1.1.1

In the worst case, the cow starts walking in the direction opposite to where the gap is, i.e., if the gap is closer to it by walking in the left direction, the cow decides to walk in the right direction or vice versa.

The maximum number of steps it has to then walk = (L-k)

⇒ Worst-case complexity = $\Theta(L)$, assuming $L >> k$

1.1.2

<u>Pseudocode:</u>

```
function findgap (k):
        steps ← 0
        while steps < k do
                walk one step to the left
                steps ← steps + 1
        if gap found then
           return

        steps ← 0
        while steps < 2k do
                walk one step to the right
                steps ← steps + 1
```

The algorithm works by picking left as the direction towards which the cow always starts walking. It walks k steps to the left. If the gap is not found, it turns around and moves to 2k steps towards the right, to reach the position k steps towards the right of origin.

In the best case, the gap would be k steps to the left of the cow. So, it will have to walk only k steps.
⇒ Best-case complexity = $\Theta(k)$

In the worst case, the gap would be k steps to the right of the cow. So, it will have to walk 3k steps before reaching the gap.
⇒ Worst-case complexity = $\Theta(k)$, ignoring the coefficient 3

The complexity is in terms of Big Theta because the cow has to walk a minimum of k steps regardless of the direction in which the gap lies.

1.1.3

<u>Pseudocode:</u>

```
function findgap (L)
    i ← i + 1
    while i <= L/2 do
        if traverse(i) = 1 then
            return 1
        else i ← i + 1
    return 0


function traverse(i)
    if i is odd then
        j ← 0
        while j < i do
            walk one step to the right
            if gap is found then
                return 1
            j ← j + 1
        walk i steps to the left
        return 0

    else
        j ← 0
        while j < i do
            walk one step to the left
            if gap is found then
                return 1
            j ← j + 1
        walk i steps to the right
        return 0
```

<u>Worst-case complexity:</u>

In the worst case, the gap is at the opposite end, i.e., L/2 steps away from the cow's starting position. In this case, the total number of steps,

$S = 2(1) + 2(2) + 2(3) + \dots + 2(L/2 - 1) + L/2$

This is because each time the cow takes i steps in either direction, it takes another i steps to move back to the start. So, steps in each iteration = 2(i)

Note that the last term is L/2 and not 2(L/2) because once the cow reaches the gap it doesn't move back to the start.

Now the sum, $S = 2(1 + 2 + 3 + ... + (L/2 - 1)) + L/2$

$= 2\{(L/2 - 1)(L/2)/2\} + L/2$
$= L^2/4 - L/2 + L/2$
$= L^2/4$

$\Rightarrow$ Worst-case complexity = $O(L^2)$, ignoring the constant 1/4

<u>Comparison with part A:</u>
The algorithm in part A has a worst-case complexity of $\Theta(L)$ which is better than the worst-case complexity of the one described above, i.e., $\Theta(L^2)$

1.1.4
This algorithm takes a similar approach as the one in part C, with slight modifications.

Description:
The cow moves 1 step to the right, if gap not found, it moves back to the start. Then it moves 1 step to the left and returns to the start.
Next, it walks 2 steps to the right, moves back to the start, 2 steps to the left, back to the start.

So let the number of steps moved in a single direction = i
We start with i = 1
Every iteration goes as follows:
- Move i steps to the right. If gap not found, move back to the start.
- Move i steps to the left, If gap not found, move back to the start.
- Update i = 2i

So, in every iteration, the cow moves 4(i) steps in total

Iteration (n)	**i**	**Steps walked**
1	1	4(1) = 4
2	2	4(2) = 8

3	4	4(4) = 16
...	...	...
n	2^{n-1}	$4(2^{n-1}) = 2^2.2^{n-1} = 2^{n+1}$

Total steps walked = $\sum_{k=1}^{n} 2^{k+1}$

$= 2 \sum_{k=1}^{n} 2^{k}$ - eq(1)

Now, we know that $\sum_{k=0}^{n} 2^{k} = 2^{n+1} - 1$

$\Rightarrow 1 + \sum_{k=1}^{n} 2^{k} = 2^{n+1} - 1$

$\Rightarrow \sum_{k=1}^{n} 2^{k} = 2^{n+1} - 2$ - eq(2)

Substitute eq(2) in (1),
Total steps walked = $2(2^{n+1} - 2) = (2^{n+2} - 4)$

The gap (located k steps from the origin) is found in the nth iteration, when the distance travelled along one direction(i) = 2^{n-1} steps.

$\Rightarrow 2^{n-1} \approx k$ $\Rightarrow n-1 = \log_2 k$ $\Rightarrow$ **$n = \log_2 k + 1$**

Also, the cow never completes the last iteration because it doesn't move back to the start once it finds the gap. Thus, the total number of steps taken by it is always less than $(2^{n+2} - 4)$, where $n = \log_2 k + 1$

Total steps = $2^{n+2} - 4 = 2^{\log_2 k + 1 + 2} - 4$
$= 2^{\log_2 k + 3} - 4$
$= (2^{\log_2 k})2^3 - 4$
$= 8k - 4$

Therefore the complexity = O(k), ignoring coefficient 8 and constant term -4

1.2.2

The program in part A calculates cost of radio installation by multiplying cost of installing one antenna by the total number of houses (excluding the data center)

For calculating the cost of cable-based installation, we implement Prim's algorithm to get a minimum spanning tree connecting the data center and houses. This works because a house is said to be "connected" even if it has a transitive (and not direct) connection with the data center via another house(s).

To implement this, we make use of a priority queue which contains all vertices of the graph with their corresponding minimum costs. In addition, we make use of two arrays – cost and prev.
The cost array keeps track of the minimum distance required to reach a vertex and prev keeps track of the node which is directly preceding the vertex in the minimum spanning tree, representing the graph's structure

Starting off, the data center (vertex 0) is chosen as the starting point and the cost[0] is assigned zero, while rest all costs are assigned to infinity(INT_MAX). Similarly, the prev[0] is assigned to -1 while rest all are nil.
We pop a vertex(u) with the least priority(cost) from the priority queue while the queue is not empty using function getmin(), which in the first iteration is vertex 0. Then we look at all the vertices(w) adjacent to u and update their costs if we find that the weight of the edge between u and w is less than the current cost of reaching w. It should be noted that the graph in this problem is undirected, so the actual code looks for edges (u,w) or (w,u) in the edgelist.

(The following pseudocode is taken from Lecture slides Week 4 Lecture 8)

```
function prims(vertices, edgelist)
    for each v ∈ vertices do
        cost[v] ← ∞
        prev[v]← nil

    initialVertex ← 0      //data centres
    cost[initialVertex] ← 0

    pq ← initialisePriorityQueue(V, cost)     //priorities of the vertices are their costs

    while pq is not empty do
        u ← getmin(pq)
        for each (u, w) ∈ edgelist        //find all edges from or to vertex u
            if w ∈ pq and weight(u,w) < cost[w])
                cost[w] ← weight(u,w)
                prev[w] ← u
                update (pq, w, cost[w]) // update the cost of w in the
                priority queue because a shorter path has been found
```

Once the queue is empty, we have the minimum spanning tree and the minimum cost to reach each vertex. Next, we take the sum of values in cost array, which are necessarily the minimum possible distances between nodes. This gives us the cable-based installation cost.
Finally, we compare the two installation costs and print which one is more efficient.