CS 667 : Homework 1(Due: Sep 23, 2009)

Problem 1. (50 points)

The Fibonacci sequence is given by the following recurrence $F_{n+1} = F_n + F_{n-1}$ for $n \geq 1$ and $F_0 = F_1 = 1$.

(a) Show how to compute F_n in $O(n)$ time.

(b) Given an $n \times n$ matrix A show how you can find A^n in $O(n^3 \lg n)$ time.

(c) Can you improve the obvious time bound in (a)? In particular prove that F_n can be computed in $O(\lg n)$ time. Hint: You may need to use the result of part (b), i.e. formulate the F_n as a matrix problem. The discussion on page 902 and 903 (Problem section at the end of the Chapter on Number-Theoretic Algorithms may offer you some insight).

Problem 2. (50 points)

We are interested in determining whether n is a perfect power i.e. whether there exist integer x, y such that $n = x^y$ and if such x, y exist also determining the values of x, y. We will use in the remainder the word-model for operation counting.

(a) If n is indeed a perfect power, how large can y be? Express your answer as a function of n only.

(b) Suppose you have a black box `YRoot(N,Y)` that given N and Y it finds the integer Y-th root of N (i.e. $N^{1/Y}$) if such exists or returns `-1` if such root does not exist. For example `YRoot(8,3)` would return 2 since $8^{1/3} = 2$, whereas `YRoot(8,2)` would return -1. Suppose that the time it takes for `YRoot` to return an answer is $O(1)$. How could you use `YRoot` to determine whether n is a perfect power or not? How many times do you need to call `YRoot` in an efficient determination of whether n is a perfect power or not?

(c) `YRoot` in $O(1)$ time is rather unrealistic. How fast could you implement it? Does a solution that is $O(\log^3 n)$ (or $o(\log^3 n)$) exist? Explain. Can you do it in $o(\log^2 n)$ time?

Problem 3. (50 points)

Suppose that we try to insert n keys into a hash table of size m using open addressing and uniform hashing. Let $p(n, m)$ be the probability that no collisions occur. Show that $p(n, m) \leq exp(-n(n-1)/(2m))$. Argue that when n exceeds $\sqrt{m}$, the probability of avoiding collisions goes rapidly to zero.

Hint: Use induction... Also use $exp(x) \geq 1 + x$ for any real x. Note that $exp(x) = e^x$. If you argue probabilistically (too much) you might get confused.

Problem 4. (50 points)

(a) You are given six polynomials $f_1, \ldots f_6$ of degrees $1, 2, 3, 1, 4, 5$ respectively. We are interested in finding the product $f = f_1 f_2 f_3 f_4 f_5 f_6$. Assume that the cost of multiplying two polynomials of degree a and b is $a \cdot b$. Find a schedule for multiplying the six polynomials that is of the lowest possible total cost.

(b) You are given six polynomials $f_1, \ldots f_6$ of degrees $1, 2, 3, 1, 4, 5$ respectively. We are interested in finding the product $f = f_1 f_2 f_3 f_4 f_5 f_6$. Assume that the cost of multiplying two polynomials of degree a and b is $a + b$ (note the difference from the previous problem) i.e. it is proportional to the space required to store the product which is a polynomial of degree $a + b$.

Find a schedule for multiplying the six polynomials that is of the lowest possible total cost for this non-traditional definition of a cost function.

Example. If you have three polynomials g_1, g_2, g_3 of degrees $1, 2, 3$ respectively and you first compute $g_2 g_3$ and then multiply the result by g_1, the cost of the first multiplication is 5 $(= 2 + 3)$ and the cost of the second multiplication is 6 since you multiply the result, a degree 5 polynomial to a degree one polynomial. Total cost is $5 + 6 = 11$. Is this the best you can do for these three polynomials?

Problem 5. (50 points)

n choose k. $\binom{n}{k}$ is defined as

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

where $k!$ is defined as $k! = k(k-1)(k-2)\ldots 1$.

Find $\binom{n}{k}$ in $O(n^2)$ additions/subtractions. Multiplications and divisions are not allowed.

If you are not very familiar with the properties of $\binom{n}{k}$, a good reading is section C.1 of the textbook, including the exercises. A way to derive the answer is hidden somewhere there.

Problem 6. (100 POINTS)

You can replace 100 points worth of Problems 1-5 (i.e. two problems) with this Problem.

Implement hashing by open-addressing consistent with the operations Insert/Delete/Search as defined in the textbook (Chapter 11) or the associated notes and possibly enhanced as follows. A hash table will consist of two entities: one the table T itself as used in open-addressing. The table itself won't store key values but indexes to a dictionary of words separated by null values (\0). Call the latter dictionary D. Insertion operations affect both T and D. Deletion operations won't delete strings from D, only indexes from T. Search operations will return not only the location in T but also the corresponding location in D (location of first character of word). Implement (an approximate interface is provided) the following functions. You may deviate from the interface depending on the use of C, C++, Java. However the testing interface must be maintained intact.

```
1.   int HashFunction(string k, probe i) // Hash function takes key k as input returns 0..m-1

   /* For open addressing  implement
    * h(string,i) --->   (h(key(string)) % m + i**2 + i ) % m
    *  where k=key(string) is a numerical conversion of the string.
    *  choice of key is up to you. You may use for example key=MD5
    * and isolate a subset of the 128 bits of it if necessary.
    * Deviation from standard: Location h(string,i) stores not the key
    * but a pointer/reference/index into an array D of characters.
    * Strings there are sequences of characters separated by \0
    * Thus the array of length 20 stores three strings alex, algorithm,data
    * 0        9         19
      alex0algorithm0data0
    */
```

and

```
   /* In the remainder T refers to both T and its associated dictionary D */
2.   HashCreate(table T, int m); // Create a hash table with m slots /Initialize
3.   HashEmpty (table T); //Check if Table is empty
4.   HashFull  (table T); // or full; this is different from overflow.
5.   HashInsert(table T, string k); //Insert string to T (and D)
6.   HashDelete(table T, string k); //Delete string from T (but not necessarily from D)
7.   HashSearch(table T, string k, int m); //Search for string in T (and D)

8.   HashTable(table T, operation o, string k); //Generic Interface for operations
                                                 // o id 10,11,12,13
9.   HashPrint(table T);                         //Print D not T
10.  ProcessHash(file file-name)
```

The end result is the implementation of HashTable, a function that can implement an open-addressing scheme with quadratic probing as defined above. An operation can be defined in a single line with two arguments. The first being the operation (10 for Insertion, 11 for Deletion, 12 for search, 13 for HashPrint) and the second the key value involved (except for HashPrint) given in the form of a string with no more than 20 characters (if this is important to your implementation). If the operation is 13 (HashPrint) no key value needs to be present and the dictionary D is printed in the following compact form

```
0  alex
5  algorithm
15 data
```

I will test your code, through the command line, by typing in `% ./ProcessHash my-test-file` . A test file might look like the following one. Your program should support such an interface. The semantics of the other functions is thus not important; the semantics of ProcessHash however needs to be maintained intact.

```
10 alex
10 algorithm
10 data
12 alex
11 algorithm
10 structures
12 algorithm
12 structures
13
12 algorithm
```

An operation unable to perform correctly should return a `-1`, which is equivalent to string not found. An operation that successfully terminates should return the position of the hash table relevant to the operation and the position in D containing the relevant word.

Problem 7. (100 POINTS)

You can replace 100 points worth of Problems 1-5 (i.e. two problems) with this Problem.

Huffman coding. Says all. Arguments ($N \leq 10$) in the command-line won't exceed 10 in the file option case, there is only one argument in the directory argument case. File inputs can be arbitrary files (it will mostly be test on binary files such as .class Java files or executable C/C++ files).

```
% ./huffman-encode file1 file2 ... fileN
% ./huffman-encode dir1
% ./huffman-decode file1 file2 ... fileN
% ./huffman-decode dir1
 // Encode : converts file1 ---> file1.huf
 // Decode : From command-line file1 reads (if it exists) file1.huf and converts it into file1
 // file1 ... fileN may have suffixes: eg myfile.pdf ---> myfile.pdf.huf --> myfile.pdf

 Example
% ./huffman-encode myfile.tex
  // encode myfile.tex into myfile.tex.huf using Huffman coding. myfile.tex still exists.
  // If a previous myfile.tex.huf exists, it will be overwritten. No warning required.
% ./huffman-decode myfile.tex
  // It locates a myfile.tex.huf and decodes it by overwriting a myfile.tex if it still
  // exists. myfile.tex.huf ceases to exist.
  // If a previous myfile.tex exists, it will be  overwritten. No warning required.
```