Problem Bejeweled

(a) how to store data

- Create an `enum` to store different types of gems.
- Store the whole board using an 8x8 2-dimensional array of `enum`.
- Create a vector to store multiple possible swaps

(b) Description methodology

We would like to brute force and keep track of a max number of score along the process.

During the process, for each row and for each column, pretend to swap a pair of gems and count the max score. Update the variable to keep track of the max score and revert the swap so that the board will be in its original condition.

The number of check we need: 100 * 8 * 8 * (the check to get score in this swap)

(c) Pseudocode if needed

```
max_score <- 0
vector_possibleSwaps = []
for r in each row:
    for elements in the 1st to the second last in row r:
        Pretend to swap the current gem with the gem to the right of it;
        count <- sum of same consequence gems in the current row and current column (w/o overlapping gems);
        if (count > max_score):
            vector_possibleSwaps.clear()
            max_score <- count;
            vector_possibleSwaps.push_back(row, column, the direction 'R' or 'D')
        else if (count == max_score):
            # keep track of this new swap as a possible swap in the vector
            vector_possibleSwaps.push_back(row, column, the direction 'R' or 'D')

        revert the swap so that the board remains the original one;

```

```
repeat the same procedure above for each column
```

Problem Binary

(a) how to store data

1. Preprocess an array A to store the range of K with different length binary number (more on part(b))

(b) Description methodology

- DP with 1d array

1. Preprocess an array A where A[i] stores the first digit number of value 2^i
    - ex.
        - A looks like [1 2 6 18 50 .....]
        - i = 2, A[2] = 6, indicate value 4 (2^2) 's first digit number is 6
        - 0 1 10 11 100, => $4_{10} = 100_2$, where 1 in 100_2 in the 6_{th} digit in binary sequence
    - Base case:
        - A[0] = 1, A[1] = 2
    - State function
        - $A[x] = (2^x - 2^{x-1})x + A[x-1]$
        - ex. x=3
            - A[3] = $(2^3 - 2^2) * 3 + A[2] = 18$, where A[2] = 6
    - Stop allocate A until A[n] >= max k = 10^9
2. Recover
    - ex. input k = 30
    - Search first element in A that A[i] > k, in this case A[3] = 18, A[4] = 50, i = 4
    - k falls between 3 and 4 (i -1 to i)
        - between 18_{th} digit and 50_{th} digit are all 4 digits number
        - 1000, 1001, 1010, ... ,1111, there are
            - $(50 - 18)/4 = 8$

- $(A[i] - A[i-1])/i$ decimal numbers
- 30_{th} digits falls in (30-18)/4 + 1= 4_{th} decimal numbers
    - **To sum up**, input k falls into$\{2^{i-1} + (k - A[i-1])/i\}$th decimal number
- Then print the according bit in binary

(c) Pseudocode if needed

```
% Init array
i = 2; % iterator
A[0] = 1, A[1] = 2
while element in A < 10^9
    A[i] = (2^i - 2^(i-1)i + A[i-1];
    i++
end while

% read each k
for each input k
    find first element in A larger than k;
    j = index of this element;
    k falls into decimal value D = 2^(j-1) + (k-A[j-1])/j;
    D_b = binary of D;
    print according bit;
end for

```

Problem CitySlickers

(a) how to store data

- Generate a adjacency matrix to present each edge and vertices.
- Creat a class to contain the weight of corresponding edges

(b) Description methodology

1. Store the graph

2. Treat weights of mountain as 1 and others as 0
    - Now, it becomes a question to find the shortest path of a weighted undirect graph. Weights are either 0 or 1.
3. Using Dijkstra's shortest path algorithm to find the solution

(c) Pseudocode if needed

The implementation of Dijkstra's algorithm codebook can be found below.

https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/

Example:

Convert the following input into code:

```
...n.
.rmnx
xxvn.
xkzr.
.y...
```

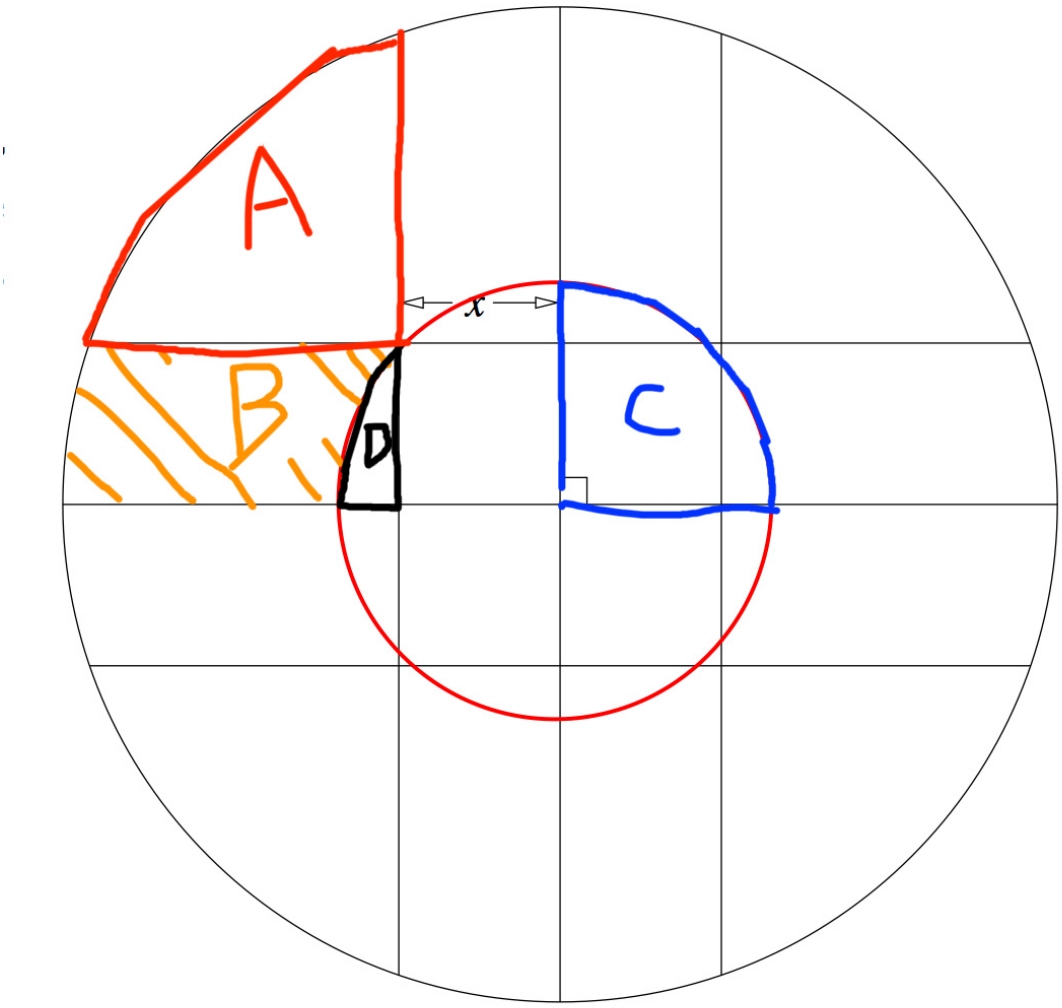
```
// Diskstra's algorithm cal be found in the codebook link above.
void dijkstra(int graph[V][V], int src);

// Input and how to run:
int graph[5[5] = {
    {0, 0, 0, 1, 0},
    {0, 0, 1, 1, 1},
    {1, 1, 1, 1, 0},
    {1, 1, 1, 1, 0},
    {0, 1, 0, 0, 0}
};
dijkstra(graph, 0);

```

Problem Cutting Pizza

(a) how to store data

No data store in this question

(b) Description methodology

- Draw a circle inside with r = dignonal of square
- We have A B C D areas as follow:
  - 

- Denote
  - S_{oc} = area of outer circle
  - S_{ic} = area of inner circle
  - S_{sq} = area of small square
- We concluded 3 equations:
  - $A + 2B + C = 1/4S_{oc}$
  - B+D = A (from question)
  - D = $1/4S_{ic} - 1/2S_{sq}$
- D, C are known, the only unknown in 3 equations is B, which we can easily solve

- $B = 1/12S_{oc} - 1/6S_{ic} - 1/4S_{sq}$
- FInal Area = B+D = $1/12S_{oc} + 1/12S_{ic} - 1/4S_{sq}$

(c) Implementation:

Solve equation with X and `double pi = acos(-1)`

Problem DoingLaundry

(a) how to store data

- Read each test case into a vector

(b) Description methodology

- Read each test case into a vector
- Sort vector in decreasing order
- Assuming all washing were done one after one
- Maintain a variable to record each job's start drying time
    - if `a >(1+i)*30` , which means job `i-1`'s **drying** finished after job `i`'s **washing**
        - So job i's drying can only start after job `i-1`'s **drying**
    - if `a < (1+i)*30`, job i **drying** can start right after its **washing**
    - a = max(a, (1+i)*30) + t[i];

(c) Pseudocode if needed

```
int main() {
  int n, x;
  while (cin >> n && n) {
    vector<int> t;
    for (int i = 0; i < n && cin >> x; ++i)
      t.push_back(x);

    sort(t.rbegin(), t.rend());
    int a = 0;
```

```
    for (int i = 0; i < n; ++i)
      a = max(a, (1+i)*30) + t[i];
    cout << a/60 << ":" << (a%60 < 10 ? "0" : "")
     << a%60 << endl;
  }
}

```

Problem LinkingLogos

(a) how to store data

- Create 2 arrays to emunerate stacking up legos: One array is to store legos on the lower level while the other one is for the upper level.
    - The maximum size per array is 15×5.
    - Arrays will be initialized as 0 for all elements.

(b) Description methodology

1. Read in the data.
2. Traverse and calculate the sum of all elements in one line.
    - **A case to prune half of cases**: If the sum is not divisible by 2 then we can assert that these legos will not be linked together as described in the question.
3. Calculate the permutations of all building blocks.
    - We will put blocks to the lower and upper level arrays alternately.
    - **Case we can prune**: If the front part of the sequence is the same as the remaining part (1x1, 1x3, 1x1, 1x3), we can be sure that this does not work as the lego we built will not be able to link together.
4. Loop through the upper and lower arrays to check whether indeed it can form a linked lego
    - by checking if the element in the upper level is **not** same as the one in the lower level **in the same index** (ignoring overlapping).
    - Print "yes" in this case, otherwise print "no".

Examples:

*

```
Upper level: [ 1 2 2 3 3 3 ]
Lower level: [ 1 3 3 3 2 2 ]

This denotes we have
    3 blocks on the upper level: 1x1, 1x2, 1x3
    3 blocks on the lower level: 1x1, 1x3, 1x2

This case is not OK because the first index we are checking is the same.
This denotes that the lego will be separated after the first index.
```

*

```
Upper level: [ 1 3 3 3 ]
Lower level: [ 3 3 3 1 ]

This is OK.
```

*

```
Upper level: [ 1 3 3 3 ]
Lower level: [ 1 3 3 3 ]

This is not OK and can be pruned because the sequence generated is 1 3 3 3 1 3 3 3,
where the first half of the sequence is the same as the second half.
```

In this examples, when we are checking index-wise, we need to account for overlapping blocks so that they will be considered correctly.

Problem Longshot

(a) how to store data

- Store each test case into a 2d array win

(b) Description methodology

1. Read the input
2. **Permutation** for all winning tree
    1. Assign an array A with length 16, initialize with all -1

    2. calculate all permutations of A having 8 1s, indicating **8** corresponding players go into **Quarterfinals**
    3. Use all permutations in step 2.2, each calculate all permutations of array having 4 1s, indicating 4 corresponding players go into **Semifinals**
    4. Use all permutations in step 2.3, each calculate two players of **finals**
3. During calculating of permutation, if the permutation doesn't satisfy input, skip this permutation
    1. ex. if a permutation of 8 players doesn't satisfy input, we don't need to calculate its 4 players permutations.
4. If a player stays in final two players then return true since its previous 8 and 4 permutation **survived** for input

(c) Pseudocode if needed

```
play(win, m1, m2){
    check who wins, denote I winner's index
    result = 16 bits binary variable where I=1 and everywhere else = 0
    //ex. I = 2, result = 0100000000000000
    return result
}

win4(win,P4){
    for all possible final match up m in P4
        result = result or play(win,m1, m2);
    return result;
}
win8(win,P8){
    for all permutation P4 of 4 1s
            result = result or win4(win,P4);
    return result;
}

main
    for each test case
        read data in array win
        result = 0;
        for all permutation P8 of 8 1s
            result = result or win8(win,P8);
        endfor
        for i = 0 to 15
            if ((result>>i)&1)
               output result;
        endfor
    endfor
```

Problem PrefixGoodness

(a) how to store data

- Create a suffix array of all the strings so that we can calculate the common prefix of 2 strings

(b) Description methodology

1. Compute all the prefixes of the strings
2. For each prefix, determine how many strings have the same prefix
3. Also record the length of the prefix
4. Compute the prefix goodness b y multiplying the length of the prefix by the number of strings having the same prefix

(c) Pseudocode if needed

```
all_prefixes = []
for string in all_strings:
    sa = suffixArray(string)
    prefixes = sa.findPrefix()
    all_prefixes.append(prefixes)

largest_goodness = 0
for prefix in all_prefixes:
    num = determine how many strings have the same prefix
    length = len(prefix)
    prefix_goodness = num * length

    if prefix_goodness > largest_goodness:
        largest_goodness = prefix_goodness
```

Problem QuantumTeleporters

(a) how to store data

1. A 1-D array of state A to store each node whose state is A.
2. A 1-D array of state B to store each node whose state is B.
3. A 2-D array that stores a pair of <int, int> to identify the state of each node.
4. A queue to store the nodes and then traverse the graph as the BFS does.
5. A variable that maintain the current minimum distance.
6. A 3-D array to store the weight of each edge since state is 2-D.

(b) Description methodology

1. Read the input
2. Using a queue as the data structure
    - Dequeue next new one
    - Decide if the pair is a valid state
    - Ignore the pair if it is invalid
    - Otherwise, append it to the tail of the queue
    - Update and compare the current shortest distance with the one that after adding the current weight
3. Comparing and output results by iterating all states'' combination.

(c) Pseudocode if needed

```
for TC in range(T) # <-- reading input based on test case number
    for i in range(N)
        # initialize the connector array

    for i in range(M)
        # initialize the current shortest distance for each states

    while (queue != empty)
        dequeue the _next_ avaiable item

        for each neighbour of item
            if (item.states != neighbour.states)
                invalid;
                continue;

        else
            for each neighbour of item
                shortest_minimum = max(shortest_minimum, previous_distance + weight(item, neighbour))

            queue.add(neighbour)
```

```
21
22      print result
```

Problem TennisProbability

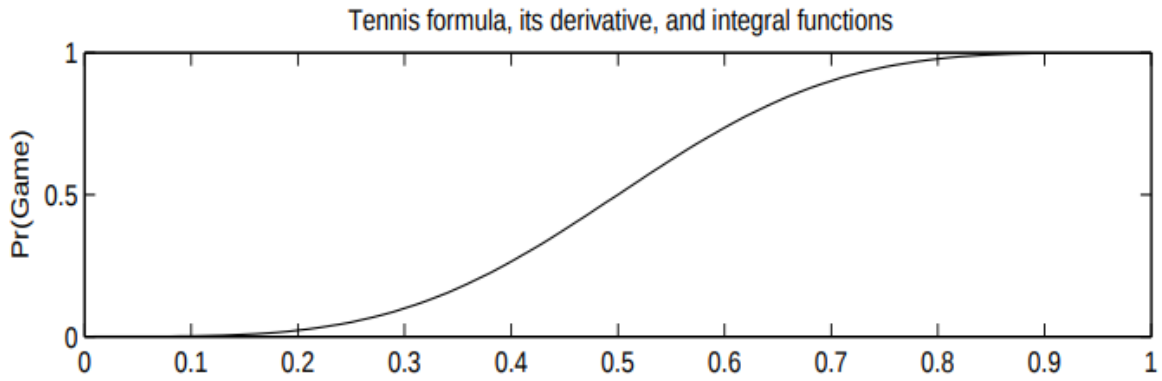
(a) how to store data

We do not need to store any data for this problem.

(b) Description methodology

According to a formula presented in page 4 in this document, the probability of winning a game given the probability p of winning a single point is $Pr(Win\ game) = p^4 + 4p^4(1-p) + 10p^4(1-p)^2 + 20p^3(1-p)^3 \times \frac{p^2}{1-2p(1-p)}$

The probability density function plot will look like this:



(c) Pseudocode if needed

Problem ZurchTrees

(a) how to store data

- Adjacent list with Integer, Set to store the graph
- Queue to store nodes

(b) Description methodology

To find the minimum number of policy is to find the number of common ancestors in the graph.

1. Read the input and generate the corresponding adjacency list
2. Using BFS to traverse the graph and count the number of edges for each node as degrees
3. Print the number of degrees that is larger than 3. Then we believe these nodes are common ancestors.

(c) Pseudocode if needed

```
for each input
  initializ the adjacency list <Integer, Set<Integer>>

  # BFS
  while (queue != empty)
    BFS to count the number of degrees for each nodes

  for each degree
    print the number of degrees that are larger than or equal to 2.
```