ECE G205 Fundamentals of Computer Engineering
Fall 2003

Exercises in Preparation to the Midterm

The following problems can be solved by either providing the pseudo-codes of the required algorithms or the C++ code for the corresponding functions.

- **Problem # 1**. Write two functions *summa* and *summaRec*, one iterative and one recursive, that given as input an array of *N* integer numbers returns the sum of the "even elements" of the array (i.e., the elements whose position is even, such as the 0th element of the array, the 2nd, the 4th and so on).

  Possible solutions are given by the following function:

```
int summa( int E[], int N ) {
  int sum = E[ 0 ];
  for( int i = 2; i < N; i += 2 )
    sum += E[ i ];
  return sum;
}

int summaRec( int E[], int i ) {
  if ( i == 0 )
    return E[ i ];
  else
    return E[ i ] + summaRec( E, i - 2 );
}
```

  In the case of the recursive version the call from `main` should be either `summaRec( A, n - 1 )` (*n* odd) or `summaRec( A, n - 2 )` (*n* even), where A is the given array and *n* its size.

- **Problem # 2**. Write an iterative and a recursive function for computing the *n*th element F_n of the *Fibonacci* sequence. The Fibonacci sequence is a sequence of integers so defined: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$. Discuss the worst case time complexity of the two solutions.

  Possible solution are given by the following functions:

```
int fibo( int N ) {
  if ( N == 0 ) || ( N == 1 )
    return N;
  else
    return fibo( N - 1 ) + fibo( N - 2 );
}

int fibIter( int N ) {
  if ( N == 0 ) || ( N == 1 )
    return N;
  int fN2 = 0, fN1 = 1, fN;
  while ( N > 1 ) {
    fN = fN1 + fN2;
    fN2 = fN1;
    fN1 = fN;
    N--;
  }
  return fN;
}
```

  The time complexity $T(n)$ of the recursive function is expressed by the following recursive equation:

$$T(n) = T(n-1) + T(n-2).$$

  The solution to this equation is quite involved. Let us try to guess it. Since $T(n)$ nearly doubles, one can think that the solution could be 2^n. However, this is not the case, since $2^n \neq 2^{n-1} + 2^{n-2}$. Let us try with a generic a^n. To determine if such an a exists, we notice that the solution must work for every n, i.e., also for $n = 2$. So, such an a should be such that $a^2 = a + 1$. This equation has the two (famous!) solutions $a_1 = \frac{1+\sqrt(5)}{2}$ (the *Golden Ratio*) and $a_2 = \frac{1-\sqrt(5)}{2}$. Hence, the algorithm is exponential!

  The time complexity of the iterative function depends on the while loop. Since N is decremented by one at every iteration of the loop, after at most $N - 1$ iterations the loop is exited. Therefore, the time complexity of the function is $O(N)$.

- **Problem # 3**. Prove that the following recursive algorithm for the multiplication of natural numbers is correct:

```
MULTIPLY(y,z)
  if z = 0
    then return 0
    else if z is odd
           then return MULTIPLY( 2y, int(z/2) ) + y
           else return MULTIPLY( 2y, int(z/2) )
```

  The correctness proof is by induction on z. We claim that `MULTIPLY(y,z)` return the product yz. The claim holds true when $z = 0$ since `MULTIPLY(y,0)` returns 0 independently of y. Now, suppose that for $z \geq 0$ `MULTIPLY(y,z)` returns yz. We must prove that `MULTIPLY(y,z+1)` returns $y(z+1)$. There are two cases to be considered, depending on whether $z+1$ is odd or even.

  By inspection, if $z+1$ is odd, then `MULTIPLY(y,z+1)` returns `MULTIPLY(2y,int((z+1)/2))+y` $=$ $2y\lfloor\frac{(z+1)}{2}\rfloor + y$ (by the induction hypothesis) $= 2y\frac{z}{2} + y$ (since z is even) $= y(z+1)$.

  By inspection, if $z+1$ is even, then `MULTIPLY(y,z+1)` returns `MULTIPLY(2y,int((z+1)/2))` $=$ $2y\lfloor\frac{(z+1)}{2}\rfloor$ (by the induction hypothesis) $= 2y\frac{(z+1)}{2}$ (since z is odd) $= y(z+1)$.

- **Problem # 4**. Consider the following, fundamental question: "Given a number $n > 1$, is n prime, i.e., is n divisible only by 1 and itself?"

  The question is answered by the following function (where the C++ operator `%` is the modulus operator, which returns the remainder of the integer division between its operands):

```
bool prime( int n ) {
  int j = 2;
  while ( j < n )
    if ( n % j == 0 )
      return false;
    else
      j++;
  return true;
}
```

  Function `prime` attempts to divide n by every number j in the range $2,\ldots,n-1$ and returns *true* only if no number j that divides n has been found.

  Is this function time complexity polynomial in the *size of the input*? Justify your answer.

  The body of the while loop is executed $n-2$ times, and therefore the time complexity of the function is $O(n)$. However, this complexity is *not* polynomial in the size of the input. Each integer n is represented in the computer "concisely" as a sequence of $d = \log n + 1$ bits. Hence the size of the input (also called the *dimension of the problem*) is d. The complexity of the function `prime` is therefore $O(n) = O(2^d)$, i.e., *exponential* in the size of the input.

  The problem of determining whether a number is prime or *composite* has been interesting human beings for millennia. The best minds of each generations, from Eratosthenes (born circa 275 BC) and up, have been trying to develop efficient (i.e., non-exponential) algorithms for this problems. The *first* algorithm that produces (deterministically) an answer in polynomial time has been introduced on August 6 2002.

- **Problem # 5**. Write a recursive function that given as input an array A of n integers, returns the maximum element. Prove its correctness and determine its worst-case time complexity.

  A possible solution is given by the following function:

```
int maxR( int j, int A[] ) {
  if ( j == 0 )
    return A[ 0 ];
  else
    return max( A[ j ], maxR( j - 1, A ) );
}
```

  The correctness can be proved by induction on the number of recursive calls. When j equals 0 the function terminates and return $A[0]$. Assume that for a generic $j > 0$ the function `maxR( j - 1, A )` terminates and correctly returns the maximum in $A[0 \dots j-1]$. Then, by inspection, since the library function `max` terminates and correctly returns the maximum between two numbers, the generic recursive call `maxR( j, A )` also terminates and return the maximum in $A[0 \dots j]$. When $j = n-1$ (initial call) the function correctly terminates returning the maximum element of the array.

  The complexity of the function is expressed by the following recurrence relation:

$$T(n) = \begin{cases} c & \text{if } n = 0 \\ \\ T(n-1) + c & \text{if } n > 0. \end{cases}$$

  which can be solved by substitution to yield $T(n) \in O(n)$.

- **Problem # 6**. Consider the following 2×2 matrix:

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

(a) Write a function to *efficiently* compute the nth power of A, namely:

$$A^n = \underbrace{A \cdot A \cdots A}_{n \text{ times}}.$$

One solution is obtained by multiplying the matrix A by itself for $n-1$ times, yielding A^n. The corresponding code is the following (n is required as input).

```
int main() {
  int n;
  cout << "Insert a number n (> 0): ";
  cin >> n;
  cout << endl;
  int A[ 2 ][ 2 ] = { { 0, 1 }, { 1, 1 } };
  int B[ 2 ][ 2 ], C[ 2 ][2 ];
  copyMat( B, A );
  for( int a = 0; a < n - 1; a++ ) {
    mulMat( A, B, C );
    copyMat( A, C );
  }
  return 0;
}
// Multiply X by Y and return the result in Z
void mulMat( int X[][ 2 ], int Y[][ 2 ], int Z[][ 2 ] ) {
  Z[ 0 ][ 0 ] = X[ 0 ][ 0 ] * Y[ 0 ][ 0 ] + X[ 0 ][ 1 ] * Y[ 1 ][ 0 ];
  Z[ 0 ][ 1 ] = X[ 0 ][ 0 ] * Y[ 0 ][ 1 ] + X[ 0 ][ 1 ] * Y[ 1 ][ 1 ];
  Z[ 1 ][ 0 ] = X[ 1 ][ 0 ] * Y[ 0 ][ 0 ] + X[ 1 ][ 1 ] * Y[ 1 ][ 0 ];
  Z[ 1 ][ 1 ] = X[ 1 ][ 0 ] * Y[ 0 ][ 1 ] + X[ 1 ][ 1 ] * Y[ 1 ][ 1 ];
}
// Copy Y onto X
void copyMat( int X[][ 2 ], int Y[][ 2 ] ) {
  X[ 0 ][ 0 ] = Y[ 0 ][ 0 ];
  X[ 0 ][ 1 ] = Y[ 0 ][ 1 ];
  X[ 1 ][ 0 ] = Y[ 1 ][ 0 ];
  X[ 1 ][ 1 ] = Y[ 1 ][ 1 ];
}
```

The following is a more efficient way to compute A^n. It is based on the fact that since we want just A^n we can avoid to compute many of the intermediate A^is, $2 \leq i \leq n$. This is possible by repeatedly squaring the matrix A till we get to A^n.

$$\begin{array}{rcl} A \cdot A & = & A^2 \\ A^2 \cdot A^2 & = & A^4 \\ A^4 \cdot A^4 & = & A^8 \\ & \cdots & \\ A^{2^{\lceil(\log n)-1\rceil}} \cdot A^{2^{\lceil(\log n)-1\rceil}} & = & A^{2^{\lceil \log n \rceil}} \end{array}$$

The following code works for $n = 2^h$ (the code must be adjusted when n is not a power of two).

```
// Compute the logarithm base 2
double log2( int x ) {
  return ( log( x )/ log( 2 ) );
}
// MAIN
int main() {
  // Defs as above
  int k = (int)( log2( n ) );
  for( int b = 0; b < k; b++ ) {
    squareMat( A, B );
    copyMat( A, B );
  }
  ...
}
// Multiply X by itself and return the result in Y
void squareMat( int X[][ 2 ], int Y[][ 2 ] ) {
  Y[ 0 ][ 0 ] = X[ 0 ][ 0 ] * X[ 0 ][ 0 ] + X[ 0 ][ 1 ] * X[ 1 ][ 0 ];
  Y[ 0 ][ 1 ] = X[ 0 ][ 0 ] * X[ 0 ][ 1 ] + X[ 0 ][ 1 ] * X[ 1 ][ 1 ];
  Y[ 1 ][ 0 ] = X[ 1 ][ 0 ] * X[ 0 ][ 0 ] + X[ 1 ][ 1 ] * X[ 1 ][ 0 ];
  Y[ 1 ][ 1 ] = X[ 1 ][ 0 ] * X[ 0 ][ 1 ] + X[ 1 ][ 1 ] * X[ 1 ][ 1 ];
}
```

(b) Determine the time complexity of your solution.

Since functions `copyMat`, `mulMat` and `squareMat` are executed in constant time, the first method for computing A^n requires $O(n)$ time, while the second method produces A^n in $O(\log n)$ time.

(c) How can a solution to point **(a)** be used to return the nth number of the Fibonacci sequence.

It can be proved (by mathematical induction) that, when $n > 0$,

$$A^n = \begin{pmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{pmatrix}$$

Hence, to return (in logarithmic time!) the nth Fibonacci number, it is enough to compute A^n and then return $a^n_{1,0}$.

- **Problem # 7**. The recursive definition of the binomial coefficient when $k \leq n$

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n. \end{cases}$$

write a function that returns $\binom{n}{k}$ in a way that is different from the more "natural" recursive implementation. The binomial coefficient $\binom{n}{k}$ can be seen as the last entry $c_{n,k}$ of a $(n+1) \times (k+1)$ matrix which can be filled out line by line in the following way: $c_{0,j} = 0$, $1 \leq j \leq k$, $c_{i,0} = 1$, $0 \leq i \leq n$, and

$$c_{i,j} = c_{i-1,j-1} + c_{i-1,j}, 1 \leq i \leq n, 1 \leq j \leq k.$$

Use this characterization (also known as *Pascal's triangle*) to write a function that taken as input n and k returns $\binom{n}{k}$. Discuss the time *and* the *space* complexity of your solution. Can this problem be solved in linear space?

The following function works in O(nk) (which given that $k \leq n$ is $O(n^2)$).

```
int bcI( int n, int k ) {
  int T[ k + 1 ];
  T[ 0 ] = 1;
  for( int j = 1; j < k + 1; j++ )
    T[ j ] = 0;
  for( int i = 0; i < n; i++ )
    for( int j = k; j > 0; j-- )
      T[ j ] += T[ j - 1 ];
  return T[ k ];
}
```

The only data structure used (over an over again) is the unidimensional array T with $k+1 \leq n+1$ elements. Hence the space is linear in n.

- **Problem # 8**. Write an *optimal* Boolean C++ function that, given an array of *n* integers ≥ 0, returns *true* is there are two elements in the array whose sum equals 23, *false* otherwise.

  A possible solution is given by the following function:

```
bool twentyThree( int A[], int n ) {
  bool S[ 24 ];
  for( int a = 0; a < 24; a++ )
    S[ a ] = false;
  for( int b = 0; b < n; b++ )
    if ( A[ b ] < 24 )
      S[ A[ b ] ] = true;
  for( int c = 0; c < 12; c++ )
    if ( S[ c ] && S[ 23 - c ] )
      return true;
  return false;
}
```