One-dimensional Dynamic Programming

Dimensionality of the DP program depends on the number of parameters that define the state.

If there is one parameter then the problem is called one dimensional. We usually use an array for storing the results. Usually these type of problems are easy to solve. Lets use our step by step approach to solve few such problems.

1. A robber is planning to rob a row of houses. Each house has an amount of money stashed in it. He has to be careful as adjacent houses are installed with a security system which calls the police if two adjacent houses are robbed. The robber cannot rob two adjacent houses, he knows before hand how much stash each house has. He has to come up with a strategy to rob the houses such that he can make maximum profit without being caught by the police.

20 25 30 15 10

S =

20	25	20	15	10

1. State

1. State

Parameters

i - Index of the last house being robbed.

Cost function

maxProfit(i,S) -

Function returns the maximum profit that can be acehived by robbing the houses ending at index i. i.e robbing houses indexed 0 to i without being caught by the police.

S - is the array having the values of stash in each house.

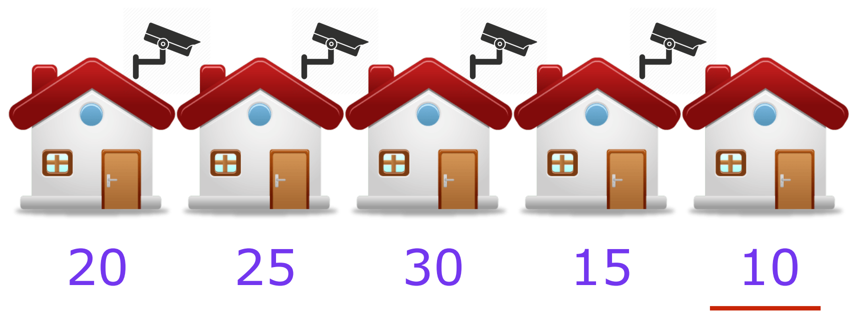
2. Transitions

2. Transitions

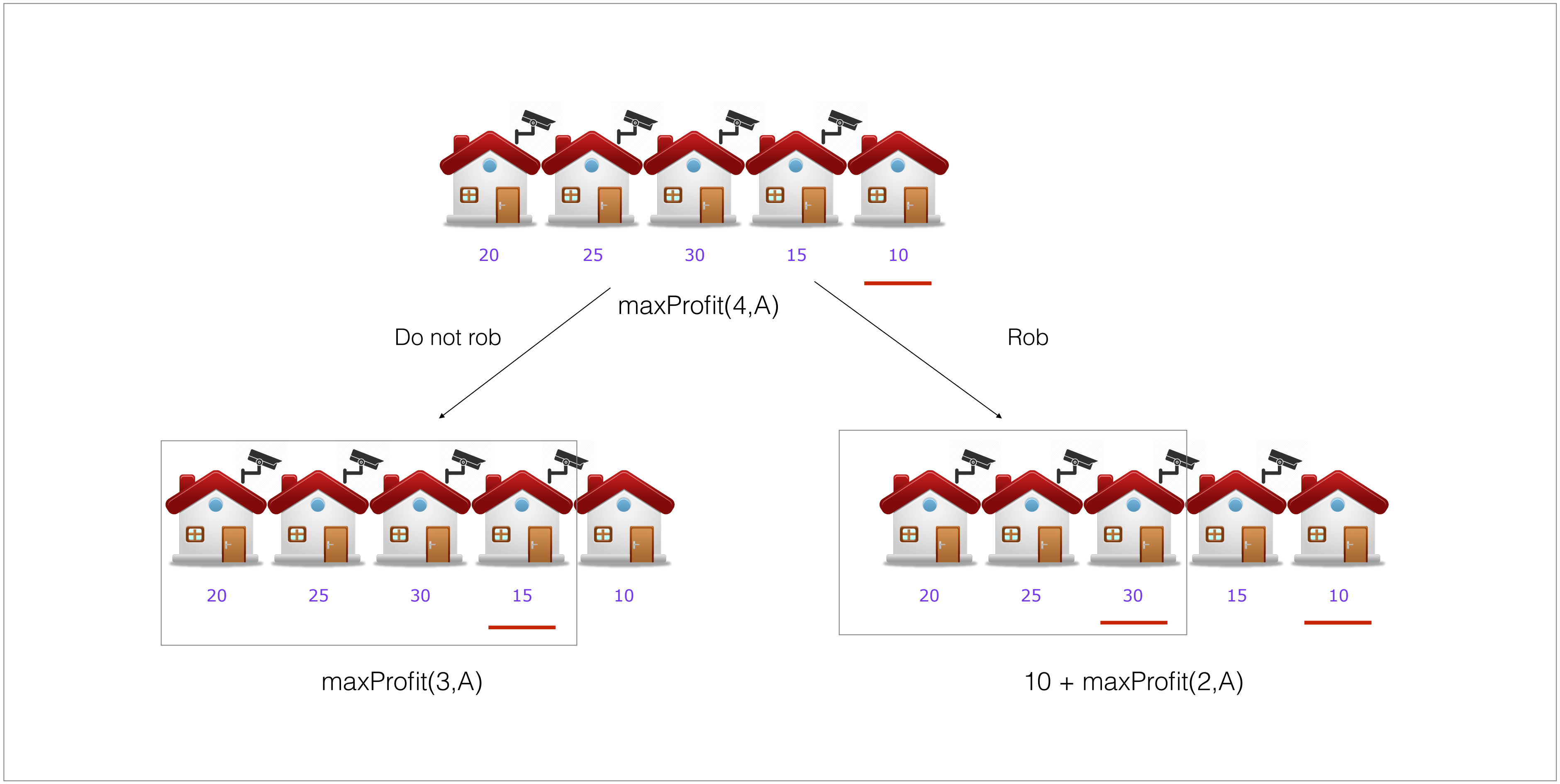
Base case

i=0 , return S[0]

House robber



House robber



Candidates

maxProfit(i,S)

Rob the i^{th} house

S[i]+maxProfit(i-2,S)

Skip the house

maxProfit(i-1,S)

Optimal choice

We want to maximize the profit from robbing, choose the maximum option.

Recurrence relation

maxProfit(i,S) = MAX(S[i]+maxProfit(i-2,S),maxProfit(i-1,S))

3. Recursive solution

3. Recursive solution

Pseudo code

```
maxProfit(i,S)
  if i<0
     return 0
   if i==0
     return S[0]
   return MAX(S[i]+maxProfit(i-2,S),maxProfit(i-1,S))
```

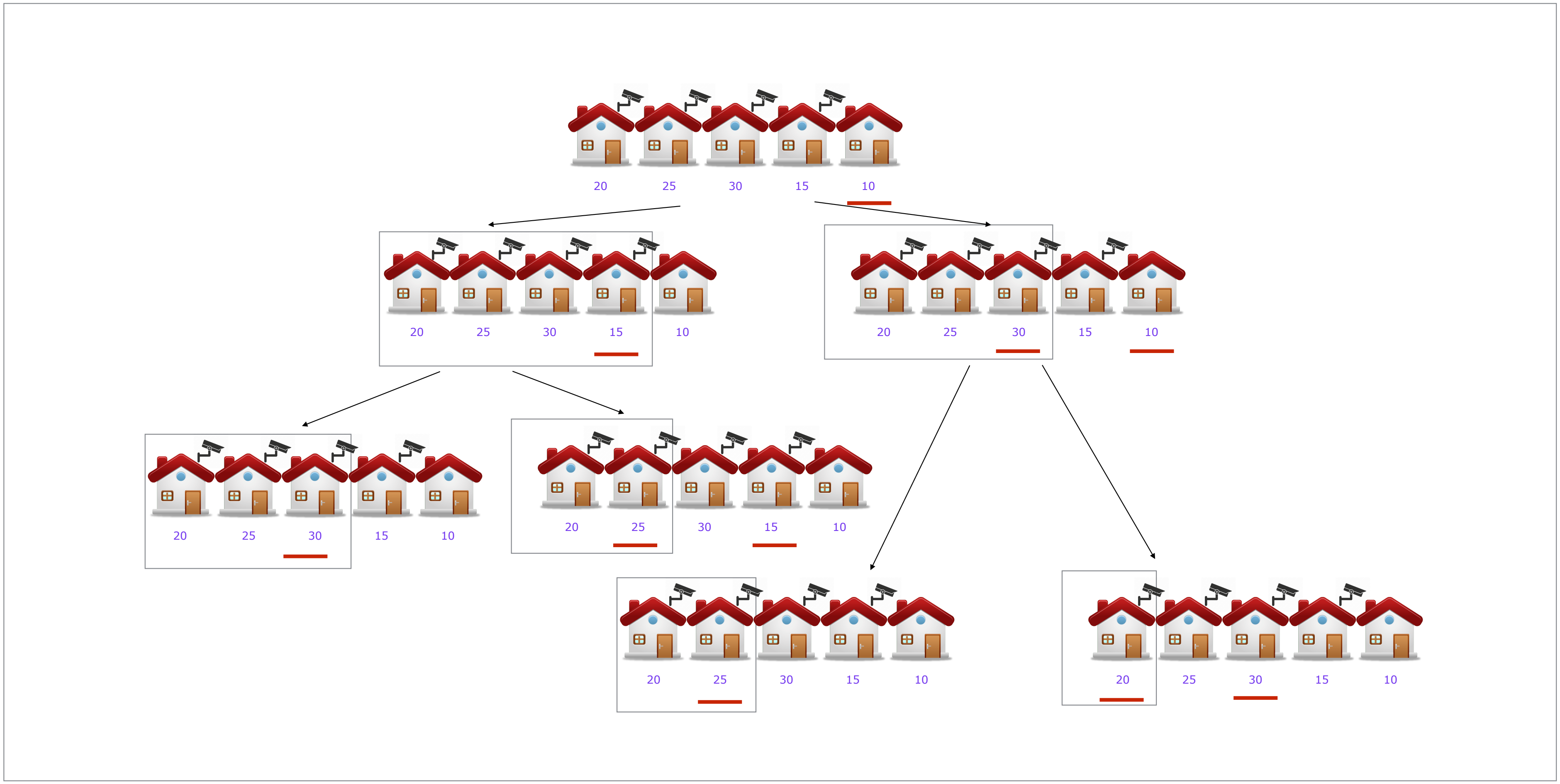
Java

```java
public static int maxProfit(int i,int[] S){
    if(i<0){
        return 0;
    }
    if(i == 0){
        return S[0];
    }
    return Math.max(S[i]+maxProfit(i-2,S),maxProfit(i-1,S));
}
```

Python

```python
def max_profit(i, S):
    if i == 0:
        return S[0]
    if i < 0:
        return 0
    return max(S[i] + max_profit(i - 2, S), max_profit(i - 1,
S))
```

Overlapping subproblems



4. Memoize

4. Memoization

maxProfit(i,S) = MAX(S[i]+maxProfit(i-2,S),maxProfit(i-1,S))

There is only one parameter which defines the state.

We can use an array to cache the results.

Key -> i , Index of the house

Value -> Maximum profit that can be acheived by robbing houses index 0 to i

Java

```java
public static int maxProfitMemo(int i,int[] S,int[] cache){
    if(i<0){
        return 0;
    }
    if(i == 0){
        return S[0];
    }
    if(cache[i] != 0){
        return cache[i];
    }
    int profit = Math.max(S[i]
+maxProfitMemo(i-2,S,cache),maxProfitMemo(i-1,S,cache));
    cache[i] = profit;
    return profit;
}
```

Python

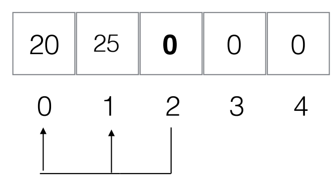
```python
def max_profit_memo(i, S, cache):
    if i == 0:
        return S[0]
    if i < 0:
        return 0
    if cache[i] != 0:
        return cache[i]
    profit = max(S[i] + max_profit_memo(i - 2, S, cache),
max_profit_memo(i - 1, S, cache))
    cache[i] = profit
    return profit
```

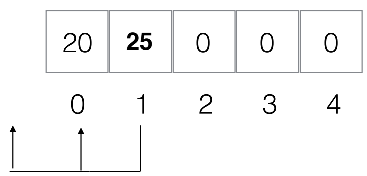
5. Bottom up approach

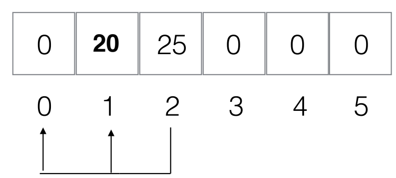
5. Bottom up

maxProfit(i,S) = MAX(S[i]+maxProfit(i-2,S),maxProfit(i-1,S))

dp[i] - Stores maximum profit obtained by robbing houses from 0 to i-1


20
25
0
0
0
0
1
2
3
4


20
25
0
0
0
0
1
2
3
4


0 20 25 0 0 0
0 1 2 3 4 5

5. Bottom up

maxProfit(i,S) = MAX(S[i]+maxProfit(i-2,S),maxProfit(i-1,S))

dp[i] - Stores maximum profit obtained by robbing houses from 0 to i-1

dp[0] - 0

dp[1] - S[0]

dp[i] = MAX(S[i-1]+dp[i-2],dp[i-1])

Bottom up

S =

20	25	20	15	10

Bottom up

S =	20	25	20	15	10	
DP =	0	0	0	0	0	0
	0	1	2	3	4	5

Bottom up

S =

20	25	20	15	10

DP =

0	0	0	0	0	0
0	1	2	3	4	5

0	**20**	0	0	0	0
0	1	2	3	4	5

Bottom up

S =

20	25	20	15	10

DP =

0	0	0	0	0	0
0	1	2	3	4	5

0	**20**	0	0	0	0
0	1	2	3	4	5

0	20	**25**	0	0	0
0	1	2	3	4	5

dp[2] = MAX(25+0,20) = 25

0	20	25	**40**	0	0
0	1	2	3	4	5

dp[3] = MAX(20+20,25) = 40

0	20	25	40	40	**50**
0	1	2	3	4	5

Java

```java
public static int maxProfitDP(int[] S){
    int N = S.length;
    int[] dp = new int[N+1];
    dp[1] = S[0];
    for(int i=2;i<=N;i++){
        dp[i] = Math.max(S[i-1]+dp[i-2],dp[i-1]);
    }
    return dp[N];
}
```

Python

```python
def max_profit_dp(S):
    N = len(S)
    dp = [0 for _ in range(0, N + 1)]
    dp[1] = S[0]
    for i in range(2, N + 1):
        dp[i] = max(S[i - 1] + dp[i - 2], dp[i - 1])
    return dp[N]
```

Time and space complexity

Recursive solution

Binary tree.

Height of the tree = N

Number of nodes = 2^N

Time complexity is $O(2^N)$, Exponential

Space complexity , $O(1)$

Dynamic programming solutions

There is one for loop, it runs from 0 to N

$$\sum_{i=0}^{N} 1 = 1+1+1+\ldots+1 = N$$

Time complexity O(N) , Linear

Space complexity O(N)

Difference in execution time

N = 12

Java

Recursion:	370 micro seconds
Memoization:	46 micro seconds
DP :	33 micro seconds

Python

Recursion:	168 micro seconds
Memoization:	16 micro seconds
DP :	12 micro seconds

Reconstruct the solution

Record the decision at every index.

robbed[i] - True , Its beneficial to rob house i than not robbing it.

robbed[i] - False , Its best to not rob house i,

robbed[0] = True, Base case, if there is only one house then rob it.

Reconstructing

We start from the last. i=N-1

We check the decision recorded at rob[i]

if its true, then house was robbed. i = i-2

If its false, then skip the house, i = i-1

We repeat this until we hit i=0 or -1.

Java

```java
public static int maxProfitDPReconstruct(int[] S){
    int N = S.length;
    int[] dp = new int[N+1];
    boolean[] rob = new boolean[N];
    rob[0]=true;
    dp[1] = S[0];
    for(int i=2;i<=N;i++){
        if(S[i-1]+dp[i-2] > dp[i-1]){
            dp[i] = S[i-1]+dp[i-2];
            rob[i-1] = true;
        }else{
            dp[i] = dp[i-1];
            rob[i-1] = false;
        }
    }
```

Java

```java
    int i = N-1;
    while(i>=0){
        if(rob[i]){
            System.out.println(i+" "+S[i]);
            i=i-2;
        }else{
            i--;
        }
    }
    return dp[N];
}
```

Python

```python
def max_profit_dp_reconstruct(S):
    N = len(S)
    dp = [0 for _ in range(0, N + 1)]
    rob = [False for _ in range(0, N)]
    rob[0] = True
    dp[1] = S[0]

    for i in range(2, N + 1):
        if S[i - 1] + dp[i - 2] > dp[i - 1]:
            rob[i - 1] = True
            dp[i] = S[i - 1] + dp[i - 2]
        else:
            dp[i] = dp[i - 1]
            rob[i - 1] = False
    i = N - 1
    while i >= 0:
        if rob[i]:
            print("{} = {}".format(i, S[i]))
            i -= 2
        else:
            i -= 1
    return dp[N]
```