the formula for the summation, Appendix A. Note that we must subtract 2^0 here, since the formula starts with j=0)
$= n + (lgn)*2n + 2n -1 -1$ (using $n=2^k$, k = lgn)

$= 2nlgn + 3n -2$ (simplifying)

We can (and should) confirm our result for the solution of the recurrence relation using induction.

Claim: W(n) = 2nlgn + 3n -2 for n>=1, n a power of 2.
Proof: (By induction)
Base case: If n = 1, then by the recurrence relation, W(1) = 1, and by our formula, W(1) = 2lg1 + 3 - 2 = 1. So the claim is true for n = 1.

Induction Hypothesis: Assume the claim is true for all 1<= m < n, m a power of 2, i.e.
W(m) = 2mlgm + 3m - 2.

Now consider any n>1, n a power of 2. We have

$W(n) = 2W(n/2) + 2n + 2$ (by the recurrence relation)

$= 2(2(n/2)\lg(n/2) + 3(n/2) - 2) + 2n + 2$ (here we are using the fact that n/2 is a power of 2 and <n and >=1, and thus we can use the induction hypothesis and substitute for W(n/2))

$= 2n\lg(n/2) + 3n - 4 + 2n + 2$ (simplifying)

$= 2n(\lg n - \lg 2) + 5n - 2$ (simplifying, and using $\lg(a/b) = \lg a - \lg b$)

$= 2n\lg n + 3n - 2$ (simplifying)

as required.

b) Since $W(n) = 2n\lg n + 3n - 2$, this algorithm is $\Theta(n\lg n)$ when n is a power of 2.

4 a)

Usual algorithm (improved):

Number of multiplications = $n^3 = 16^3$

Number of additions = $n^3 - n^2 = 16^3 - 16^2$

Total number of operations = $2*16^3 - 16^2 = 7{,}936$

Strassen's Algorithm:

Number of multiplications = $7^{\lg n} = 7^{\lg 16} = 7^4$

Number of additions/subtractions = $6*7^{\lg n} - 6n^2 = 6(7^{\lg 16}) - 6(16^2)$

Total number of operations = $7(7^4) - 6(16^2) = 15{,}271$

So, when counting the operations listed above, the usual algorithm would be faster for n=16.

b) From class, the recurrence relation for Strassen's algorithm counting multiplications with a threshold of 4 will be:

$W(n) = 7W(n/2)$ for $n>4$, n a power of 2

$W(4) = 4^3 = 64$

$W(2) = 2^3 = 8$

$W(1) = 1$

(Note: For n<= 4 we will use the usual algorithm to multiply the matrices, which requires n^3 multiplications).

5.

Input size: n, the number of keys in the list. We are assuming $n = 2^k$ for some $k >= 0$.

Operation being counted: Comparison of keys.

Let K(n) represent the number of comparisons for the reverse sorted ordered list.

(Note: For this analysis, I am assuming that the numbers are distinct.)

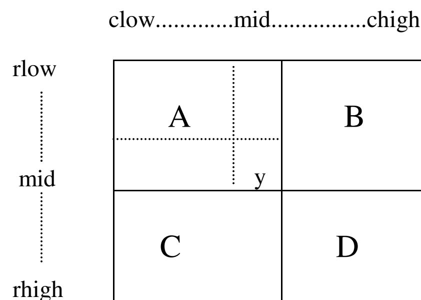
Since the list is in reverse sorted order, as every Merge step, we will be merging two lists where one list has values that are all smaller than the other. Given 2 sorted lists of length n/2 where one list has values that are all smaller than those in the second list, the Merge algorithm will only do n/2 comparisons, rather than n-1 comparisons as in the worst-case. Thus from the Mergesort algorithm we have

K(n) = 0 for n=1 (from the base case).
For n>1,
K(n) =(the work to sort the left half of the list) + (the work to sort the right half of the list) + (the work to merge the two 1/2 lists)
=W(n/2) + W(n/2) + n/2
= 2W(n/2) + n/2.

6. a) There are many different solutions for this problem. Below I give two possible divide and conquer solutions. The first uses ideas very similar to those used in binary search. The second is a more efficient, slicker algorithm that runs in linear time.

<u>Algorithm 1: A binary search type of algorithm</u>

<u>Idea of algorithm</u>:
We have a row range rlow....rhigh, and a column range clow....chigh. To begin with this range will be 1...n for the rows, and 1...m for the columns. We look at the number in the table that is in the middle of both ranges, call it y. (See figure below—note that each of the quadrants A, B, C and D would have size n/2 by n/2 for an n by n table, in the case where n is a power of 2).



If x=y, we are done. If x>y, then we know that x will not lie anywhere in quadrant A in the table (since the entries in the rows and columns are sorted). So we search each of quadrants B, D and C recursively for x. If x<y, then we know that x cannot lie in quandrant D, and we search quadrants A, B and C recursively. Finally, if the row range or the column range becomes empty (i.e. we have rlow>rhigh or clow>chigh) then we know x in not in the table.

Algorithm (in pseudocode):

Input: an n by m array *table*, sorted as described in the question, and a number x.

Output: a 2 element array *location*, where *location*[1] is the row location of x in table, and *location*[2] is the column location of x in *table*. If x is not in the table, these are 0.

```
void findx1(index rlow, index rhigh, index clow, index chigh, index location[])

{
  index rmid, cmid;

  if ((rlow>rhigh) || (clow>chigh)) {
       location[1]=0;
       location[2]=0;}
  else {
    rmid = ⌊(rlow + rhigh)/2⌋
    cmid = ⌊(clow + chigh)/2⌋
    if (x==table[rmid][cmid]){
         location[1] = rmid;
         location[2] = cmid; }
    else {
       if (x > table[rmid][cmid]){
          findx1(rmid+1, rhigh, clow, cmid, location);    \\look in quadrant C
           if (location[1] = = 0) \\not found in C
              findx1(rmid+1, rhigh, cmid+1, chigh, location);  \\look in quadrant D
              if (location[1] = = 0)  \\ not found in C or D
                  findx1(rlow, rmid, cmid+1, chigh, location); }    \\look in quadrant B

       else }
          findx1(rmid+1, rhigh, clow, cmid, location);    \\look in quadrant C
           if (location[1] = = 0) \\ not found in C
              findx1(rlow, rmid, clow, cmid, location);  \\look in quadrant A
              if (location[1] = = 0) \\ not found in C or A
                  findx1(rlow, rmid, cmid+1, chigh, location); }    \\look in quadrant B

     }
}
```

Worst-case Analysis (we assume table is n by n, where n is a power of 2, i.e. $n = 2^k$, $k>=0$):

Input size: n, the number of rows and columns in table being searched

Basic operation counted: Comparison of x with keys

Input that gives worst case: x not in table, and bigger than all the numbers in table

Analysis:

Each of the quadrants has size n/2 by n/2, we search 3 quadrants recursively and do 2 comparisons outside of the recursive calls, so we have

$W(n) = 3W(n/2) + 2$, $n>=2$

$W(1) = 2$ (since when n is 1 and x is bigger than the single element in the table, then there are 2 comparisons, followed by 3 recursive calls where rlow>rhigh, each of which thus require 0 comparisons with x).

Back substitution:

$W(2^k) = 3W(2^{k-1}) + 2$

$= 3(3W(2^{k-2}) + 2) + 2$ (back substitute)

$=3^2W(2^{k-2}) +3*2 + 2$ (simplify)

⋮

(ith line)

$=3^iW(2^{k-i}) + 2(3^{i-1} + 3^{i-2} + ... + 1)$

⋮

(line k)

$=3^kW(2^{k-k}) + 2(3^{k-1} + 3^{k-2} + ... + 1)$

$=(3^k)*2 + 2(3^{k-1} + 3^{k-2} + ... + 1)$ (using $W(1) = 2$)

$=2(\Sigma (3^j: j=0 \text{ to } k))$ (simplify)

$= 2 ((3^{k+1} -1)/(3-1))$ (from Appendix A)

$= 3^{k+1} -1$ (simplify)

$=3* 3^k - 1$ (simplify)

$= 3*3^{\lg n} - 1$ (simplify)

$= 3*n^{\lg 3} - 1$ (using $a^{\lg b} = b^{\lg a}$)

$\approx 3*n^{1.585} - 1$

So this algorithm is $\approx\Theta(n^{1.585})$.

Algorithm 2: A linear-time algorithm.

Idea of algorithm: Suppose we have an array that is n by m.

We look at entry y = table[n][1]. If x=y, we are done. If x <y, then we can conclude that x is not in row n. So we now search the array consisting of the rows 1 to n-1, and columns 1 to m (i.e. the table with the nth row removed—so we have eliminated one row). If x > y, then we can conclude that x is not in the first column. So we now search the array consisting of the rows 1 to n, and the columns 2 to m (i.e. the table with the 1st column removed, so we have eliminated one column). So in general, given the search space in in the row range 1 to lastrow, and the column range firstcol to m, we look at the bottom left entry in this search space, i.e. table[lastrow][firstcol] and decide to either eliminate row rowlast or column firstcol from the space. Once either the number of rows

or the number of columns of the table area we are searching is 0, we stop and conclude that x is not in the array.

Pseudocode for algorithm

Input: an n by m array *table*, sorted as described in the question, and a number x.

Output: a 2 element array *location*, where *location*[1] is the row location of x in table, and *location*[2] is the column location of x in *table*. If x is not in the table, these are 0.

```
void findx2(index lastrow, index firstcol, index location[])
{
if (lastrow==0) || (firstcol==m+1) {
    location[1]=0;
    location[2]=0; }
else
   if (x==table[lastrow][firstcol]){
       location[1]=lastrow;
       location[2]=lastcol;}
  else {
       if (x < table[lastrow][firstcol])
              lastrow=lastrow – 1;
        else
              firstcol=firstcol+1;
        findx2(lastrow, firstcol, location)  }
}
```

Worst-case analysis (Actually, for this worst case, I don't need to assume that n is a power of 2, so here I show the analysis for a general n by m table).

Input size: n+m, where n and m are the number of rows and columns in the table.
Basic operation counted: comparison of x with entries in table
Input that gives worst case: At each stage of the recursion, we either eliminate a row or a column from our search space, until either the number of rows left or the number of columns left becomes 0. Thus we get the most stages in the recursion if, at the very last stage, both the number of rows and the number of columns is 1 (so when we reach the base case, the number of rows and columns left are 1 and 0 or 0 and 1). An example of an input which creates this situation is one in which all of the entries in the table are 1, except for the nth row, where we have 2's in position 1 to m-1, and the mth column is all 4's, and the value of x is 3.

Analysis:
For our worst case input, we have
$W(n+m) = W(n+m-1) + 2$ for $n+m>=2$,
since at each stage of the recursion we decrease the total number of rows and columns by 1, and we do 2 comparisons outside of the recursive call.

For the base case, $W(1) = 0$ (i.e. when we have the number of columns is 1 and rows is 0).

Back substitution:

$W(n+m) = W(m+n-1) + 2$
$= W(m+n-2) + 2*2$ (back substitute)
$= W(m+n-3) + 2*3$ (back substitute)
⋮

(line i)
$= W(m+n-i) + 2*i$
⋮

(line m+n-1)
$=W(1) + 2*(m+n-1)$
$=2*(m+n-1)$. (since $W(1) = 0$)
So this algorithm is $\Theta(n+m)$.