Math / Stat.

P1.

(Update, Thanks to Ji Yang's idea)

For any girl, the next one to her (clockwise) is a boy with probability B/(B+G-1);

For any boy, the next one to him (clockwise) is a girl with probability G/(B+G-1).

So the total expected number of neighbors is G* B/(B+G-1)+ B* G/(B+G-1) = 2 GB/(B+G-1).

P2.

Uncorrelated random variables are defined as random variables which have zero correlation. If for two random variables, the conditional distribution of one on the other equals the distribution of itself, these two random variables are independent with each other. Obviously, independence is stronger statement.

Example of random variables that are uncorrelated but not independent:

Assume random variable X satisfies f(X) = f(-X), set Y= |X|. We have Cor(X, Y) = E(X)E(Y) - E(XY) = 0 - 0 = 0. Then X and Y are uncorrelated. But obviously, X and Y are not independent.

P3.

This is a typical dynamic programming problem, appeared in the Green book page 123, Dice Game.

If I am given only 1 time to toss, the value is 1/6*(1+2+3+4+5+6)=3.5

If given 2 times, the value is 1/6*(4+5+6+3*3.5) =4.25 considering that I will continue if the first toss gets less than 3.5, the expected value of the next toss if I do not stop after the first.

If given 3 times, the value is 1/6*(5+6+4*4.25)=14/3 considering that I will continue if the first toss gets less than 4.25, the expected value of the next two tosses if I do not stop after the first.

P5

covariant matrix is symmetric positive semidefinite. According to Sylvester's criterion, its principal minors are greater than or equal to 0.

$$\Omega = \begin{vmatrix} 1 & r & 0 \\ r & 1 & r \\ 0 & r & 1 \end{vmatrix}, |\Omega| = 1 - r^2 > 0, \begin{vmatrix} 1 & r \\ r & 1 \end{vmatrix} > 0 \Longrightarrow -1 < r < 1$$

P6

S_n as a symmetric random walk. (Where $S_n = X_1 + X_2 + X_3 \cdots + X_n$ with n being the stopping time). We know that both S_n and $S_n^2 - n$ are martingales.

a) Let P(−1) be the probability that the drunk man go through the left door, and N be the stopping time. $E[S_N] = P(-1)*(-1)+(1-P(-1))*99 = S_0 = 0$, $E[S_N^2 - N] = E[P(-1)\times 1+(1-P(-1))\times 99^2]-E[N] = S_0^2 - 0 = 0$, $\Longrightarrow P(-1) = 99/100$, $N = 99$

b) if left door is locked, the drunk man need to walk right for the next step at the left door, and exit from the right door. we could use symetric analysis, it's equivalent that the drunk man could exit at either 99 or -101. Use the same method, we get P (99) = 0.505, N = 9999

P7&8. What is Ridge regression and Lasso regression?

Ridge and Lasso are two shrinkage methods. They shrink the regression coefficients by imposing a penalty on their size. There are two motivations to use Ridge and Lasso regressions:

Motivation 1: too many predictors

♣ It is not unusual to see the number of input variables greatly exceed the number of observations, e.g. micro-array data analysis, environmental pollution studies.

♣ With many predictors, fitting the full model without penalization will result in large prediction intervals, and LS regression estimator may not uniquely exist.

Motivation 2: correlated variables X in a linear regression model

♣ Because the LS estimates depend upon , we would have problems in computing if X ′X were singular or nearly singular.

♣ In those cases, small changes to the elements of X lead to large changes in . In other words, due to the correlated variables, the coefficients are poorly determined and exhibit high variance.

The Ridge and Lasso coefficients minimize a penalized residual sum of squares:

Here is a complexity parameter that controls the amount of shrinkage: the larger the value of the greater the amount of shrinkage. The coefficients are shrunk toward zero.
The difference between Ridge and Lasso regression is that when the estimated coefficients have sharp boundary, e.g. the coefficients are piece-wise, Lasso can capture this sharp boundary better than Ridge.

P9.

Assumes the wealth of the player is w and the ticket price is c. The player would only want to play if the expected exponential rate of growth g(w,c) is positive:

where D_k = 2^k is the payoff at round k and p_k = 1/(2^k) is the probability.
g(w,c) is a decreasing function of c. The constaint g(w,c) >0 gives the upper bound of c, which is the maximum price the player would want to pay for the game. Numerically, for w=100, c ~= 3; for w = 1million, c~=8; for w=100million, c~=12.

Programming

P10.

The use of 'default' in C++ is a class' member functions automatically generated by the complier if not declared. They include constructor, copy constructor, copy assignment operator, and destructor. All these functions will be both public and inline.
(see item 5 of Effective C++" third version).

You should generally write a default constructor for every class you define, to guarantee the state of any "default constructed" variable. If you don't declare a default constructor, the complier will supply one for you; however, since it does not know much about your class, it won't be able to guarantee very much about the initial stat of one of your variables. In fact, all the native types will be left in random state, as though they were declared but not initialized; this is an undesirable condition.
Why did I say 'generally', but not always? Because there are some times when you do not want to allow an object to be created unless the 'real' data is available.
If you want to make it impossible to create an object via complied-generated default constructor, you can declare a private default constructor that will cause a complier error in any user code that tried to define an object of that class without specifying an initial value. You do not have to implement this constructor, because a program that tried to use it won't compile. (see page 328 of C++: A Dialogue)

P11.

http://stackoverflow.com/questions/2576022/efficient-thread-safe-singleton-in-c

A. This Meyers/Alexandrescu paper (*www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf*) explains why - but that paper is also widely misunderstood. It started the 'double checked locking is unsafe in C++' meme - but its actual conclusion is that double checked locking in C++ can be implemented safely, it just requires the use of memory barriers in a non-obvious place.

The paper contains pseudocode demonstrating how to use memory barriers to safely implement the DLCP, so it shouldn't be difficult for you to correct your implementation.

B. If you are using C++11, here is a right way to do this:

```
Foo& getInst()

{

  static Foo inst(...);

  return inst;

}
```

According to new standard there is no need to care about this problem any more. Object initialization will be made only by one thread, other threads will wait till it complete. Or you can use std::call_once. (more info here)

C. ACE singleton implementation uses double-checked locking pattern for thread safety, you can refer to it if you like. You can find source code here.

```
// Singleton.cpp,v 4.54 2005/10/28 16:14:55 ossama Exp
#ifndef ACE_SINGLETON_CPP
#define ACE_SINGLETON_CPP
#include "ace/Singleton.h"
#if !defined (ACE_LACKS_PRAGMA_ONCE)
# pragma once
#endif /* ACE_LACKS_PRAGMA_ONCE */
#if !defined (__ACE_INLINE__)
#include "ace/Singleton.inl"
#endif /* __ACE_INLINE__ */
#include "ace/Object_Manager.h"
#include "ace/Log_Msg.h"
#include "ace/Framework_Component.h"
#include "ace/Guard_T.h"
```

```
ACE_RCSID (ace,
           Singleton,
           "Singleton.cpp,v 4.54 2005/10/28 16:14:55 ossama Exp")


ACE_BEGIN_VERSIONED_NAMESPACE_DECL

template <class TYPE, class ACE_LOCK> void
ACE_Singleton<TYPE, ACE_LOCK>::dump (void)
{
#if defined (ACE_HAS_DUMP)
  ACE_TRACE ("ACE_Singleton<TYPE, ACE_LOCK>::dump");

#if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
  ACE_DEBUG ((LM_DEBUG,  ACE_LIB_TEXT ("instance_ = %x"),
              ACE_Singleton<TYPE, ACE_LOCK>::instance_i ()));
  ACE_DEBUG ((LM_DEBUG, ACE_END_DUMP));
#endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
#endif /* ACE_HAS_DUMP */
}

template <class TYPE, class ACE_LOCK> ACE_Singleton<TYPE, ACE_LOCK> *&
ACE_Singleton<TYPE, ACE_LOCK>::instance_i (void)
{
#if defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
  // Pointer to the Singleton instance.  This works around a bug with
  // G++ and it's (mis-)handling of templates and statics...
  static ACE_Singleton<TYPE, ACE_LOCK> *singleton_ = 0;

  return singleton_;
#else
  return ACE_Singleton<TYPE, ACE_LOCK>::singleton_;
#endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
}

template <class TYPE, class ACE_LOCK> TYPE *
ACE_Singleton<TYPE, ACE_LOCK>::instance (void)
{
  ACE_TRACE ("ACE_Singleton<TYPE, ACE_LOCK>::instance");

  ACE_Singleton<TYPE, ACE_LOCK> *&singleton =
    ACE_Singleton<TYPE, ACE_LOCK>::instance_i ();
```

```

  // Perform the Double-Check pattern...
  if (singleton == 0)
    {
      if (ACE_Object_Manager::starting_up () ||
          ACE_Object_Manager::shutting_down ())
        {
          // The program is still starting up, and therefore assumed
          // to be single threaded.  There's no need to double-check.
          // Or, the ACE_Object_Manager instance has been destroyed,
          // so the preallocated lock is not available.  Either way,
          // don't register for destruction with the
          // ACE_Object_Manager:  we'll have to leak this instance.

          ACE_NEW_RETURN (singleton, (ACE_Singleton<TYPE, ACE_LOCK>), 0);
        }
      else
        {
#if defined (ACE_MT_SAFE) && (ACE_MT_SAFE != 0)
          // Obtain a lock from the ACE_Object_Manager.  The pointer
          // is static, so we only obtain one per ACE_Singleton
          // instantiation.
          static ACE_LOCK *lock = 0;
          if (ACE_Object_Manager::get_singleton_lock (lock) != 0)
            // Failed to acquire the lock!
            return 0;

          ACE_GUARD_RETURN (ACE_LOCK, ace_mon, *lock, 0);

          if (singleton == 0)
            {
#endif /* ACE_MT_SAFE */
              ACE_NEW_RETURN (singleton, (ACE_Singleton<TYPE, ACE_LOCK>), 0);

              // Register for destruction with ACE_Object_Manager.
              ACE_Object_Manager::at_exit (singleton);
#if defined (ACE_MT_SAFE) && (ACE_MT_SAFE != 0)
            }
#endif /* ACE_MT_SAFE */
        }
    }

  return &singleton->instance_;
```

```
}

template <class TYPE, class ACE_LOCK> void
ACE_Singleton<TYPE, ACE_LOCK>::cleanup (void *)
{
  delete this;
  ACE_Singleton<TYPE, ACE_LOCK>::instance_i () = 0;
}

#if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
// Pointer to the Singleton instance.
template <class TYPE, class ACE_LOCK> ACE_Singleton<TYPE, ACE_LOCK> *
ACE_Singleton<TYPE, ACE_LOCK>::singleton_ = 0;

template <class TYPE, class ACE_LOCK> ACE_Unmanaged_Singleton<TYPE,
ACE_LOCK> *
ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::singleton_ = 0;
#endif /* !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES) */

template <class TYPE, class ACE_LOCK> void
ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::dump (void)
{
#if defined (ACE_HAS_DUMP)
  ACE_TRACE ("ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::dump");

#if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
  ACE_DEBUG ((LM_DEBUG,  ACE_LIB_TEXT ("instance_ = %x"),
              ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::instance_i ()));
  ACE_DEBUG ((LM_DEBUG, ACE_END_DUMP));
#endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
#endif /* ACE_HAS_DUMP */
}

template <class TYPE, class ACE_LOCK>
ACE_Unmanaged_Singleton<TYPE, ACE_LOCK> *&
ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::instance_i (void)
{
#if defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
  // Pointer to the Singleton instance.  This works around a bug with
  // G++ and it's (mis-)handling of templates and statics...
  static ACE_Unmanaged_Singleton<TYPE, ACE_LOCK> *singleton_ = 0;

  return singleton_;
```

```
#else
  return ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::singleton_;
#endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
}

template <class TYPE, class ACE_LOCK> TYPE *
ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::instance (void)
{
  ACE_TRACE ("ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::instance");

  ACE_Unmanaged_Singleton<TYPE, ACE_LOCK> *&singleton =
    ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::instance_i ();

  // Perform the Double-Check pattern...
  if (singleton == 0)
    {
      if (ACE_Object_Manager::starting_up () ||
          ACE_Object_Manager::shutting_down ())
        {
          // The program is still starting up, and therefore assumed
          // to be single threaded.  There's no need to double-check.
          // Or, the ACE_Object_Manager instance has been destroyed,
          // so the preallocated lock is not available.  Either way,
          // don't register for destruction with the
          // ACE_Object_Manager:  we'll have to leak this instance.

          ACE_NEW_RETURN (singleton, (ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>),
                          0);
        }
      else
        {
#if defined (ACE_MT_SAFE) && (ACE_MT_SAFE != 0)
          // Obtain a lock from the ACE_Object_Manager.  The pointer
          // is static, so we only obtain one per
          // ACE_Unmanaged_Singleton instantiation.
          static ACE_LOCK *lock = 0;
          if (ACE_Object_Manager::get_singleton_lock (lock) != 0)
            // Failed to acquire the lock!
            return 0;

          ACE_GUARD_RETURN (ACE_LOCK, ace_mon, *lock, 0);
#endif /* ACE_MT_SAFE */
```

```

          if (singleton == 0)
            ACE_NEW_RETURN (singleton,
                            (ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>),
                            0);
        }
    }

  return &singleton->instance_;
}

template <class TYPE, class ACE_LOCK> void
ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::close (void)
{
  ACE_Unmanaged_Singleton<TYPE, ACE_LOCK> *&singleton =
    ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::instance_i ();

  if (singleton)
    {
      singleton->cleanup ();
      ACE_Unmanaged_Singleton<TYPE, ACE_LOCK>::instance_i () = 0;
    }
}

template <class TYPE, class ACE_LOCK> void
ACE_TSS_Singleton<TYPE, ACE_LOCK>::dump (void)
{
#if defined (ACE_HAS_DUMP)
  ACE_TRACE ("ACE_TSS_Singleton<TYPE, ACE_LOCK>::dump");

#if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
  ACE_DEBUG ((LM_DEBUG,  ACE_LIB_TEXT ("instance_ = %x"),
              ACE_TSS_Singleton<TYPE, ACE_LOCK>::instance_i ()));
  ACE_DEBUG ((LM_DEBUG, ACE_END_DUMP));
#endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
#endif /* ACE_HAS_DUMP */
}

template <class TYPE, class ACE_LOCK> ACE_TSS_Singleton<TYPE, ACE_LOCK>
*&
ACE_TSS_Singleton<TYPE, ACE_LOCK>::instance_i (void)
{
#if defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
```

```
  // Pointer to the Singleton instance.  This works around a bug with
  // G++ and it's (mis-)handling of templates and statics...
  static ACE_TSS_Singleton<TYPE, ACE_LOCK> *singleton_ = 0;

  return singleton_;
#else
  return ACE_TSS_Singleton<TYPE, ACE_LOCK>::singleton_;
#endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
}

template <class TYPE, class ACE_LOCK> TYPE *
ACE_TSS_Singleton<TYPE, ACE_LOCK>::instance (void)
{
  ACE_TRACE ("ACE_TSS_Singleton<TYPE, ACE_LOCK>::instance");

  ACE_TSS_Singleton<TYPE, ACE_LOCK> *&singleton =
    ACE_TSS_Singleton<TYPE, ACE_LOCK>::instance_i ();

  // Perform the Double-Check pattern...
  if (singleton == 0)
    {
      if (ACE_Object_Manager::starting_up () ||
          ACE_Object_Manager::shutting_down ())
        {
          // The program is still starting up, and therefore assumed
          // to be single threaded.  There's no need to double-check.
          // Or, the ACE_Object_Manager instance has been destroyed,
          // so the preallocated lock is not available.  Either way,
          // don't register for destruction with the
          // ACE_Object_Manager:  we'll have to leak this instance.

          ACE_NEW_RETURN (singleton, (ACE_TSS_Singleton<TYPE, ACE_LOCK>), 0);
        }
      else
        {
#if defined (ACE_MT_SAFE) && (ACE_MT_SAFE != 0)

          // Obtain a lock from the ACE_Object_Manager.  The pointer
          // is static, so we only obtain one per ACE_Singleton instantiation.
          static ACE_LOCK *lock = 0;
          if (ACE_Object_Manager::get_singleton_lock (lock) != 0)
            // Failed to acquire the lock!
```

```
          return 0;

        ACE_GUARD_RETURN (ACE_LOCK, ace_mon, *lock, 0);

        if (singleton == 0)
          {
#endif /* ACE_MT_SAFE */
            ACE_NEW_RETURN (singleton, (ACE_TSS_Singleton<TYPE, ACE_LOCK>),
                            0);

            // Register for destruction with ACE_Object_Manager.
            ACE_Object_Manager::at_exit (singleton);
#if defined (ACE_MT_SAFE) && (ACE_MT_SAFE != 0)
          }
#endif /* ACE_MT_SAFE */
      }
  }

  return ACE_TSS_GET (&singleton->instance_, TYPE);
}

template <class TYPE, class ACE_LOCK> void
ACE_TSS_Singleton<TYPE, ACE_LOCK>::cleanup (void *)
{
  delete this;
  ACE_TSS_Singleton<TYPE, ACE_LOCK>::instance_i () = 0;
}

template <class TYPE, class ACE_LOCK> void
ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::dump (void)
{
#if defined (ACE_HAS_DUMP)
  ACE_TRACE ("ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::dump");

#if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
  ACE_DEBUG ((LM_DEBUG,  ACE_LIB_TEXT ("instance_ = %x"),
              ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::instance_i ()));
  ACE_DEBUG ((LM_DEBUG, ACE_END_DUMP));
#endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
#endif /* ACE_HAS_DUMP */
}

template <class TYPE, class ACE_LOCK>
```

```
ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK> *&
ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::instance_i (void)
{
#if defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
  // Pointer to the Singleton instance.  This works around a bug with
  // G++ and it's (mis-)handling of templates and statics...
  static ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK> *singleton_ = 0;

  return singleton_;
#else
  return ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::singleton_;
#endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
}

template <class TYPE, class ACE_LOCK> TYPE *
ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::instance (void)
{
  ACE_TRACE ("ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::instance");

  ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK> *&singleton =
    ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::instance_i ();

  // Perform the Double-Check pattern...
  if (singleton == 0)
    {
      if (ACE_Object_Manager::starting_up () ||
          ACE_Object_Manager::shutting_down ())
        {
          // The program is still starting up, and therefore assumed
          // to be single threaded.  There's no need to double-check.
          // Or, the ACE_Object_Manager instance has been destroyed,
          // so the preallocated lock is not available.  Either way,
          // don't register for destruction with the
          // ACE_Object_Manager:  we'll have to leak this instance.

          ACE_NEW_RETURN (singleton,
                          (ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>),
                          0);
        }
      else
        {
#if defined (ACE_MT_SAFE) && (ACE_MT_SAFE != 0)
          // Obtain a lock from the ACE_Object_Manager.  The pointer
```

```
          // is static, so we only obtain one per
          // ACE_Unmanaged_Singleton instantiation.
          static ACE_LOCK *lock = 0;
          if (ACE_Object_Manager::get_singleton_lock (lock) != 0)
            // Failed to acquire the lock!
            return 0;

          ACE_GUARD_RETURN (ACE_LOCK, ace_mon, *lock, 0);
#endif /* ACE_MT_SAFE */

          if (singleton == 0)
            ACE_NEW_RETURN (singleton,
                            (ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>),
                            0);
        }
    }

  return ACE_TSS_GET (&singleton->instance_, TYPE);
}

template <class TYPE, class ACE_LOCK> void
ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::close (void)
{
  ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK> *&singleton =
    ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::instance_i ();

  if (singleton)
    singleton->cleanup ();
}

#if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
// Pointer to the Singleton instance.
template <class TYPE, class ACE_LOCK> ACE_TSS_Singleton <TYPE, ACE_LOCK>
*
ACE_TSS_Singleton<TYPE, ACE_LOCK>::singleton_ = 0;

template <class TYPE, class ACE_LOCK>
ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK> *
ACE_Unmanaged_TSS_Singleton<TYPE, ACE_LOCK>::singleton_ = 0;
#endif /* !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES) */

/*************************************************************************/

```

```
#if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
// Pointer to the Singleton instance.
template <class TYPE, class ACE_LOCK> ACE_DLL_Singleton_T<TYPE, ACE_LOCK> *
ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::singleton_ = 0;
#endif /* !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES) */

template <class TYPE, class ACE_LOCK> void
ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::dump (void)
{
#if defined (ACE_HAS_DUMP)
  ACE_TRACE ("ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::dump");

#if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
  ACE_DEBUG ((LM_DEBUG,  ACE_LIB_TEXT ("instance_ = %x"),
              ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::instance_i ()));
  ACE_DEBUG ((LM_DEBUG, ACE_END_DUMP));
#endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
#endif /* ACE_HAS_DUMP */
}

template <class TYPE, class ACE_LOCK>
ACE_DLL_Singleton_T<TYPE, ACE_LOCK> *&
ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::instance_i (void)
{
  ACE_TRACE ("ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::instance_i");

#if defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
  // Pointer to the Singleton instance.  This works around a bug with
  // G++ and it's (mis-)handling of templates and statics...
  static ACE_DLL_Singleton_T<TYPE, ACE_LOCK> *singleton_ = 0;

  return singleton_;
#else
  return ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::singleton_;
#endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */
}

template <class TYPE, class ACE_LOCK> TYPE *
ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::instance (void)
{
  ACE_TRACE ("ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::instance");

```

```
  ACE_DLL_Singleton_T<TYPE, ACE_LOCK> *&singleton =
    ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::instance_i ();

  // Perform the Double-Check pattern...
  if (singleton == 0)
    {
      if (ACE_Object_Manager::starting_up () ||
          ACE_Object_Manager::shutting_down ())
        {
          // The program is still starting up, and therefore assumed
          // to be single threaded.  There's no need to double-check.
          // Or, the ACE_Object_Manager instance has been destroyed,
          // so the preallocated lock is not available.  Either way,
          // don't register for destruction with the
          // ACE_Object_Manager:  we'll have to leak this instance.

          ACE_NEW_RETURN (singleton, (ACE_DLL_Singleton_T<TYPE, ACE_LOCK>),
                          0);
        }
      else
        {
#if defined (ACE_MT_SAFE) && (ACE_MT_SAFE != 0)
          // Obtain a lock from the ACE_Object_Manager.  The pointer
          // is static, so we only obtain one per
          // ACE_Unmanaged_Singleton instantiation.
          static ACE_LOCK *lock = 0;
          if (ACE_Object_Manager::get_singleton_lock (lock) != 0)
            // Failed to acquire the lock!
            return 0;

          ACE_GUARD_RETURN (ACE_LOCK, ace_mon, *lock, 0);
#endif /* ACE_MT_SAFE */

          if (singleton == 0)
            ACE_NEW_RETURN (singleton,
                            (ACE_DLL_Singleton_T<TYPE, ACE_LOCK>),
                            0);
        }
      //ACE_REGISTER_FRAMEWORK_COMPONENT(ACE_DLL_Singleton<TYPE,ACE_LOCK>, singleton);
      ACE_Framework_Repository::instance ()->register_component
        (new ACE_Framework_Component_T<ACE_DLL_Singleton_T<TYPE, ACE_LOCK> > (singleton));
```

```
    }

  return &singleton->instance_;
}

template <class TYPE, class ACE_LOCK> void
ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::close (void)
{
  ACE_TRACE ("ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::close");

  ACE_DLL_Singleton_T<TYPE, ACE_LOCK> *&singleton =
    ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::instance_i ();

  delete singleton;
  singleton = 0;
}

template <class TYPE, class ACE_LOCK> void
ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::close_singleton (void)
{
  ACE_TRACE ("ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::close_singleton");
  ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::close ();
}

template <class TYPE, class ACE_LOCK> const ACE_TCHAR *
ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::dll_name (void)
{
  return this->instance ()->dll_name ();
}

template <class TYPE, class ACE_LOCK> const ACE_TCHAR *
ACE_DLL_Singleton_T<TYPE, ACE_LOCK>::name (void)
{
  return this->instance ()->name ();
}


/**********************************************************************/

template <class TYPE> const ACE_TCHAR*
ACE_DLL_Singleton_Adapter_T<TYPE>::dll_name (void)
{
  // @todo make this a constant somewhere (or it there already is one
```

```
  // then use it.
  return ACE_TEXT("ACE");
}

ACE_END_VERSIONED_NAMESPACE_DECL

#endif /* ACE_SINGLETON_CPP */
```

P13.

There are several ways to generate random numbers. One of the most common PRNG is the linear congruential generator, which uses the recurrence

to generate numbers, where a, b and m are large integers, and is the next in X as a series of pseudo-random numbers.
Testing
Assume that f is a function taking any finite sequence of zeros and ones, and returning a non-negative real value. Then, given a sequence of independent and uniformly distributed random variables X_n, applying f to the finite sequence of random variables $(X_1,\ldots,X_n)$ yields a new random variable, Y_n. This new variable has a certain cumulative probability distribution $F_n(x)=\mathbb{P}(Y_n\leq x)$, which in some cases approaches a function F as ngrows large. This limit function F can be seen as the cumulative probability distribution of a new random variable, Y, and in these cases Y_n is said to converge in distribution to Y.

P15.

Exclusive or operation is a logical operation that outputs true only when both inputs differ. In this problem we could sum all elements in the array using exclusive or operation. All elements that appear twice would sum to zero and the final result would be the element that appears only once.

P16.

We flip a coin n times to generate a binary sequence of size n. This number is in the range of $0 - (2^n - 1)$. If we treat binary sequence $000 \cdots 0$ as value 2^n, we get a variable ranging from $1 - 2^n$. This number mode 6 will generate output $0 - 5$ equally if we reject number $2^n > 6^m$ given we want m die flips, r is the rejection rate.

$$m = [nlog_3 2], r = 1 - \frac{3^m}{2^n}$$

P17.

Basicly, we move both robots in the same direction, the robot which gets to zero first stays at zero, the other robot reverts its direction until it gets to zero as well. Two robots will meet at zero.

```
#include<iostream> using namespace std ;

class robot{

private : int pos;

public : robot(int x){ pos = x; }

void Go_Left (){ pos −=1; }

void Go Right (){ pos +=1; }

bool at_zero (){
return ( pos == 0 );

} };

int main(){

robot robot1(−8); robot robot2 (7);

int count=0;
while( !robot1.at zero() || !robot2.at zero() ){

if ( robot1.at zero() || robot2.at zero() ){ if( robot1.at zero() ) {

robot2 . Go Left (); count++;

}else{ robot1 . Go Left ();

count++;

}

}else{
robot1.Go Right();

robot2.Go Right(); count++;

}

}

cout << count << endl ;

return 0; }
```