Solution 3

Group A

1) Let I_i be the indicator if i-th person and the next (clockwise) person forms a boy-girl neighbor. Then
$$E\sum_{i=1}^{16} I_i = \sum_{i=1}^{16} EI_i = 16 * \frac{9*7*2}{16*15} = \frac{42}{5}.$$
2) In general, *independence* implies *uncorrelated*, but not vice versa.
   E.g. Two random variables X and Y are independent if and only if $F_{X,Y}(x,y) = F_X(x)F_Y(y)$. They are uncorrelated if and if $\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$. Assuming independence, it is easy to check that
$$\mathbb{E}[XY] = \int xydF_{X,Y}(x,y) = \int xdF_X(x) \cdot \int ydF_Y(y) = \mathbb{E}[X]\mathbb{E}[Y].$$
   Moreover, it can be shown that for any two measurable functions f and g where $\mathbb{E}[|f(X)|] < \infty$ and $\mathbb{E}[|g(Y)|] < \infty$, we have
$$\mathbb{E}[f(X)g(Y)] = \mathbb{E}[f(X)]\mathbb{E}[g(Y)].$$
   Now consider random variables $X = \mathbf{1}_{[0,1/4)} - \mathbf{1}_{[1/4,1/2)}$ and $Y = \mathbf{1}_{[1/2,3/4)} - \mathbf{1}_{[3/4,1)}$ on the probability space $\Omega = [0,1)$ with standard Lebesgue measure. We have $\mathbb{E}[X] = 0$, $\mathbb{E}[Y] = 0$, and $\mathbb{E}[XY] = \mathbb{E}[0] = 0$, so they are uncorrelated. However, $\mathbb{E}[X^2] = 1/2$, $\mathbb{E}[Y^2] = 1/2$, but $\mathbb{E}[X^2Y^2] = \mathbb{E}[0] = 0$, so they are not independent.
3) The expectation of the last toss is 3.5, that means we should continue after second toss only if second toss is less than 3.5. That means if the second toss is $1, 2, 3$ we should continue, otherwise stop. The maximum expectation for last two tosses is 4.25, thus if the first toss is 5 or 6 we should stop, otherwise we continue. The maximum expectation for this problem is 4.67.
4) Ref: http://math.stackexchange.com/questions/14190/average-length-of-the-longest-segment
   This problem is discussed extensively in David and Nagaraja's Order Statistics (pp. 133-135, and p. 153).
   If $X_1, X_2, , X_{n1}$ denote the positions on the rope where the cuts are made, let $V_i = X_i - X_{i1}$, where $X_0 = 0$ and $X_n = 1$. So the V_i's are the lengths of the pieces of rope.
   The key idea is that the probability that any particular k of the V_i's simultaneously have lengths longer than $c_1, c_2, , c_k$, respectively (where $\sum_{i=1}^{k} c_i \leq 1$), is
$$(1 - c_1 - c_2 - \ldots - c_k)^{n-1}.$$
   This is proved formally in David and Nagaraja's Order Statistics, p. 135. Intuitively, the idea is that in order to have pieces of size at least $c_1, c_2, , c_k$, all $n1$ of the cuts have to occur in intervals of the rope of total length $1 - c_1 - c_2 - \ldots - c_k$.

If $V_{(1)}$ denotes the shortest piece of rope, then for $x \leq 1/n$,

$$P(V_{(1)} > x) = P(V_1 > x, V_2 > x, , V_n > x) = (1 - nx)^{n-1}.$$

Therefore,

$$E[V_{(1)}] = \int_0^\infty P(V_{(1)} > x)dx = \int_0^{1/n} (1 - nx)^{n-1} = 1/n^2.$$

David and Nagaraja also give the formula Yuval Filmus mentions (as Problem 6.4.2):

$$E[V_{(r)}] = 1/n \sum_{j=1}^{r} 1/(n - j + 1).$$

5) Consider the correlation matrix of X, Y, Z, we have

$$\begin{pmatrix} 1 & r & 0 \\ r & 1 & r \\ 0 & r & 1 \end{pmatrix}$$

This is a semi-definite matrix. So we need to have non-negative determinants of all its diagonal submatrics, i.e. $1 - r^2 \geq 0$ and $1 - 2r^2 \geq 0$. Hence, we have

$$-\frac{\sqrt{2}}{2} \leq r \leq \frac{\sqrt{2}}{2}.$$

6) (a) Denote the drunk man's position at time t as P_t, then we know P_t is a martingale and $P_t^2 - t$ is also a martingale. Denote stopping time τ as the first time the drunk man at position -1 or 99. Then we know a martingale stops at a stopping time is also a martingale. Thus if we denote p as the probability that the drunk man will first visit -1 and q as the probability that the drunk man will first visit 99, we have

$$p + q = 1$$

$$p \times (-1) + q \times 99 = 0.$$

Thus, we know $p = \frac{99}{100}$ and $q = \frac{1}{100}$. Since

$$E[P_\tau - \tau] = (-1)^2 \times p + 99^2 \times q - E[\tau] = 0,$$

thus we know $E[\tau] = 99$.

(b) If the left door is locked, then when the drunk man hit -1, he still need to move right for 100 steps. Since left and right are symmetric, we know this is equivalent that we have two doors at -101 and 99. Thus, by the same analysis as part (a), we know $E[\tau] = 9999$.

7) Ref: Element of Statistical learning

Ridge regression is one of the regularized regression methods. Ridge regression estimate $\hat{\beta}$ by minimizing

$$\sum_i (y_i - x_i^T \beta)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

where λ is the tuning parameter.

The solution to the ridge regression is

$$\hat{\beta} = (X^T X + \lambda I)^{-1} X^T y$$

Applying the ridge regression penalty has the effect of shrinking the estimates towards zero, introducing bias but reducing the variance of the estimates.

8) The LASSO, which is short for Least Absolute Shrinkage and Selection Operator, is a regression method that involves penalizing the absolute size of the regression coefficients. Its objective function is trying to minimize the following:

$$\|Y - X\omega\|_2^2 + \alpha\|\omega\|_1.$$

Compared to ordinary linear regression, the L1 penalization term can yield sparse models, and thus it can be used to do feature selection.

9) . This is classic St. Petersberg Paradox, helpful link at: https://en.wikipedia.org/wiki/St._Petersburg_paradox http://plato.stanford.edu/entries/paradox-stpetersburg/

10) Ref: Thinking in C++ P340
A default constructor will only be automatically generated by the compiler if no other constructors are defined. One case we want to define our own default constructor is, for example, to declare a private constructor, and never define it, to prevent the compiler from implicitly defining any others.

11) Problem only happens at instantiation of singleton

```
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;

std::mutex mtx;

class Singleton
{
private:
    static bool instanceFlag;
    static Singleton *single;
    Singleton(){
        //private constructor
    }
public:
    static Singleton* getInstance();
    void method();
    ~Singleton(){
        instanceFlag = false;
    }
};
```

```
bool Singleton::instanceFlag = false;
Singleton* Singleton::single = NULL;
Singleton* Singleton::getInstance()
{
    if(! instanceFlag){
        mtx.lock();
        if(! instanceFlag){
            single = new Singleton();
            instanceFlag = true;
        }
        mtx.unlock();
    }
    return single;
}
```

12) This is a dynamic programming problem. Let

global[i][j]: Max profit upto day i when we have completed at most j transactions

local[i][j]: Max profit upto day i when we have completed at most j transactions, *AND* the last transaction is to sell the stock at day j

The state transition functions are

global[i][j] = max(local[i][j], global[i-1][j])

local[i][j] = max(global[i-1][j-1] + max(diff, 0), local[i-1][j] + diff)

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

class Solution {
    int helper(vector<int> &prices) {
        int profit = 0;
        for (int i = 0; i < prices.size() - 1; i++) {
            profit = max(profit, profit + prices[i + 1] - prices[i]);
        }
        return profit;
    }
public:
    int maxProfit(int k, vector<int> &prices) {
        int len = prices.size();
        if (len == 0) { return 0; }
        if (k >= len) { return helper(prices); }

        // rolling array
        vector<int> max_local(k + 1, 0);
        vector<int> max_global(k + 1, 0);
        int diff;
        for (int i = 0; i < len - 1; i++) {
            diff = prices[i + 1] - prices[i];
            for (int j = k; j >= 1; j--) {
                max_local[j] = max(max_global[j - 1] + max(diff, 0), max_local[j] + diff);
                max_global[j] = max(max_local[j], max_global[j]);
            }
        }
        return max_global[k];
    }
};
```

13)
```
class Solution {
public:
    int singleNumber(int A[], int n) {
        int result = 0;
        for (int i = 0; i < n; i++) {
            result = result ^ A[i];
        }
        return result;
    }
};
```

14) The solution is found on this website: https://isocpp.org/wiki/faq/strange-inheritance#calling-virtuals-from-base. Yes. Its sometimes (not always!) a great idea. For example, suppose all Shape objects have a common algorithm for printing, but this algorithm depends on their area and they all have a potentially different way to compute their area. In this case Shapes area() method would necessarily have to be virtual (probably pure virtual) but Shape::print() could, if we were guaranteed no derived class wanted a different algorithm for printing, be a non-virtual defined in the base class Shape.

```
#include "Shape.h"
void Shape::print() const
{
    float a = this->area();  // area() is pure virtual
    // ...
}
```

15) To generate the random numbers, we first need to generate uniform distribution. Once we have uniform distribution, we can transform it to any distribution. There are two main methods to generate uniform distribution: physical method and computation method.

Computer can sensor some random phenomena, e.g., thermal noise, radio noise., audio noise, a computer can transform these random information to random number. For example, to mimic a fair coin, we can use last digit of the current time to module 2. However, physical method has asymmetries and systematic biases that make their outcomes not uniformly random. Also, the rate to generate is restricted due to the sampling limit of sensor.

The second method actually generate pseudo-random number. Such generator creates long runs of numbers with good random properties but eventually the sequence repeats. One of the most common pseudo-random number is the linear congruential generator, which uses the recurrence

$$X_{n+1} = (aX_n + b) \mod m.$$

a, b and m are large numbers which are prespecified.

To test the goodness, we can use goodness-of-fit tests, which include χ^2 test for discrete distribution and also KS test for continuous distribution. Also the generator should generate i.i.d sequence, so a good generator should have low autocorrelation.

16) First let's see how to mimic one dice. Let the result of a coin flip be 0 or 1, 3 tosses can be encoded as 3 bits where the i-th bit is the i-th toss. The encoded number ranges from 0 to 7, if we get 0 and 1 we restart immediately, otherwise map the result as a dice uniquely. The expectation of tosses needed is $\frac{11}{3}$.

    Notice that if we drop 0 and 7, the expectation of tosses is $4 > 11/3$. The reason is that the first two digits for 0 and 1 are the same, so if we restart we do it right after 2 tosses.

    Follow the same logic, if we mimic m dices, we can use n tosses of a coin where n is the smallest number s.t. $6^n < 2^n$. The outcomes can be encoded to 0 to $2^n - 1$. We drop 0 to $2^n - 6^m - 1$ and map the rest uniquely with the result of m dices. In this way, the expectation of tosses is minimized.

17) Idea: first, both robots move slowly to the right (right, left, right) until left robot hit 0 (right robot never hit zero), then left robot speed up (goright() only works for left robot) and finally they will meet

```
while(!at_zero()){
    go_right();
    go_left();
    go_right();
}
go_right();
```