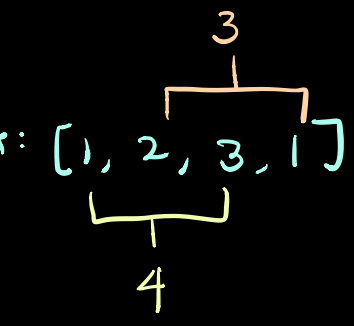
House Robber: You are a robber planning to rob houses along a street. Each house has a certain amount of money stashed.
If two adjacent houses are robbed, an alarm goes off.

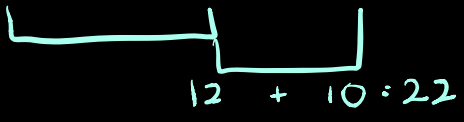
Given a list of non-negative integers representing the amount of money for each house, determine the max amount of money that can be stolen.

Ex)

① input: [1, 2, 3, 1] output: 4



② input: [9, 1, 2, 3, 4, 10] output: 22



Thoughts:

Upon seeing the first example, my initial thought is to compare a sum of values per odd index against a sum of values per even index. However, it would not satisfy the second case.

Let's visualize the array with letters to represent arbitrary values:

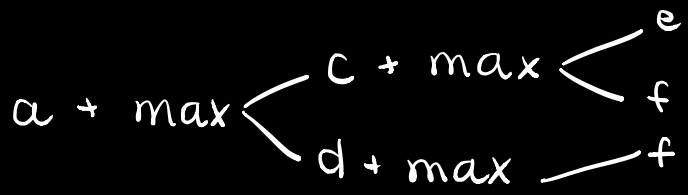
array.length = 6

from a, we have two valid choices for which index we can travel to next: c & d. b is not valid since it is adjacent to a.

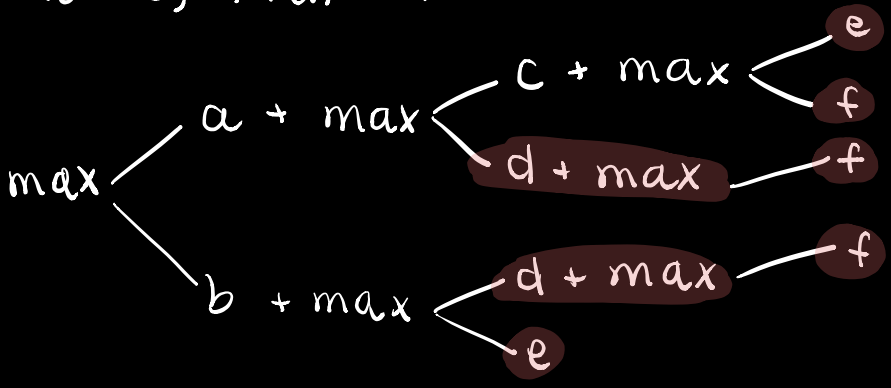
If a + c is > a + d, then we would need to choose e or f next.

If a + d is > a + c, our final house would be f.

The resulting decision tree looks something like:



However... b is not in this tree at all. If we account for b, then the answer looks more like:



This solution would recursively add sums, but for 2 paths, one starting at 0 & one starting at 1.

Looks nice and all but there's some duplicate work occurring in recalculating the sums!

Another way to go about this is to iteratively sum elements of the array, and also determine what produces the max sum (a+c vs a+d) for that segment.

Since this problem is reminiscent of Fibonacci numbers, we can use some base cases to start off with, developing a dynamic programming solution.

for an array of size one, the max sum would be arr[0].

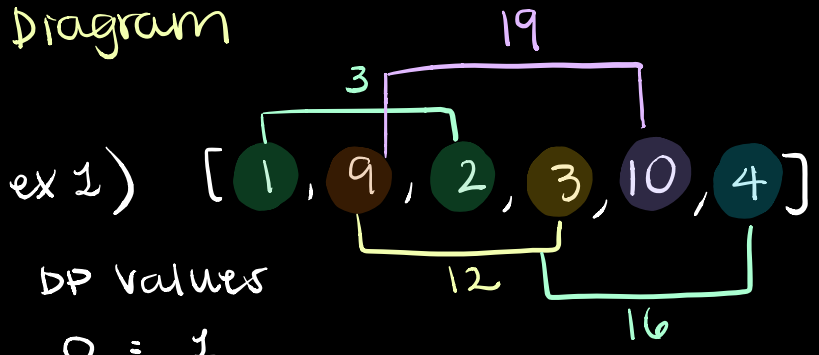
for an array of size two, the max sum would be the larger number between arr[0] & arr[1].

We can store these values in an array for dp values.

for arrays of size > 2, we begin storing the max number between the following sums:

1) one dp value behind the current index
2) two dp values behind + the current house

Diagram



DP values

0 = 1

1 = max(1, 9) = 9

2 = max(9, 1+2) = 9

3 = max(9, 9+3) = 12

4 = max(12, 19) = 19

5 = max(19, 16) = 19

also matches to current index of given array of houses

in both these cases, we are comparing whether we should skip the current house, producing the left value, or include it (right value). choosing to include the current house results in the previous house being skipped.

Implementation

```
function rob (houses) {
    if (houses.length === 0) {
        return 0;
    }
    let dpvalues = [ ];
    dpValues[0] = houses[0];
    dpvalues[1] = Math.max(houses(0), houses[1]);
    for (let i= 2; i< houses.length; i++) {
        dpvalues[i] = Math.max(dpvalues[i-1] + houses[i],
                               dpvalues[i-2]);
    }
    return dpvalues[houses.length - 1];
}
```

Complexity

Time: O(n) where n is the number of houses

Space: O(n) for the array to store cumulative sum values