(a) $DP(n, W) = \begin{cases} DP(n-1, W) & \text{if } w_n > W; \\ \max(v_n + DP(n-1, W - w_n), DP(n-1, W)) & \end{cases}$

(b)

Value	Size	Item	Maximum Available Space					
			0	1	2	3	4	5
0	0	0	0	0	0	0	0	0
12	4	1	0	0	0	0	12	12
8	2	2	0	0	8	8	12	12
9	3	3	0	0	8	9	12	17
7	2	4	0	0	8	9	15	17
5	1	5	0	5	8	13	15	20

Take items 2,4,5

(c) $DP(n, W, k) = \max(v_n + DP(n-1, W - w_n, k),\ v_n + DP(n-1, W - w_n/2, k-1),\ DP(n-1, W, k))$

Explanation:
In Dynamic programming, we can consider a 3D DP table where the state DP[i][j][k] will denote the maximum value we can obtain if we are considering values from 1 to i-th, weight of the knapsack is j and we can half the weight of at most k values. Basically, we are adding one extra state, the number of weights that can be halved in a traditional 2-D 01 knapsack DP matrix.

Now, three possibilities can take place:

- Include item with full weight if the item's weight does not exceed the remaining weight
- Include the item with half weight if the item's half weight does not exceed the remaining weight
- Skip the item

Now we have to take the maximum of these three possibilities.

- If we do not take the i-th weight then dp[i][j][k] would remain equal to dp[i – 1][j][k], just like traditional knapsack.
- If we include item with half weight then dp[i][j][k] would be equal to dp[i – 1][j – wt[i] / 2][k – 1] + val[i] as after including i-th value our remaining knapsack capacity would be j – wt[i] / 2 and our number of half operations (k) would increase by 1.
- Similarly, if we include item with full weight then dp[i][j][k] would be equal to dp[i – 1][j – wt[i]][k] + val[i] as knapsack capacity in this case would reduce to j – wt[i].

We simply take the maximum of all three choices.

2.

1. **B**
2. **B,C**
3. **C**