Question 1:
a).

Step1: Define Subset
	OPT(i, houses) = max sum subset of house from 0, …., i without adjacent houses being picked at the same time

Step2: Find Recurrences

Case1:
Include house[i]: OPT(i, houses) = houses[i] + OPT(i - 2, houses)

Case2:
Exclude houses[i]: OPT(i, houses) = max{OPT(i - 1, houses), OPT(i - 2, houses)}

OPT(i, houses) = max{houses[i] + OPT(i - 2, houses) , max{OPT(i - 1, houses), OPT(i - 2, houses)}}

Step3: Solve the base cases

	OPT(0, houses) = houses[0]

b).
Inductive claim:
	At iteration k, we know the max value of excluding houses[k] and including houses[k].

Base case:
	When k=0, the max value should be including houses[0] or the value would be 0.

Inductive Hypothesis:
	We assume our inductive claim holds for k=l−1

Inductive Step: For k = l:
	At the l-th step of the algorithm, we have already known the max value of including houses[l-1] and excluding houses[l-1]. By our inductive hypothesis, this corresponds to the first (l - 1)-th houses. And now we update the inclusion and exclusion value by including and excluding houses[l]. As such, we know whether including houses[l] or excluding houses[l] has the greater sum value.

Corollary to Inductive Claim:
	At iteration n, we know the max sum for including or excluding houses[n].

c).
Since our dynamic programming algorithm is working on the array of houses from the start to the end and each time of the iteration, the algorithm can take the advantage from the calculate result from the previous one. So there would not be any redundant calculation. So the time complexity should be O(n) since we only have 1 for loop.

Question 2:

a).

Step1: Define Subset

OPT(root_index, T) = max sum of sub tree with root at root_index in given array T

Step2: Find Recurrences

Case1:
Include root element, OPT selects the sum of the max sum of left sub tree without the adjacent left child and the max sum of right sub tree without adjacent right child and the root value

- OPT(root_index, T[root_index:] - T[root_index * 2 + 1] - T[root_index * 2 + 2])
- T[root_index:] : means indicates the sub-tree from T[root_index] to the end
- T[root_index:] - T[root_index * 2 + 1] - T[root_index * 2 + 2] : exclude T[root_index * 2 + 1] and T[root_index * 2 + 2]

Case2:
Exclude root element, OPT selects the max sum of the left sub-tree (either include or exclude the adjacent left child of the root element) and the max sum of the right sub-tree(either include or exclude the adjacent right child of the root element).

- OPT(root_index, T[root_index + 1:])
- T[root_index:] : means indicates the sub-tree from T[root_index + 1] to the end

OPT(root_index, T) = max{OPT(root_index, T[root_index:] - T[root_index * 2 + 1] - T[root_index * 2 + 2]), OPT(root_index, T[root_index + 1:])}

Step3: Solve the base cases

OPT[root_index, T] = 0 if root_index >= T's length

b).

Inductive claim:

At iteration when root_index is k, we have the max sum of sub-tree whose root_index is k.

Base case:

When k >= T's length, a tuple (0,0), whose first 0 indicates the max sum with root element and the second 0 indicates the max sum without root element.

Inductive Hypothesis:

We assume our inductive claim holds for k= l –1.

Inductive Step: For k = l:

At the l-th step of the algorithm, when root_index is l, we have already known the max sum of left sub-tree and right sub-tree. By our inductive hypothesis, this corresponds to all of the sub-trees which share the same ancestor at index l. As such, we perform a max operation to find the max sum of the tree whose root_index is l, this is equivalent to finding the greatest value among the sum with the root element and without the root element.

Corollary to Inductive Claim:

At iteration n, we have the greatest sum of the tree whose root index is n.

c).

Our algorithm actually scans the element of the array from right to left (from the bottom to the root in the tree). By taking the advantage of the recursion, so we simply find the max sum by including and excluding the root element of the sub-tree during the iteration. Each time after the trace back with we calculating the max sum of the current deeper tree by using the max sum of the sub-tree. So in the whole process, we only traverse each element in the tree once, therefore the time complexity should be O(n).