CSCI E-124-Spring, 2004 Homework 1

Russell Lowke, February 21st 2004

1. Suppose you are given a six-sided die, that might be biased. Explain how to use die rolls to generate unbiased coin flips

To get unbiased coin flips from a biased die, we can reduce the die rolls to coin flips by mapping 1, 2, 3 to "heads" and 4, 5, 6 to "tails". Let's assign the probabilities p1, p2,...., p6 to the die rolling a 1, 2,..., 6 respectively. Then the probability that this die will produce a "heads" (p) will be p1 + p2 + p3 and the probability of it producing a "tails" (q) will be p4 + p5 + p6. So now that we have a valid mapping and a p and q, we can proceed with any of the algorithms that were given in class for coin flips, say the Advanced Multi-Level strategy. This has already been proven to produce, on average, one bit per flip. This will provide for us, on average, one bit per roll.

Now suppose you want to generate unbiased die rolls (from a six-sided die) given a potentially biased die.

The equivalent of the Von Neumann method when expanded to a six sided die is to simply define a round as six rolls and keep all permutations of the sequence of [1, 2 , 3 , 4, 5, 6], because all such permutations (excluding double and triples, etc.) have the same probability of occurring. This yields 6! (720) results that can be kept, all occurring with probability p1*p2*...*p6 where pn is the probability of the number n being rolled.

Using the formula given in class, with th being the expected number of total rolls, e being the probability of generating a bit each round, and f being the number of rolls per round:

t = f / e

Since for us f = 6 and e = 6! (p1*p2*...*p6) we have

f = (6) / (6! (p1*p2*p3*p4*p5*p6)) =

<u>f = 1 / (5! (p1*p2*p3*p4*p5*p6))</u>

2. Implement the three different methods for computing the Fibonacci numbers (recursive, iterative, and matrix) discussed in lecture.

I have implemented the three versions using C++, the code for each which starts on page 4. The recursive method is `fibonacci_r.cc`, the iterative method is `fibonacci_i.cc`, and the matrix method is `fibonacci_m.cc`

Can you determine the first Fibonacci number where you reach integer overflow?

- My machine reaches integer overflow at 2^{32} (which is = 4294967296).

In the case of integer overflow at 2^{32} the largest fibonacci number that can be calculated is the 47th, which is 2971215073.

- Many machines reach integer overflow at 2^{16} (which is = 65536).

In the case of integer overflow at 2^{16} the largest fibonacci number that can be calculated is the 24th, which is 75025.

How fast does each method appear to be? Give precise timings if possible.

I have used calculation of the 24th fibonacci number as a benchmark. To amplify the result I have each program calculate this 10000 times, and time the duration . Note: the resolution of the `clock()` C++ method is 1/100th of a second.

The recursive `fibonacci_r.cc` program takes 73.62 seconds to calculate the 24th Fibonacci number 10000 times.
Recursive version, one progression to the 24th number takes ≈0.007362ths of a second.
The iterative `fibonacci_i.cc` program takes .35 seconds to calculate the 24th Fibonacci number 10000 times.
Iterative version, one progression to the 24th number takes ≈0.000035ths of a second.
The matrix `fibonacci_m.cc` program takes 7.77 seconds to calculate the 24th Fibonacci number 10000 times.
Matrices version, one progression to the 24th number takes ≈0.000779ths of a second.

When calculating *low* fibonacci numbers the iterative version is fastest and *appears* quickest.

What is the largest Fibonacci number you can compute in one minute of machine time?

In 1 minute, recursive `fibonacci_r.cc` program yields ≈ 41 fibonacci numbers.

In 1 minute, iterative `fibonacci_i.cc` program yields ≈ 9154 fibonacci numbers.

In 1 minute, matrix `fibonacci_m.cc` program yields ≈ 26525 fibonacci numbers.

The matrices version *easily* computes the most numbers in one minute, calculating 26525 fibonacci numbers.

```
//
//
//   fibonacci_r.cc
//
//        AUTHOR: Russell Lowke    Mon Feb 16th 2004
//
// Description: Calculates the Fibonacci sequence recursively
//
//

#include <iostream>
#include <ctime>

using namespace std;

enum {OVERFLOW_AT2_16 = 65536};
enum {OVERFLOW_AT2_32 = 4294967296};      // not used, but nice to know

//
//   recursive call that calculates the fibonacci sequence
//
// I have included  % OVERFLOW_AT2_16 in arithmetic, as specified,
// even though my machine does not overflow using ints at 2^16
//
unsigned int fib_r (int n) {

    if (n <= 0)                             // base case 0        = 0
            return 0;
      else if (n <= 2)                      // base case 1 or 2   = 1
            return 1;
      else
            return ( (fib_r(n - 1) + fib_r(n - 2)) % OVERFLOW_AT2_16 );
}

//
// Calculates the number of fibonacci numbers calculated in [seconds] seconds
//
int  nFibsInTime(int seconds) {

    int n = 0;
      clock_t endwait =   clock() + seconds * CLK_TCK;
    while (clock() < endwait) {
        ++n;
        fib_r(n);
    }

    return n;

}

//
// Calculates to the [fibNum] fibonacci number [nTimes] times
//   returning the duration taken
//
clock_t  timedFib(int fibNum, int nTimes) {
     clock_t startTime = clock();
```

```
    for (int i; i < nTimes; i++ )
         fib_r(fibNum);
     clock_t endTime = clock();

    return (endTime - startTime);
}

int main() {

     clock_t duration = timedFib(24, 10000);
     cout << "It took " << duration << " 1/" << CLK_TCK <<
        "ths of a second for fib_r to compute the 24th fibonacci number 10000
times" << endl;

     cout << endl;

    int n = nFibsInTime(60);
      cout << "After 1 minute fib_r computed " << n << " fibonacci numbers."
<< endl;

    return 0;
}
```

```
//
//
//   fibonacci_i.cc
//
//        AUTHOR: Russell Lowke    Mon Feb 16th 2004
//
// Description: Calculates the Fibonacci sequence using iteration
//

#include <iostream>
#include <ctime>
#include <vector>

using namespace std;

typedef vector<int>     IntList;

enum {OVERFLOW_AT2_16 = 65536};
enum {OVERFLOW_AT2_32 = 4294967296}; // not used, but nice to know

//
//   iterative call that calculates a fibonacci number
//
// I have included   % OVERFLOW_AT2_16 in arithmetic, as specified,
// even though my machine does not overflow using ints at 2^16
//
unsigned int fib_i (int n) {

     IntList f(n + 1);                 // +1 as [0] included

     f[0] = 0;
     f[1] = 1;

    for (int i = 2; i <= n; ++i)
         f[i] = ( f[i - 1]  + f[i - 2] ) % OVERFLOW_AT2_16;

     return f[n];                      // OVERFLOW_AT2_16 avoids overflow
}

//
// Calculates the number of fibonacci numbers calculated in [seconds] seconds
//
```

```
int  nFibsInTime(int  seconds)  {

    int n = 0;
    clock_t  endwait =   clock() + seconds * CLK_TCK;
    while  (clock() < endwait)  {
         ++n;
         fib_i(n);
    }

    return  n;
}

//
// Calculates  to  the  [fibNum]  fibonacci  number  [nTimes]  times
//   returning  the  duration  taken
//
clock_t  timedFib(int  fibNum,  int  nTimes)  {
     clock_t  startTime  =  clock();

    for  (int  i;  i  <  nTimes;  i++ )
         fib_i(fibNum);
     clock_t  endTime  =  clock();

    return  (endTime  -  startTime);
}

int  main()  {

     clock_t  duration  =  timedFib(24,  10000);
    cout  <<  "It  took  "  <<  duration  <<  "  1/"  <<  CLK_TCK  <<
        "ths  of  a  second  for  fib_i  to  compute  the  24th  fibonacci  number  10000
times"  <<  endl;

     cout  <<  endl;

    int  n  =  nFibsInTime(60);
    cout  <<  "After  1  minute  fib_i  computed  "  <<  n  <<  "  fibonacci  numbers."  <<
endl;

    return  0;
}
```

```
//
//
//   fibonacci_m.cc
//
//       AUTHOR: Russell Lowke    Mon Feb 21st 2004
//
// Description: Calculates the Fibonacci sequence using matrices
//

#include <iostream>
#include <ctime>
#include <vector>
#include <cmath>

using namespace std;

typedef vector<int>     IntList;
typedef vector<IntList> Matrix;
typedef vector<Matrix>  MatrixList;

enum {OVERFLOW_AT2_16 = 65536};
enum {OVERFLOW_AT2_32 = 4294967296}; // not used, but nice to know

//
//   utility to create a matrix of specified width and height from an array
//
Matrix buildMatrix(int width, int height, int data[]) {

     Matrix n;

    for (int x = 0; x < width; ++x) {
         IntList col;
        for (int y = 0; y < height; ++y)
              col.push_back( data[ width * y + x] );
         n.push_back( col );
    }
    return n;
}

//
// multiply matrices    m x n
//
```

```
Matrix muliplyMatrices (Matrix matrixM, Matrix matrixN) {

     Matrix n;

    // resultant matrix will have m's height and n's width
    int width   = matrixN.size();
    int height = matrixM[0].size();

    for (int x = 0; x < width; ++x) {

         IntList col;

        for (int y = 0; y < height; ++y ) {

            int val = 0;
            for (int i = 0; i < height; ++i )
                  val += (matrixM[i][y] % OVERFLOW_AT2_16 * matrixN[x][i] %
OVERFLOW_AT2_16  );

            // % OVERFLOW_AT2_16 avoids overflow
              col.push_back( val );
        }
         n.push_back( col );
    }
    return n;
}

//
//   matrix call that calculates a fibonacci number
//
int fib_m (int n) {

    if (n == 0) {
        return 0;
    } else {
        //  matrixA = [ 0 1 ]
        //            [ 1 1 ]
        //
        int dataA[] = {0, 1, 1, 1};
         Matrix matrixA = buildMatrix(2, 2, dataA );
         Matrix matrixB = matrixA;
```

```
        /*
         // matrixA^n       simple version for iterative matrix (not used)
        for (int i = 1; i < n; ++i )
              matrixA = muliplyMatrices(matrixA, matrixB);
        */

         MatrixList mList;
         matrixA = buildMatrix(2, 2, dataA );
         matrixB = matrixA;
         mList.push_back( matrixA );           //  mList[0]
        int place = 0;
        int indice = 1;                        // as in x^1

        // use repeated squaring
        do {
             indice *= 2;                      // double the indice
             ++place;
               matrixA = muliplyMatrices(matrixA, matrixB);
              mList.push_back( matrixA );
             matrixB = matrixA;
        } while (indice*2 < n);

        for (; place >= 0; place--) {

            // get the indice of this "place"      2^place
            int pwr = (int) pow(2.0, place);

            if (indice + pwr <= n) {          // include if < n
                    matrixA = muliplyMatrices(matrixA, mList.at(place));
                 indice += pwr;               // update indice
            }
        }

        //  matrixC = [ 0 ]
        //            [ 1 ]
        //
        int dataC[] = {0, 1};
         Matrix matrixC = buildMatrix(1, 2, dataC);

        return muliplyMatrices(matrixA, matrixC)[0][0];
    }
}
```

```
//
// Calculates the number of fibonacci numbers calculated in [seconds] seconds
//
int  nFibsInTime(int seconds) {

    int n = 0;
    clock_t endwait =   clock() + seconds * CLK_TCK;
    while (clock() < endwait) {
         ++n;
         fib_m(n);
    }

    return n;

}

//
// Calculates to the [fibNum] fibonacci number [nTimes] times
//   returning the duration taken
//
clock_t  timedFib(int fibNum, int nTimes) {
     clock_t startTime = clock();

    for (int i; i < nTimes; i++ )
         fib_m(fibNum);
     clock_t endTime = clock();

    return (endTime - startTime);
}

int main() {
     clock_t duration = timedFib(24, 10000);
     cout << "It took " << duration << " 1/" << CLK_TCK <<
        "ths of a second for fib_m to compute the 24th fibonacci number 10000
times" << endl;

     cout << endl;
    int n = nFibsInTime(60);
    cout << "After 1 minute fib_m computed " << n << " fibonacci numbers." <<
endl;
    return 0;
}
```

3. Indicate for each pair of expressions (A, B) in the table below the relationship between A and B.

Note: I couldn't get characters for Little Omega and Theta to display properly, so I have used "w" for Little Omega and "ø" for Theta.

		≤	≪	≥	≫	=
A	B	O	o	Ω	w	ø
$\log n$	$\log(n^2)$	Y	N	Y	N	Y
$\log(n!)$	$\log(n^n)$	Y	Y	N	N	N
$n^{1/3}$	$(\log n)^6$	Y	Y	N	N	N
$n^2 2^n$	3^n	Y	Y	N	N	N
$(n^2)!$	n^n	N	N	Y	Y	N
$n^2/\log n$	$n\log(n^2)$	N	N	Y	Y	N
$(\log n)^{\log n}$	$n/\log(n)$	N	N	Y	Y	N
$100n + \log n$	$(\log n)^3 + n$	Y	N	Y	N	Y

[Notes]

O f(n) is O(g(n)) if f(n)/g(n) tends to c

o f(n) is o(g(n)) if f(n)/g(n) tends to 0

Ω f(n) is Ω(g(n)) if g(n)/f(n) tends to c

w f(n) is w(g(n)) if g(n)/f(n) tends to 0

ø if O and Ω are True then True

4. For all of the problems below, when asked to give an example, you should give an example whose domain and range is the positive integers.

• Find (with proof) a function f_1 such that $f_1(2n)$ is $O(f_1(n))$

Let $f_1(n) = n$ to prove $f_1(2n) = O(f_1(n))$

$f_1(2n) \le c \cdot f_1(n)$ for all $n \ge N$

Let $c = 2$ then

For $n = 1$ $2 \cdot 1 \le 2 \cdot 1$

For n

$f_1(2n) \le 2(f_1(n))$

$2n \le 2(n)$

$2n \le 2n$

which is true for all $n \ge N$

$f_1(n2) \le cg(n)$ which is O

• Find (with proof) a function f_2 such that $f_2(2n)$ is not $O(f_2(n))$.

Let $f_2(n) = e^n$ then $f_2(2n) = e^{2n}$

Let $c = 1$

For $n = 1$ $e^2 > e$ so true for $n = 1$

For n $e^{2n} > e^n$ so

multiply each side by e^2

$e^{2n+2} > e^{n+2}$

$e^{2(n+1)} > e^{(n+2)}$

$e^{2(n+1)} > e \cdot e^{(n+1)} > e^{(n+1)}$

$e^{2(n+1)} > e^{(n+1)}$

Therefore by induction it is true for all n.

• Show that if f(n) is O(g(n)) and g(n) is O(h(n)), then f(n) is O(h(n)).

$f(n) \leq c \cdot g(n)$
$g(n) \leq c_2 \cdot h(n)$

if f(n) is O(g(n)) then
$f(n) \leq c \cdot g(n)$

if g(n) is O(h(n)) then
$g(n) \leq c_2 \cdot h(n)$

Therefore $f(n) \leq c \cdot c_2 \cdot h(n)$
Therefore $f(n) \leq c_3 \cdot h(n)$ where $c_3 = c \cdot c_2$

Therefore f(n) is O(h(n))

• Give a proof or a counterexample: if f is not O(g), then g is O(f)

If f is not O(g), then g is O(f)

If $f \neq\leq c_1 \cdot g$ then
$f > c_1 \cdot g$
$c_1 \cdot g < f$
$g < 1/c_1 \cdot f$
so g is O(f)

- **Give a proof or a counterexample: if f is o(g), then f is O(g)**

If	f is o(g), then f is O(g)		
lim n -> ∞	f/g is 0	definition of o	
Therefore	f(n)/g(n)	is < c	for all n > N
	f(n)	is < c . g(n)	for all n > N
By definition	f(n)	is O(g(n))	

5. Prove StoogeSort correctly sorts.

| A1 | A2 | A3 |

A2 ≥ A3 else a large number could be left behind

A2 ≥ A1 else a small number could be left behind

So, when defining our breaks, let the sizes be as follows:
A1 = n / 3,
A2 = n / 3 + n % 3
A3 = n / 3
and the above conditions will always be met.

Why ScroogeSort works

Because we are given that StoogeSort *correctly sorts* on every pass:
<u>After the 1st sort we are guaranteed that the n / 3 largest numbers are not in A1</u>
To see why this is true, consider the case when they start in A1 (when they start anywhere else, it's obvious that they won't somehow migrate *down* to A1 because, again, StoogeSort actually sorts correctly). If the n / 3 largest numbers start in A1, they are pulled into A2 after the first sort because, due to the way we defined the size of our partitions, A2 is always *at least as large as A1* and thus able to hold every one of the n / 3 values.

Next, we know that:
<u>After 2nd sort that the n / 3 largest numbers are in A3 in sorted order and the smallest numbers are are in A1 & A2</u>
Remember that the largest n / 3 numbers were NOT in A1 by the end of the first sort. What that means is they were either in A2 or A3. In either scenario, after the list A2A3 had been sorted, they will migrate to A3 and stay there, in sorted order (again, because we are given that StoogeSort sorts correctly on every pass). So we are guaranteed that, even if the largest numbers started in A1, they will have migrated to A3 by the end of this sort.
The smallest numbers, if they started in A1, are still in A1. If they were in A2, they also were pushed to A1 after the first sort. If they were in A3, they've now migrated to A2 because A2 is large enough to hold everything in A3 and, after that section of the list was sorted, would have the smallest numbers pushed into it.

Lastly, we know that:

<u>After the 3rd sort the rest of the list has been sorted.</u>

Before starting the third sort, we knew that the largest numbers are sitting (sorted) in A3. So that part of the list is done**.** We also know that the smallest numbers are in A1&A2. Therefore, when we sort again, this portion of the list is going to be organized in sorted order. The smallest numbers are in A1, the largest are in A3, and the middle values are sitting in A2 (why? If they smallest are in A1 and the largest are in A3 there's nowhere else for these middle values to be).

In terms of running time, StoogeSort has to partially sort the list *three* times. The length of the list that it is sorting is size n/3. So a recurrence for its running time would be

$T(n) = 3\ T\ (2n / 3)$

Subsititute $m = 2n$ giving $T(1/2m) = 3\ T\ (m / 3)$

and, using the master formula given in class,

Formula $T(n) = aT(n/b) + cn^k$ where $a = 3,\ b = 3,\ c = 1,\ k = 0$

$T(1/2m)$ is $O(1/2m^{\log_3 3})$	if $3 > 1$	TRUE
$T(1/2m)$ is $O(1 \log 1/2m)$	if $3 = 1$	FALSE
$T(1/2m)$ is $O(1)$	if $3 < 1$	FALSE

<u>Asymptotic runningtime for $T (n) = 3\ T\ (2n / 3)$ is $O(n)$</u>

6. Derivation of Expression for T(n) where T(1) = 1 and T(n) = T(n-1) + 3n -3

Note T(1) = 1, T(2) = 4, T(3) = 10.

We can derive:

$T(n) = \{3n-3\} + \{3(n-1)-3\} + \{3(n-2)-3\} + \{3(n-3)-3\} + \ldots(n \text{ terms}) + 1$

$= (\text{ n terms of } 3n) + (n \text{ terms of } -3 + -6 + -9 + -12 \ldots) + 1.$

$= 3n^2 + (n/2)[-6 + (n-1)(-3)] + 1$

$= 3n^2/2 - 3n/2 + 1$

In the third line, use is made of the formula for the sum of an arithmetic progression
$S = A + (A+D) + (A+2D) \ldots (n \text{ terms}) = (n/2)[2A + (n-1)D]$ where A = -3 and D = -3.

Note that using **$T(n) = 3n^2/2 - 3n/2 + 1$**, T(1) = 1, T(2) = 4 and T(3) = 10 as required.

Proof by induction:

For n=1, $T(n) = 3n^2/2 - 3n/2 + 1$ gives T(1) = 1 as required.

Assume $T(n) = 3n^2/2 - 3n/2 + 1$ for n.

Then from T(n) = T(n-1) + 3n -3,

$T(n+1) = T(n) + 3(n+1) - 3$ (substitute n+1 instead of n)

$= 3n^2/2 - 3n/2 + 1 + 3(n+1) - 3$

$= 3n^2/2 + 3n/2 + 1,$

$= 3(n+1)^2/2 - 3(n+1)/2 + 1$ as follows by removing the brackets.

The above expression is the same form as $T(n) = 3n^2/2 - 3n/2 + 1$, but with n+1 instead of n. So if $T(n) = 3n^2/2 - 3n/2 + 1$ for n it is true for n+1.

So, by induction, $T(n) = 3n^2/2 - 3n/2 + 1$ for all n.

• **T(1) = 1, T(n) = 2T(n - 1) + 2n -1**

If $T(n) = 2T(n - 1) + 2n - 1$, then
$T(n + 1) = 2T(n) + 2n + 1$

Base For n = 1, $T(1) = 3 \cdot 2^1 - 2(1) - 3 = 6 - 2 - 3 = 1$

Assume $T(n) = 3 \cdot 2^n - 2n - 3$ (from Mathematica)

We show $T(n + 1) = 2T(n) + 2(n + 1) - 1$ using substitution

$= 2(3 \cdot 2^n - 2n - 3) + 2(n + 1) - 1$

$= 2 \cdot 3 \cdot 2^n - 4n - 6 + 2n + 1$

$= 3 \cdot 2^{(n + 1)} - 4n - 5 + 2n$

$= 3 \cdot 2^{n+1} - 2n - 5$

$= 3 \cdot 2^{n+1} - 2(n + 1) - 3$

Therefore true for n + 1
Therefore by induction this form is true for all n.

7. Give asymptotic bounds for T(n) in each of the following recurrences.

Formula: $T(n) = aT(n/b) + cn^k$ $a \geq 1$ $b \geq 2$ are integers c, k are positive constants

$T(n)$ is $O(n^{\log_b a})$ if $a > b^k$

$T(n)$ is $O(n^k \log n)$ if $a = b^k$

$T(n)$ is $O(n^k)$ if $a < b^k$

• $T(n) = 4T(n/2) + n^3$

Using $T(n) = aT(n/b) + cn^k$ where $a = 4,\ b = 2,\ c = 1,\ k = 3$

$T(n)$ is $O(n^{\log_2 4})$	if $4 > 2^3$	FALSE
$T(n)$ is $O(n^3 \log n)$	if $4 = 2^3$	FALSE
$T(n)$ is $O(n^3)$	if $4 < 2^3$	TRUE

Asymptotic bound for $T(n) = 4T(n/2) + n^3$ is $O(n^3)$

• $T(n) = 17T(n/4) + n^2$

Using $T(n) = aT(n/b) + cn^k$ where $a = 17,\ b = 4,\ c = 1,\ k = 2$

$T(n)$ is $O(n^{\log_4 17})$	if $17 > 4^2$	TRUE
$T(n)$ is $O(n^2 \log n)$	if $17 = 4^2$	FALSE
$T(n)$ is $O(n^2)$	if $17 < 4^2$	FALSE

Asymptotic bound for $T(n) = 17T(n/4) + n^2$ is $O(n^{\log_4 17})$

which simplifies to approximately $\approx O(n^{2.05})$

- **$T(n) = 9T(n/3) + n^2$**

Using $T(n) = aT(n/b) + cn^k$ where $a = 9$ $b = 3$, $c = 1$, $k = 2$

$T(n)$ is $O(n^{\log_3 9})$	if $9 > 3^2$	TRUE
$T(n)$ is $O(n^2 \log n)$	if $9 = 3^2$	FALSE
$T(n)$ is $O(n^2)$	if $9 < 3^2$	FALSE

Asymptotic bound for $T(n) = 9T(n/3) + n^2$ is $O(n^2 \log n)$

- **$T(n) = T(\sqrt{n}) + 1$**

Let $m = \sqrt{n}$

$T(m^2) = T(m) + 1$

Using $T(m^2) = aT(m/b) + cm^k$ where $a = 1$ $b = 1$, $c = 1$, $k = 0$

$T(m^2)$ is $O(1^{\log m})$

$T(m^2)$ is $O(\log m)$

$T(n)$ is $O(\log n^{1/2})$ substitute back m

$T(n)$ is $O(1/2 \log n)$

$T(n)$ is $O(\log n)$

Asymptotic bound for $T(n) = T(\sqrt{n}) + 1$ is $O(\log n)$