CS 667 : Homework 5(Due: Nov 28, 2005)

Problem 1. (50 points)

We want to compute the product $m = m_1 m_2 \ldots m_k$. Show that m can be computed in $O((\lg m)^2)$ bit operations.

Problem 2. (50 points)

(a) Compute $n!$ (n factorial) in $O((n \lg n)^2)$ bit operations.

(b) Given integers a and b how long (in bit, NOT word operations) does it take to efficiently compute integer $n = a^b$, i.e. raise a to the power of b? Express the number of bit operations required in asymptotic notation as a function of n in an efficient solution to this problem. **Note**. An answer of the form: "*At most* $2 \lg n - 1$ *multiplications*" is not suitable as an answer to this problem.

Problem 3. (50 points)

Show that the GCD computation requires $O(\lg a \lg b)$ bit operations thus improving the rough $O(\lg^3 a)$ bound given in class, i.e. complete the proof of Corollary 5 of the notes (Subject 12, page 10).

Problem 4. (50 points)

You are given an a-bit positive integer A and an b-bit positive integer B. You may assume $A \geq B$. How fast can you compute $A + B$, $A - B$, $A \cdot B$, and find Q, R such that $A = BQ + R$, $0 \leq R < B$? Express your answer in terms of a and b and justify it by pointing to the appropriate reference or give the algorithm. You can cite results of the notes/textbook covered in class.

Problem 5. (50 points)

Fibonacci Revisited. Show that F_n can be computed in $O(n^2)$ bit operations even if n-bit multiplication can only be done in $\Theta(n^2)$ bit operations.

Problem P1. (100 points)

Implement the algorithm for the perfect power problem (see Solutions of Problem 1 of HW 1) in C or C++ or Java.However A can be arbitrarily long (eg. 1024 or 8192 bits). You are allowed to use libraries for arbitrary precision arithmetic as long as (a) they are for free (i.e. I can use them and install them on a linux or AFS machine to test your code without having to obtain root privileges) and (b) they are easily installable (i.e. even I can install them easily). Alternatively, you can implement your own functions for auxiliary operations (eg. arbitrarily long multiplication, exponentiation, etc).

I expect as an answer a .tar file sent by email. The .tar will be untarred on a linux workstation and compiled through gcc/g++ or Java. I expect a make install command to compile everything and create an executable file named powertest. powertest will accept as input a file containing n in decimal notation.

Thus if file myfile contains a base-10 integer such as

```
17487686712733928413644063750880864826316326531082890839798047131393
06920507121153268532108269241880671097659463948848837902966613039193
61801624726151915125942668649508993665419986623407409256320591797254
65901781768127877188903984695217669171609482765052925389918678373962
03339268758977282743385306120289869213112851446870454351518286400633
04862801177174173384780775389699560332291136389670468588940721006781
36076427022136733286307364152390825026213339153406940132529505642288
77265570344122667987101745048846965145 6
```

then the following command should return

```
% powertest myfile
  x= 123456 y=101
  It's a power!
```

within a reasonable amount of time (eg. under 60 seconds for integers as long as 8192 bits, i.e with up to 2000-3000 digits). `myfile` is a string corresponding to an arbitrarily-named file (in this instance `myfile`).

Note that even if the library you are using has a built-in function to test whether an integer is the integer power of an integer, you are not allowed to use it. You need to build YOUR OWN IMPLEMENTATION from scratch. If you don't you should not expect any credit.

Problem P2. (80 points)

Write a simple (recursive or non-recursive) FFT program that works for n a power of two with the following definition `comp` of a complex number

```
typedef struct {
double i,r;
} comp;
```

For example the high level function call may look like

```
 my_fft(comp *input, comp *output, int n);
```

if `output` has been preallocated or

```
 my_fft(comp *input, comp **output, int n);
```

I should be able to test your implementation through an input file.

```
./myfft input-file result-file
or
java myfft input-file result-file
```

The syntax of the input file `input-file` is straighforward.

```
8
1.0 0.0
2.0 0.0
3.0 0.0
4.0 0.0
5.0 0.0
6.0 0.0
7.0 0.0
8.0 0.0
```

The first line contains an integer the size of the vector to follow. You may size that vector size is always a power of two. Each line contains one element of the vector where the real-part of a complex number appears before the imaginary part. Therefore the 5-th element of the vector contains real number $5.0 + i0 = 5.0$. All input should be store in double precision. The output file that will be generated in `result-file` should also have the same format.