Quiz 4

Group 2

1. Problem 1

If the drunk man takes his own seat, every one following him will take his/her own seat too. If the drunk man takes a wrong seat that is i, everyone between the drunk man and i-th guy will take his/her own but the i-th guy becomes the new drunk man.

That being said, for each drunk man, if he takes the 1st seat, everyone following him will take the correct seat including the last two guys. But if he takes 49th or 50th seat, the last two guys will never take the correct seats. The former probability is alway half of the later.

Thus the probability is 1/3.

2. Problem 2

For 5 girls, there are $C_5^2 = 10$ different two girls pairs. The question asks us to choose 5 out of 10 pairs so that each girls shows up twice. Let's note the girls with 'A, B, C, D, E'. The possible combinations are:

AB	AC	AD	AE
	BC	BD	BE
		CD	CE
			DE

To guarantee each girl appears twice, we have $C_4^2 = 6$ ways to choose 2 first row, $C_2^1 = 2$ ways to choose 1 from second row and no chose for the rest rows. This gives us $6 \times 2 = 12$ possible choices. Because all girls have equal chance to be at position of A,B,C,D,E. So the total number is $5! \times 12 = 1440$.

3. Problem 3

Similarly we have $C_7^2 = 21$ different pairs and can get table below.

AB	AC	AD	AE	AF	AG
	BC	BD	BE	BF	BG
		CD	CE	CF	CG
			DE	DF	DG
				EF	EG
					FG

Total possible choices $C_6^2 \times 1 + C_6^2 \times C_4^1 \times 1 + C_6^2 C_4^1 C_3^1 C_2^1 = 15 \times 31 = 465$.
The total number possible arrangements is $7! \times 465 = 2343600$

4. Problem 4

The two prices we are considering are:

$$S_{t_1} = S_0 \exp[(\mu_1 - \sigma_1^2/2)t_1 + \sigma_1 W_1] = \bar{S}_{t_1} e^{\sigma_1 W_1 + \sigma_1^2 t_1/2}$$
$$S_{t_2} = S_0 \exp[(\mu_2 - \sigma_2^2/2)t_2 + \sigma_2 W_2] = \bar{S}_{t_2} e^{\sigma_2 W_2 + \sigma_2^2 t_2/2}$$

The correlation we want is:

$$corr(S_{t_1}, S_{t_2}) = \frac{Cov(S_{t_1}, S_{t_2})}{\sqrt{Var(S_{t_1})Var(S_{t_1})}} = \frac{< S_{t_1}S_{t_2} > -\bar{S}_{t_1}\bar{S}_{t_2}}{\sqrt{Var(S_{t_1})Var(S_{t_1})}}$$

The variance can be calculated as:

$$Var(S_{t_1}) =< S_{t_1}^2 > - < S_{t_1} >^2 = S_0^2 e^{(2\mu_1+\sigma_1^2)t_1} - S_0^2 e^{2\mu_1 t_1} = \bar{S}_{t_1}^2 (e^{\sigma_1^2 t_1} - 1)$$

Similarly we have:

$$Var(S_{t_2}) = S_0^2 e^{(2\mu_2+\sigma_2^2)t_2} - S_0^2 e^{2\mu_2 t_2} = \bar{S}_{t_2}^2 (e^{\sigma_2^2 t_2} - 1)$$

and

$$< S_{t_1}S_{t_2} >= \bar{S}_{t_1}\bar{S}_{t_2} < e^{\sigma_2 W_1 + \sigma_2 W_2 + (\sigma_1^2 t_1 + \sigma_2^2 t_2)/2} >= \bar{S}_{t_1}\bar{S}_{t_2}$$

So we have $corr(S_{t_1}, S_{t_2}) = 0$. Brownian motion has no memory.

5. Problem 5

From the definition of martingale, we need to consider :

$$Exp(W_{t+1}^N | W_t^N) = Exp(W_t^N + NW_t dW_t^{N-1} + \cdots + dW_t^N | W_t^N)$$

Consider $N > 1$. If N is even, the last term in the series contributes $Exp(dW_t^N | W_t) = dt^{N/2}$ which is non-zero. If N is odd, the second in the series contributes $Exp(NW_t dW_t^{N-1} | W_t) = Exp(W_t) dt^{(N-1)/2}$ which is non-zero. So W_t^N is not martingale when N¿1.

Only when N=1 it is martingale.

6. Problem 6

Reference: Xinfeng Zhou's book, pg 149

Under risk neutral measure $dS = rSdt + \sigma S dW(t)$, random variable $V = 1/S$ follows geometric brownian motion:

$$dV = (-r + \sigma^2)Vdt - \sigma V dW(t)$$

We can apply Ito's lemma:

$$d(\ln V) = (-r + \frac{1}{2}\sigma^2)dt - \sigma dW(t)$$

Hence $E[V_T] = \frac{1}{S_t} e^{-rt+\sigma^2 t}$.

Discount the payoff we have:

$$V = \frac{1}{S_t} e^{-2rt+\sigma^2 t}$$

7. Problem 7

Reference: `https://en.wikipedia.org/wiki/Numerical_stability`
`https://en.wikipedia.org/wiki/Explicit_and_implicit_methods`

An algorithm for solving a linear evolutionary partial differential equation is stable if the total variation of the numerical solution at a fixed time remains bounded as the step size goes to zero

Explicit methods calculate the state of a system at a later time from the state of the system at the current time, while implicit methods find a solution by solving an equation involving both the current state of the system and the later one

8. Problem 8

The answer is yes.

Consider the time right before the rate cross 0.5 from below. If the total shoots is even $2n$, then the current rate is: $(n-1)/2n$. The next successful shoot makes it $n/(2n+1)$. This is the case the total shoots is odd.

If the total shoots is odd $2n+1$, then the current rate is: $n/(2n+1)$. The next successful shoot takes it to $n+1/(2n+2)$ which is 0.5.

So there will always be a moment the rate equals 0.5.

9. Problem 9

We need to consider two things: expected return and risk.

Consider the two strategy returns have different distribution/variance, it's not safe to use average return directly, instead we can use Welch Test to compare the expected return.

Knowing expected return, risk free return and variance, we can calculate Sharpe Ratio of each strategy. Sharpe ratio measures return per volatility. Higher sharpe ratio means expected higher return with risk.

To take a higher expected return or higher sharpe ratio is a question of risk preference.

10. Problem 10

We have three ways.

1. Use cross product of vectors. 2. Barycentric Technique 3. Say we have a triangle ABC and want to know if point P is inside of this triangle. What we can do is to express AP as the linear combination of AB and AC

$$(\vec{AP}) = a(\vec{AB}) + b(\vec{AC})$$

Solve for a and b. P is inside the triangle if 1. a,b¿0; 2. a+b¡1.

Solution 1:

```
# P10 of quiz10
# reference: http://www.blackpawn.com/texts/pointinpoly/
import numpy as np

# check if point (x,y) is inside triangle defined by
# three verticies v1(x1, y1), v2(x2, y2), v3(x3, y3)
def SameSide(p1,p2, a,b):
    cp1 = np.cross(b-a, p1-a)
    cp2 = np.cross(b-a, p2-a)
    if cp1.dot(cp2) >= 0: return True
    else return False
```

```
def PointInTriangle(p, a,b,c):
    if SameSide(p,a, b,c) and SameSide(p,b, a,c)
        and SameSide(p,c, a,b): return True
    else return False
```

Solution 3:

```
struct Double3
{
    double x = 0;
    double y = 0;
    double z = 0;
public:
    Double3();
    Double3(double a, double b, double c);
};
Double3 add(Double3 A, Double3 B);
Double3 subtra(Double3 A, Double3 B);
Double3 cross(Double3 A, Double3 B);
double dot(Double3 A, Double3 B);
Double3::Double3()
{}
Double3::Double3(double a, double b, double c)
{
    x = a;
    y = b;
    z = c;
}
Double3 add(Double3 A, Double3 B)
{
    Double3 temp;
    temp.x = A.x + B.x;
    temp.y = A.y + B.y;
    temp.z = A.z + B.z;
    return temp;
}
Double3 subtra(Double3 A, Double3 B)
{
    Double3 temp;
    temp.x = A.x - B.x;
    temp.y = A.y - B.y;
    temp.z = A.z - B.z;
    return temp;
}
Double3 cross(Double3 A, Double3 B)
{
    Double3 temp;
    temp.x = A.y*B.z - A.z*B.y;
    temp.y = A.z*B.x - A.x*B.z;
    temp.z = A.x*B.y - A.y*B.x;
    return temp;
}
double dot(Double3 A, Double3 B)
{
    return A.x*B.x + A.y*B.y + A.z*B.z;
}
bool inTri(Double3 A, Double3 B, Double3 C, Double3 P)
{
    Double3 AB = subtra(B, A);
    Double3 AC = subtra(C, A);
    Double3 AP = subtra(P, A);

    double norm = dot(AB, AB)*dot(AC, AC) - dot(AB, AC)*dot(AB, AC);
    double a = (dot(AP, AB)*dot(AC, AC) - dot(AP, AC)*dot(AC, AB)) / norm;
    double b = (dot(AP, AC)*dot(AB, AB) - dot(AP, AB)*dot(AC, AB)) / norm;

    return (a > 0 && b > 0 && a + b < 1);
}
int main()
{
    Double3 A = { 0, 0, 0 };
    Double3 B = { 1, 0, 0 };
    Double3 C = { 0, 1, 0 };
    Double3 P = { 0.25, 0.25, 0 };
    std::cout << inTri(A, B, C, P)<<std::endl;
    char wait;
    std::cin >> wait;
    return 1;   72.      }
```

11. Problem 11

This can be done by checking the pair of chars that located at opposite position.

```
#include <iostream>
#include <string>

using namespace std;

bool checkPal(string *s)
{
          int i=0;
          while(i<s->length()/2)
    {
          if ((*s)[i]==(*s)[s->length()-1-i])
          {
              i++;
          }
          else
          {
              return false;
          }
    }
    return true;
}

void main()
{
    string s("112211");
    cout<<checkPal(&s)<<endl;
}
```

12. Problem 12

12. DP solution $O(N^2)$ time complexity, O(N) space complexity. Basic idea is to use an array of same size as the data to store the length of the longest decreasing array end with arr[i]. And that can be done by finding the max(table[i],table[j]+1) for all $arr[j] > arr[i], j < i$.

```
#include <iostream>
#include <math.h>
#include <vector>

using namespace std;
```

```
void findLDS(vector<double> *arr, vector<int>* table, vector<double>* result
{
    int i, j;
    int max = 0;

    for (i = 0; i < (*arr).size(); i++)
    {
        for (j = 0; j < i; j++)
        {
            if ((*arr)[j] > (*arr)[i])
            {
                (*table)[i] = (*table)[i]>(*table)[j] + 1 ? (*table)[i] : (*
            }
        }
        max = max > (*table)[i] ? max : (*table)[i];
    }

    for (i = (*table).size() - 1; i >=0; i--)
    {
        if ((*table)[i] == max)
        {
            (*result).push_back((*arr)[(*arr).size() - i - 1]);
            max--;
        }
    }
}
int main()
{
    vector<double> arr={5,4,4,3,3,2,2,1,1};
    vector<int> table(arr.size(), 1 );
    vector<double> result;
    int i;
    findLDS(&arr, &table, &result);
    for (i = 0; i < table.size(); i++)
    {
        cout << table[i] << ',';
    }
    cout << endl;
    for (i = 0; i < result.size(); i++)
    {
        cout << result[i] << ',';
    }
    return 1;   50.      }
```

There is a O(NlogN) algorithm using binary search + DP. Reference: `https://swiyu.wordpress.com/2012/10/15/longest-increasing-subarray/`

13. Problem 13

This is a depth first search problem. Use back-tracking algorithm.

```
// A Backtracking program  in C++ to solve Sudoku problem
#include <stdio.h>

// UNASSIGNED is used for empty cells in sudoku grid
#define UNASSIGNED 0

// N is used for size of Sudoku grid. Size will be NxN
#define N 9

// This function finds an entry in grid that is still unassigned
bool FindUnassignedLocation(int grid[N][N], int &row, int &col);

// Checks whether it will be legal to assign num to the given row,col
bool isSafe(int grid[N][N], int row, int col, int num);

/* Takes a partially filled-in grid and attempts to assign values to
all unassigned locations in such a way to meet the requirements
for Sudoku solution (non-duplication across rows, columns, and boxes) */
bool SolveSudoku(int grid[N][N])
{
    int row, col;

    // If there is no unassigned location, we are done
    if (!FindUnassignedLocation(grid, row, col))
        return true; // success!

    // consider digits 1 to 9
    for (int num = 1; num <= 9; num++)
    {
        // if looks promising
        if (isSafe(grid, row, col, num))
        {
            // make tentative assignment
            grid[row][col] = num;

            // return, if success, yay!
            if (SolveSudoku(grid))
                return true;

            // failure, unmake & try again
            grid[row][col] = UNASSIGNED;
        }
    }
    return false; // this triggers backtracking
}

/* Searches the grid to find an entry that is still unassigned. If
found, the reference parameters row, col will be set the location
that is unassigned, and true is returned. If no unassigned entries
remain, false is returned. */
bool FindUnassignedLocation(int grid[N][N], int &row, int &col)
{
    for (row = 0; row < N; row++)
        for (col = 0; col < N; col++)
            if (grid[row][col] == UNASSIGNED)
                return true;
    return false;
}

/* Returns a boolean which indicates whether any assigned entry
in the specified row matches the given number. */
bool UsedInRow(int grid[N][N], int row, int num)
{
    for (int col = 0; col < N; col++)
        if (grid[row][col] == num)
            return true;
    return false;
}

/* Returns a boolean which indicates whether any assigned entry
in the specified column matches the given number. */
bool UsedInCol(int grid[N][N], int col, int num)
{
    for (int row = 0; row < N; row++)
        if (grid[row][col] == num)
```

```
            return true;
    return false;
}

/* Returns a boolean which indicates whether any assigned entry
within the specified 3x3 box matches the given number. */
bool UsedInBox(int grid[N][N], int boxStartRow, int boxStartCol, int num)
{
    for (int row = 0; row < 3; row++)
        for (int col = 0; col < 3; col++)
            if (grid[row + boxStartRow][col + boxStartCol] == num)
                return true;
    return false;
}

/* Returns a boolean which indicates whether it will be legal to assign
num to the given row,col location. */
bool isSafe(int grid[N][N], int row, int col, int num)
{
    /* Check if 'num' is not already placed in current row,
    current column and current 3x3 box */
    return !UsedInRow(grid, row, num) &&
        !UsedInCol(grid, col, num) &&
        !UsedInBox(grid, row - row % 3, col - col % 3, num);
}

/* A utility function to print grid  */
void printGrid(int grid[N][N])
{
    for (int row = 0; row < N; row++)
    {
        for (int col = 0; col < N; col++)
            printf("%2d", grid[row][col]);
        printf("\n");
    }
}

/* Driver Program to test above functions */
int main()
{
    // 0 means unassigned cells
    int grid[N][N] = { { 3, 0, 6, 5, 0, 8, 4, 0, 0 },
    { 5, 0, 0, 0, 0, 0, 0, 0, 0 },
    { 0, 8, 7, 0, 0, 0, 0, 3, 1 },
    { 0, 0, 3, 0, 1, 0, 0, 8, 0 },
    { 9, 0, 0, 8, 6, 3, 0, 0, 5 },
    { 0, 5, 0, 0, 9, 0, 6, 0, 0 },
    { 1, 3, 0, 0, 0, 0, 2, 5, 0 },
    { 0, 0, 0, 0, 0, 0, 0, 7, 4 },
    { 0, 0, 5, 2, 0, 6, 3, 0, 0 } };
    if (SolveSudoku(grid) == true)
        printGrid(grid);
    else
        printf("No solution exists");

    return 0;  132.     }
```

14. Problem 14

This is a dynamical problem. max_value[i][j]=max(max_value[i][j-1]+value[i][j], max_value[i-1][j]+value[i][j]). Use a 2d-array to record local max_value so we don't need to do repeated calculation.

```
#include <iostream>

using namespace std;
int max(int a, int b)
{
    return a > b ? a : b;
}

int main()
{
    const int N = 3;
    const int M = 3;
    int board[N][M];
    int sum[N][M];
    int i, j;
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < M; j++)
        {
            board[i][j] = rand() % 10;
```

```
                sum[N][M] = 0;
            }
        }
        for (i = 0; i < N; i++)
        {
            for (j = 0; j < M; j++)
            {
                cout<<board[i][j]<<' ';
            }
            cout << endl;
        }
        //assume we need to find the max sum for reaching [i][j], then the only
        //optimum sum for reaching [i-1][j] and [i][j-1] we can then decide whic
        sum[0][0] = board[0][0];
        for (i = 0; i < N; i++)
        {
            for (j = 0; j < M; j++)
            {
                if (i == 0 && j>0)
                {
                    sum[i][j] = board[i][j] + sum[i][j - 1];
                }
                else if (j == 0 && i>0)
                {
                    sum[i][j] = board[i][j] + sum[i-1][j];
                }
                else if (i > 0 && j>0)
                {
                    sum[i][j] = board[i][j] + max(sum[i - 1][j], sum[i][j - 1]);
                }
            }
        }
        cout << sum[i - 1][j - 1];
}
```