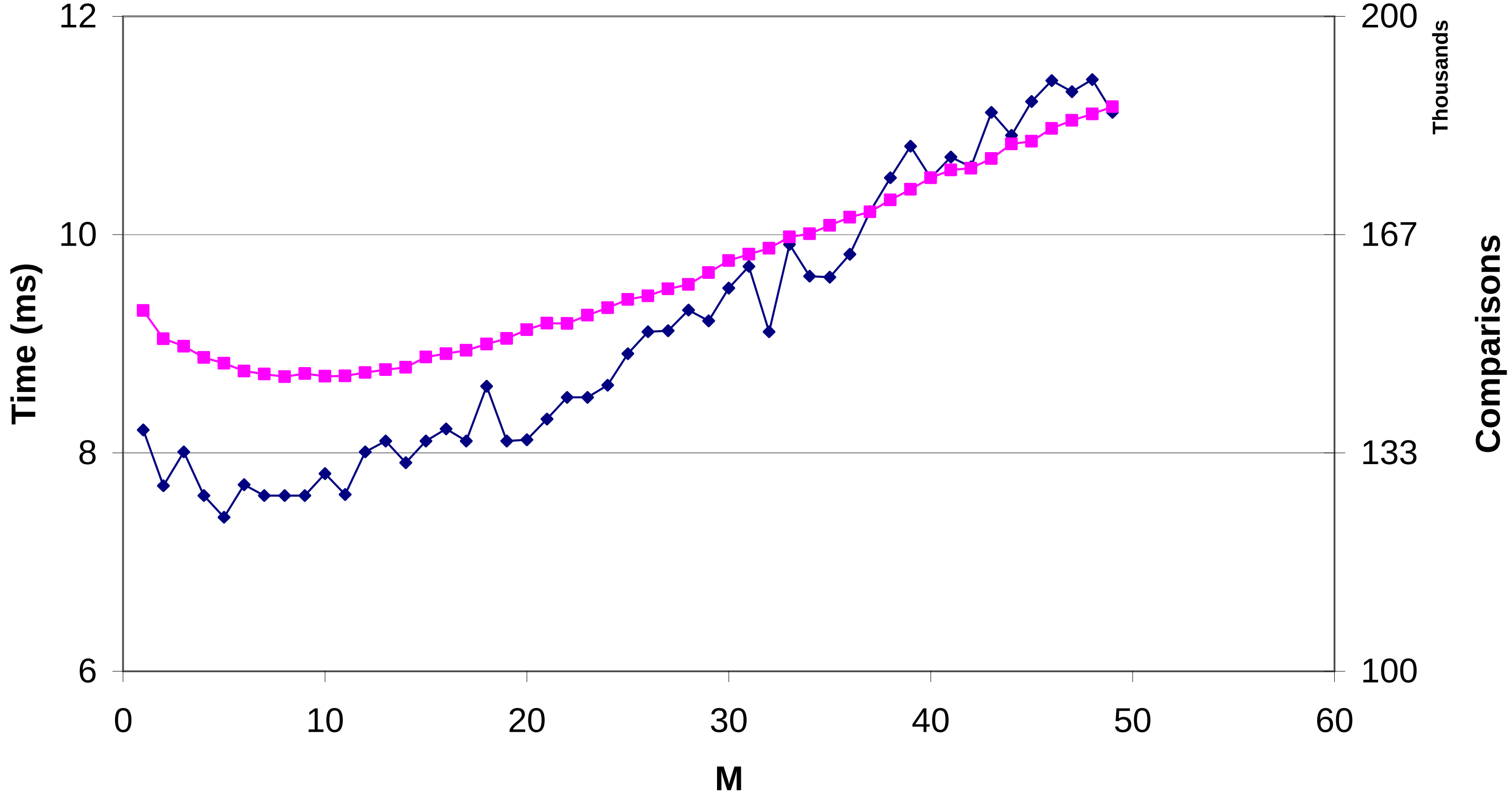
Random Number Generation

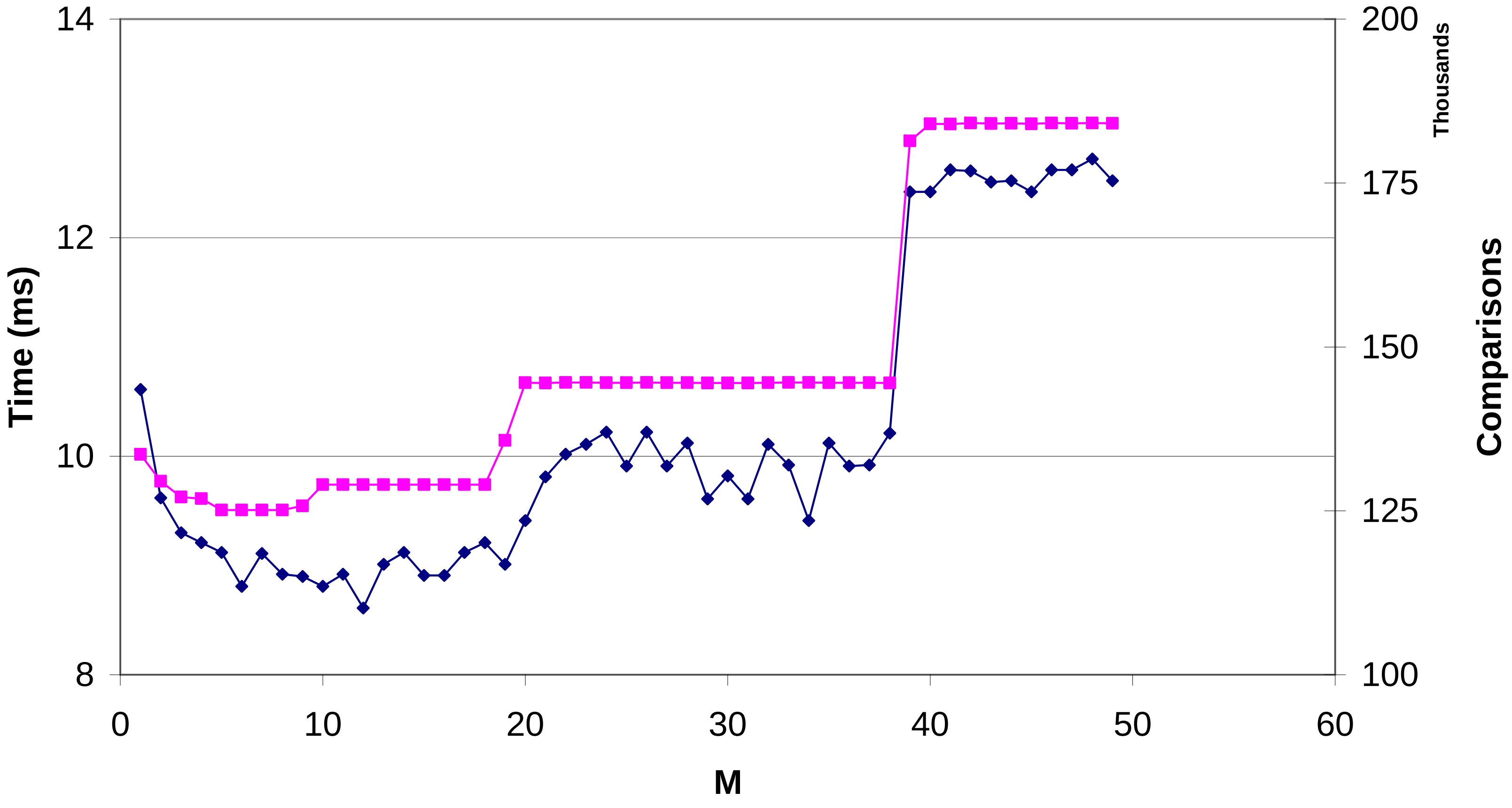
Biostatistics 615/815

Lecture 14

Homework 5, Question 1: Quick Sort Optimization …



Homework 5, Question 1: Merge-Sort Optimization



Homework 5, Question 2:

- Comparison of Hashing Strategies
  - Linear hashing
  - Double hashing

- Interesting aspects:
  - Memory dramatically impacts performance
  - In double-hashing, it is important to choose the second hash function carefully:
    - Specifically, it is key to avoid that it might return the values 0, 1 and any multiple of the table size M

Today

- Random Number Generators
  - Key ingredient of statistical computing

- Discuss properties and defects of alternative generators

Some Uses of Random Numbers

- Simulating data
  - Evaluate statistical procedures
  - Evaluate study designs
  - Evaluate program implementations

- Controlling stochastic processes
  - Markov-Chain Monte-Carlo methods

- Selecting questions for exams

Random Numbers and Computers

- Most modern computers do not generate truly random sequences

- Instead, they can be programmed to produce *pseudo-random* sequences
    - These will behave the same as random sequences for a wide-variety of applications

Uniform Deviates

- Fall within specific interval (usually 0..1)
- Potential outcomes have equal probability

- Usually, one or more of these deviates are used to generate other types of random numbers

C Library Implementation

```
// RAND_MAX is the largest value returned by rand
// RAND_MAX is 32767 on MS VC++ and on Sun Workstations
// RAND_MAX is 2147483647 on my Linux server
#define RAND_MAX    XXXXX

// This function generates a new pseudo-random number
int rand();

// This function resets the sequence of
// pseudo-random numbers to be generated by rand
void srand(unsigned int seed);
```

Example Usage

```
#include <stdlib.h>
#include <stdio.h>

int main()
   {
   int i;

   printf(“10 random numbers between 0 and %d\n”, RAND_MAX);

   /* Seed the random-number generator with
    * current time so that numbers will be
    * different for every run.
    */
   srand( (unsigned) time(NULL) );

   /* Display 10 random numbers. */
   for( i = 0; i < 10; i++ )
      printf( "  %6d\n", rand() );
   }
```

Unfortunately …

- Many library implementations of `rand()` are botched

- Referring to an early IBM implementation, a computer consultant said …
  - *We guarantee each number is random individually, but we don't guarantee that more than one of them is random.*

Good Advice

- Always use a random number generator that is known to produce “good quality” random numbers

- “Strange looking, apparently unpredictable sequences are not enough”
  - Park and Miller (1988) in Communications of the ACM provide several examples

Lehmer's (1951) Algorithm

- Multiplicative linear congruential generator
  - $I_{j+1} = aI_j \bmod m$
- Where
  - I_j is the j^{th} number in the sequence
  - m is a large *prime* integer
  - a is an integer $2 \,..\, m - 1$

Rescaling

- To produce numbers in the interval 0..1:
  - $U_j = I_j / m$
- These will range between *1/m* and *1 – 1/m*

Example 1

- $I_{j+1} = 6\, I_j \bmod 13$

- Produces the sequence:
    - … 1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1, …

- Which includes all values $1 .. m\text{-}1$ before repeating itself

Example 2

- $I_{j+1} = 7\,I_j \bmod 13$
- Produces the sequence:
  - … 1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1 …
- This sequence still has a full period, but looks a little less “random” …

Example 3

- $I_{j+1} = 5 I_j \bmod 13$
- Produces one of the sequences:
  - … 1, 5, 12, 8, 1, …
  - … 2, 10, 11, 3, 2, …
  - … 4, 7, 9, 6, 4, …
- In this case, if m = 13, a = 5 is a very poor choice

Practical Values for *a* and *m*

- Do not choose your own (dangerous!)
- Rely on values that are known to work.

- Good sources:
  - Numerical Recipes in C
  - Park and Miller (1988) Communications of the ACM

- We will use $a = 16807$ and $m = 2147483647$

A Random Number Generator

```
/* This implementation will not work in
 * many systems, due to integer overflows
 */

static int seed = 1;
double Random()
   {
   int a = 16807;
   int m = 2147483647; /* 2^31 – 1 */

   seed = (a * seed) % m;
   return seed / (double) m;
   }

/* If this is working properly, starting with seed = 1,
 * the 10,000th call produces seed = 1043618065
 */
```

A Random Number Generator

```
/* This implementation will not work in newer compilers that
 * support 64-bit integer variables of type long long
 */

static long long seed = 1;
double Random()
   {
   long long a = 16807;
   long long m = 2147483647; /* 2^31 – 1 */

   seed = (a * seed) % m;
   return seed / (double) m;
   }

/* If this is working properly, starting with seed = 1,
 * the 10,000th call produces seed = 1043618065
 */
```

Practical Computation

- Many systems will not represent integers larger than 2^{32}
- We need a practical calculation where:
  - Results Cover nearly all possible integers
  - Intermediate values do not exceed 2^{32}

The Solution

- Let $m = aq + r$
- Where
  - $q = m / a$
  - $r = m \bmod a$
  - $r < q$
- Then $$aI_j \bmod m = \begin{cases} a(I_j \bmod q) - r[I_j / q] & if \geq 0 \\ a(I_j \bmod q) - r[I_j / q] + m & \end{cases}$$

Random Number Generator:
A Portable Implementation

```
#define RAND_A      16807
#define RAND_M      2147483647
#define RAND_Q      127773
#define RAND_R      2836
#define RAND_SCALE  (1.0 / RAND_M)

static int seed = 1;

double Random()
   {
   int k = seed / RAND_Q;

   seed = RAND_A * (seed – k * RAND_Q) – k * RAND_R;

   if (seed < 0) seed += RAND_M;

   return seed * (double) RAND_SCALE;
   }
```

Reliable Generator

- Fast
- Some slight improvements possible:
  - Use $a = 48271$ ($q = 44488$ and $r = 3399$)
  - Use $a = 69621$ ($q = 30845$ and $r = 23902$)
- Still has some subtle weaknesses …
  - E.g. whenever a value $< 10^{-6}$ occurs, it will be followed by a value < 0.017, which is 10^{-6} * RAND_A

Further Improvements

- **Shuffle Output.**
  - Generate two sequences, and use one to permute the output of the other.
- **Sum Two Sequences.**
  - Generate two sequences, and return the sum of the two (modulus the period for either).

Example: Shuffling (Part I)

```
// Define RAND_A, RAND_M, RAND_Q, RAND_R as before
#define RAND_TBL 32
#define RAND_DIV (1 + (RAND_M – 1) / RAND_TBL)

static int random_next = 0;
static int random_tbl[RAND_TBL];

void SetupRandomNumbers(int seed)
   {
   int j;

   if (seed == 0) seed = 1;

   for (j = RAND_TBL – 1; j >= 0; j--)
       {
       int k = seed / RAND_Q;
       seed = RAND_A * (seed – k * RAND_Q) – k * RAND_R;
       if (seed < 0) seed += RAND_M;
       random_tbl[j] = seed;
       }

   random_next = random_tbl[0];
   }
```

Example: Shuffling (Part II)

```
double Random()
   {
   // Generate the next number in the sequence
   int k = seed / RAND_Q, index;
   seed = RAND_A * (seed – k * RAND_Q) – k * RAND_R;
   if (seed < 0) seed += RAND_M;

   // Swap it for a previously generated number
   index = random_next / RAND_DIV;
   random_next = random_tbl[index];
   random_tbl[index] = seed;

   // And return the shuffled result …
   return random_next * (double) RAND_SCALE;
   }
```

Shuffling ...

- Shuffling improves things, however …
- Requires additional storage …
- If an extremely small value occurs (e.g. $< 10^{-6}$) it will be slightly correlated with other nearby extreme values.

Summing Two Sequences (I)

```
#define RAND_A1      40014
#define RAND_M1      2147483563
#define RAND_Q1      53668
#define RAND_R1      12211

#define RAND_A2      40692
#define RAND_M2      2147483399
#define RAND_Q2      52744
#define RAND_R2      3791

#define RAND_SCALE1  (1.0 / RAND_M1)
```

Summing Two Sequences (II)

```
static int seed1 = 1, seed2 = 1;

double Random()
   {
   int k, result;

   k = seed1 / RAND_Q1;
   seed1 = RAND_A1 * (seed1 – k * RAND_Q1) – k * RAND_R1;
   if (seed1 < 0) seed1 += RAND_M1;

   k = seed2 / RAND_Q2;
   seed2 = RAND_A2 * (seed2 – k * RAND_Q2) – k * RAND_R2;
   if (seed2 < 0) seed2 += RAND_M2;

   result = seed1 – seed2;
   if (result < 1) result += RAND_M1 – 1;

   return result * (double) RAND_SCALE1;
   }
```

Summing Two Sequences

- If the sequences are uncorrelated, we can do no harm:
  - If the original sequence is “random”, summing a second sequence will preserve the original randomness
- In the ideal case, the period of the combined sequence will be the least common multiple of the individual periods

Summing More Sequences

- It is possible to sum more sequences to increase randomness

- One example is the Wichman Hill random number generator, where:
  - A1 = 171, M1 = 30269
  - A2 = 172, M2 = 30307
  - A3 = 170, M3 = 30323

- Values for each sequence are:
  - Scaled to the interval (0,1)
  - Summed
  - Integer part of sum is discarded

So far …

- Uniformly distributed random numbers
  - Using Lehmer’s algorithm
  - Work well for carefully selected parameters

- “Randomness” can be improved:
  - Through shuffling
  - Summing two sequences
  - Or both (see Numerical Recipes for an example)

Random Numbers in R

- In R, multiple generators are supported

- To select a specific sequence use:
  - `RNGkind()` -- select algorithm
  - `RNGversion()` -- mimics older R versions
  - `set.seed()` -- selects specific sequence

- Use `help(RNGkind)` for details

Random Numbers in R

- Many custom functions:
  - `runif(n, min = 0, max = 1)`
  - `rnorm(n, mean = 0, sd = 1)`
  - `rt(n, df)`
  - `rchisq(n, df, ncp = 0)`
  - `rf(n, df1, df2)`
  - `rexp(n, rate = 1)`
  - `rgamma(n, shape, rate = 1)`

Sampling from Arbitrary Distributions

- The general approach for sampling from an arbitrary distribution is to:

- Define
  - Cumulative density function F(x)
  - Inverse cumulative density function $F^{-1}(x)$

- Sample x ~ U(0,1)
- Evaluate $F^{-1}(x)$

Example: Exponential Distribution

- Consider:
  - $f(x) = e^{-x}$
  - $F(x) = 1 - e^{-x}$
  - $F^{-1}(y) = -\ln(1 - y)$

```
double RandomExp()
    {
    return –log(Random());
    }
```

Example: Categorical Data

- To sample from a discrete set of outcomes, use:

```
int SampleCategorical(int outcomes, double * probs)
  {
  double prob = Random();
  int    outcome = 0;

  while (outcome + 1 < outcomes && prob > probs[outcome])
     {
     prob -= probs[outcome];
     outcome++;
     }

  return outcome;
  }
```

More Useful Examples

- Numerical Recipes in C has additional examples, including algorithms for sampling from normal and gamma distributions

The Mersenne Twister

- Current gold standard random generator
- Web: www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
  - Or Google for “Mersenne Twister”
- Has a very long period ($2^{19937} - 1$)
- Equi-distributed in up to 623 dimensions

Recommended Reading

- Numerical Recipes in C
  - Chapters 7.1 – 7.3
- Park and Miller (1998)

  “Random Number Generators:

  Good Ones Are Hard To Find”

  Communications of the ACM

Implementation Without Division

- Let *a = 16807* and *m = 2147483647*
- It is actually possible to implement Park-Miller generator without any divisions
  - Division is 20-40x slower than other operations
- Solution proposed by D. Carta (1990)

A Random Number Generator

```
/* This implementation is very fast, because there is no division */

static unsigned int seed = 1;
int RandomInt()
   {
   // After calculation below, (hi << 16) + lo = seed * 16807
   unsigned int lo = 16807 * (seed & 0xFFFF);  // Multiply lower 16 bits by 16807
   unsigned int hi = 16807 * (seed >> 16);     // Multiply higher 16 bits by 16807

   // After these lines, lo has the bottom 31 bits of result, hi has bits 32 and up
   lo += (hi & 0x7FFF) << 16;       // Combine lower 15 bits of hi with lo’s upper bits
   hi >>= 15;                       // Discard the lower 15 bits of hi

   // value % (2^31 - 1)) = ((2^31) * hi + lo) % (2^31 – 1)
   //                     = ((2^31 - 1) * hi + hi + lo) % (2^31-1)
   //                     = (hi + lo) % (2^31 – 1)
   lo += hi;

   // No division required, since hi + lo is always < 2^32 - 2
   if (lo > 2147483647) lo -= 2147483647;

   return (seed = lo);
   }
```