Step by step approach

Steps

1. Define the state
2. List out all the state transitions
3. Implement a recursive solution
4. Memoize
5. Make it bottom up

1. Define the state

States : Set of parameters which define the state of the system. We need to choose least possible number of parameters. The values of the parameters reflect the consequence of a decision.

Define the cost function which represent the state and its return is the cost that we are trying to optimize for.

In our knapsack problem:

States :

1. W - Available Capacity of the bag.

2. i - Index of the item being considered

Cost function: knapsack(W,i)

2. Define the state transitions and optimal choice

Identify the base cases.

For eg:

1. Very last stage , where there are no more items left or if its the last item.

2. Some parameters are 0 or reached some final value after which we cannot proceed further.

Transitions

In backtracking,

1. Enumerate all the candidates,

2. Tried one candidate at a time and called the recursive function.

3. Once it returned we backtrack and try another candidate.

We need to identify all the valid candidates and try each of them one at a time , change the parameters based on the candidate chosen and call the recursive function.

Optimal choice

Once we have tried out all the candidates, we need to combine those results and find the optimal value which we return

In our knapsack problem

Base cases.

1. If W = 0, it means the knapsack is full. No space available , so we output will be 0.

2. If i = -1, No more items available to consider

if W==0 or i == -1

return 0

Knapsack problem

Transitions

Decisions that we can make are whether we should take the item in the bag or not.

1. If we take the i^{th} item if it fits in the knapsack, then W will become W-weight[i]. To check if the item fits in the knapsack we need check weight[i] less than or equal to W.

2. Skip the i^{th} item, then W will remain same

KS(W,i)

value[i]+KS(W-weight[i],i-1)   KS(W,i-1)

Optimal choice

MAX(val[i]+KS(W-weight[i],i-1) , KS(W,i-1))

We choose the decision which yeilds the maximum output.

Recurrence relation

KS(W,i) = MAX(val[i]+KS(W-weight[i],i-1) , KS(W,i-1))

if weight[i] <= W

else

KS(W,i) = KS(W,i-1)

Base cases

KS(0,i) = 0

KS(W,-1) = 0

2. Implement recursive solution

Java

```java
public static int knapsack(int[] ws, int[] vs, int W, int i) {
    if (i == -1 || W == 0) {
        return 0;
    }
    if (ws[i] <= W) {
        int include = vs[i] + knapsack(ws, vs, W - ws[i], i -
1);
        int exclude = knapsack(ws, vs, W, i - 1);
        return Math.max(include, exclude);
    } else {
        return knapsack(ws, vs, W, i - 1);
    }
}
```

Python

```python
def knapsack(W, i, weights, values):
    if W == 0 or i == -1:
        return 0
    if weights[i] <= W:
        return max(values[i] + knapsack(W - weights[i], i - 1,
weights, values),
                   knapsack(W, i - 1, weights, values))
    else:
        return knapsack(W, i - 1, weights, values)
```

4. Memoize

- We can speed up the naive recursive solution using memoization.
- In memoization, we cache the results of subproblems so that we avoid solving overlapping subproblems.

- If there is only one parameter then we can use an array and we use the value of the parameter as the index to store the result.
- If there are two parameters which define the state then we use a 2D matrix.
- We initialize the array with some default values, for eg 0 or -1. We should choose the default value such that the result of the function can never be equal to the default value.

Java

```java
public static int knapsackMemo(int[] weights, int[] values, int W, int
i, int[][] dp) {
    if (i == -1 || W == 0) {
        return 0;
    }
    if (dp[W][i] != -1) {
        return dp[W][i];
    }
    if (weights[i] <= W) {
        int include = values[i] + knapsack(weights, values, W -
weights[i], i - 1);
        int exclude = knapsack(weights, values, W, i - 1);
        dp[W][i] = Math.max(include, exclude);
        return dp[W][i];
    } else {
        return dp[W][i] = knapsack(weights, values, W, i - 1);
    }
}
```

Python

```python
def knapsack(W, i, weights, values, dp):
    if W == 0 or i == -1: return 0
    if dp[W][i] != -1:
        return dp[W][i]
    if weights[i] <= W:
        res = max(values[i] + knapsack(W - weights[i], i - 1,
weights, values, dp),
                  knapsack(W, i - 1, weights, values, dp))
        dp[W][i] = res
        return res
    else:
        res = knapsack(W, i - 1, weights, values, dp)
        dp[W][i] = res
        return res
```

5. Bottom up approach

- In bottom up approach you have to think in reverse direction. i.e you have to start at the bottom of the tree, solve all the problems starting from the leaves and make your way up.

- We will start with base case and make our way upto final state solving all the subproblems sequentially.
- We use one for-loop for each parameter and solve problems for all the values of the parameter. This is why its called table filling method, because we fill the entire table bottom up whether we need all the results or not.
- We will solve all the smaller problems first which will be needed by the larger problems.

How do we know which problem to solve first ?

It depends on how the parameters change in recurrence relation.

If a parameter becomes smaller in the subproblem then we start from 0 and go all the way up to the maximum value of the parameter.

Base case

if W=0 , return 0

if i=-1 , return 0

Recurrence relation

KS(W,i) = MAX(val[i]+KS(W-weight[i],i-1) , KS(W,i-1))

if weight[i] <= W

else

KS(W,i) = KS(W,i-1)

In bottom up approach, we can do

Base case

if i = -1 , return 0

if w = 0 , return 0

Bottom up equation

dp[w][i] = MAX(val[i]+dp[w-weight[i]][i],dp[w][i-1])

In bottom up approach, we can do

Base case

if i = -1 , return 0

if w = 0 , return 0

Bottom up equation

dp[w][i] = MAX(val[i]+dp[w-weight[i]][i-1], **dp[w][i-1]**)

In bottom up approach, we can do

Base case

if i = 0, return 0

if w = 0 , return 0

Bottom up equation

dp[w][i] = MAX(val[i-1]+dp[w-weight[i-1]][i],dp[w][i-1])

For eg:

ks(W=20) depends on ks(W=17), and ks(W=17) depends on ks(W=14) and so on.

ks(i=5) depends on ks(i=4) and ks(i=4) depends on ks(i=3) which means to solve problem at stage 2, we need the results of stage 1.

We use one for loop for each of the parameter and build all the solutions.

Rule of thumb

In your recurrence relationship , if the value of the parameter passed to the recursive call to the function is **less** than the current value of the parameter, then your for loop for the parameter should iterate in **asending order**. Otherwise your for loop for the parameter should iterate in descending order.

In bottom up approach, we can do

Base case

if i = 0, return 0

if w = 0 , return 0

Bottom up equation

dp[w][i] = MAX(val[i-1]+dp[w-weight[i-1]][i],dp[w][i-1])

for all w=1,2,…W

for all i = 1,2,3,…N

Java

```java
public static int knapsackDP(int[] weights, int[] values, int W) {
    int N = weights.length;
    int[][] dp = new int[W + 1][N + 1];
    for (int i = 1; i <= N; i++) {
        for (int w = 1; w <= W; w++) {
            if (weights[i - 1] <= w) {
                dp[w][i] = Math.max(dp[w - weights[i - 1]][i - 1] + values[i - 1], dp[w][i - 1]);
            } else {
                dp[w][i] = dp[w][i - 1];
            }
        }
    }
    return dp[W][N];
}
```

Python

```python
def knapsack_DP(W, weights, values):
    dp = [[0 for i in range(0, len(weights) + 1)] for j in range(0, W + 1)]
   for i in range(1, len(weights) + 1):
        for w in range(0, W + 1):
            if weights[i - 1] <= w:
                dp[w][i] = max(dp[w][i - 1], dp[w - weights[i - 1]][i - 1] + values[i - 1])
            else:
                dp[w][i] = dp[w][i - 1]
    return dp[W][len(weights)]
```

	Item	Weight	Value
0		3	4
1		7	14
2		10	10
3		6	5

					
	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	4	4	4	4
4	0	4	4	4	4
5	0	4	4	4	4
6	0	4	4	4	5
7	0	4	14	14	14
8	0	4	14	14	14
9	0	4	14	14	14
10	0	4	18	18	18
11	0	4	18	18	18
12	0	4	18	18	18
13	0	4	18	18	19
14	0	4	18	18	19
15	0	4	18	18	19
16	0	4	18	18	23
17	0	4	18	24	24
18	0	4	18	24	24
19	0	4	18	24	24
20	0	4	18	28	28

Analyzing the time and space complexity

Analyzing the recursive algorithm

Draw a recursion tree.

1. Start with the initial state. This will be our root node.

2. Enumerate all the state transitions as discussed and draw nodes for each of them and connect them to the parent state using an edge.

3. Then for each of the child node repeat the process.

4. Base cases will be the leaves of the tree.

5. Once you draw the recursion tree. Determine the height of the tree.

6. Then the time complexity will be number of nodes in the recursion tree.

We know that given the height of the tree h, and if each node has C children, then number of nodes in the tree is $O(C^h)$

In our knapsack example C = 2 and height of the recursion tree is N. Then the time complexity is $O(2^N)$. This is exponential time

The main objective of Dynamic Programming is to implement a solution which runs in linear time $O(N)$ or polynomial time $O(N^2)$ or $O(N^3)$

Analyzing the memoization and bottom up approach

There are two for loops

Outer for loop goes from i=0,1,2,...N

Inner loop goes from w=0,1,2,...W

$$\sum_{i=0}^{N} \sum_{w=0}^{W} 1 = W \sum_{i=0}^{N} 1 = NW$$

Then the time complexity of the DP algorithm is O(NW).

Space complexity is O(NW)

Reconstruct the solution

Max Profit = 28

	Item	Weight	Value
0		3	4
1		7	14
2		10	10
3		6	5

We need to also record the decision that was made at each step and using that we will reconstruct the solution.

Knapsack

We will use a boolean cache of same dimension to record the decisions.

In Knapsack the decision is to either take the item or not depending on whichever yields the maximum value, so we can use a boolean value to record the decision.

We will record the decision at each subproblem.

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	4	4	4	4
4	0	4	4	4	4
5	0	4	4	4	4
6	0	4	4	4	5
7	0	4	14	14	14
8	0	4	14	14	14
9	0	4	14	14	14
10	0	4	18	18	18
11	0	4	18	18	18
12	0	4	18	18	18
13	0	4	18	18	19
14	0	4	18	18	19
15	0	4	18	18	19
16	0	4	18	18	23
17	0	4	18	24	24
18	0	4	18	24	24
19	0	4	18	24	24
20	0	4	18	28	28

	0	1	2	3	4
0	F	F	F	F	F
1	F	F	F	F	F
2	F	F	F	F	F
3	F	T	F	F	F
4	F	T	F	F	F
5	F	T	F	F	F
6	F	T	F	F	T
7	F	T	T	F	F
8	F	T	T	F	F
9	F	T	T	F	F
10	F	T	T	F	F
11	F	T	T	F	F
12	F	T	T	F	F
13	F	T	T	F	T
14	F	T	T	F	T
15	F	T	T	F	T
16	F	T	T	F	T
17	F	T	T	T	F
18	F	T	T	T	F
19	F	T	T	T	F
20	F	T	T	T	F

To reconstruct the solution

1. We start at the final state of the problem, w=W,i=N

2. If the decison at (w,i) is true

Then we print that we picked up that item , then update the parameters accordingly.

w = w-weights[i-1]

i = i-1

3. If the decision at w,i was to not pick the item, then we just update the index

i = i-1

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	4	4	4	4
4	0	4	4	4	4
5	0	4	4	4	4
6	0	4	4	4	5
7	0	4	14	14	14
8	0	4	14	14	14
9	0	4	14	14	14
10	0	4	18	18	18
11	0	4	18	18	18
12	0	4	18	18	18
13	0	4	18	18	19
14	0	4	18	18	19
15	0	4	18	18	19
16	0	4	18	18	23
17	0	4	18	24	24
18	0	4	18	24	24
19	0	4	18	24	24
20	0	4	18	28	28

	0	1	2	3	4
0	F	F	F	F	F
1	F	F	F	F	F
2	F	F	F	F	F
3	F	T	F	F	F
4	F	T	F	F	F
5	F	T	F	F	F
6	F	T	F	F	T
7	F	T	T	F	F
8	F	T	T	F	F
9	F	T	T	F	F
10	F	T	T	F	F
11	F	T	T	F	F
12	F	T	T	F	F
13	F	T	T	F	T
14	F	T	T	F	T
15	F	T	T	F	T
16	F	T	T	F	T
17	F	T	T	T	F
18	F	T	T	T	F
19	F	T	T	T	F
20	F	T	T	T	F

Skip 

W = 20

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	4	4	4	4
4	0	4	4	4	4
5	0	4	4	4	4
6	0	4	4	4	5
7	0	4	14	14	14
8	0	4	14	14	14
9	0	4	14	14	14
10	0	4	18	18	18
11	0	4	18	18	18
12	0	4	18	18	18
13	0	4	18	18	19
14	0	4	18	18	19
15	0	4	18	18	19
16	0	4	18	18	23
17	0	4	18	24	24
18	0	4	18	24	24
19	0	4	18	24	24
20	0	4	18	28	28

	0	1	2	3	4
0	F	F	F	F	F
1	F	F	F	F	F
2	F	F	F	F	F
3	F	T	F	F	F
4	F	T	F	F	F
5	F	T	F	F	F
6	F	T	F	F	T
7	F	T	T	F	F
8	F	T	T	F	F
9	F	T	T	F	F
10	F	T	T	F	F
11	F	T	T	F	F
12	F	T	T	F	F
13	F	T	T	F	T
14	F	T	T	F	T
15	F	T	T	F	T
16	F	T	T	F	T
17	F	T	T	T	F
18	F	T	T	T	F
19	F	T	T	T	F
20	F	T	T	T	F

Include 

W = 20 -10 = 10

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	4	4	4	4
4	0	4	4	4	4
5	0	4	4	4	4
6	0	4	4	4	5
7	0	4	14	14	14
8	0	4	14	14	14
9	0	4	14	14	14
10	0	4	18	18	18
11	0	4	18	18	18
12	0	4	18	18	18
13	0	4	18	18	19
14	0	4	18	18	19
15	0	4	18	18	19
16	0	4	18	18	23
17	0	4	18	24	24
18	0	4	18	24	24
19	0	4	18	24	24
20	0	4	18	28	28

	0	1	2	3	4
0	F	F	F	F	F
1	F	F	F	F	F
2	F	F	F	F	F
3	F	T	F	F	F
4	F	T	F	F	F
5	F	T	F	F	F
6	F	T	F	F	T
7	F	T	T	F	F
8	F	T	T	F	F
9	F	T	T	F	F
10	F	T	T	F	F
11	F	T	T	F	F
12	F	T	T	F	F
13	F	T	T	F	T
14	F	T	T	F	T
15	F	T	T	F	T
16	F	T	T	F	T
17	F	T	T	T	F
18	F	T	T	T	F
19	F	T	T	T	F
20	F	T	T	T	F

Include 

W = 10 - 7 = 3

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	4	4	4	4
4	0	4	4	4	4
5	0	4	4	4	4
6	0	4	4	4	5
7	0	4	14	14	14
8	0	4	14	14	14
9	0	4	14	14	14
10	0	4	18	18	18
11	0	4	18	18	18
12	0	4	18	18	18
13	0	4	18	18	19
14	0	4	18	18	19
15	0	4	18	18	19
16	0	4	18	18	23
17	0	4	18	24	24
18	0	4	18	24	24
19	0	4	18	24	24
20	0	4	18	28	28

	0	1	2	3	4
0	F	F	F	F	F
1	F	F	F	F	F
2	F	F	F	F	F
3	F	T	F	F	F
4	F	T	F	F	F
5	F	T	F	F	F
6	F	T	F	F	T
7	F	T	T	F	F
8	F	T	T	F	F
9	F	T	T	F	F
10	F	T	T	F	F
11	F	T	T	F	F
12	F	T	T	F	F
13	F	T	T	F	T
14	F	T	T	F	T
15	F	T	T	F	T
16	F	T	T	F	T
17	F	T	T	T	F
18	F	T	T	T	F
19	F	T	T	T	F
20	F	T	T	T	F

Include 

W = 3 - 3 = 0

Java

```java
public static int knapsackDPReconstruction(int[] weights, int[] values, int W) {
    int N = weights.length;
    int[][] dp = new int[W + 1][N + 1];
    boolean[][] decisions = new boolean[W + 1][N + 1];
    for (int i = 1; i <= N; i++) {
        for (int w = 1; w <= W; w++) {
            if (weights[i - 1] <= w) {
                if (dp[w - weights[i - 1]][i - 1] + values[i - 1] > dp[w][i - 1]) {
                    decisions[w][i] = true;
                    dp[w][i] = dp[w - weights[i - 1]][i - 1] + values[i - 1];
                } else {
                    dp[w][i] = dp[w][i - 1];
                }
            } else {
                dp[w][i] = dp[w][i - 1];
            }
        }
    }
```

Java

```java
    int i = N;
    int w = W;
    while (i >= 0 && w >= 0) {
        boolean picked = decisions[w][i];
        if (picked) {
            System.out.println("Picked : " + (i-1) + ", Weight "
+ weights[i - 1] + ", Value " + values[i - 1]);
            w -= weights[i - 1];
            i--;
        } else {
            i--;
        }
    }
    return dp[W][N];
}
```

Python

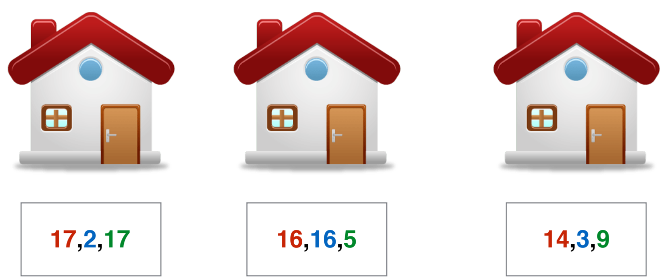
```python
def knapsack_DP_reconstruct(W, weights, values):
    dp = [[0 for i in range(0, len(weights) + 1)] for j in range(0, W + 1)]
    n = len(weights)
    decisions = [[False for i in range(0, len(weights) + 1)] for j in range(0, W + 1)]
    dp[W][0] = 0
    for i in range(1, n + 1):
        for w in range(0, W + 1):
            if weights[i - 1] <= w:
                if dp[w - weights[i - 1]][i - 1] + values[i - 1] > dp[w][i-1]:
                    # We record the decision here that its beneficial to pick the ith item
                    decisions[w][i] = True
                    dp[w][i] = dp[w - weights[i - 1]][i - 1] + values[i - 1]
                else:
                    dp[w][i] = dp[w][i - 1]
            else:
                dp[w][i] = dp[w][i - 1]
```

Python

```
    i = n
    w = W
    while i >= 0 and w >= 0:
        if decisions[w][i]:
            print("Picked up {} , Weight {} , Value 
{}".format(i-1,weights[i-1],values[i-1]))
            w -= weights[i - 1]
            i -= 1;
        else:
            i -= 1
    return dp[W][n]
```

Exercise

1. There are a row of n houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color. What is the minimum cost ?



cost[][] =
[[17,2,17],
[16,16,5],
[14, 3, 9]]

1. State

State

i - Index of the house

c - color of the paint , It can be Red,Blue or Green. We use 0,1,2 to denote it.

Cost function

min_cost(i,c) - returns the minimum cost of painting house i with paint color c.

2. Transitions

Transitions

We start at house indexed at 0.

House 0 can be painted with RED,BLUE or GREEN,

We try all three options.

min_cost(0,RED)

min_cost(0,BLUE)

min_cost(0,GREEN)

Transitions

If we paint house number 0 with RED then house number 1 can be either painted with GREEN or BLUE.

min_cost(0,RED)

min_cost(1,BLUE)   min_cost(1,GREEN)

Optimal choice

Because we are interested in the painting sequence which costs minimum amount, at each step we choose the option which returns minimum cost.

min_cost(0,RED) = cost[i][RED] +
MIN(min_cost(1,BLUE),min_cost(1,GREEN))

Recurrence relation

min_cost(i,RED) = cost[i][RED] + MIN(min_cost(i+1,BLUE),min_cost(i+1,GREEN))

min_cost(i,BLUE) = cost[i][BLUE] + MIN(min_cost(i+1,RED),min_cost(i+1,BLUE))

min_cost(i,GREEN) = cost[i][GREEN] + MIN(min_cost(i+1,BLUE),min_cost(i+1,GREEN))

Base case

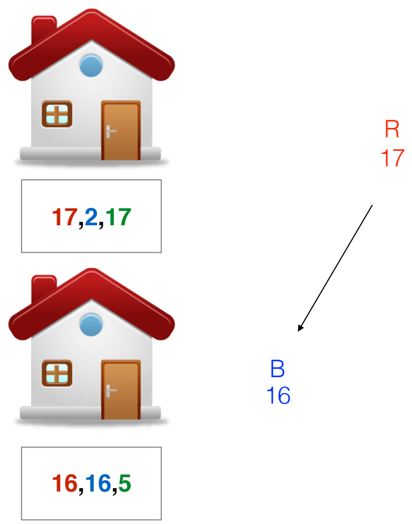
When i == n , where n is number of houses, we return 0. We have reached the end of the array.

Painting houses

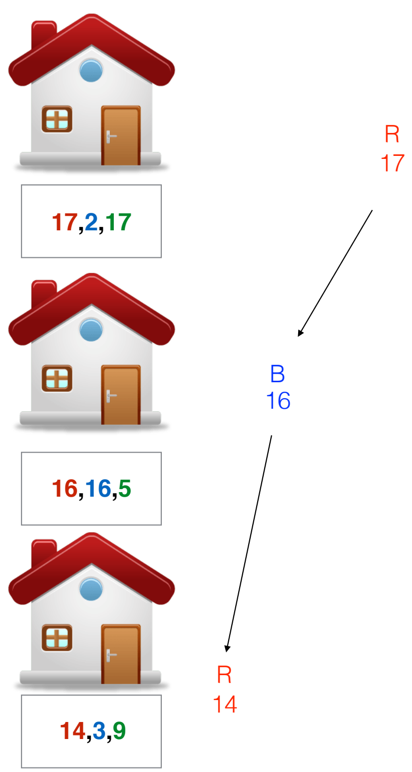
17,2,17

R
17

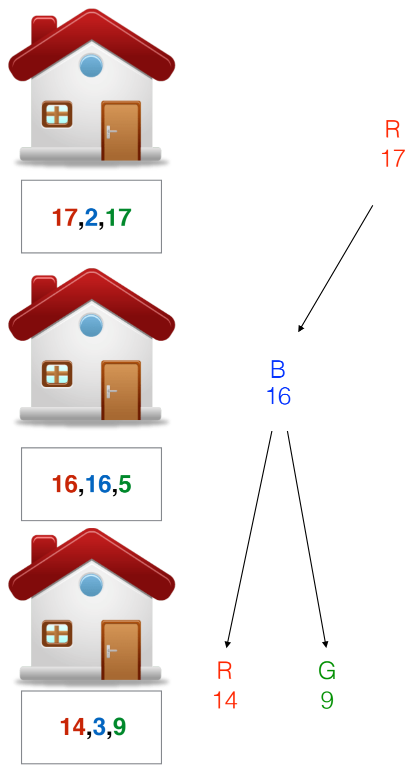
Painting houses



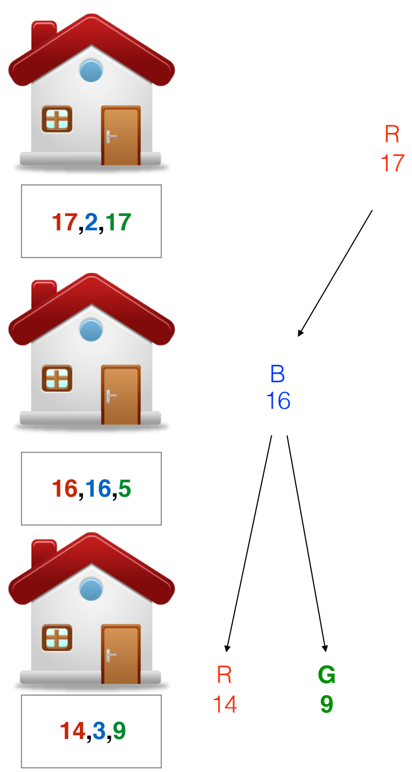
Painting houses



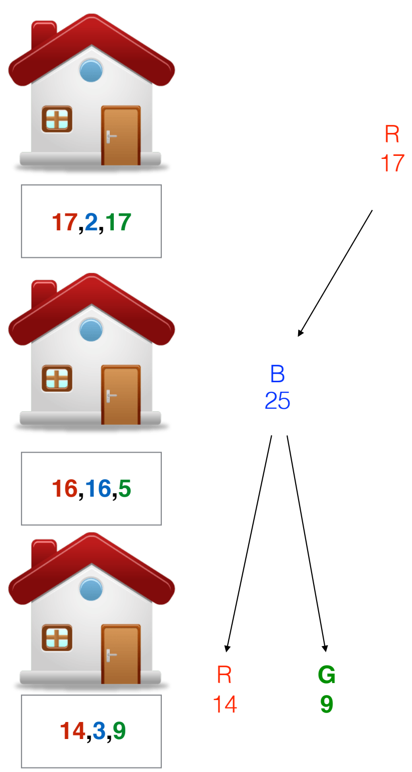
Painting houses



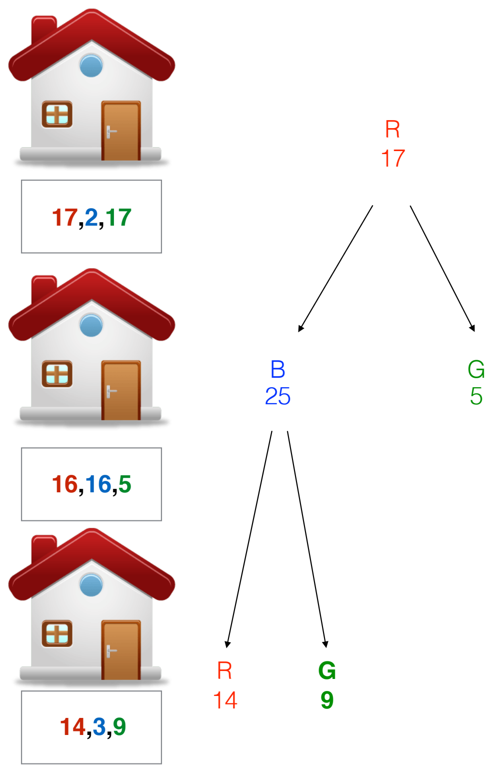
Painting houses



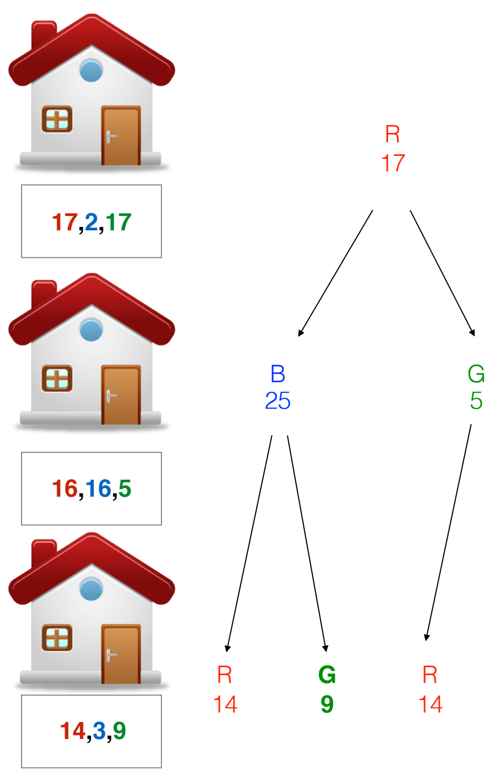
Painting houses



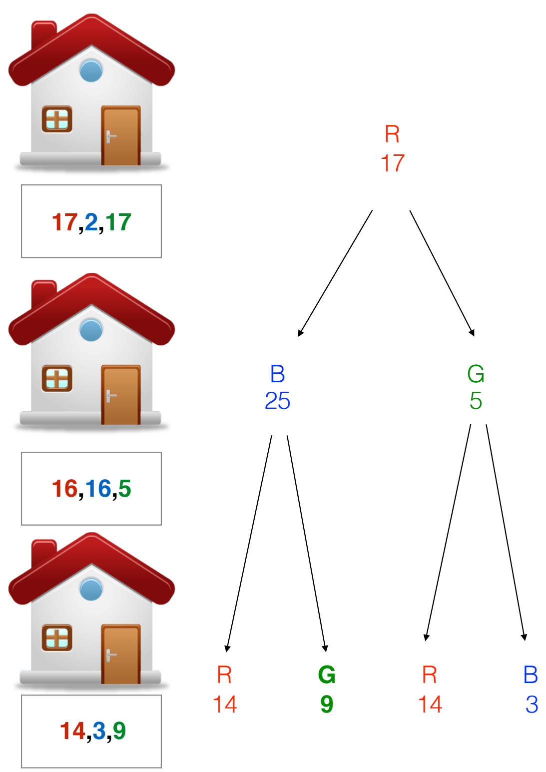
Painting houses



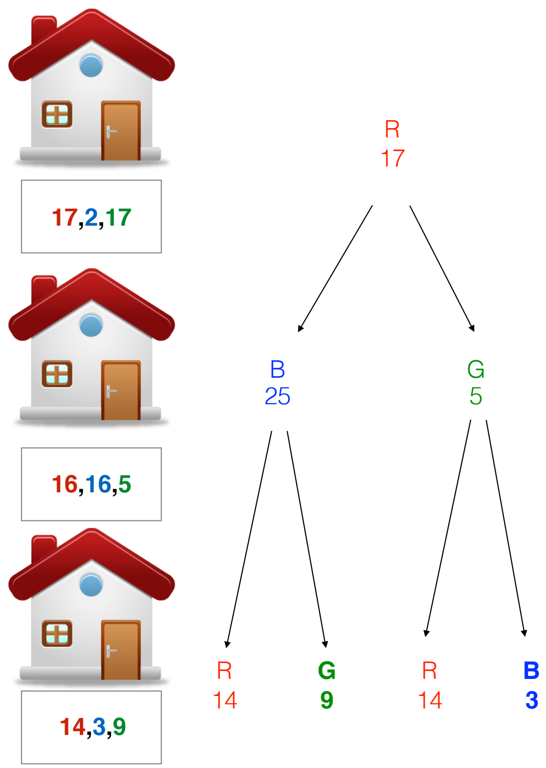
Painting houses



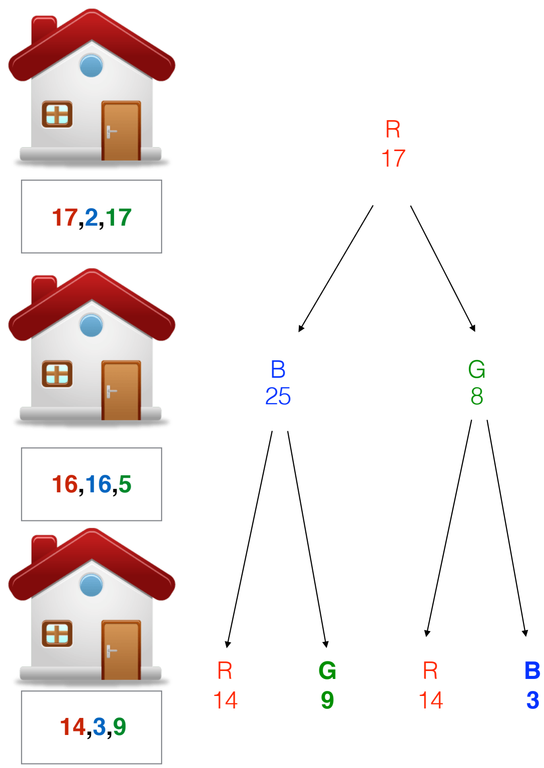
Painting houses



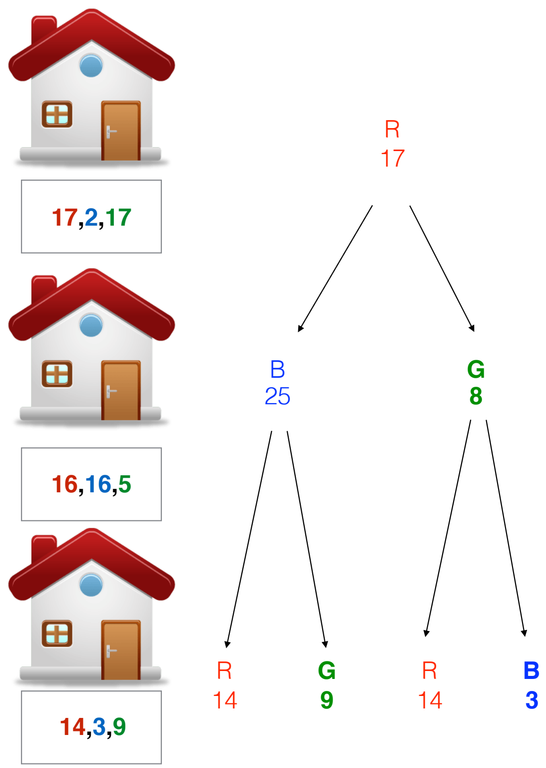
Painting houses



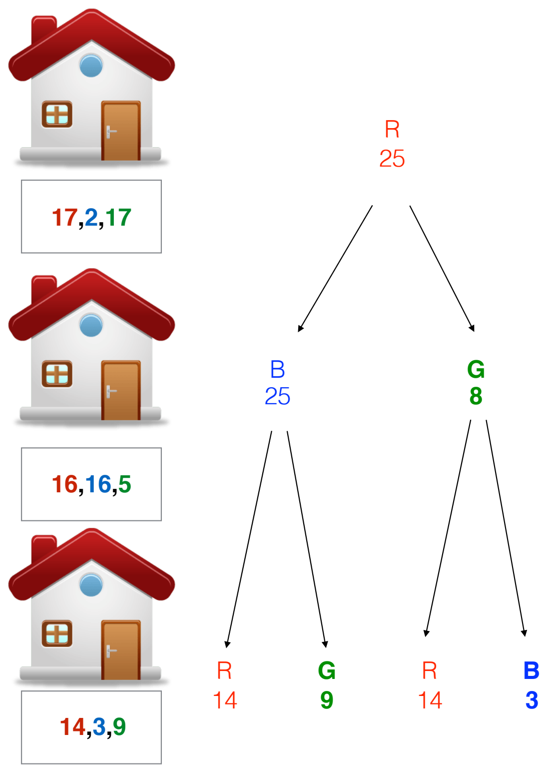
Painting houses



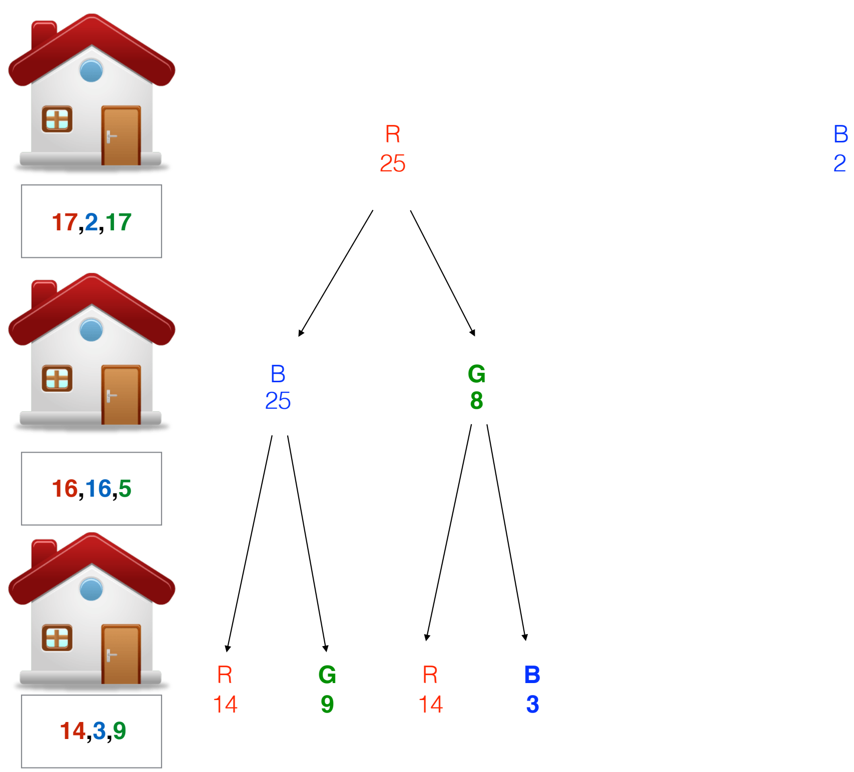
Painting houses



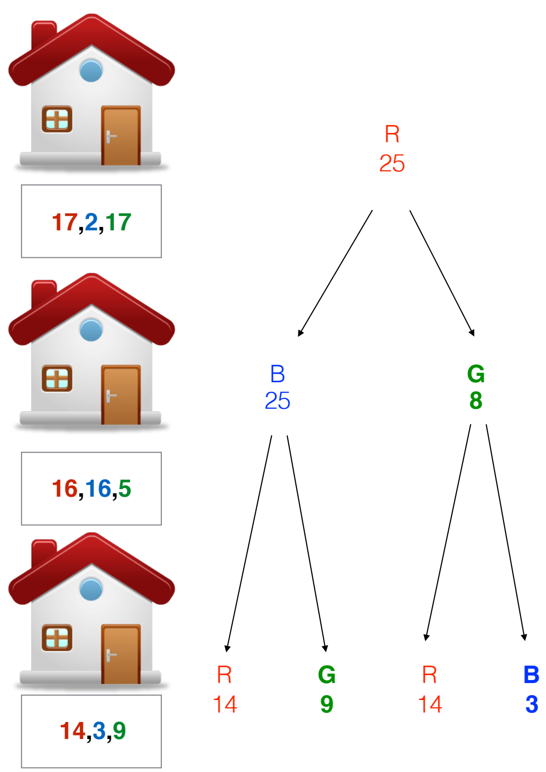
Painting houses



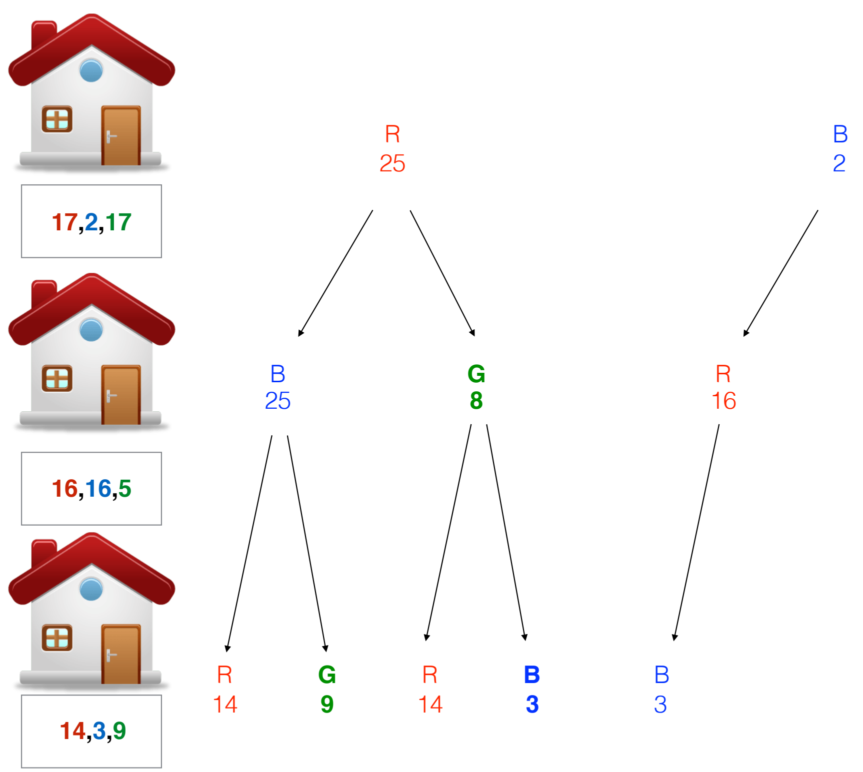
Painting houses



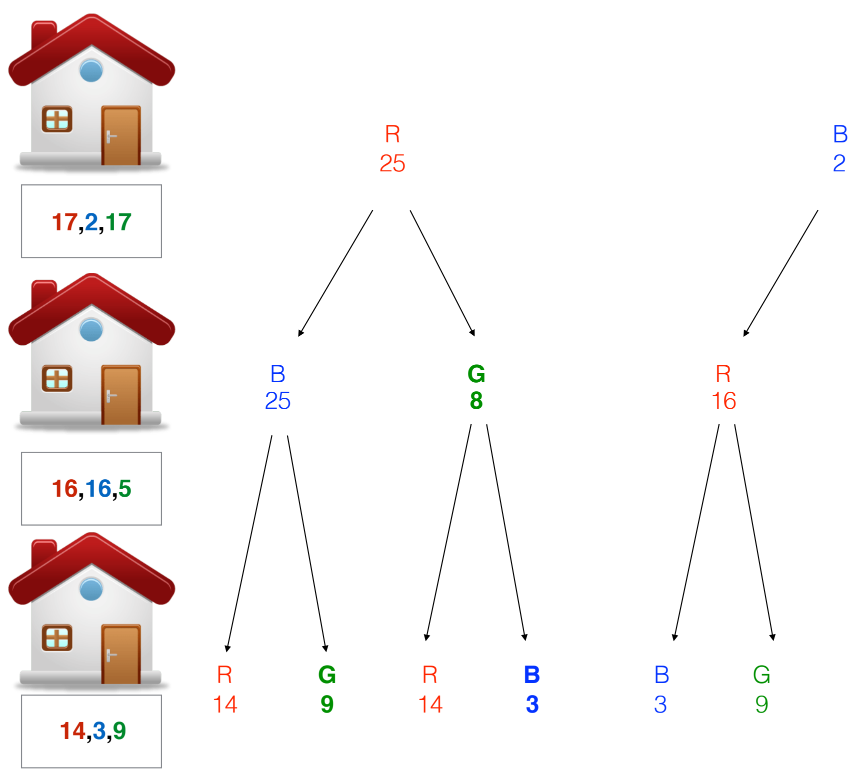
Painting houses



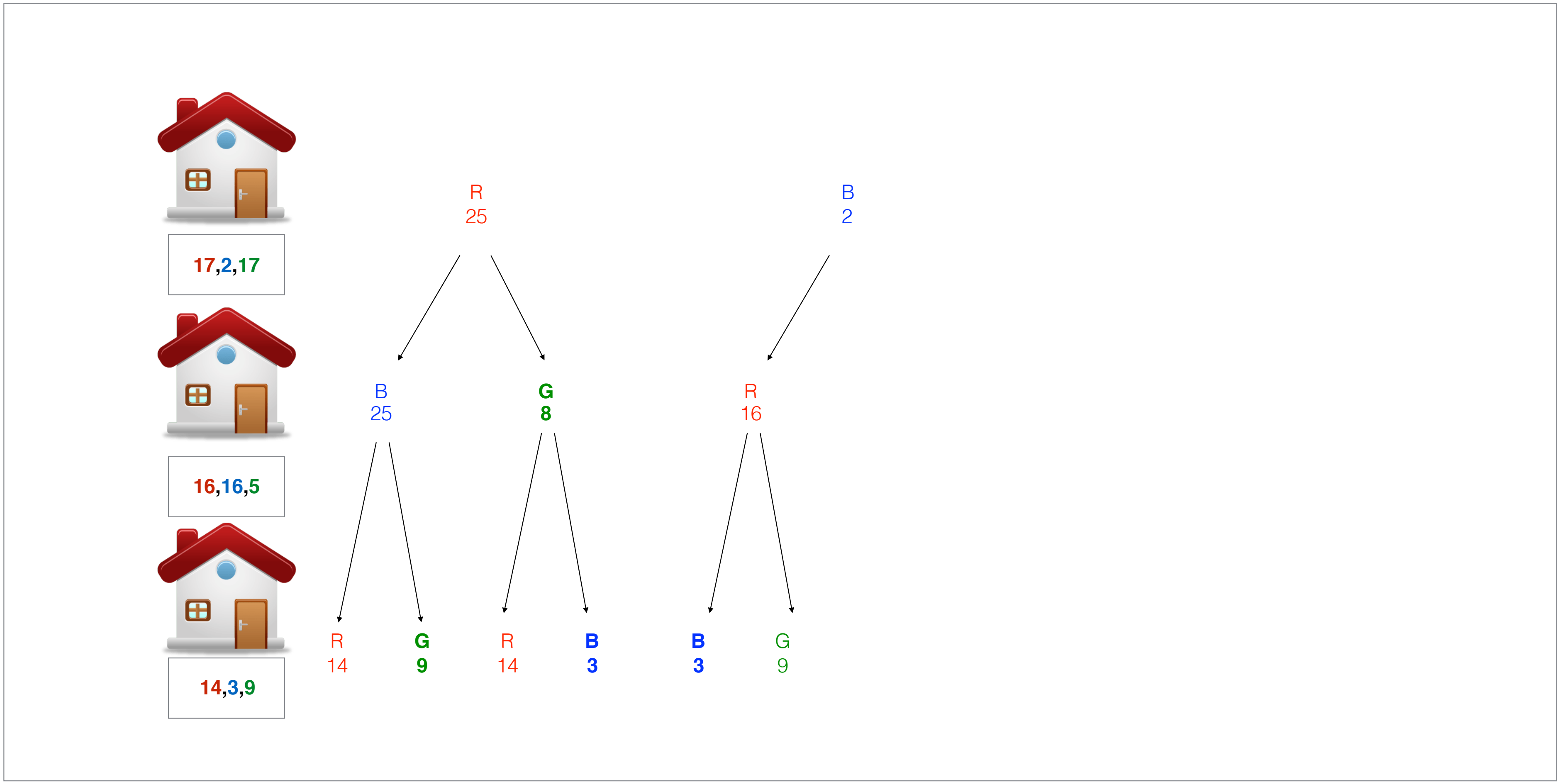
Painting houses



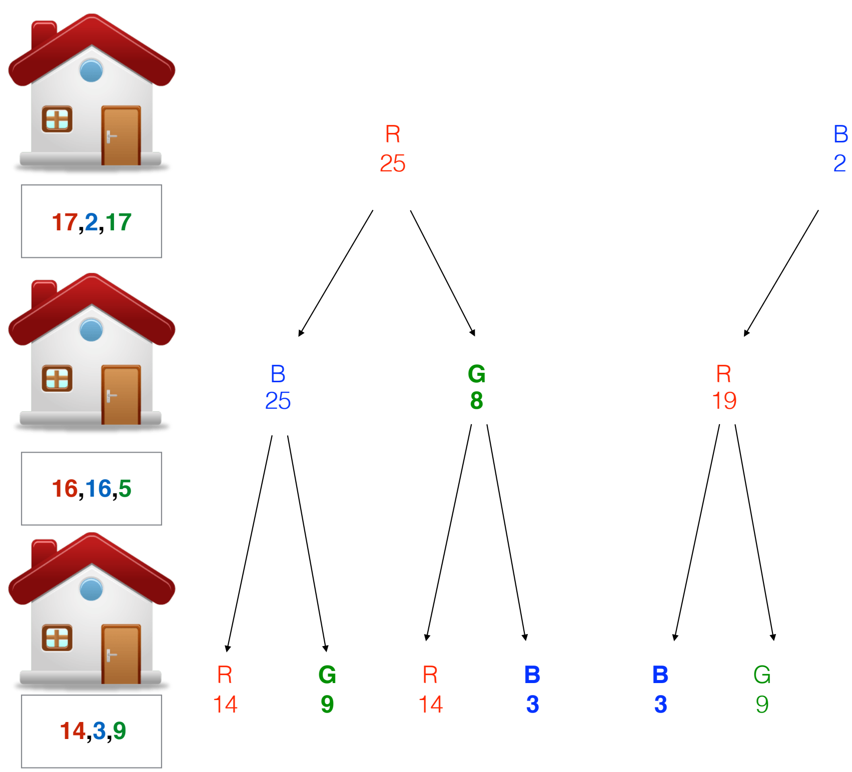
Painting houses



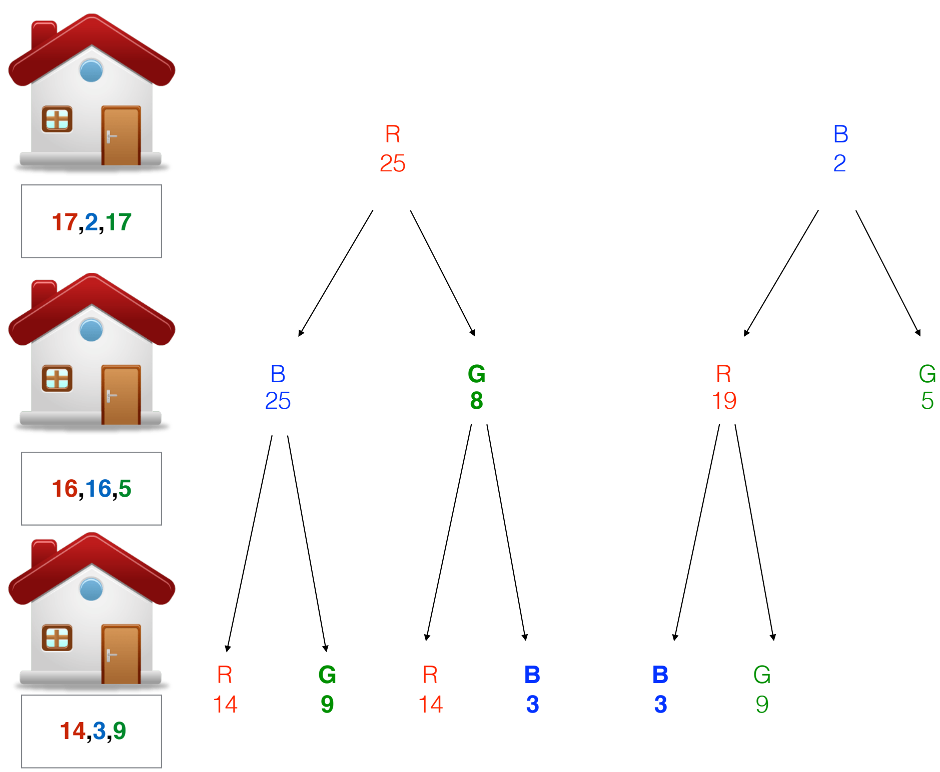
Painting houses



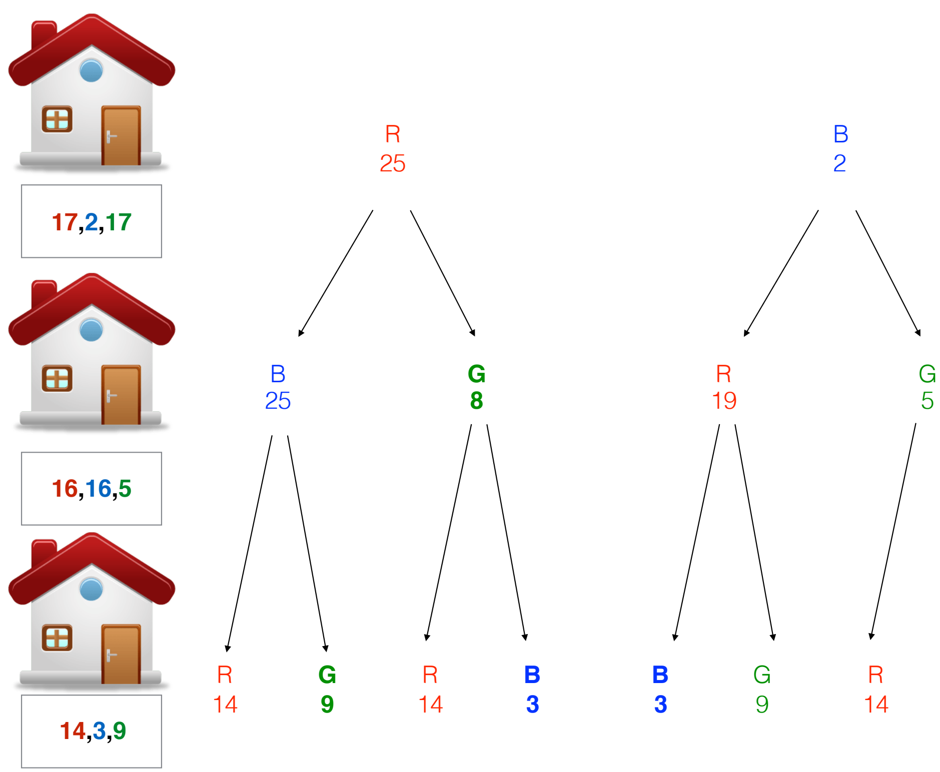
Painting houses



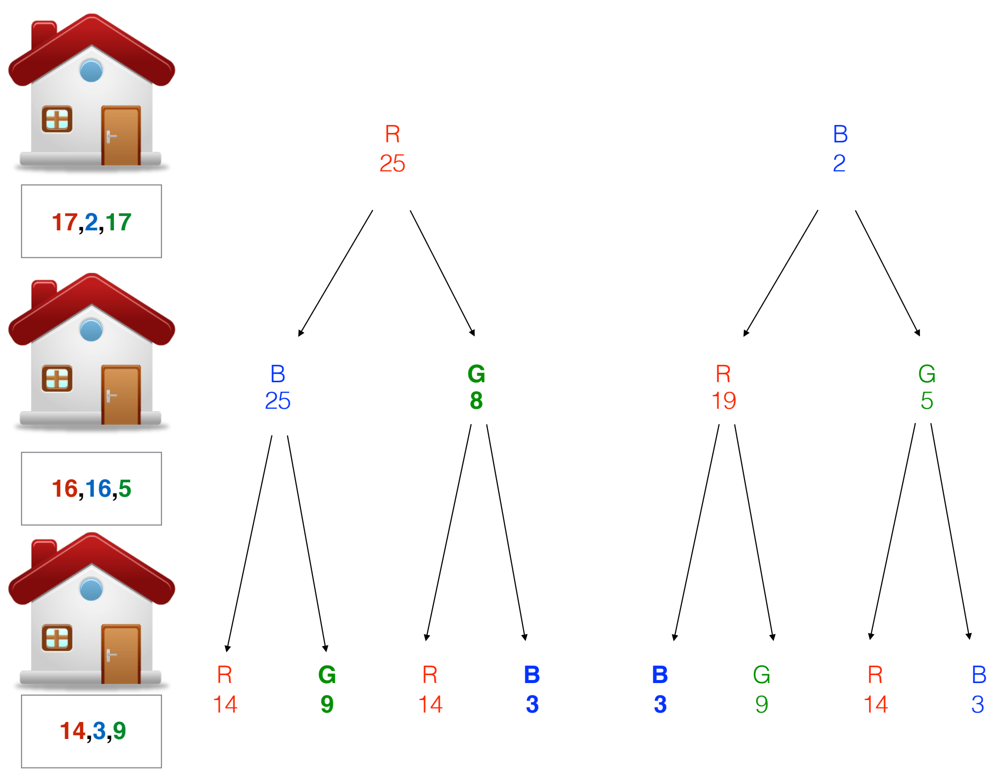
Painting houses



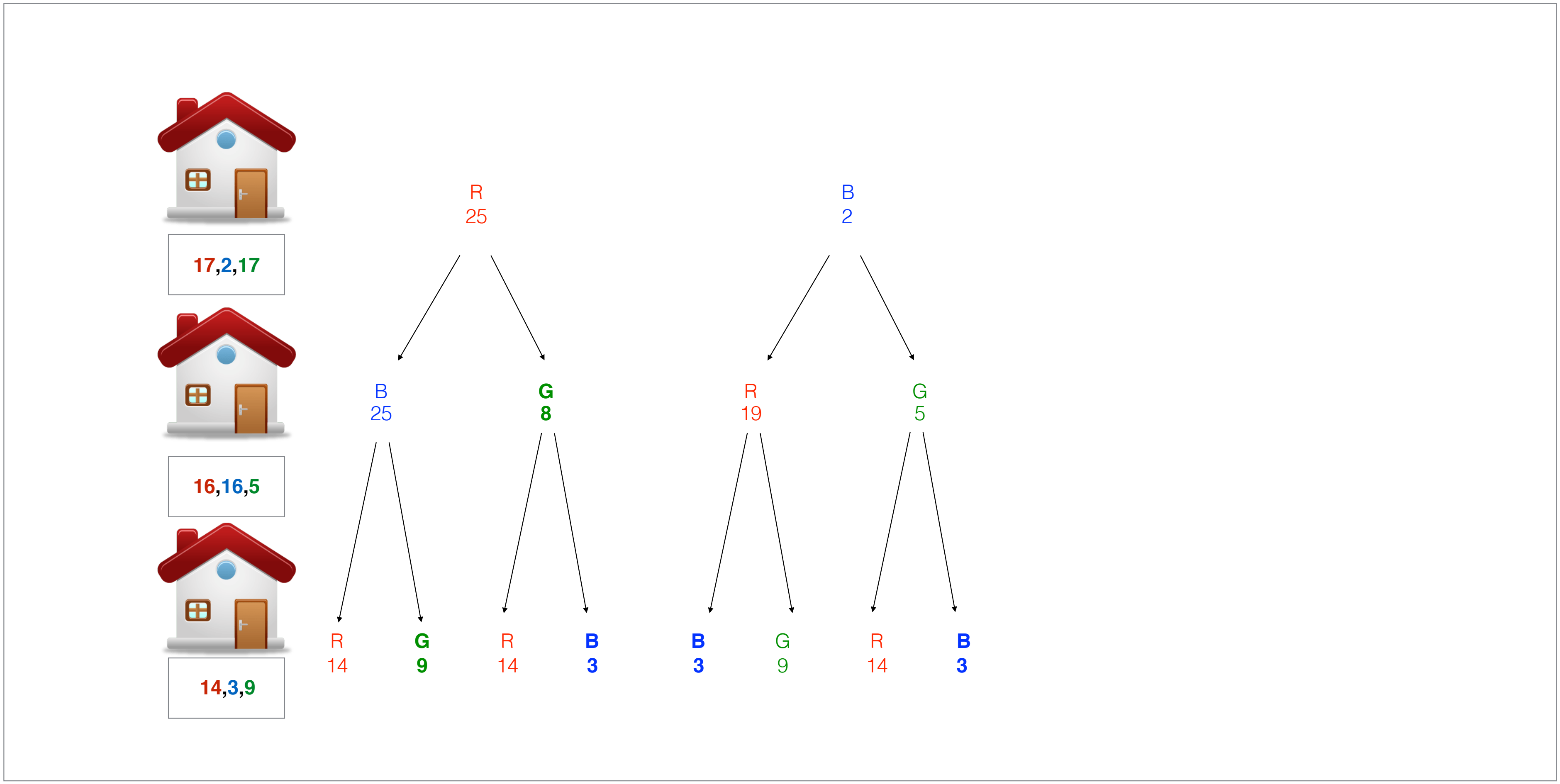
Painting houses



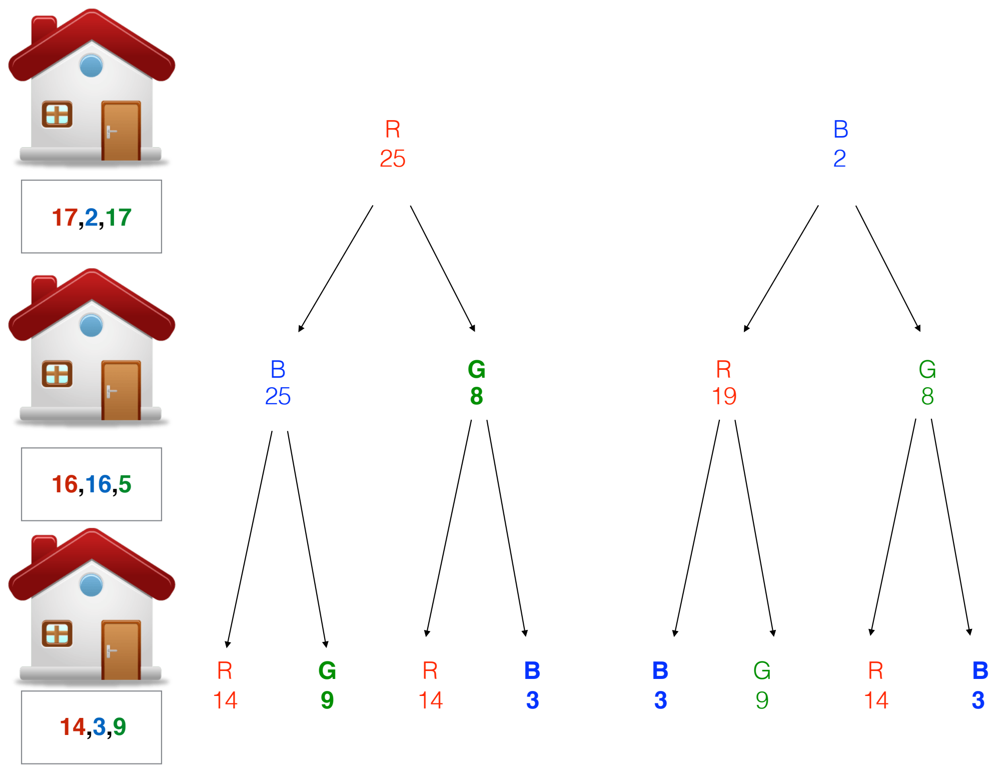
Painting houses



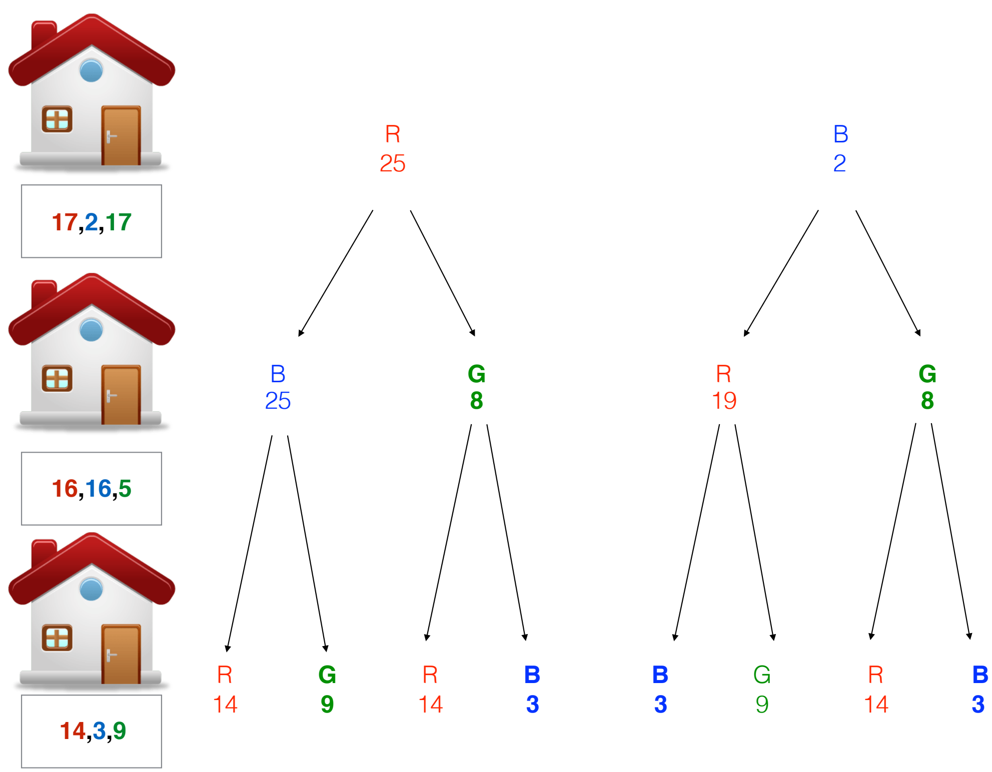
Painting houses



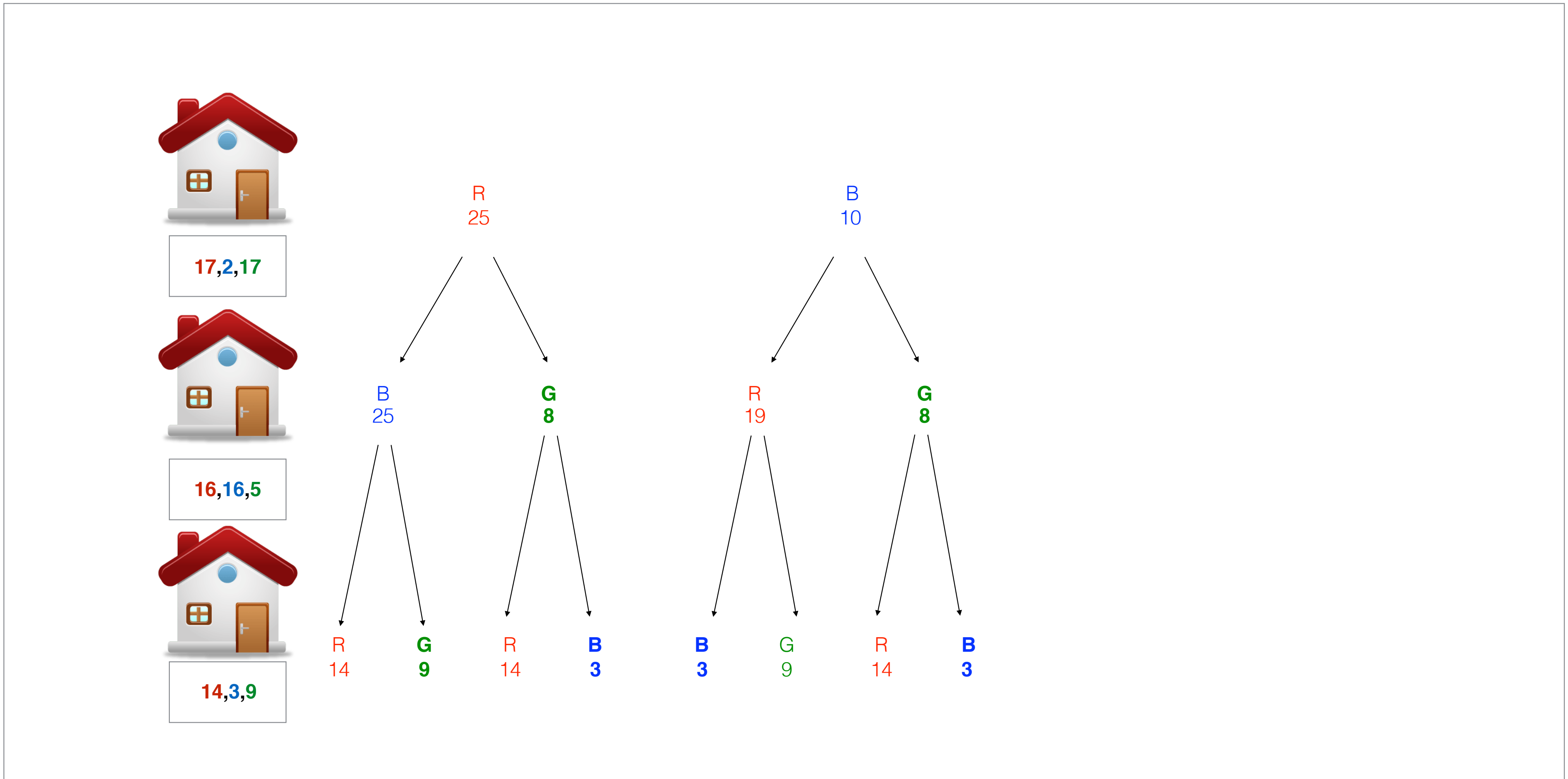
Painting houses



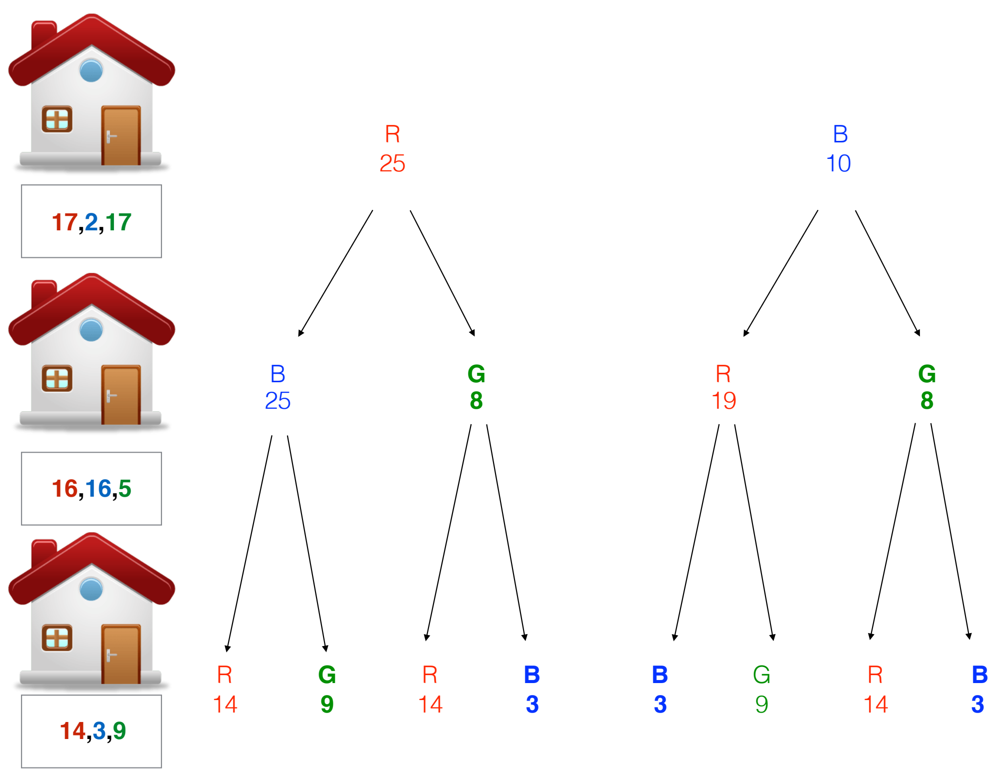
Painting houses



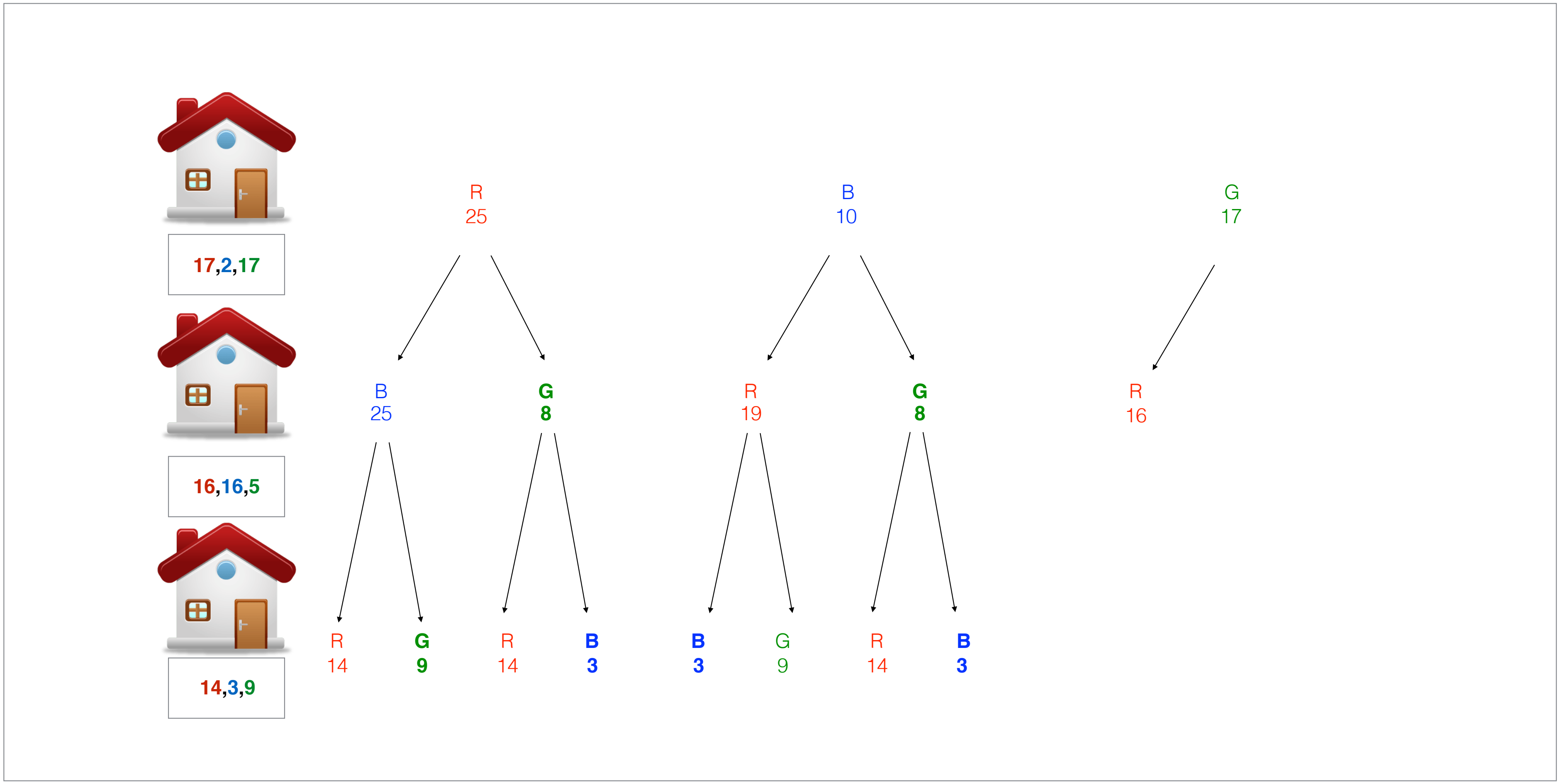
Painting houses



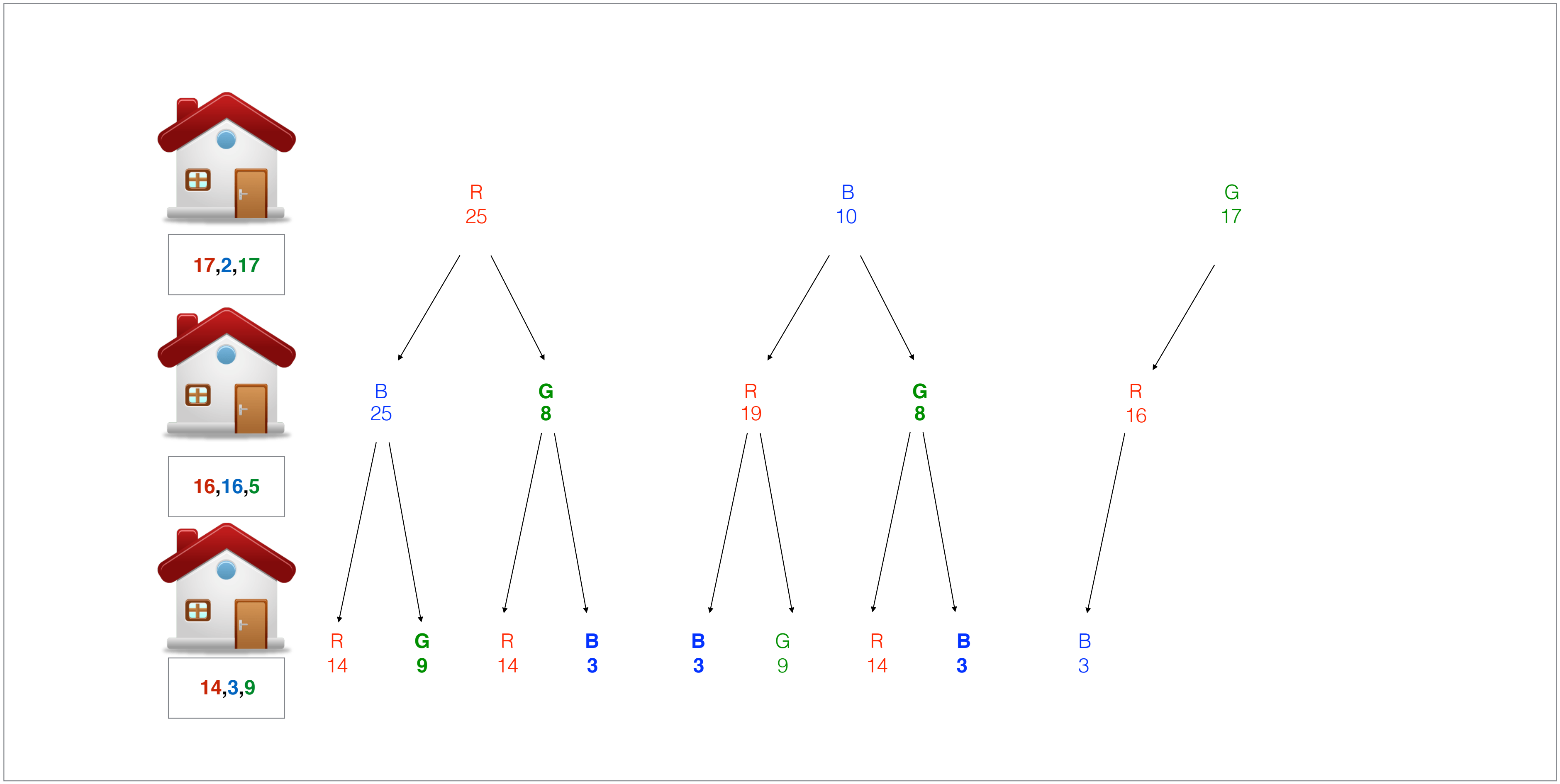
Painting houses



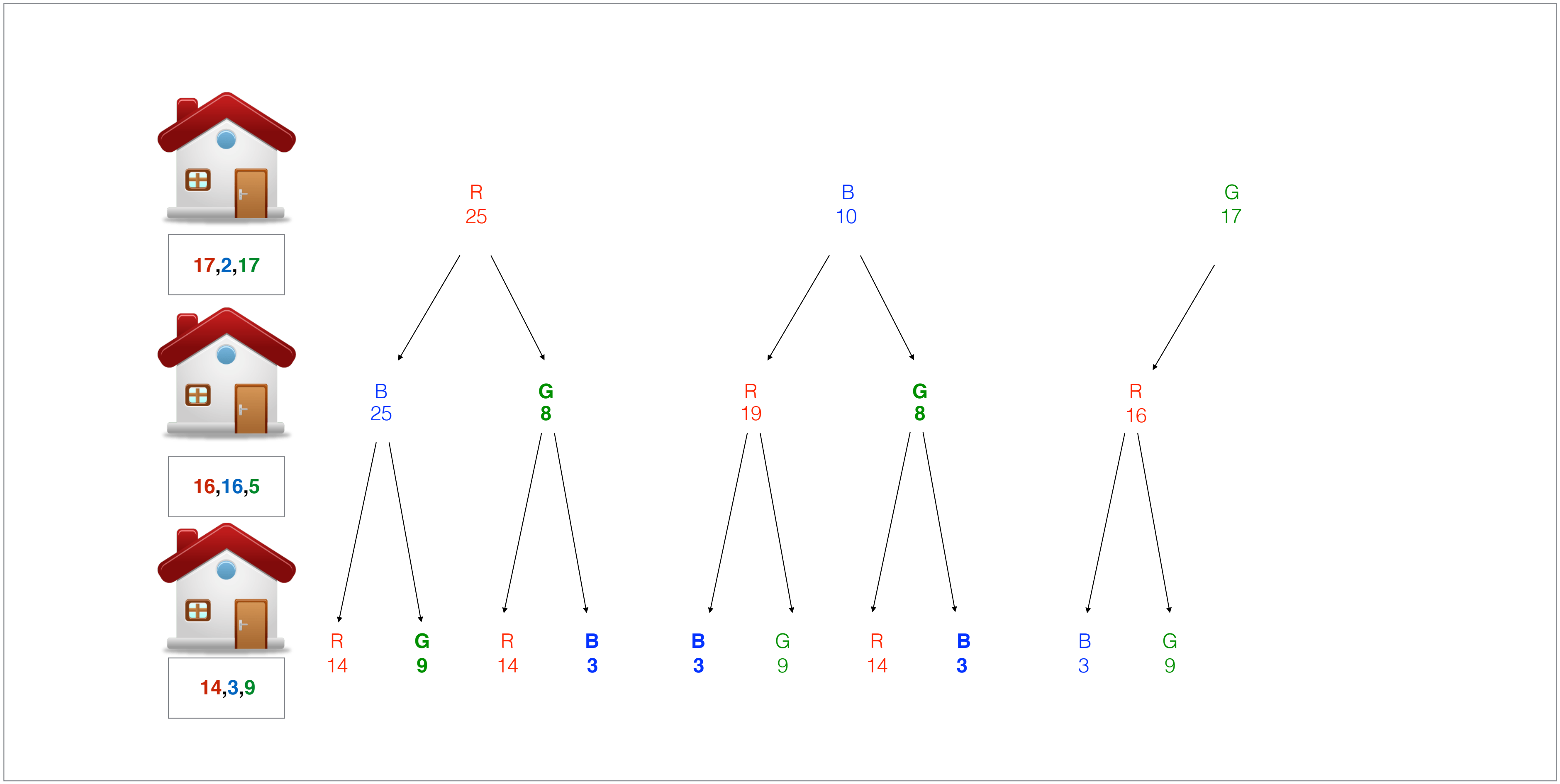
Painting houses



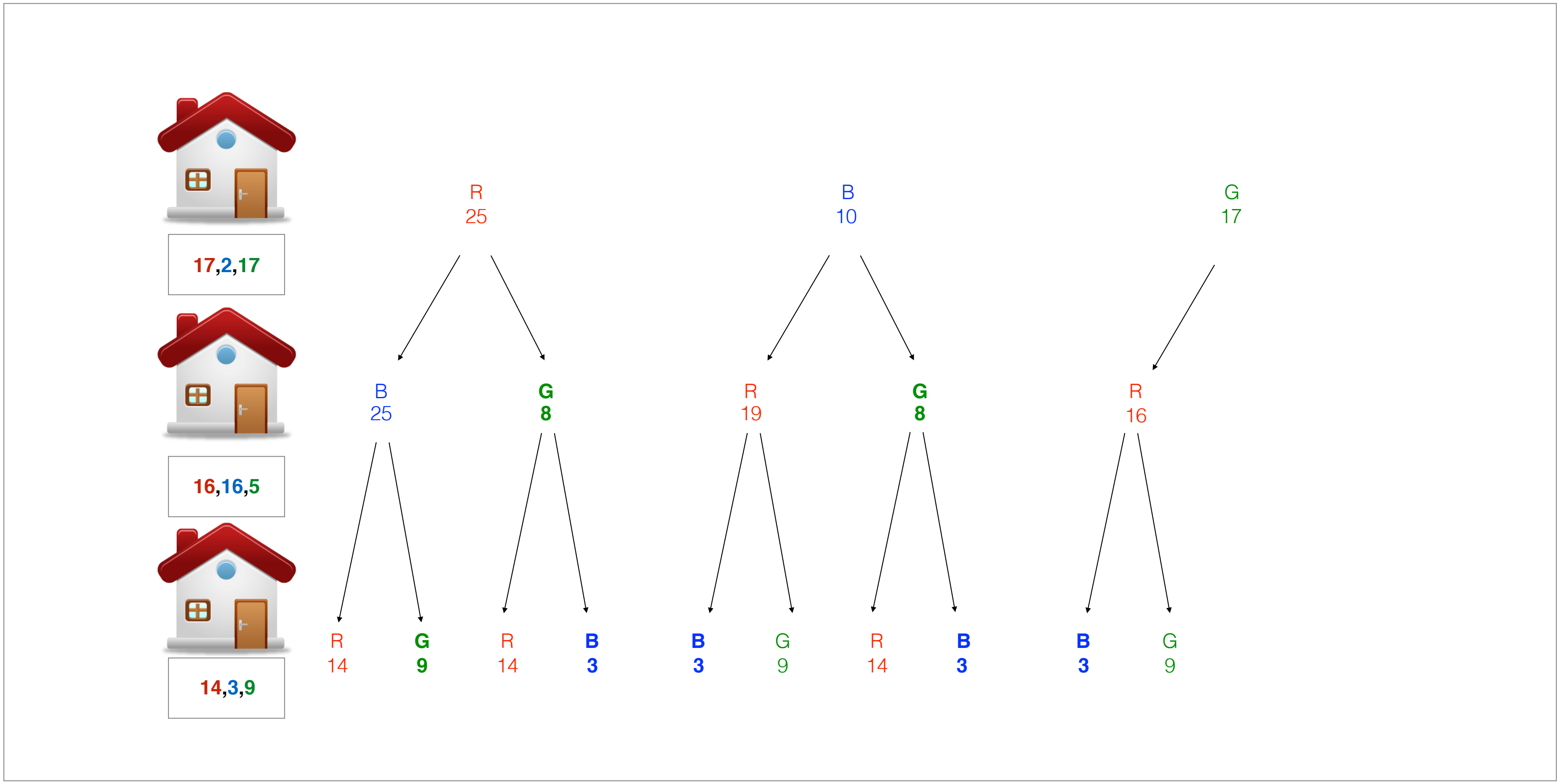
Painting houses



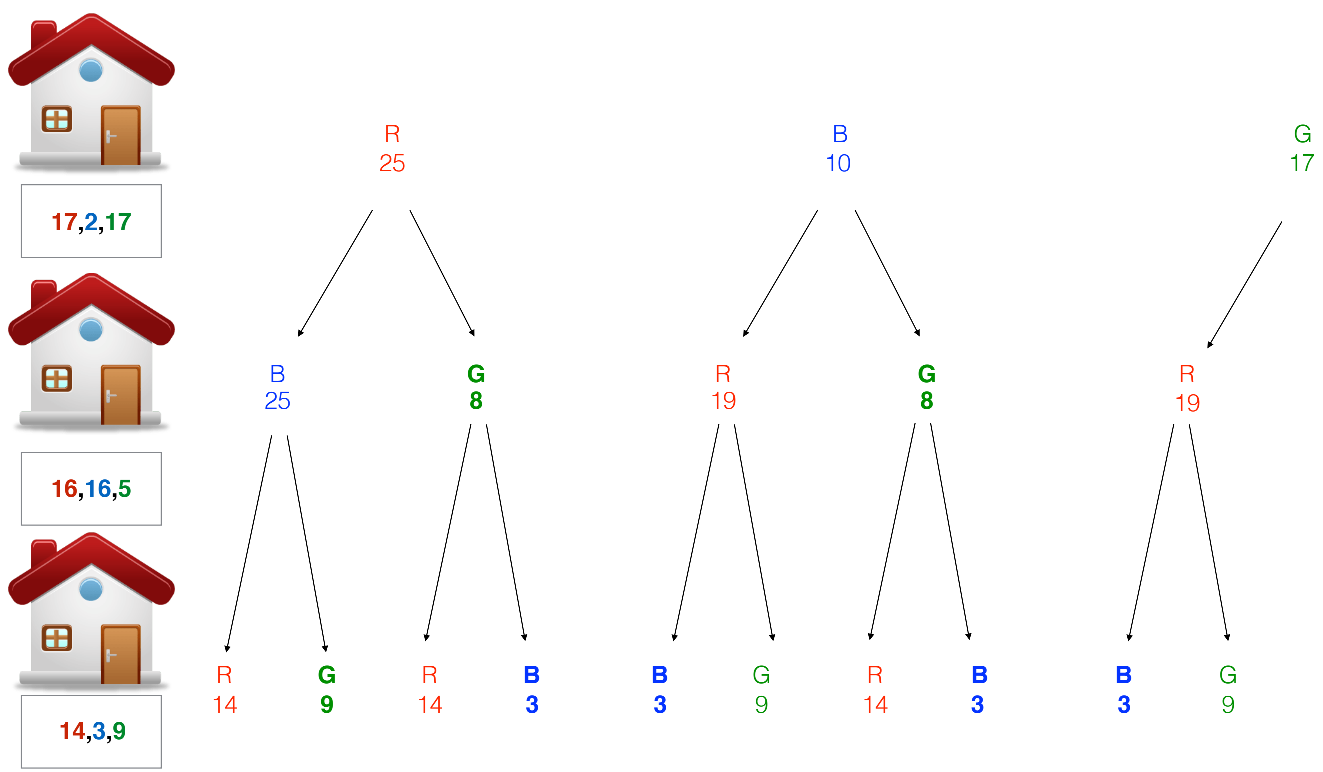
Painting houses



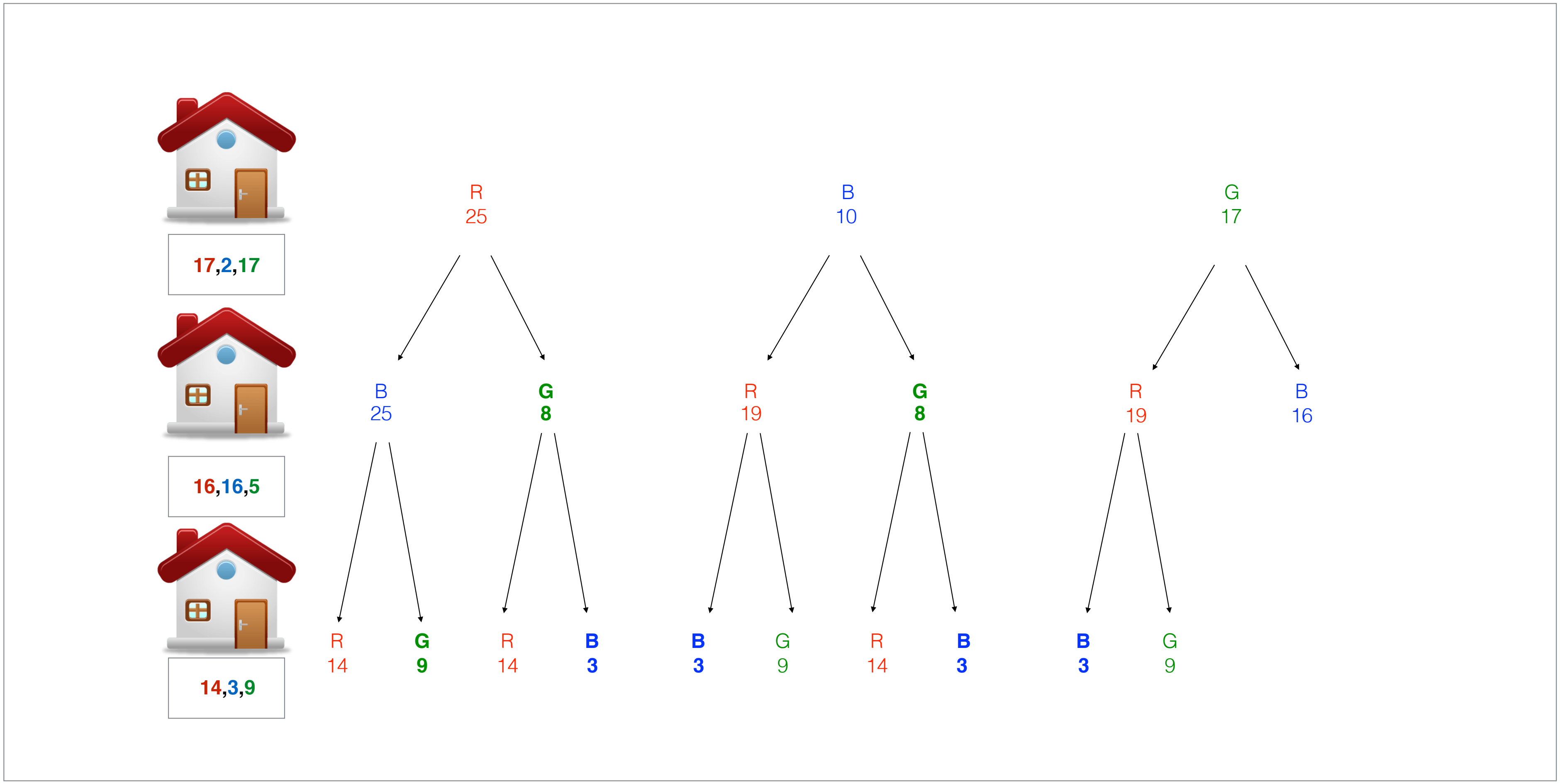
Painting houses



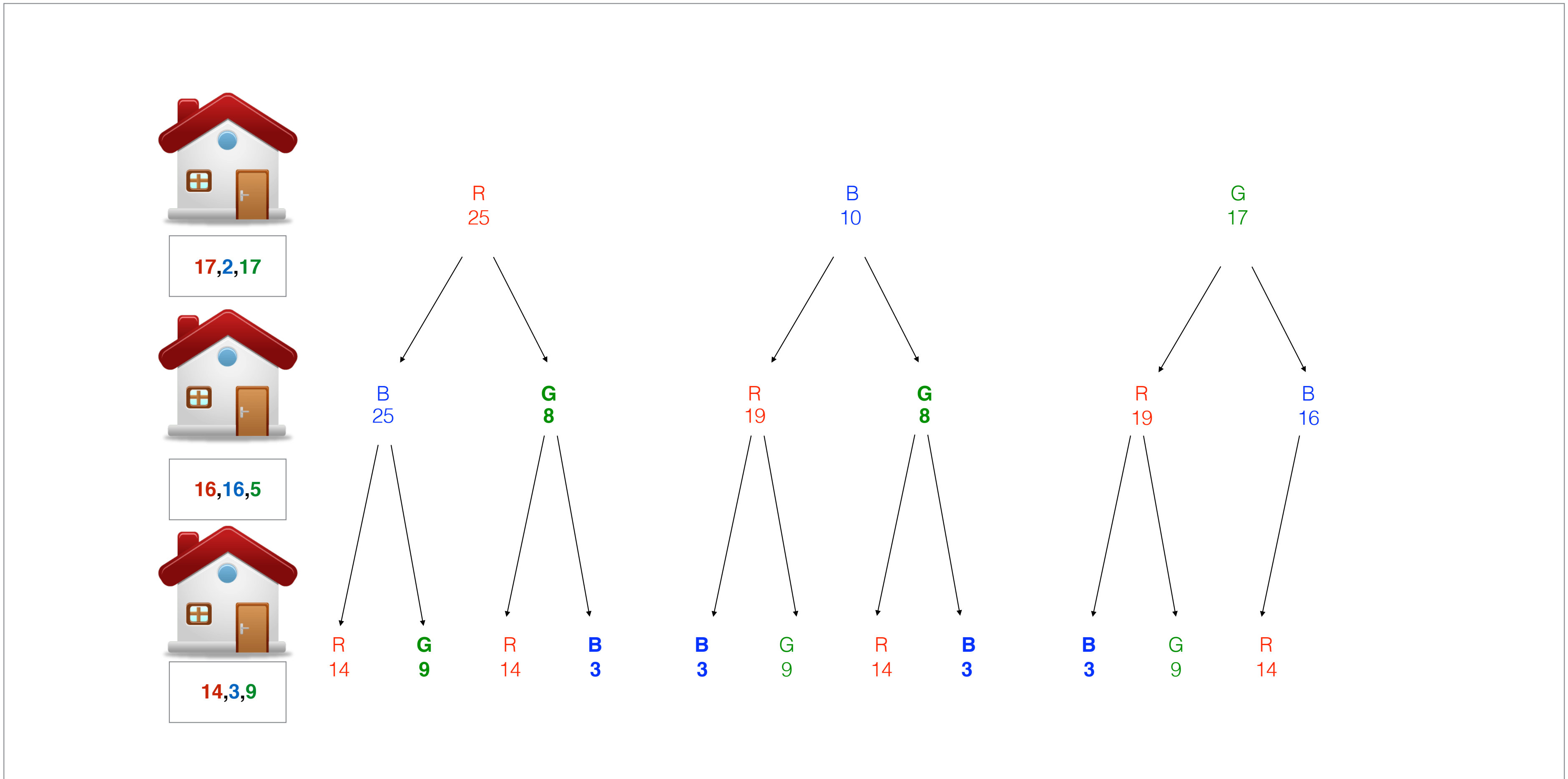
Painting houses



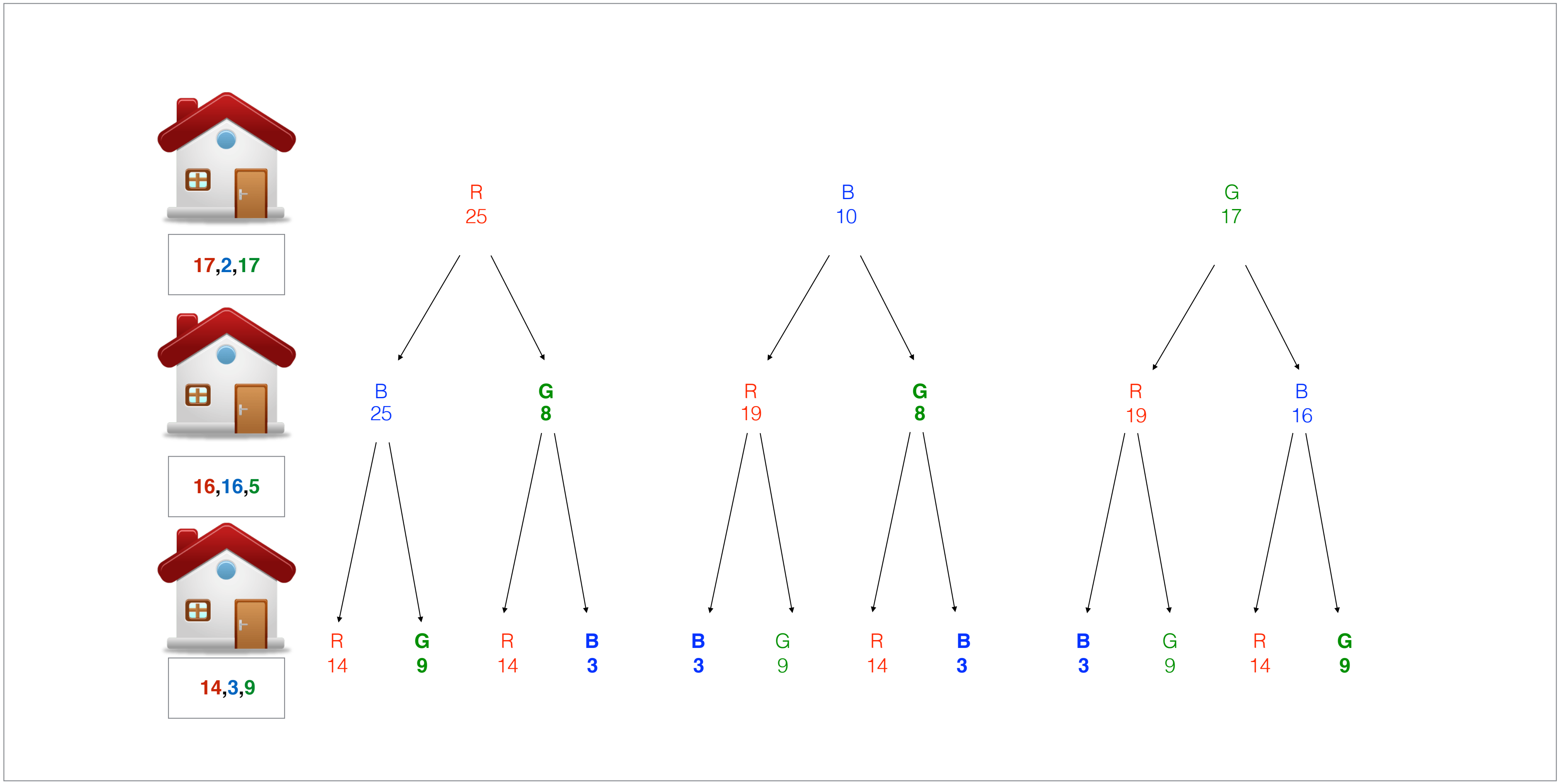
Painting houses



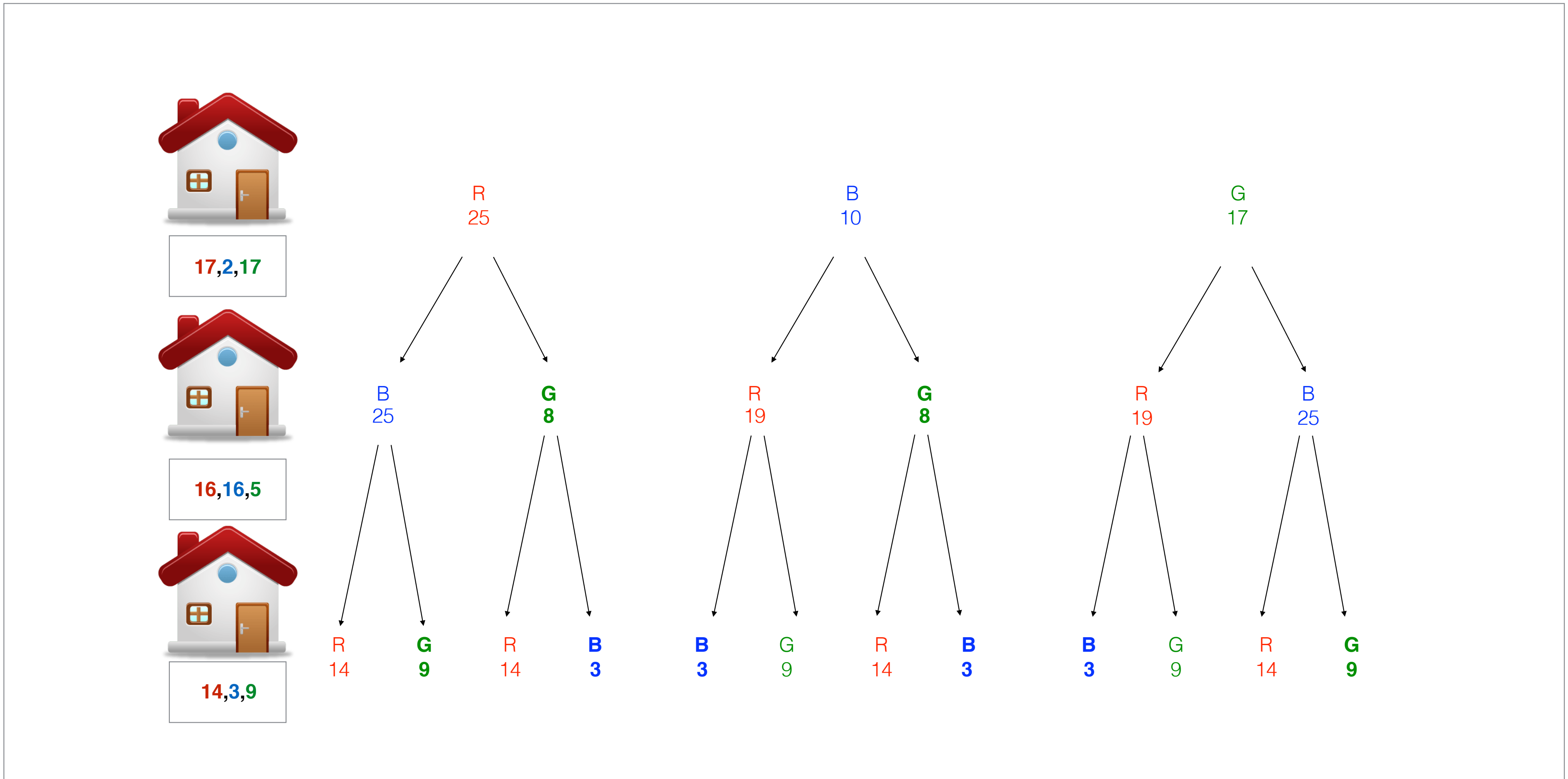
Painting houses



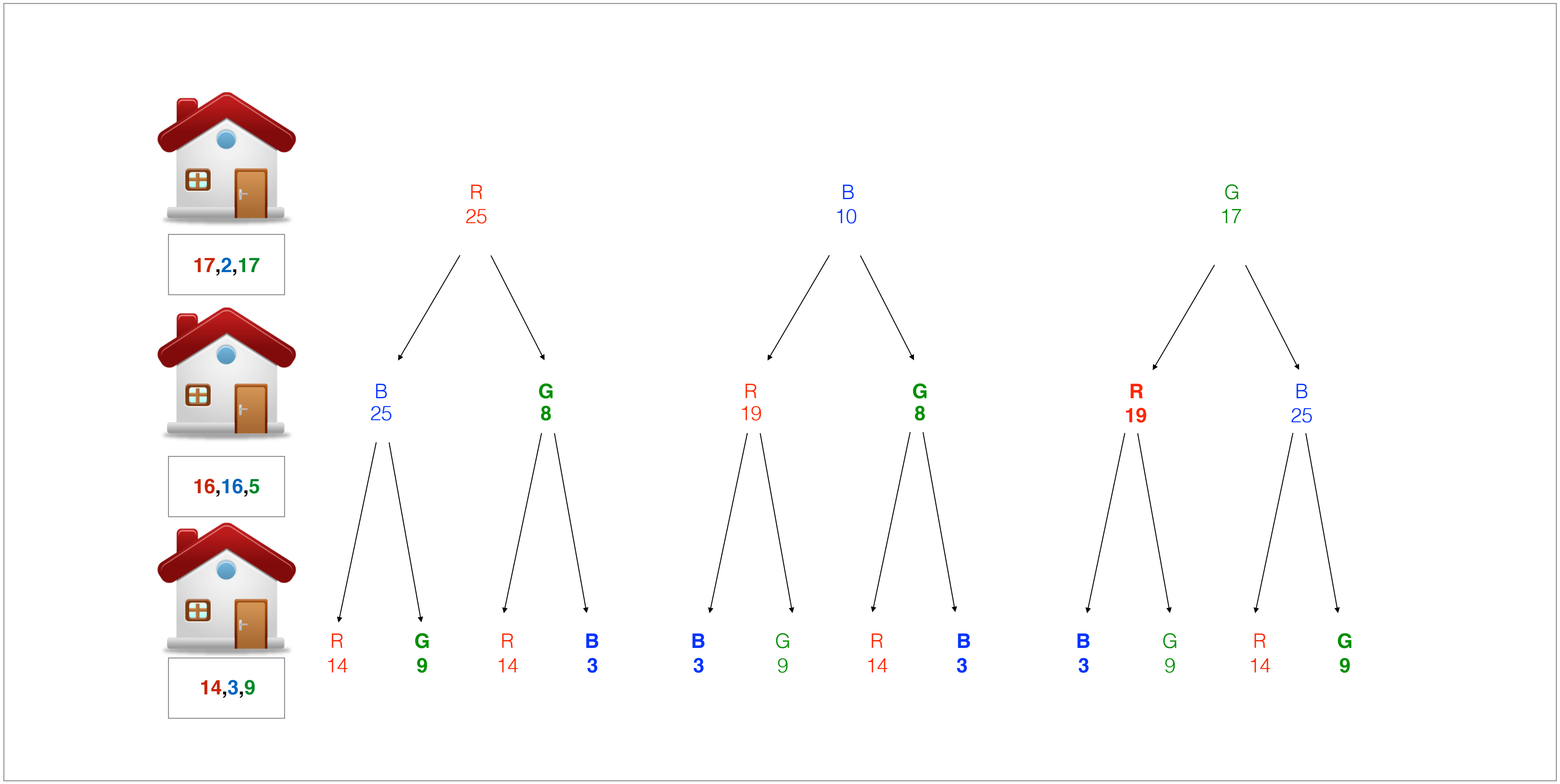
Painting houses



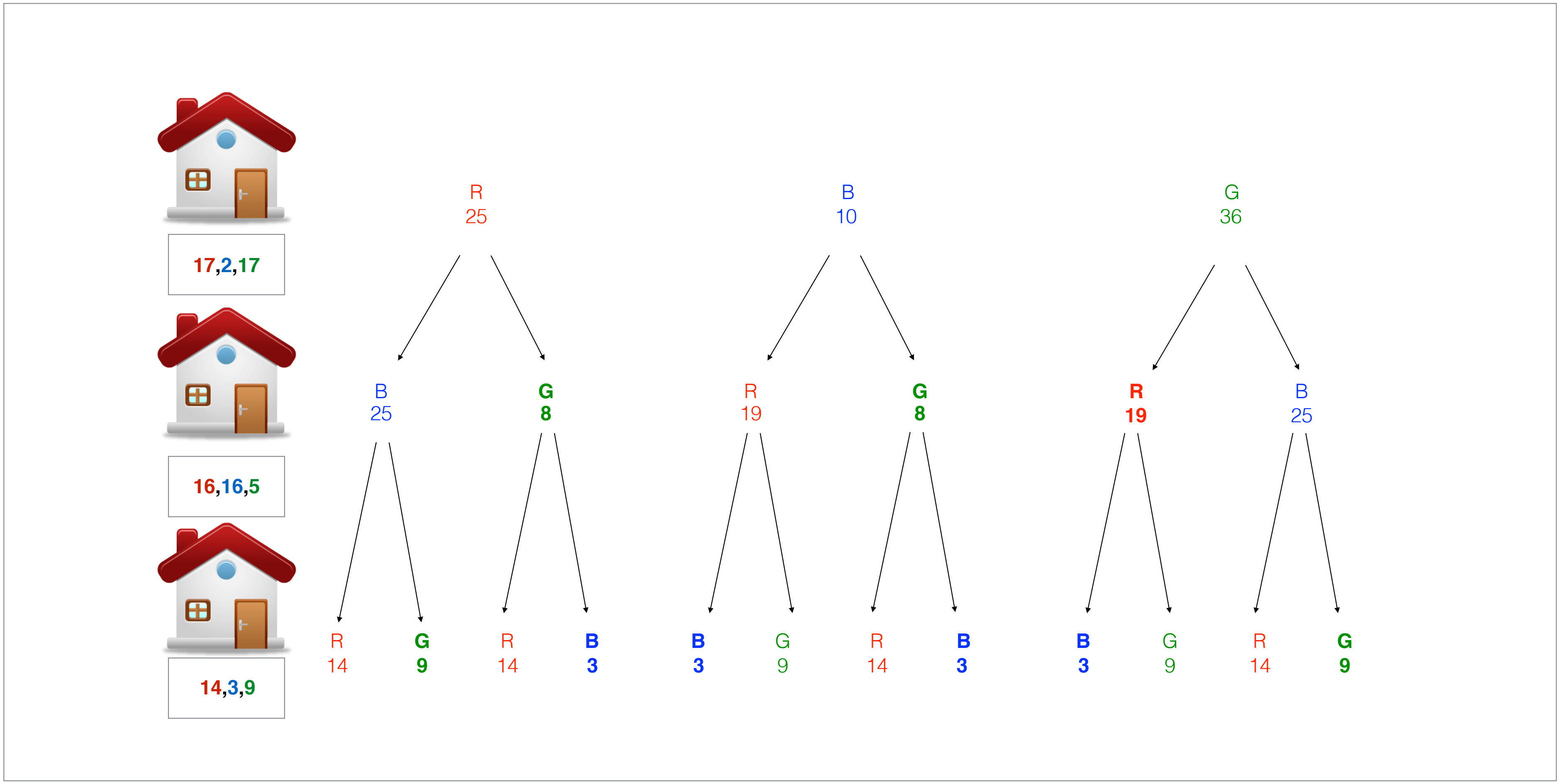
Painting houses



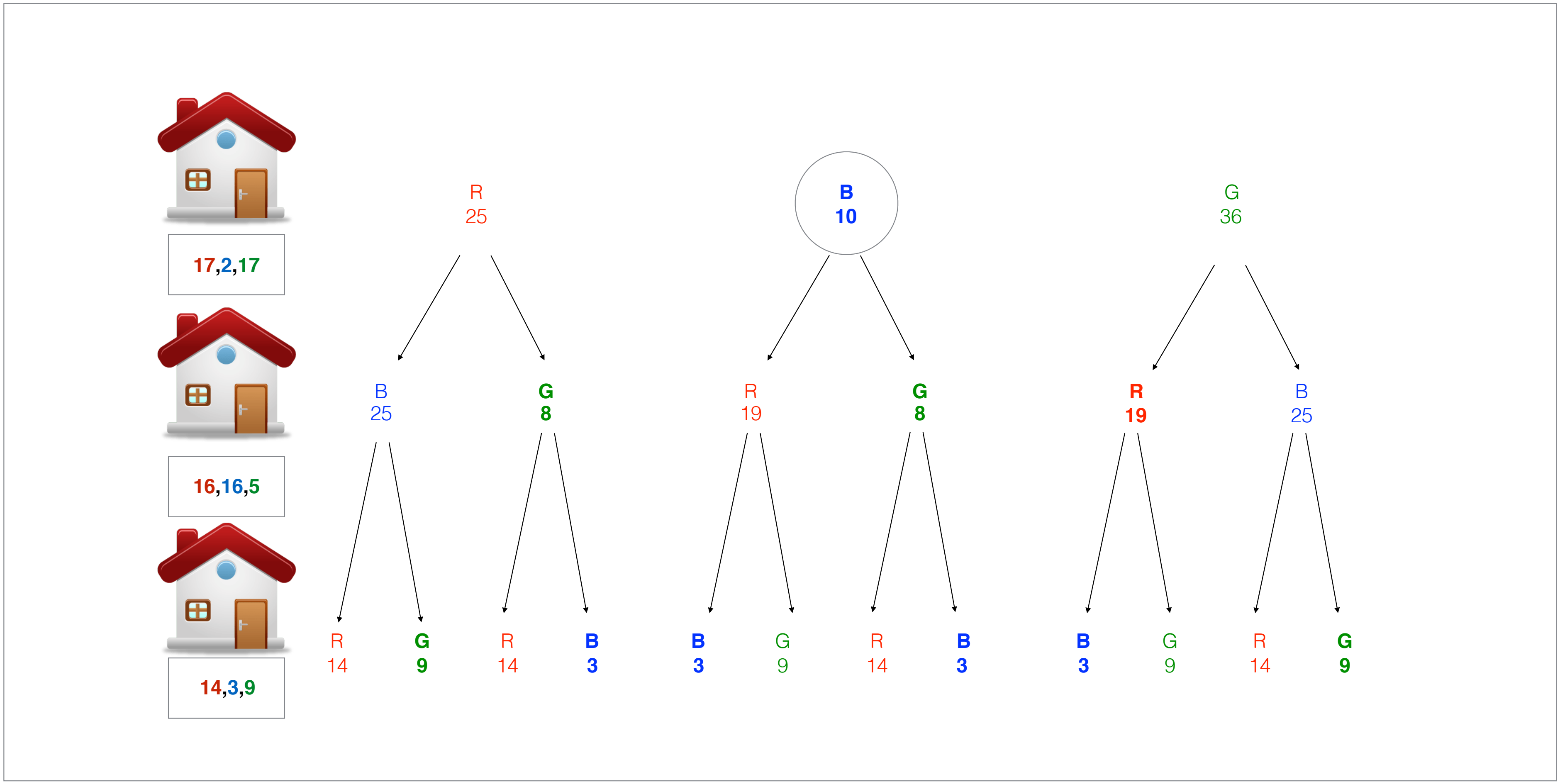
Painting houses



Painting houses



Painting houses



3. Recursive solution

Java

```java
public static final int RED = 0; public static final int BLUE = 1; public static final int GREEN = 2;

public static int minCost(int[][] cost, int i, int color) {
    if (i == cost.length) {
        return 0;
    }
    switch (color) {
        case RED: {
            int costBlue = minCost(cost, i + 1, BLUE);
            int costGreen = minCost(cost, i + 1, GREEN);
            return cost[i][RED] + Math.min(costBlue, costGreen);
        }
        case BLUE: {
            int costRed = minCost(cost, i + 1, RED);
            int costGreen = minCost(cost, i + 1, GREEN);
            return cost[i][BLUE] + Math.min(costRed, costGreen);
        }
        case GREEN: {
            int costRed = minCost(cost, i + 1, RED);
            int costBlue = minCost(cost, i + 1, BLUE);
            return cost[i][GREEN] + Math.min(costRed, costBlue);
        }
    }
    return 0;
}
```

Java

```java
public static int minCost(int[][] cost) {
    int costRed = minCost(cost, 0, RED);
    int costBlue = minCost(cost, 0, BLUE);
    int costGreen = minCost(cost, 0, GREEN);
    return Math.min(costRed, Math.min(costBlue, costGreen));
}
```

Python

```python
RED = 0
BLUE = 1
GREEN = 2

def min_cost(cost, i, color):
    if i == len(cost):
        return 0
    if color == RED:
        cost_blue = min_cost(cost, i + 1, BLUE)
        cost_green = min_cost(cost, i + 1, GREEN)
        return min(cost_blue, cost_green)
    elif color == BLUE:
        cost_red = min_cost(cost, i + 1, RED)
        cost_green = min_cost(cost, i + 1, GREEN)
        return min(cost_red, cost_green)
    elif color == GREEN:
        cost_red = min_cost(cost, i + 1, RED)
        cost_blue = min_cost(cost, i + 1, BLUE)
        return min(cost_red, cost_blue)

cost = [[17, 2, 17], [16, 16, 5], [14, 3, 19]]
cost_red = min_cost(cost,0,RED)
cost_blue = min_cost(cost,0,BLUE)
cost_green = min_cost(cost,0,GREEN)
min_cost = min(cost_red,min(cost_blue,cost_green))
```

4. Memoize

Memoize

There are two state variables, we have to use a 2D array to cache the result.

Java

```java
public static int minCostMemo(int[][] cost, int i, int color, int[][] cache) {
    if (i == cost.length) {
        return 0;
    }
    if (cache[i][color] != -1) {
        return cache[i][color];
    }
    switch (color) {
        case RED: {
            int costBlue = minCostMemo(cost, i + 1, BLUE, cache);
            int costGreen = minCostMemo(cost, i + 1, GREEN, cache);
            return cache[i][color] = cost[i][RED] + Math.min(costBlue, costGreen);
        }
        case BLUE: {
            int costRed = minCostMemo(cost, i + 1, RED, cache);
            int costGreen = minCostMemo(cost, i + 1, GREEN, cache);
            return cache[i][color] = cost[i][BLUE] + Math.min(costRed, costGreen);
        }
        case GREEN: {
            int costRed = minCostMemo(cost, i + 1, RED, cache);
            int costBlue = minCostMemo(cost, i + 1, BLUE, cache);
            return cache[i][color] = cost[i][GREEN] + Math.min(costRed, costBlue);
        }
    }
    return 0;
}
```

Java

```java
public static int minCostMemo(int[][] cost) {
    int[][] cache = new int[cost.length][cost[0].length];
    for (int[] row : cache) {
        Arrays.fill(row, -1);
    }
    int costRed = minCostMemo(cost, 0, RED, cache);
    int costBlue = minCostMemo(cost, 0, BLUE, cache);
    int costGreen = minCostMemo(cost, 0, GREEN, cache);
    return Math.min(costRed, Math.min(costBlue, costGreen));
}
```

Python

```python
def min_cost_memo(costs, i, color, cache):
    if i == len(costs):
        return 0
    if cache[i][color] != -1:
        return cache[i][color]
    if color == RED:
        cost_blue = min_cost_memo(costs, i + 1, BLUE, cache)
        cost_green = min_cost_memo(costs, i + 1, GREEN, cache)
        cache[i][RED] = costs[i][RED] + min(cost_blue, cost_green)
        return cache[i][RED]
    elif color == BLUE:
        cost_red = min_cost_memo(costs, i + 1, RED, cache)
        cost_green = min_cost_memo(costs, i + 1, GREEN, cache)
        cache[i][BLUE] = costs[i][BLUE] + min(cost_red, cost_green)
        return cache[i][BLUE]
    elif color == GREEN:
        cost_red = min_cost_memo(costs, i + 1, RED, cache)
        cost_blue = min_cost_memo(costs, i + 1, BLUE, cache)
        cache[i][GREEN] = costs[i][GREEN] + min(cost_red, cost_blue)
        return cache[i][GREEN]
```

Python

```python
painting_cost = [[17, 2, 17], [16, 16, 5], [14, 3, 19]]
cache = [[-1 for _ in range(0, 3)] for _ in range(0,
len(painting_cost))]
paint_red = min_cost_memo(painting_cost,0,RED,cache)
paint_red = min_cost_memo(painting_cost,0,BLUE,cache)
paint_red = min_cost_memo(painting_cost,0,GREEN,cache)
min_cost = min(paint_red,min(paint_blue,paint_green))
print(min_cost)
```

5. Bottom up approach

Bottom up approach

We will flip the top down solution. We will solve the smaller problems first. We first try painting house 0 with all the colors and see how much it costs. Then we use the results to try out different options for house 1, then house 2 and so on until we reach the last house. We use a 2D array to store all the results, one step at a time. Finally the result to the problem will be available in the last position.

min_cost(i,RED) = cost[i][RED] + MIN(min_cost(i-1,BLUE),min_cost(i-1,GREEN))

min_cost(i,BLUE) = cost[i][BLUE] + MIN(min_cost(i-1,RED),min_cost(i-1,BLUE))

min_cost(i,GREEN) = cost[i][GREEN] + MIN(min_cost(i-1,BLUE),min_cost(i-1,GREEN))

Base case

When i == 0 , return 0

for i=0,1,2,3...N

dp[i][RED] = cost[i][RED]+MIN(dp[i-1][BLUE],dp[i-1][GREEN])

dp[i][BLUE] = cost[i][BLUE]+MIN(dp[i-1][RED],dp[i-1][GREEN])

dp[i][GREEN] = cost[i][GREEN]+MIN(dp[i-1][RED],dp[i-1][BLUE])

Painting houses

17,2,17

16,16,5

14,3,9

RED	BLUE	GREEN
0	0	0

Painting houses

RED	BLUE	GREEN
0	0	0
17+0 = 17		

Painting houses

17,2,17

16,16,5

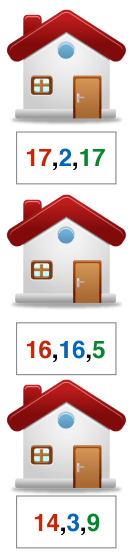
14,3,9

RED	BLUE	GREEN
0	0	0
17	2+0 = 2	

Painting houses

RED	BLUE	GREEN
0	0	0
17	2	17+0=17

Painting houses



RED	BLUE	GREEN
0	0	0
17	2	17
16+2=18		

Painting houses

17,2,17

16,16,5

14,3,9

RED	BLUE	GREEN
0	0	0
17	2	17
18	16+17 = 33	

Painting houses

RED	BLUE	GREEN
0	0	0
17	2	17
18	33	5+2=7

Painting houses

RED	BLUE	GREEN
0	0	0
17	2	17
18	33	7
14+7=21		

Painting houses

RED	BLUE	GREEN
0	0	0
17	2	17
18	33	7
21	3+7=10	

Painting houses

RED	BLUE	GREEN
0	0	0
17	2	17
18	33	7
21	10	9+18=27

Painting houses

17,2,17

16,16,5

14,3,9

RED	BLUE	GREEN
0	0	0
17	2	17
18	33	7
21	10	27

Painting houses

17,2,17

16,16,5

14,3,9

RED	BLUE	GREEN
0	0	0
17	2	17
18	33	7
21	**10**	27

Java

```
public static int minCostDP(int[][] costs) {
    int[][] dp = new int[costs.length + 1][3];
    int n = costs.length;
    if (costs.length == 0) {
        return 0;
    }
    for (int i = 1; i <= n; i++) {
        dp[i][RED] = costs[i - 1][RED] + Math.min(dp[i - 1][BLUE], dp[i - 1][GREEN]);
        dp[i][BLUE] = costs[i - 1][BLUE] + Math.min(dp[i - 1][RED], dp[i - 1][GREEN]);
        dp[i][GREEN] = costs[i - 1][GREEN] + Math.min(dp[i - 1][RED], dp[i - 1][BLUE]);
    }
    return Math.min(dp[n][RED], Math.min(dp[n][BLUE], dp[n][GREEN]));
}
```

Python

```python
def min_cost_dp(costs):
    n = len(costs)
    dp = [[0 for _ in range(0, 3)] for _ in range(0, n + 1)]
    for i in range(1, n + 1):
        dp[i][RED] = costs[i - 1][RED] + min(dp[i - 1][BLUE],
dp[i - 1][GREEN])
        dp[i][BLUE] = costs[i - 1][BLUE] + min(dp[i - 1][RED],
dp[i - 1][GREEN])
        dp[i][GREEN] = costs[i - 1][GREEN] + min(dp[i - 1]
[RED], dp[i - 1][BLUE])
    return min(dp[n][RED], min(dp[n][BLUE], dp[n][GREEN]))
```

Analyzing time and space complexity

Time and space complexity of recursive solution

Recursive solution.

The recursive function, calls itself twice. So the recursion tree looks like a binary tree.

We have 3 such binary trees, because we can paint the first house with any of the three colors.

The height of this binary tree is N, where N is number of houses.

So the number of nodes in the binary tree of height N is 2^N

The worst case time complexity is $O(3*2^N) = O(2^N)$, Exponential time

We are not using any extra space so the space complexity is O(1), constant

Time and space complexity of Dynamic Programming solution

We can analyze the time complexity of Dynamic Programming solution by counting how many subproblems are we solving.

We take a look at bottom up approach

If we look at the for loop, there is only one for loop which goes from 0 to N.

So the time complexity is O(N), linear time.

We use a chache of size 3 X N, so space complexity is O(3N) , 3 is constant so the space complexity is O(N), linear

Reconstructing solution

We need to record the decision for every subproblem.

Then we can use that to reconstruct the solution.

Painting houses

17,2,17

16,16,5

14,3,9

RED	BLUE	GREEN
0	0	0
17	2	17
18	33	7
21	**10**	27

0	0	0
BLUE	RED	RED
BLUE	RED	BLUE
BLUE	**GREEN**	RED

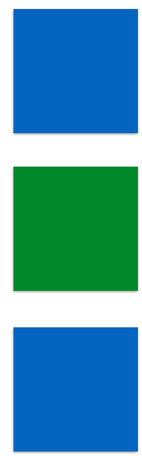
Painting houses

RED	BLUE	GREEN
0	0	0
17	2	17
18	33	7
21	**10**	27

0	0	0
BLUE	RED	RED
BLUE	RED	**BLUE**
BLUE	GREEN	RED

Painting houses

RED	BLUE	GREEN
0	0	0
17	2	17
18	33	7
21	**10**	27



0	0	0
BLUE	RED	RED
BLUE	RED	BLUE
BLUE	GREEN	RED

Java

```java
public static int minCostDPReconstruct(int[][] costs) {
    int[][] dp = new int[costs.length + 1][3];
    int[][] decision = new int[costs.length + 1][3];
    int n = costs.length;
    if (costs.length == 0) {
        return 0;
    }
    for (int i = 1; i <= n; i++) {
        // RED
        if (dp[i - 1][BLUE] < dp[i - 1][GREEN]) {
            decision[i][RED] = BLUE;
            dp[i][RED] = costs[i - 1][RED] + dp[i - 1][BLUE];
        } else {
            decision[i][RED] = GREEN;
            dp[i][RED] = costs[i - 1][RED] + dp[i - 1][GREEN];
        }

        // BLUE
        if (dp[i - 1][RED] < dp[i - 1][GREEN]) {
            decision[i][BLUE] = RED;
            dp[i][BLUE] = costs[i - 1][BLUE] + dp[i - 1][RED];
        } else {
            decision[i][BLUE] = GREEN;
            dp[i][BLUE] = costs[i - 1][BLUE] + dp[i - 1][GREEN];
        }
```

Java

```
// GREEN
        if (dp[i - 1][RED] < dp[i - 1][BLUE]) {
            decision[i][GREEN] = RED;
            dp[i][GREEN] = costs[i - 1][GREEN] + dp[i - 1][RED];
        } else {
            decision[i][GREEN] = BLUE;
            dp[i][GREEN] = costs[i - 1][GREEN] + dp[i - 1][BLUE];
        }
    }
    int ret = Math.min(dp[n][RED], Math.min(dp[n][BLUE], dp[n][GREEN]));

    // Check which color for the last house resulted in minimal cost
    int color = 0;
    if(ret == dp[n][RED]){
        color = RED;
    }else if(ret == dp[n][BLUE]){
        color = BLUE;
    }else{
        color = GREEN;
    }
    int i=n;
    do{
        System.out.println("House "+(i-1)+" , Paint "+paintColor(decision[i][color])+", Cost "+costs[i-1][color]);
        color = decision[i][color];
        i--;
    }while(i>0);
    return ret;
}
```

Python

```python
def min_cost_dp_reconstruct(costs):
    n = len(costs)
    dp = [[0 for _ in range(0, 3)] for _ in range(0, n + 1)]
    decision = [[0 for _ in range(0, 3)] for _ in range(0, n + 1)]
    for i in range(1, n + 1):
        if dp[i - 1][BLUE] < dp[i - 1][GREEN]:
            decision[i][RED] = BLUE
            dp[i][RED] = costs[i - 1][RED] + dp[i - 1][BLUE]
        else:
            decision[i][RED] = GREEN
            dp[i][RED] = costs[i - 1][RED] + dp[i - 1][GREEN]

        if dp[i - 1][RED] < dp[i - 1][GREEN]:
            decision[i][BLUE] = RED
            dp[i][BLUE] = costs[i - 1][BLUE] + dp[i - 1][RED]
        else:
            decision[i][BLUE] = GREEN
            dp[i][BLUE] = costs[i - 1][BLUE] + dp[i - 1][GREEN]

        if dp[i - 1][RED] < dp[i - 1][BLUE]:
            decision[i][GREEN] = RED
            dp[i][GREEN] = costs[i - 1][GREEN] + dp[i - 1][RED]
        else:
            decision[i][GREEN] = BLUE
            dp[i][GREEN] = costs[i - 1][GREEN] + dp[i - 1][BLUE]
```

Python

```python
    result = min(dp[n][RED], min(dp[n][BLUE], dp[n][GREEN]))
    i = n
    if result == dp[n][RED]:
        c = RED
    elif result == dp[n][BLUE]:
        c = BLUE
    else:
        c = GREEN

    while i > 0:
        print("House {} is painted with color {} ".format(i, color(c)))
        c = decision[i][c]
        i -= 1
    return result

def color(c):
    if c == RED :
        return "RED"
    elif c == BLUE:
        return "BLUE"
    else:
        return "GREEN"
```