House Robber II

Note: This is an extension of House Robber.

After robbing those houses on that street, the thief has found himself a new place for his thievery so that he will not get too much attention. This time, all houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, the security system for these houses remain the same as for those in the previous street.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.

Credits:
Special thanks to @Freezen for adding this problem and creating all test cases.

Solution 1

Since this question is a follow-up to House Robber, we can assume we already have a way to solve the simpler question, i.e. given a 1 row of house, we know how to rob them. So we already have such a helper function. We modify it a bit to rob a given range of houses.

```
private int rob(int[] num, int lo, int hi) {
    int include = 0, exclude = 0;
    for (int j = lo; j <= hi; j++) {
        int i = include, e = exclude;
        include = e + num[j];
        exclude = Math.max(e, i);
    }
    return Math.max(include, exclude);
}
```

Now the question is how to rob a circular row of houses. It is a bit complicated to solve like the simpler question. It is because in the simpler question whether to rob *num[lo]* is entirely our choice. But, it is now constrained by whether*num[hi]* is robbed.

However, since we already have a nice solution to the simpler problem. We do not want to throw it away. Then, it becomes how can we reduce this problem to the simpler one. Actually, extending from the logic that if house i is not robbed, then you are free to choose whether to rob house i + 1, you can break the circle by assuming a house is not robbed.

For example, 1 -> 2 -> 3 -> 1 becomes 2 -> 3 if 1 is not robbed.

Since every house is either robbed or not robbed and at least half of the houses are not robbed, the solution is simply the larger of two cases with consecutive houses, i.e. house i not robbed, break the circle, solve it, or house i + 1 not robbed. Hence, the following solution. I chose i = n and i + 1 = 0 for simpler coding. But, you can choose whichever two consecutive ones.

```
public int rob(int[] nums) {
    if (nums.length == 1) return nums[0];
    return Math.max(rob(nums, 0, nums.length - 2), rob(nums, 1, nums.length - 1))
;
}
```

written by lx223 original link here

Solution 2

This problem is a little tricky at first glance. However, if you have finished the **House Robber** problem, this problem can simply be **decomposed into two House Robber problems**. Suppose there are `n` houses, since house `0` and `n - 1` are now neighbors, we cannot rob them together and thus the solution is now the maximum of

1. Rob houses `0` to `n - 2`;
2. Rob houses `1` to `n - 1`.

The code is as follows. Some edge cases (`n < 2`) are handled explicitly.

```
class Solution {
public:
    int rob(vector<int>& nums) {
        int n = nums.size();
        if (n < 2) return n ? nums[0] : 0;
        return max(robber(nums, 0, n - 2), robber(nums, 1, n - 1));
    }
private:
    int robber(vector<int>& nums, int l, int r) {
        int pre = 0, cur = 0;
        for (int i = l; i <= r; i++) {
            int temp = max(pre + nums[i], cur);
            pre = cur;
            cur = temp;
        }
        return cur;
    }
};
```

written by jianchao.li.fighter original link here

Solution 3

Twice pass:

1. not rob nums[n-1]
2. not rob nums[0]

and the other is same as House Robber.

```
int rob(vector<int>& nums)
{
    if(nums.size() == 0)
        return 0;
    if(nums.size() == 1)
        return nums[0];

    int pre1 = 0, cur1 = 0;
    for(int i = 0; i < nums.size() - 1; ++ i)
    {
        int temp = pre1;
        pre1 = cur1;
        cur1 = max(temp + nums[i], pre1);
    }

    int pre2 = 0, cur2 = 0;
    for(int i = 1; i < nums.size(); ++ i)
    {
        int temp = pre2;
        pre2 = cur2;
        cur2 = max(temp + nums[i], pre2);
    }

    return max(cur1, cur2);
}
```

written by makuiyu original link here