TASK 1: Algorithmic Design

Part A:

// As River only has a single rock to determine d_max:
// **Assumption:** if the ice breaks at i meters from River's current position,
// the maximum safe distance to the opening will be i-1 meters.

// Pseudocode

```
function FindMaxDist(D)
   // Determine the maximum safe distance towards opening by linear search
   // Input: The initial distance D between River and the opening
   // Output: if the ice breaks while checking, return the largest safe distance d_max.
   // Otherwise return the longest path River can walk until reaching the opening
      d_max ← 0
      for i ← 1 to D-1 inclusive do
         if Throw(i) return True then
            d_max ← i-1
            return d_max
   return D -1
```

Part B:

// Think about the growth rate that is slower than O(D) through:
// $O(1) < O(\log n) < O(n^{\varepsilon}) < O(n)$, where $\varepsilon < 1$.
// Construct the time complexity of worst case with two sections via using each rock.
// Since O(f(n)+g(n)) = O(max{f(n), g(n)}), the number of throws for each rock must not be computed by linear relation.

// $C_{worst}(T(2,D))=O(D^{1/2})+O(D^{1/2}) \rightarrow O(D^{1/2})$

//Pseudocode

```
function FindMaxDistSqrt(D)
    // Determine the maximum safe distance towards opening by throwing each rock
    // in the multiple of square root of D meters
    // Input: The initial distance D between River and the opening
    // Output: if the ice breaks while checking, return the largest safe distance d_max.
    // Otherwise return the longest path River can walk until reaching the opening


    // Round up the interval to integer
    base_gap ← round(sqrt(D))
    d_max ← 0

    for i ← 1 to ceil(D/base_gap) do
        // The first rock breaks the ice
        if Throw (i * base_gap) return True then
            // checking the distance between River's current position and the first hole
            // by linear search
            for j ← (i-1) * base_gap +1 to i * base_gap -1 inclusive do
                if Throw (j) return True then
                    d_max ← j-1
                    return d_max

            // checking the distance between the latest landed position of the first rock
            //and the opening since the ice has not been broken by linear search
            for j ← (i -1) * base_gap + 1 to D-1 inclusive do
                if Throw (j) return True then
                    d_max ← j-1
                    return d_max

    return D - 1
```

Part C:

(1.) Best Case: **O(1)**

The ice breaks at the distance of 1 meter from River's current position, which takes single implementation for each rock as she can only throw the rock in 1 meter increments:

First rock: she throws one rock away with the distance of 1 * sqrt(D)=1 meters, and the ice breaks. ->O(1)

Second rock: she then throws another rock 1 meter away, and the ice also breaks. ->O(1)

(2.) Worst Case: **O($D^{1/2}$)**

The ice breaks at the distance D-1 meters from River's current position, we need check through both rocks to compute the total number of throws(sqrt(D)+ sqrt(D)):

First rock: she throws one rock in sqrt(D) times, ice is not broken yet. -> O($D^{1/2}$)

Second rock: linear search in at most sqrt(D) times util find d_max -> O($D^{1/2}$)

(3.) Average Case: **O($D^{1/2}$)**

The probability of breaking the ice is equal to p (0<=p<=1) and the probability of breaking the ice in the i^{th} position within the loop is the same for every i.

Part D:

//To find the index of maximum value that is smaller than the search key:
//Yes, it is possible to implement a Binary-search algorithm, therefore the time complexity will be O(log n).

//pseudocode

```
function binary_search_max(D)
 // Determine the maximum safe distance towards opening by throwing each rock
 // by binary search
 // Input: The initial distance D between River and the opening
 // Output: Return the largest safe distance d_max which is updated by "low"

 //Initialize interval of search space inclusively
 low ← 0
 high ← D
 d_max ← 0
 while low < high do
        mid ← (low+high)/2
        //check the safe distance within the first-half path
        if Throw (mid) return True then
               high ← mid -1
        //check the safe distance within the second-half path
        else if Throw (mid) return False then
               low ← mid +1
 d_max ← low
 return d_max
```

Task 2: C Problem

Part B: Describe the time complexity of the algorithm in terms of S,L,T.

The time complexity is: O(S+L+T)

Explanation:
In this part, the graph-traversal algorithm DFS is applied. We know that this algorithm will have time efficiency $O(|V|^2)$ for the adjacency matrix, given by nested-looping every input point and its adjacent points.

1. Iterating the each input point, we need to check every situation of this input point. Hence it will take **repeated (S+L+T) times** within the mapValue function loop by checking whether the point on the map is SEA, ISLAND or TREASURE.
2. Iterating the adjacent points of the input point in the DFS loop will take constant time of **6 operations**. (Number of maximum adjacent points = 6)

Therefore, the total time complexity is: O(6 * (S+L+T)) **= O(S+L+T)**

Part C:

(i.) As the extra array stores the locations of each treasure, there is no need to visit and check every point on the map and thus it saves time. Therefore, the new algorithm would be:

Firstly, looping through **every point from the treasure array** instead of looping through all the isOnMap() points. Then, still use the same strategy to visit every adjacent point connected to the input point on the island. Hence, it will neither visit the islands without treasure nor visit the point at the location of SEA which is unconnected (not an adjacent point) to the islands with treasure.

Here is the brief **pseudocode**:

```
function mapValue(map)
    //Create an array to update if the treasure point has been visited
    malloc an 2D-array   treaVisitArr[][]
    totalValue ← 0
    for each point in treaVisitArr[][] do
        if treaP is unvisited and treaP value >= 0 then
            count ← 0
            value ← 1
            DFS(map, treaP, &value, &count, treaVistArr)
            value ← count * value
            totalValue ← totalValue + value
    return toalValue



function DFS(map)
    //Mark the input treasure point as visited
    visited(treaP)
    value ← value * value of treaP
    count ← count + 1
    //Check the adjacent point of this treasure point on the island
    if treaAdj is unvisited and treaAdj value >= 0 then
        DFS(treaAdj)
```

(ii.) Best Case: **O(T)**

All the treasures are located on the same island, we only need to use the recursive DFS function once. Also, when implementing DFS, every point on this island has a treasure. Therefore, the cost of best case is equal to the number of treasures =O(T).

(iii.) Worst Case: **O(T+L)**

All the treasures are located separately on different islands. Hence, in order to check the treasure on every single island, we need to visit all locations on the map except SEA. The cost of such operation will take O(T+L).

(iv.) As N_T denotes the average number of treasures on the island, I_s denotes the average number of points on the island, where there are treasures located on the island in both cases. In addition, the definition of average case implies that all the treasures are located randomly on different islands. Based on such distribution, some islands may not have treasure whereas other islands may have many.

Therefore, we calculate:

the **expected number of islands which have treasure located: T/ N_T**

multiplied by

the **average size of islands which have treasure locates: I_s**

to get the complexity: the total number of points on all islands with treasures.

The time complexity is: **$O(T/ N_T * I_s)$**

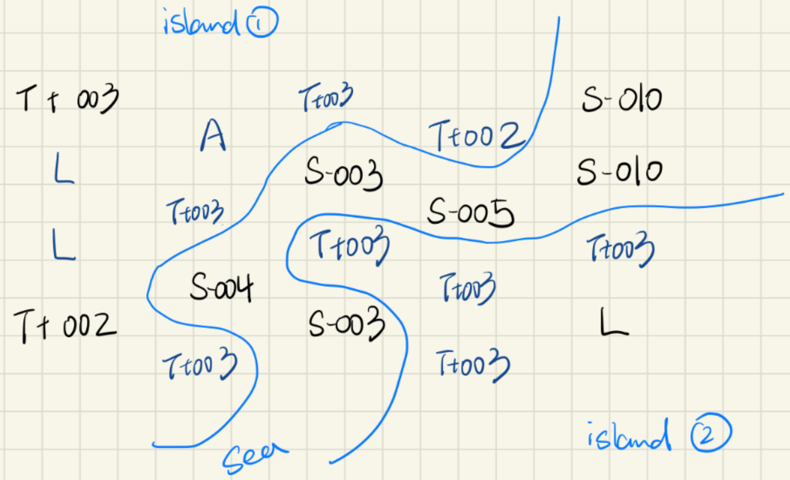
(v.) To explore as many points, the original algorithm requires us to visit every single point on the map until find the total value of the treasure, the location of points can be random.

However, when exploring with the new algorithm, we only consider the points which are already stored in the treasure array. In order to visit every point and check every situation, we expect to establish a model that (1.) every island will have at least one treasure located, because we need to visit all the islands; (2.) every sea must being adjacent (connect) to an island, because we need to visit and check all different type of locations on the map.

Here is an example of a map:

- Map Dimension : 5 4
- Map line 1 : 3 100 3 2 -10
- Map line 2 : 0 3 -3 -5 -10
- Map line 3 : 0 -4 3 3 3
- Map line 4 : 2 3 -3 3 0

Graph:



T : treasure + value
S : sea + deepth
A : airport
L : empty land

Require :
(1) S is connected with T
(2) every section must has at least one T+value

Part D: If A is the number of airports on the map, find the worst case time complexity in terms of S, L, T, A. Briefly explain your algorithm and data structures used to justify your time complexity.

Here is a brief pseudocode to explain how the **Dijkstra algorithm** is fulfilled:

```
function getShortestPath(int ** G, int s, int size, int *cost, int *pred)
    //Create a priority queue and update the vertex form G
    int u, v
    //Array initialization
    Arr ← make_array(visit, cost)                   // O(V)
    pq_node_t *pQ ← makePQ(G)                       //Heap Complexity: O(V * log(V))
    while (!empty (pQ)) do                          // V times comparisons
        u ← pull(pQ)                                // pull a single vertex: O(log V)
        // Situation when the vertex is at the airport
        if u is airport then                        // A times comparisons
            for each vertex on the map do           // V times comparisons
                if vertex is airport then           // A time comparisons
                    calculate cost                  // O(1)
                    insert the vertex to the queue  // O(log V)

        //Check the adjacent vertex to the input
        for each v connected to u do                // max number of v = 6
            calculate cost                          // O(1)
            insert v to the queue                   // O(log V)
```

The worst case time complexity is computed by changing V in terms of S,L,T,A:
The priority queue has the time complexity of O(vlog v), therefore:

(1.) Array initialization: the total number of points = O (S+L+T+A)
(2.) Create the priority queue: O((S+L+T+A) * log(S+L+T+A))
(3.) Pull function with priority queue: O((S+L+T+A) * log(S+L+T+A))
(4.) Check if the input point is airport: O(A*(S+L+Y+A) + A*A*log(S+L+T+A)), as there are A times operations in outer loop and A times operation in inner loop.
(5.) Check if the adjacent points of the input are airports: O((S+L+T+A)*6*log(S+L+T+A)), as there are 6 times operation

Hence, by combining all the time cost together,

O(S+L+T+A**)** + **O(**(S+L+T+A)*log (S+L+T+A)**)** + **O(**(S+L+T+A)*log (S+L+T+A)**)** +**O(**A*(S+L+T+A) + A*A*log(S+L+T+A)**)**+ **O(**(S+L+T+A) * 6 * log (S+L+T+A)**)**=**O(**(S+L+T+A) * (A +log(S+L+T+A)) + A*A log(S+L+T+A)**)**

we get the **time complexity** of worst case:

O((S+L+T+A) * (A +log(S+L+T+A)) + A^2 log(S+L+T+A))

Part E: What is the worst-case time complexity of this algorithm? To get full credit you must find the algorithm with the best worst case time complexity.

Here is a brief pseudocode for the modified algorithm(Changing the Dijkstra algorithm):

Idea: the algorithm in part E will have slightly faster time efficiency than part D by updating the "checking whether the searching vertex is airport " section, as the airports are defined within the function. Now, iteratively visiting the **points in airport array only** instead of the whole map.

```
function getShortestPath(int ** G, int s, int size, int *cost, int *pred)
    //Create a priority queue and update the vertex form G
    int u, v
    //Array initialization
    Arr ← make_array(visit, cost)                 // O(V)
    pq_node_t *pQ ← makePQ(G)                     //Heap Complexity: O(V * log(V))
    while (!empty (pQ)) do                        // V times comparisons
        u ← pull(pQ)                              // pull a single vertex: O(log V)
        // Situation when the vertex is at the airport
        if u is airport then                          // A times comparisons
            for each vertex in the airport array do   // numAirports times
                    calculate cost                    // O(1)
                    insert the vertex to the queue    // O(log V)

        //Check the adjacent vertex to the input
        for each v connected to u do                  // max number of v = 6
            calculate cost                            // O(1)
            insert v to the queue                     // O(log V)
```

The worst case time complexity is computed by changing V in terms of S,L,T,A:
The priority queue has the time complexity of O(vlog v), therefore:

(1.) Array initialization: the total number of points = O (S+L+T+A)
(2.) Create the priority queue: O((S+L+T+A) * log(S+L+T+A))
(3.) Pull function with priority queue: O((S+L+T+A) * log(S+L+T+A))
(4.) Check if the input point is airport: O(~~A*(S+L+Y+A) +~~ A*A*log(S+L+T+A))=**O(A*A*log(S+L+T+A))**
(5.) Check if the adjacent points of the input are airports: O((S+L+T+A)*6*log(S+L+T+A)), as there are 6 times operation

Hence, we get the **time complexity** of worst case in the **with smaller cost**:
$O((S+L+T+A) * \log(S+L+T+A) + A^2 \log(S+L+T+A))$