Task 1

Part A

```
function find_dmax(D)
    for i := 1 … (D-1) inclusive do:
        d_max := 0
        if Throw(i) is True do:
            d_max = i
        else do:
            return d_max
    return d_max
end
```

Part B

```
function find_dmax(D)
    d_step := floor(sqrt(D))
    coefficient := 1

    /* Find maximum safe distance with a step of sqrt(D) */
    while d_step * coefficient < D do:
        if Throw(d_step * coefficient) is True do:
            coefficient = coefficient + 1
        else do:
            break

    /* Precisely determine the max safe distance one-by-one */
    d_max := d_step * coefficient
    for i := (d_step * coefficient + 1) … (d_step * (coefficient + 1)) inclusive do:
        if i < D do:
            if Throw(i) is True do:
                d_max = i
            else do:
                break
        else do:
            break
    return d_max
end
```

Part C
In the worst case, the first phase of the algorithm needs sqrt(D) throws, while the second phase needs another sqrt(D) throws. Therefore, the total number of throws in the worst case will be 2*sqrt(D). In the Big-O notation, it will be O(D^0.5).

Part D
As we have no limit on the number of rocks, we can always choose to throw into the middle of the available range, so that the the number of throws will be log_2 (D). In the Big-O notation it will be O(log D).

```
function find_dmax(D)
    d_min = 1
    d_max = D - 1

    while d_max > d_min do:
        d := (d_max + d_min) / 2
        if Throw(d) is True do:
            d_min = d
        else do:
            d_max = d

    return d_max
```

Part E

Similar to the algorithm in Part B, this algorithm performs a three-phase search.
In the worst case the first phase needs D^(1/3) throws, the second phase needs another D^(1/3) throws, while the last phase also needs D^(1/3) throws. Totally, the required number of throws will be 3*D^(1/3), corresponding to Big-O notation of O(D^(1/3))

```
function find_dmax(D)
    d_step1 := floor(pow(D, 2/3))
    coefficient1 := 1
    d_step2 := floor(pow(D, 1/3))
    coefficient2 := 1

    /* Find maximum safe distance with a step of sqrt(D) */
    while d_step1 * coefficient1 < D do:
        if Throw(d_step1 * coefficient1) is True do:
            coefficient1 = coefficient1 + 1
        else do:
            break

    while d_step2 * coefficient2 < D do:
        if Throw(d_step1 * coefficient1 + d_step2 * coefficient2) is True do:
            coefficient2 = coefficient2 + 1
        else do:
            break

    /* Precisely determine the max safe distance one-by-one */
    d_start := d_step1 * coefficient1 + d_step2 * coefficient2
    d_max := d_step * coefficient
    for i := (d_step2 * coefficient2 + 1) … (d_step2 * (coefficient2 + 1)) inclusive do:
        if i < D do:
            if Throw(i) is True do:
                d_max = i
            else do:
                break
        else do:
            break
    return d_max
```

Task 2

Part B

For every point of sea, only the point itself will get considered. While for every point of land (plain land or treasure), the point itself and its 6 neighbors will be considered. So the time complexity is: S+(1+6)*(L+T), which will be O(S+L+T).

Part C

(i)

In the new algorithm, we will not have to iterate through every point, but only the treasure points, and the land points on the same island as the treasure points. In this way, the total time spent on the search will be decreased.

(ii)

The best case may be: every island with treasure has only one piece of land, which contains all the treasures on that island. In this case, the time complexity is O(T).

(iii)

Worst case: there is treasure on every island, so we will have to go through every piece of land on the map.

In this case, the time complexity will be O(T+L).

(iv)

In the average case, assume the treasure T is uniformly distributed on all islands with treasure, so that there will be (T/N_T) treasure islands on each island. As the expected size of the island is I_S, the time complexity will be O(T/N_T*I_S).

(v)

An example of such map will be:

L T

L S

All points on the map will be explored, as the sea points are all neighbor points of other land or treasure points.

Part D

The initialization of arrays require time of O(S+L+T+A), the priority queue requires O((S+L+T+A) * log(S+L+T+A)). In each iteration of the while loop, pulling from the priority queue requires O((S+L+T+A) * log(S+L+T+A)), if the current node is an airport, additional calculation is needed for distance between airports, which is O(A*(S+L+T+A) + A*A* log(S+L+T+A)),; otherwise, the time required for its 6 neighbors is: O(6 * (S+L+T+A) * log(S+L+T+A)).

Overall, the total time will be O((S+L+T+A) + (S+L+T+A) * log(S+L+T+A) + A*(S+L+T+A) + A*A* log(S+L+T+A) + 6 * (S+L+T+A) * log(S+L+T+A)) = O((S+L+T+A) * (A+log(S+L+T+A)) + A*A*log(S+L+T+A)).

Part E

Compared with the previous algorithm, the new one only goes through the list of airports when finding the distance between airports, instead of searching the whole map for airports. As the number of airports can only be 5 even in the worst case, the time required for finding other airports becomes: O(5)=O(1). Therefore, the worst case time complexity is: O((S+L+T+A) * log(S+L+T+A) + A).