Algorithms Qual Exam (Fall 2016)
(This is a closed-book exam)

Question 1

In an algorithm course, we prove a lower bound of the worst case running time for comparison-based sorting. Specifically, a lower bound for the worst case number of *comparisons* is derived.

In this question, you are asked to apply the same proof technique to derive a lower bound for the worst case number of *comparisons* for merging of two sorted sequences of n elements each.

(a) (10 points) Based on the usual merge algorithm that is given in textbooks, draw a decision tree for merging two sorted sequences of 2 elements each. (*Your design of the decision tree should be clear. Specifically, it should be clear what the nodes, edges and leaves in the tree represent.*)

(b) (10 points) Consider the merging of two sorted sequences of 3 elements each. You are asked to enumerate the leaves of a decision tree for merging the two sequences. (*Note: You are not required to draw the decision tree. Hint: there are 20 leaves.*)

(c) (15 points) Develop a lower bound on the worst case number of comparisons for merging two sorted sequences of n elements each. (*In your analysis, you cannot assume a particular algorithm of merging two sorted sequences as the result should apply to all comparison-based merging algorithms. Your development of the lower bound should mimic that for the lower bound of sorting given in textbooks. Specifically, you are not supposed to give a trivial constant time lower bound. You can make use of the Stirling's approximation $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ to simplify your answer. Also, your answer should be as simple and as accurate as possible; that is, do not suppress the multiplicative constant behind the big-Omega notation.*)

Answers:

(a)

```
                          a1 < b1
                         /        \
                       /           \
                      /             \
                     /               \
                    /                 \
                 no/                   \yes
                  /                     \
                 /                       \
                /                         \
               /                           \
           a1 < b2                       a2 < b1
            /  \                          /     \
         no/    \yes                   no/       \yes
          /      \                      /         \
b1,b2,a1,a2    a2 < b2               a2 < b2    a1,a2,b1,b2
                /   \                  /    \
             no/     \yes           no/      \yes
              /       \              /        \
    b1,a1,b2,a2   b1,a1,a2,b2  a1,b1,b2,a2  a1,b1,a2,b2
```

(b) The 20 leaves are:

```
   a1, a2, a3, b1, b2, b3            b1, b2, b3, a1, a2, a3
   a1, a2, b1, a3, b2, b3            b1, b2, a1, b3, a2, a3
   a1, a2, b1, b2, a3, b3            b1, b2, a1, a2, b3, a3
   a1, a2, b1, b2, b3, a3            b1, b2, a1, a2, a3, b3
   a1, b1, a2, a3, b2, b3            b1, a1, b2, b3, a2, a3
   a1, b1, a2, b2, a3, b3            b1, a1, b2, a2, b3, a3
   a1, b1, a2, b2, b3, a3            b1, a1, b2, a2, a3, b3
   a1, b1, b2, a2, a3, b3            b1, a1, a2, b2, b3, a3
   a1, b1, b2, a2, b3, a3            b1, a1, a2, b2, a3, b3
   a1, b1, b2, b3, a2, a3            b1, a1, a2, a3, b2, b3
```

(c) The number of leaves is $\binom{2n}{n}$. *The depth of the tree, which is the worst case number of comparisons, is* $\geq \log_2 \binom{2n}{n} = \log_2 \frac{(2n)!}{n! \cdot n!} \approx \log_2 \frac{\sqrt{2\pi\, 2n}\,(2n)^{2n}/e^{2n}}{(\sqrt{2\pi n}\,(n)^n/e^n)^2} = \log_2 \frac{\sqrt{4\pi n}\,(2n)^{2n}/e^{2n}}{2\pi n\ \ (n)^{2n}/e^{2n}} = \log_2 \frac{1}{\sqrt{\pi\, n}}\ 2^{2n} = 2n - 0.5 \log_2 n - 0.5 \log_2 \pi$.

Question 2 (30 points)

Matrix multiplication is associative. To compute a matrix to the power of k, we can use the repeated squaring technique when k is a power of 2. For example, A^{128} can be computed in 7 matrix multiplications as follows:

```
t = A
t = t * t      // t = A^2
t = t * t      // t = A^4
t = t * t      // t = A^8
t = t * t      // t = A^16
t = t * t      // t = A^32
t = t * t      // t = A^64
t = t * t      // t = A^128
```

(a) (15 points) Extend the above idea to give a *fast recursive* algorithm for computing A^k for a given matrix A and a positive integer k where k may not necessarily be a power of 2.

(b) (15 points) Analyze using recurrence the running time of your algorithm in terms of the number of matrix multiplications performed. You should analyze both the lower bound and upper bound on the running time of your algorithm.

Answers:

(a)

```
matrix square( matrix A ) { return A * A; }

matrix power( matrix A, int k ) {
   if (k == 1) return A;
   if (k % 2 == 0)
      return square( power( A, k/2 ) );
   else
      return A * square( power( A, (k-1)/2 ) );
}
```

(b) Let $t(k)$ be the number of matrix multiplications performed by the algorithm. Then $t(k) = t(k/2)+1$ when k is even, and $t(k) = t(\lfloor k/2 \rfloor)+2$ when k is odd. The best scenario is captued by $t(k) = t(k/2)+1 = \Omega(\log_2 k)$, which is the lower bound of the running time. The upper bound is given by $t(k) \le t(\lfloor k/2 \rfloor)+2 = O(\log_2 k)$. Therefore, running time is $\Theta(\log_2 k)$.

Question 3

Let $\binom{n}{k}$ (read "n choose k") denote the number of k-combinations that can be chosen from an n-element set. It is known that $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.

(a) (10 points) Write a recursive algorithm that computes $\binom{n}{k}$ where $n \geq 1$ and $0 \leq k \leq n$.

(b) (5 points) The algorithm given in part (a) is inefficient. The inefficiency can be removed by employing dynamic programming technique. Explain the inefficiency by computing $\binom{6}{3}$ using the algorithm from part (a).

(c) (10 points) Give the codes for the algorithm using dynamic programming.

(d) (10 points) Analyze the running time of your codes in part (c).

Answers:

(a)

```
int comb( int n, int k ) { // assume 0 <= k <= n
   if (k == 0 || k == n) return 1;
   return comb( n-1, k ) + comb( n-1, k-1 );
}
```

(b) To compute $C(6,3)$, we compute $C(5,3)$ and $C(5,2)$ once each. In turn, we need to compute $C(4,3)$ once, $C(4,2)$ twice and $C(4,1)$ once. Next, we need to compute $C(3,3)$ once, $C(3,2)$ three times, $C(3,1)$ three times and $C(3,0)$ once. That is, many computations of the same parameters are repeated numerous times.

(c) Below is a memoized version of the dynamic programming solution. A non-memoized verison is also acceptable.

```
int comb( int n, int k, c[][] ) { // assume 0 <= k <= n
   if (c[n][k] != 0) return c[n][k];
   if (k == 0 || k == n) return c[n][k] = 1;
   return c[n][k] = comb( n-1, k, c ) + comb( n-1, k-1, c );
}

int combinations( int n, int k ) {
    int c[n][k] = {0};   // initialize all array entries to 0
    return comb( n, k, c );
}
```

(d) Initialization takes $O(nk)$ time. Each subproblem instance `comb(i,j)`*, where $1 \leq i \leq n$ and $0 \leq j \leq k$, takes $O(1)$ time to compute. The total time is $O(nk)$.*