SCSA SC2001 Lab Example Class Project 3 Team 7

PU FANYI (U2220175K)

PUSHPARAJAN ROSHINI (U2222546A)

QIAN JIANHENG OSCAR (U2220109K)

RHEA SUSAN GEORGE (U2220116B)

(1) 2 Recursive Definitions

- Naive Approach
- Optimised Approach
  - #1 Improved optimised approach - 2 column
  - #2 Further improved optimised approach - 1 column

Recursive Definition: Naive Approach

$$P(C,j) = \max_{k=0}^{\left\lfloor \frac{C}{W_{j-1}} \right\rfloor} \Big\{ \underbrace{P\big(\overbrace{C - kw_{j-1}}^{\text{Remaining capacity for sub problem}}, j-1\big)}_{\text{Sub-problem: If don't (j-1)-th item}} + kp_{j-1} \Big\}$$

Capacity

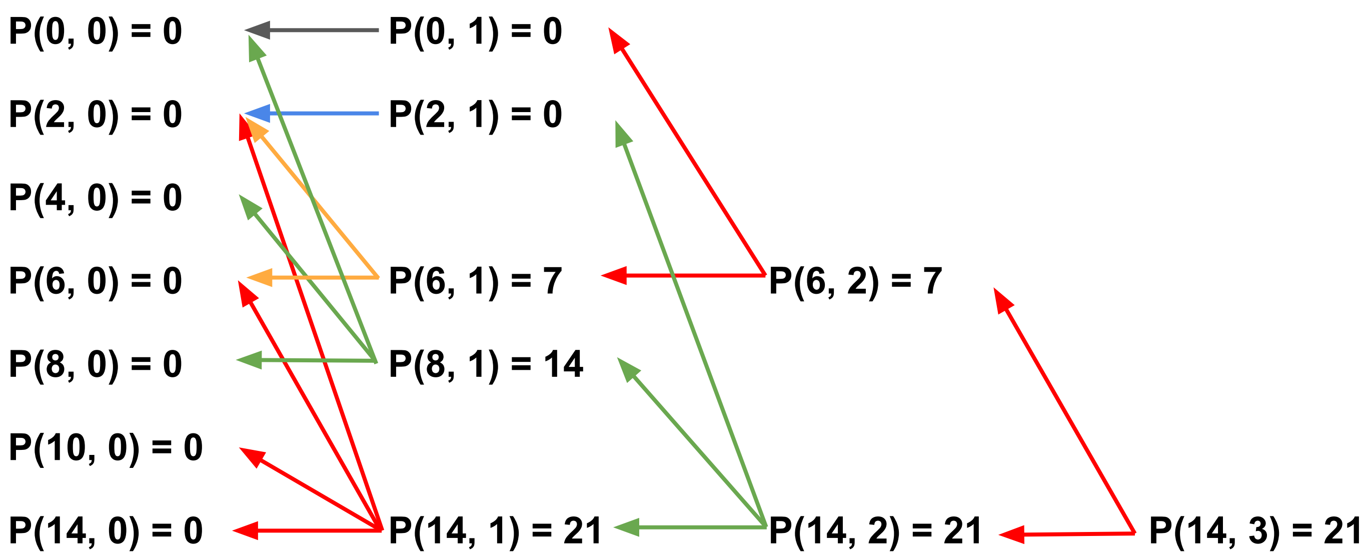
Types of items

range of k

Number of (j-1)-th item we take

Base Case: P(C,0) = 0

Subproblem Graph: Naive Approach



Example

	0	1	2
w_i	4	6	8
p_i	7	6	9

	0	1	2	3 j
0	0	0		
1	0	0		
2	0	0		
3	0	0		
4	0	7		
5	0			
6	0			
7	0			
8	0			
9	0			
10	0			
11	0			
12	0			
13	0			
14	0			
Capacity				

$$P(C,j) = \max_{k=0}^{\left\lfloor \frac{C}{W_{j-1}} \right\rfloor} \{P(C - kw_{j-1}, j-1) + kp_{j-1}\}$$

C = 4
W_{j-1} = 4
k = 4/4 = 1 → k stops at 1

k=0,
P(4, 0) = 0

k=1,
P(0,0) + 1*7 = 7

Example

	0	1	2
w_i	4	6	8
p_i	7	6	9

	0	1	2	3 j
0	0	0		
1	0	0		
2	0	0		
3	0	0		
4	0	7		
5	0	7		
6	0			
7	0			
8	0			
9	0			
10	0			
11	0			
12	0			
13	0			
14	0			
Capacity				

$$P(C,j) = \max_{k=0}^{\left\lfloor \frac{C}{W_{j-1}} \right\rfloor} \{P(C - kw_{j-1}, j-1) + kp_{j-1}\}$$

C = 5
W_{j-1} = 4
k = 5/4 = 1 → k stops at 1

k=0,
P(5, 0) = 0

k=1,
P(1,0) + 1*7 = 7

Example

	0	1	2
w_i	4	6	8
p_i	7	6	9

	0	1	2	3 j
0	0	0	0	
1	0	0	0	
2	0	0	0	
3	0	0	0	
4	0	7	7	
5	0	7	7	
6	0	7	7	
7	0	7	7	
8	0	14	14	
9	0	14	14	
10	0	14	14	
11	0	14	14	
12	0	21	21	
13	0	21	21	
14	0	21	21	

21

$$P(C,j) = \max_{k=0}^{\left\lfloor \frac{C}{W_{j-1}} \right\rfloor} \{P(C - kw_{j-1}, j-1) + kp_{j-1}\}$$

C = 14
W_{j-1} = 6
k = floor(14/6) = 2 → k stops at 2

k=0,
P(14, 1) = 21

k=1,
P(8, 1) + 1*6 = 19

k=2,
P(2, 1) + 2*6 = 12

Complexity: Naive Approach

Average Time Complexity:

$$O\left(C^2 \cdot \sum_{j=0}^{n-1} \frac{1}{w_j}\right)$$

Worst time complexity: when $w_j = 1$

$$O(C^2 \cdot n)$$

$$P(C,j) = \max_{k=0}^{\left\lfloor \frac{C}{W_{j-1}} \right\rfloor} \left\{ P\left(C - kw_{j-1}, j-1\right) + kp_{j-1} \right\}$$

```
for c in range(C):                      # O(C)
  for j in range(n):                    # O(n)
    for k in range(C / w[j - 1]): # O(C/W_{j-1})
      # Calculate P(C, j)
```

Code for Naive Approach

```c
int knapsack(const Item *items_begin, const Item *items_end, const int capacity) {
  size_t len = items_end - items_begin;
  int **f = calloc(len + 1, sizeof(int *));
  for (int i = 0; i <= len; ++i) {
    f[i] = calloc(capacity + 1, sizeof(int));
    memset(f[i], 0, (capacity + 1) * sizeof(int));
  }
  for (int i = 1; i <= len; ++i) {
    for (int j = 0; j <= capacity; ++j) {
      for (int k = 0; k * items_begin[i - 1].weight <= j; ++k) {
        f[i][j] = max(f[i][j],
                      f[i - 1][j - k * items_begin[i - 1].weight] +
                          k * items_begin[i - 1].profit);
      }
    }
  }
  int ans = f[len][capacity];
  free(f);
  return ans;
}
```

Recursive Definition: Optimised Approach

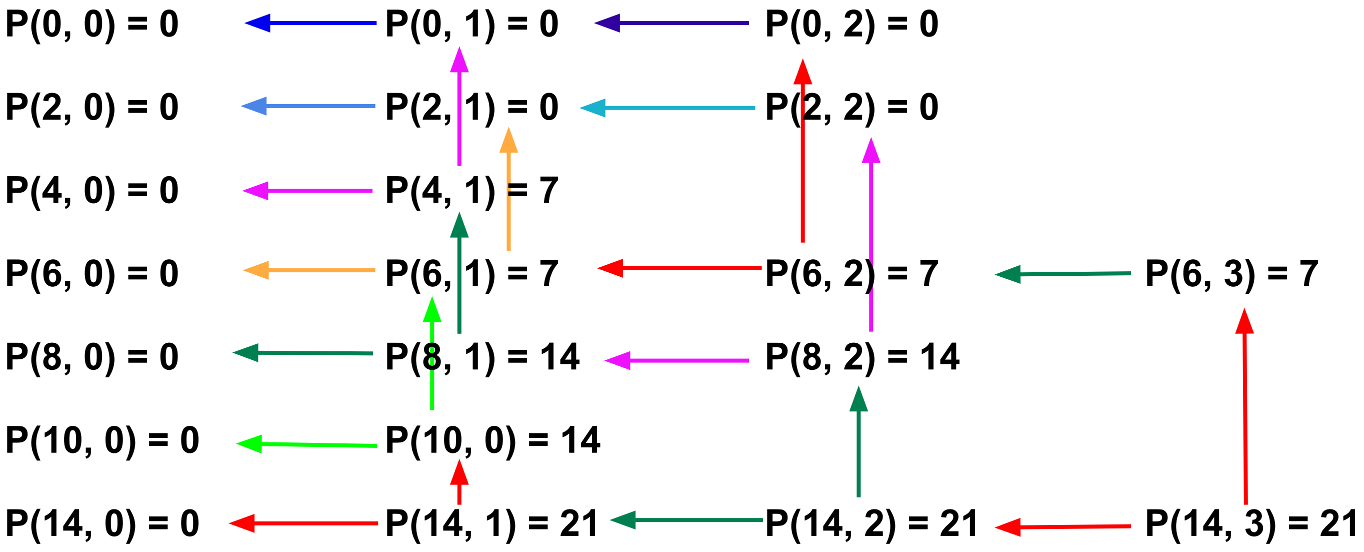
$$P(C,j) = \max\{\overbrace{P(C,j-1)}^{\text{Don't choose jth element}}, \overbrace{P(\underbrace{C - w_{j-1}}_{\text{Reserve } w_j \text{ capacity for the item}}, j) + p_{j-1}}^{\text{Choose jth element}}\}$$

You can still choose item j due to unlimited supplies

Base Case: $P(C,0) = 0$

Subproblem Graph : Optimised Approach

	0	1	2
w_i	4	6	8
p_i	7	6	9



Example

	0	1	2
w_i	4	6	8
p_i	7	6	9

	0	1	2	3	j
0	0	0	0	0	
1	0	0			
2	0	0			
3	0	0			
4	0	7			
5	0				
6	0				
7	0				
8	0				
9	0				
10	0				
11	0				
12	0				
13	0				
14	0				
Capacity					

$$P(C,j) = \max\{P(C,j-1), P(C-w_{j-1},j) + p_{j-1}\}$$

P(4,1) = max((P(4,0), P(4-4,1) + 7)
= max((P(4,0), P(0,1) + 7)
= max(0,7)
= **7**

Example

	0	1	2
w_i	4	6	8
p_i	7	6	9

	0	1	2	3	j
0	0	0	0	0	
1	0	0	0	0	
2	0	0	0	0	
3	0	0	0	0	
4	0	7	7	7	
5	0	7	7	7	
6	0	7		7	
7	0	7		7	
8	0	14			
9	0	14			
10	0	14			
11	0	14			
12	0	21			
13	0	21			
14	0	21			
Capacity					

$$P(C,j) = \max\{P(C,j-1), P\left(C - w_{j-1}, j\right) + p_{j-1}\}$$

P(12,1) = max((P(12,0), P(12-4,1) + 7)
= max((P(12,0), P(8,1) + 7)
= max(0,21)
= **21**

Wj < C

Notice the pattern!

Example

	0	1	2
w_i	4	6	8
p_i	7	6	9

	0	1	2	3	j
0	0	0	0	0	
1	0	0	0	0	
2	0	0	0	0	
3	0	0	0	0	
4	0	7	7	7	
5	0	7	7	7	
6	0	7	7	7	
7	0	7	7	7	
8	0	14	14		
9	0	14	14		
10	0	14	14		
11	0	14	14		
12	0	21	21		
13	0	21	21		
14	0	21	21		
Capacity					

MAX(14,13)

$$P(C,j) = \max\{P(C,j-1), P(C-w_{j-1},j)+p_{j-1}\}$$

P(11,2) = max((P(11,1), P(11-6,2) + 6)
= max((P(11,1), P(5,2) + 6)
= max(14,13)
= **14**

Example

	0	1	2
w_i	4	6	8
p_i	7	6	9

Object 0 costs lesser weight + profit-friendly
Example : 2 times of Object 0 is >> 1 Object 2

	0	1	2	3	j
0	0	0	0	0	
1	0	0	0	0	
2	0	0	0	0	
3	0	0	0	0	
4	0	7	7	7	
5	0	7	7	7	
6	0	7	7	7	
7	0	7	7	7	
8	0	14	14	14	
9	0	14	14	14	
10	0	14	14	14	
11	0	14	14	14	
12	0	21	21	21	
13	0	21	21	21	
14	0	21	21	21	
Capacity					

$$P(C,j) = \max\{P(C,j-1), P(C-w_{j-1},j) + p_{j-1}\}$$

P(8,3) = max((P(8,2), P(8-8,3) + 9)
= max((P(8,2), P(0,3) + 9)
= max(14,9)
= **14**

P(14,3) = max((P(14,2), P(14-8,3) + 9)
= max((P(14,2), P(6,3) + 9)
= max(21,15)
= **21**

Complexity : Optimised Approach

```python
for c in range(C + 1):          # O(C)
    for j in range(n):          # O(n)
        # O(1) calculate P(c, j)
```

$$\Theta(C \cdot n)$$

Running result for

	0	1	2
w_i	5	6	8
p_i	7	6	9

	0	1		
0	0	0		
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	7	7	7
6	0	7	7	7
7	0	7	7	7
8	0	7	7	9
9	0	7	7	9
10	0	14	14	14
11	0	14	14	14
12	0	14	14	14
13	0	14	14	16
14	0	14	14	16
Capacity				

Maximum Profit = 16

Code for Optimised Approach

```
int knapsack(const Item *items_begin, const Item *items_end, const int capacity) {
    int **f = calloc((items_end - items_begin + 1), sizeof(int *));
    for (int i = 0; i <= items_end - items_begin; ++i) {
        f[i] = calloc(capacity + 1, sizeof(int));
    }
    memset(f[0], 0, (capacity + 1) * sizeof(int));
    for (const Item *item = items_begin; item != items_end; ++item) {
        for (int i = 0; i <= capacity; ++i) {
            if (i >= item->weight) {
                f[item - items_begin + 1][i] = max(f[item - items_begin][i],
                    f[item - items_begin + 1][i - item->weight] + item->profit);
            } else {
                f[item - items_begin + 1][i] = f[item - items_begin][i];
            }
        }
    }
    int ans = f[items_end - items_begin][capacity];
    for (int i = 0; i <= items_end - items_begin; ++i) {
        free(f[i]);
    }
    free(f);
    return ans;
}
```

Analysis of both approaches

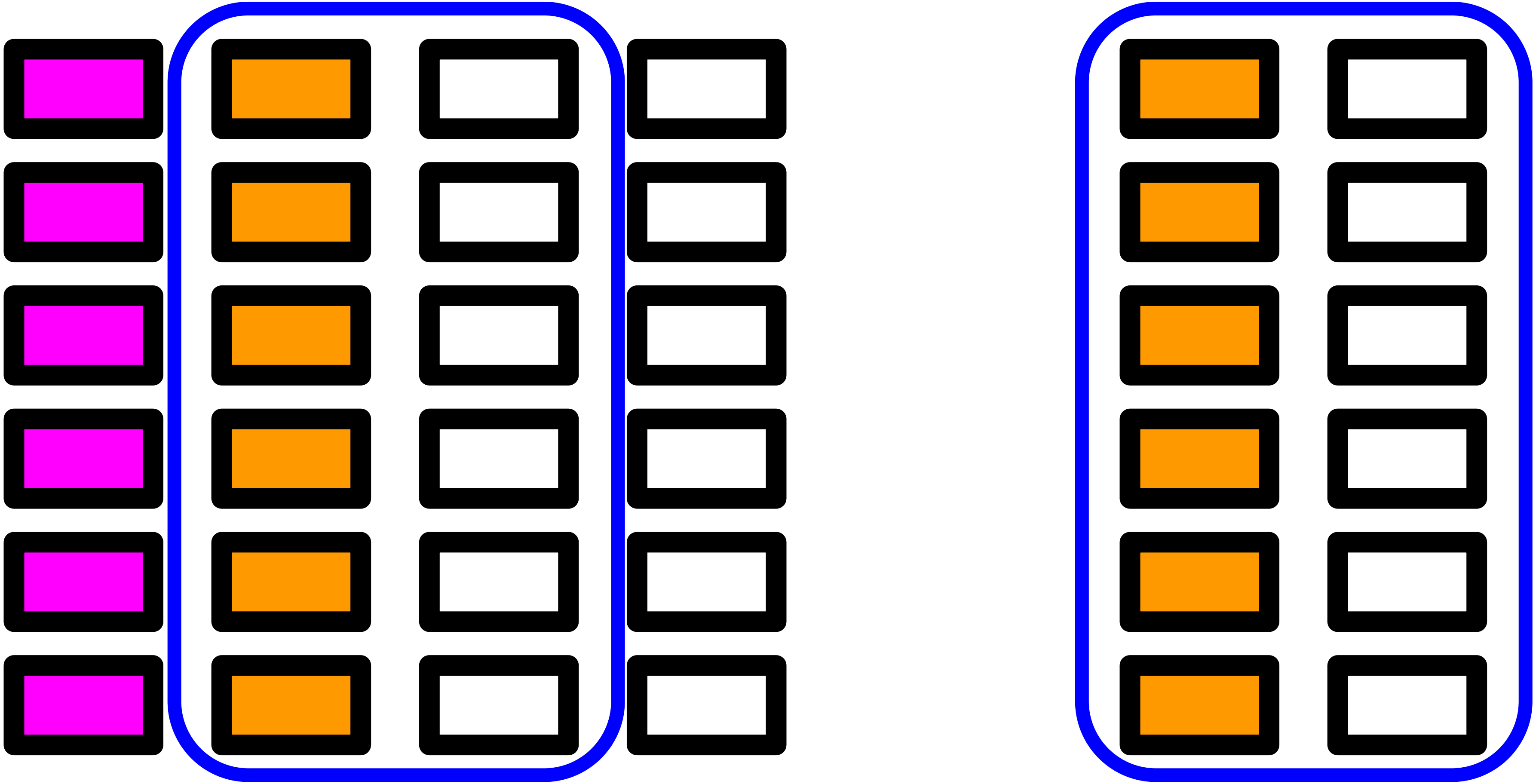
- Time Complexity for Optimised Approach is better than Naive
  - $O(Cn) << O(C^2n)$

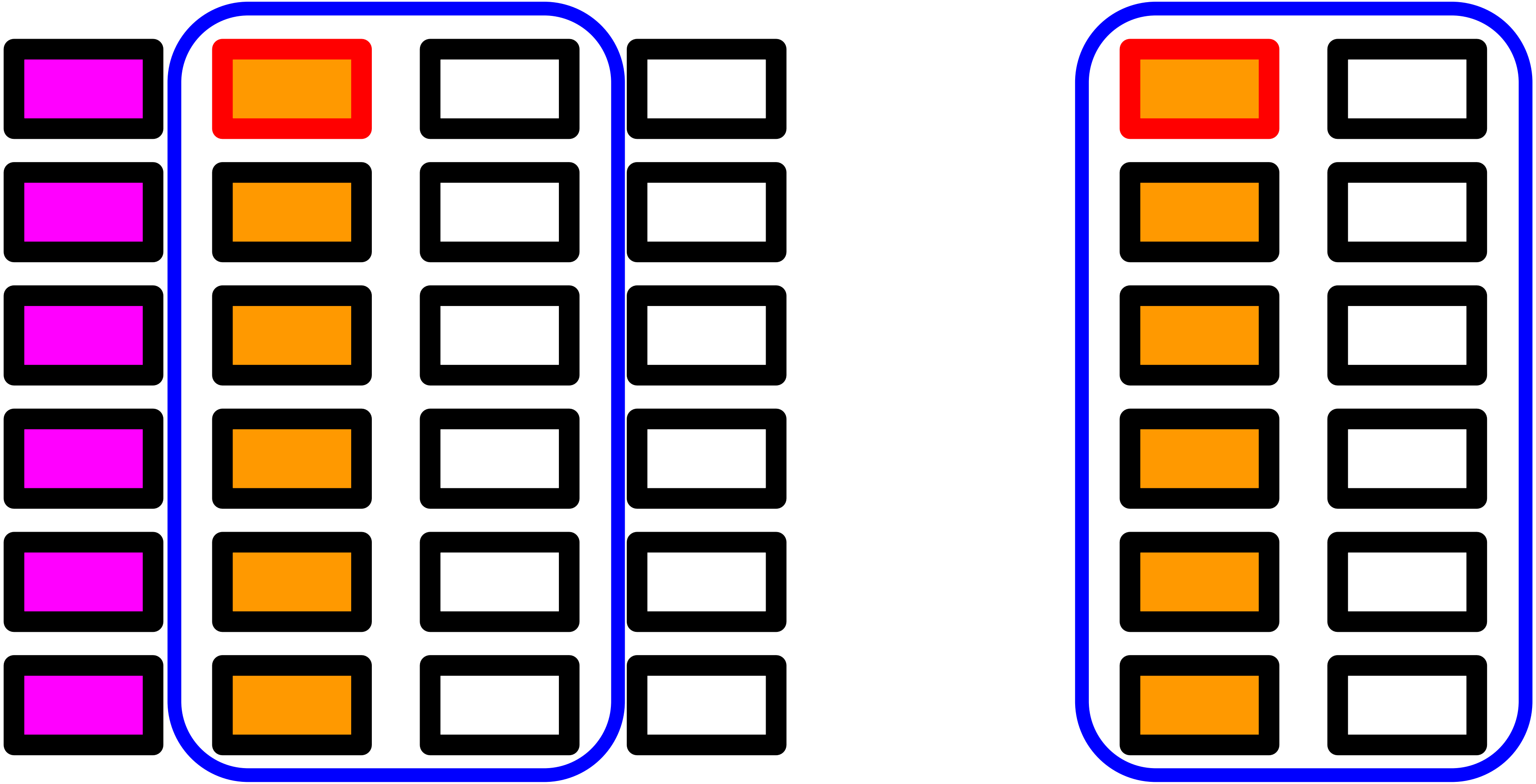
Improved optimised approach : 2 columns + 1 column

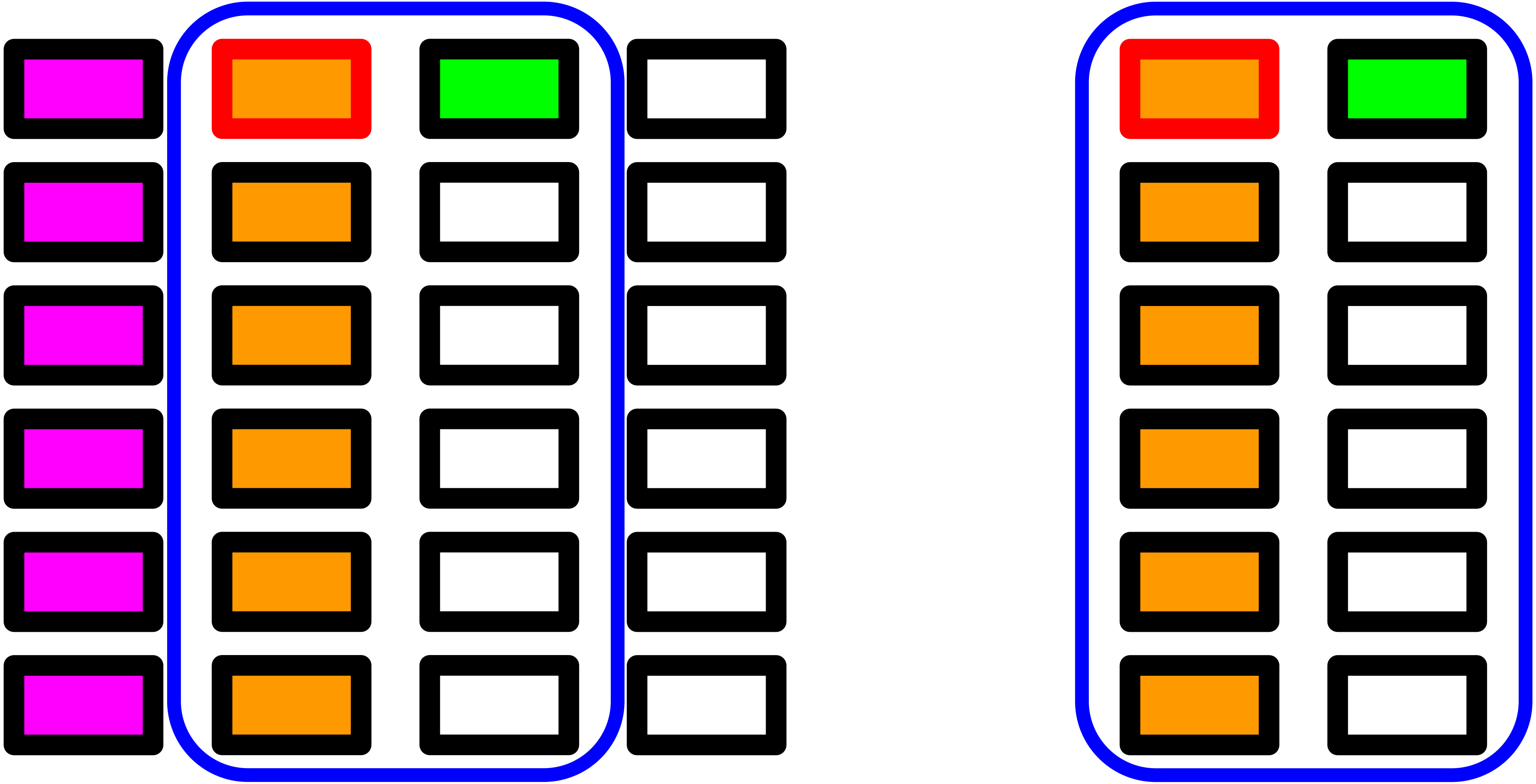
	0	1	2	3	j
0	0	0	0	0	
1	0	0	0	0	
2	0	0	0	0	
3	0	0	0	0	
4	0	7	7	7	
5	0	7	7	7	
6	0	7	7	7	
7	0	7	7	7	
8	0	14	14	14	
9	0	14	14	14	
10	0	14	14	14	
11	0	14	14	14	
12	0	21	21	21	
13	0	21	21	21	
14	0	21	21	21	
Capacity					

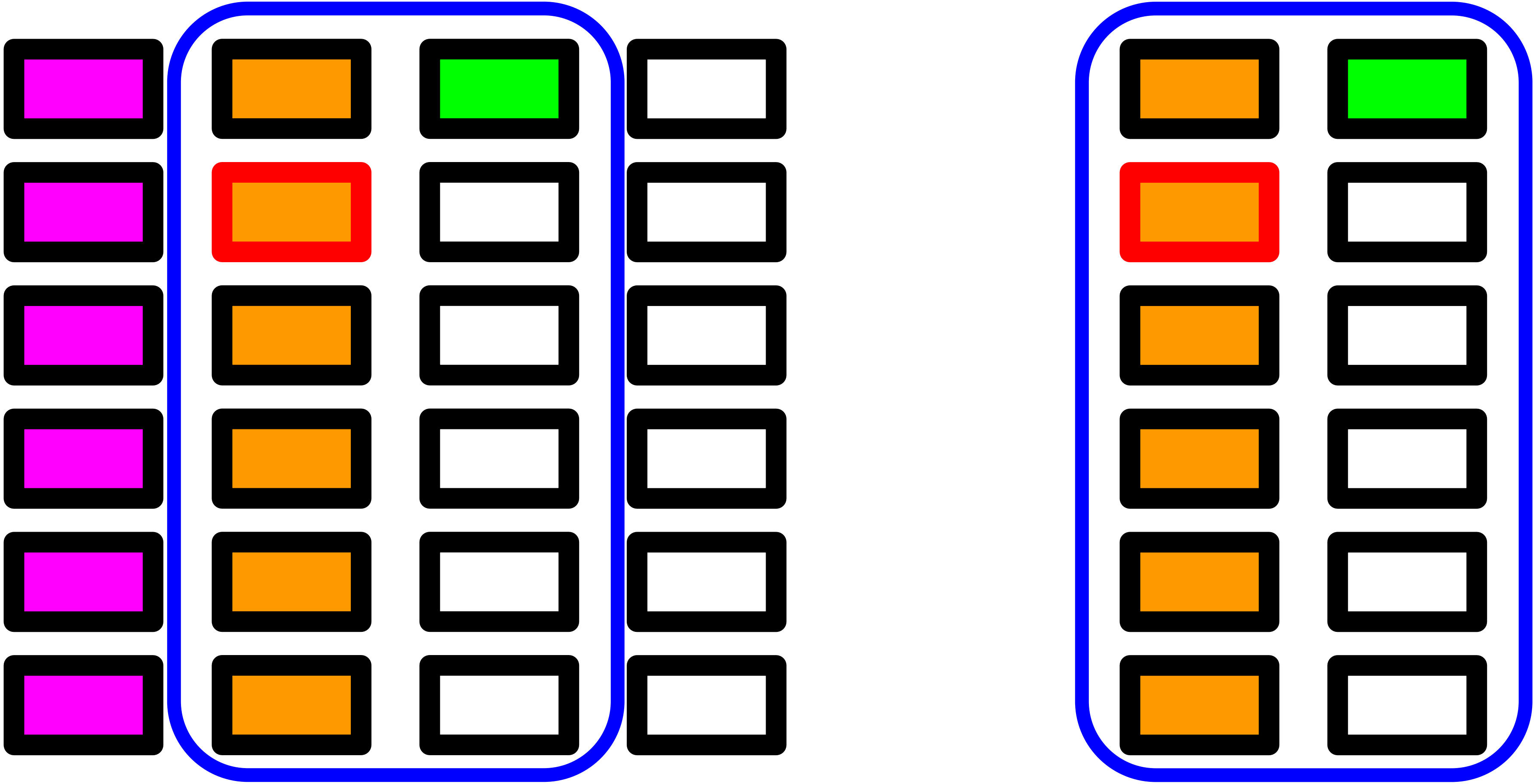
→

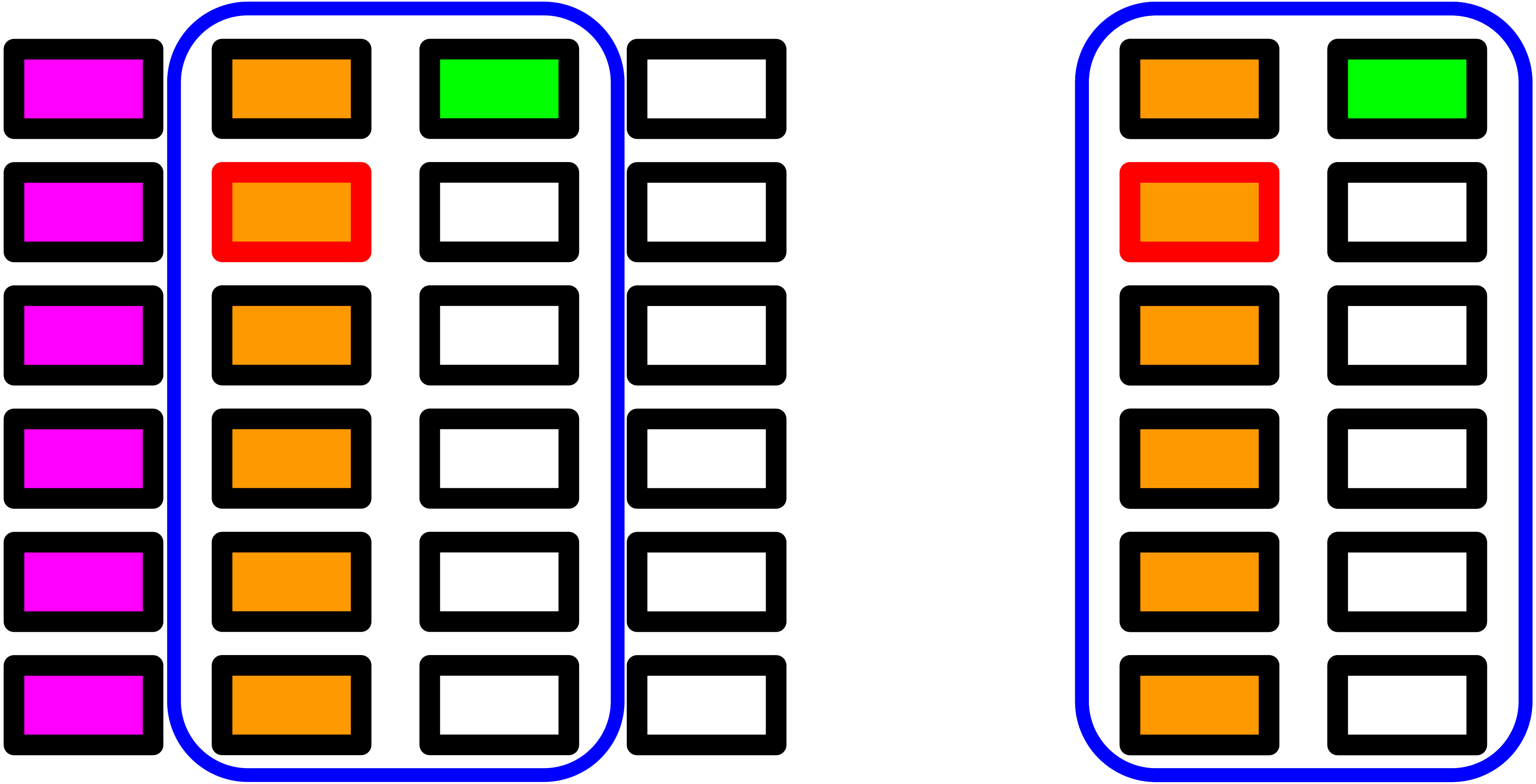
	2	3	j
0	0	0	
1	0	0	
2	0	0	
3	0	0	
4	7	7	
5	7	7	
6	7	7	
7	7	7	
8	14	14	
9	14	14	
10	14	14	
11	14	14	
12	21	21	
13	21	21	
14	21	21	
Capacity			

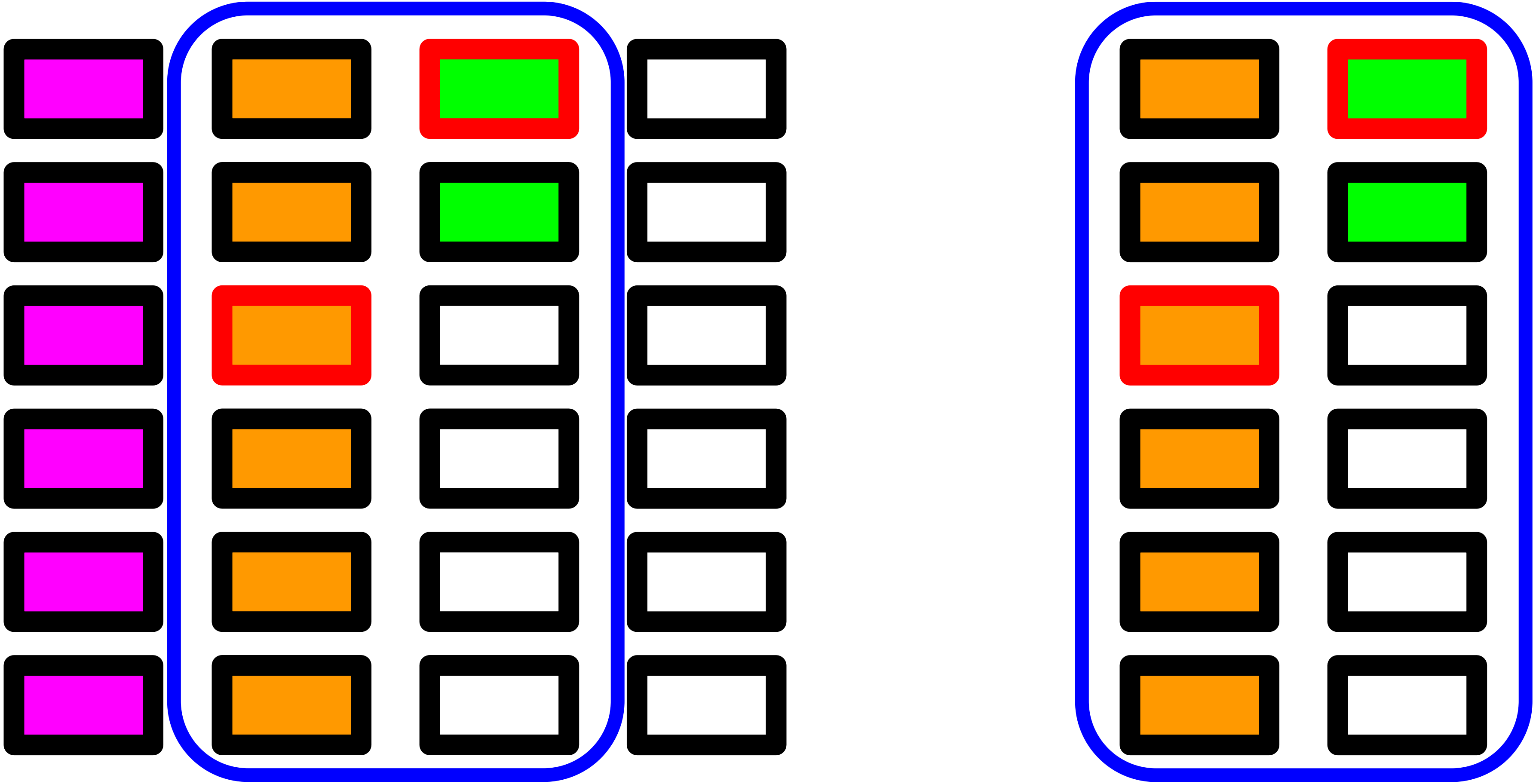


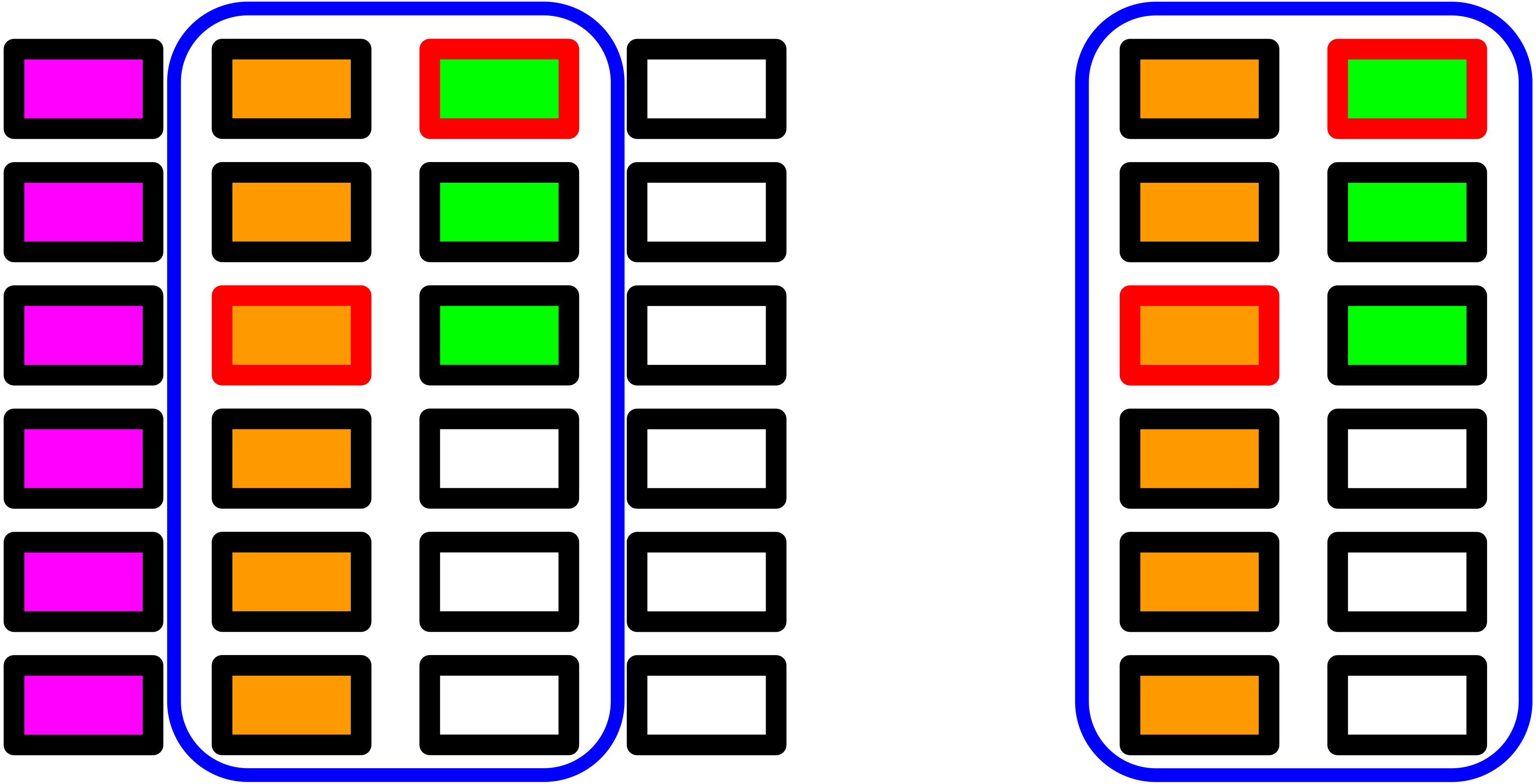


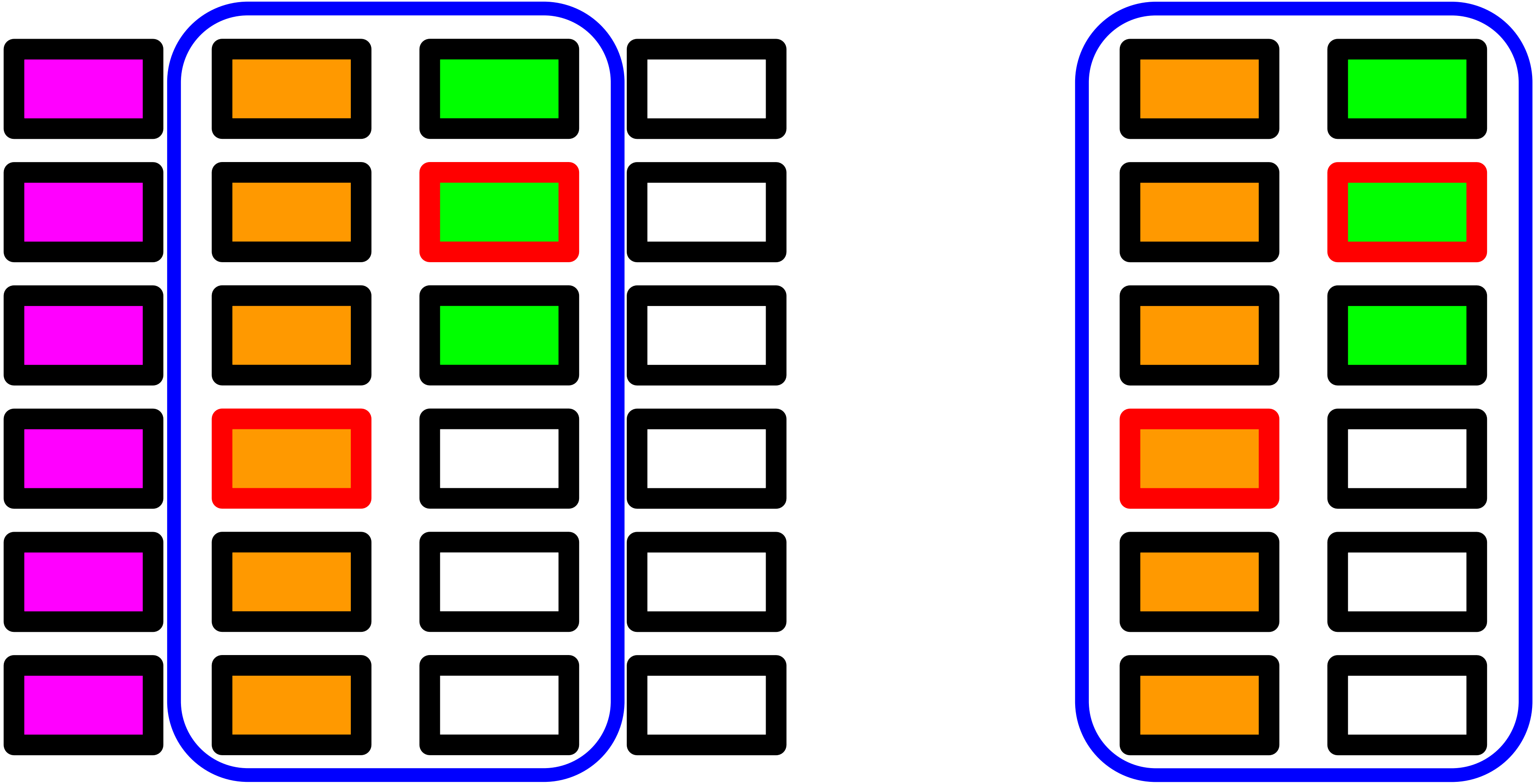


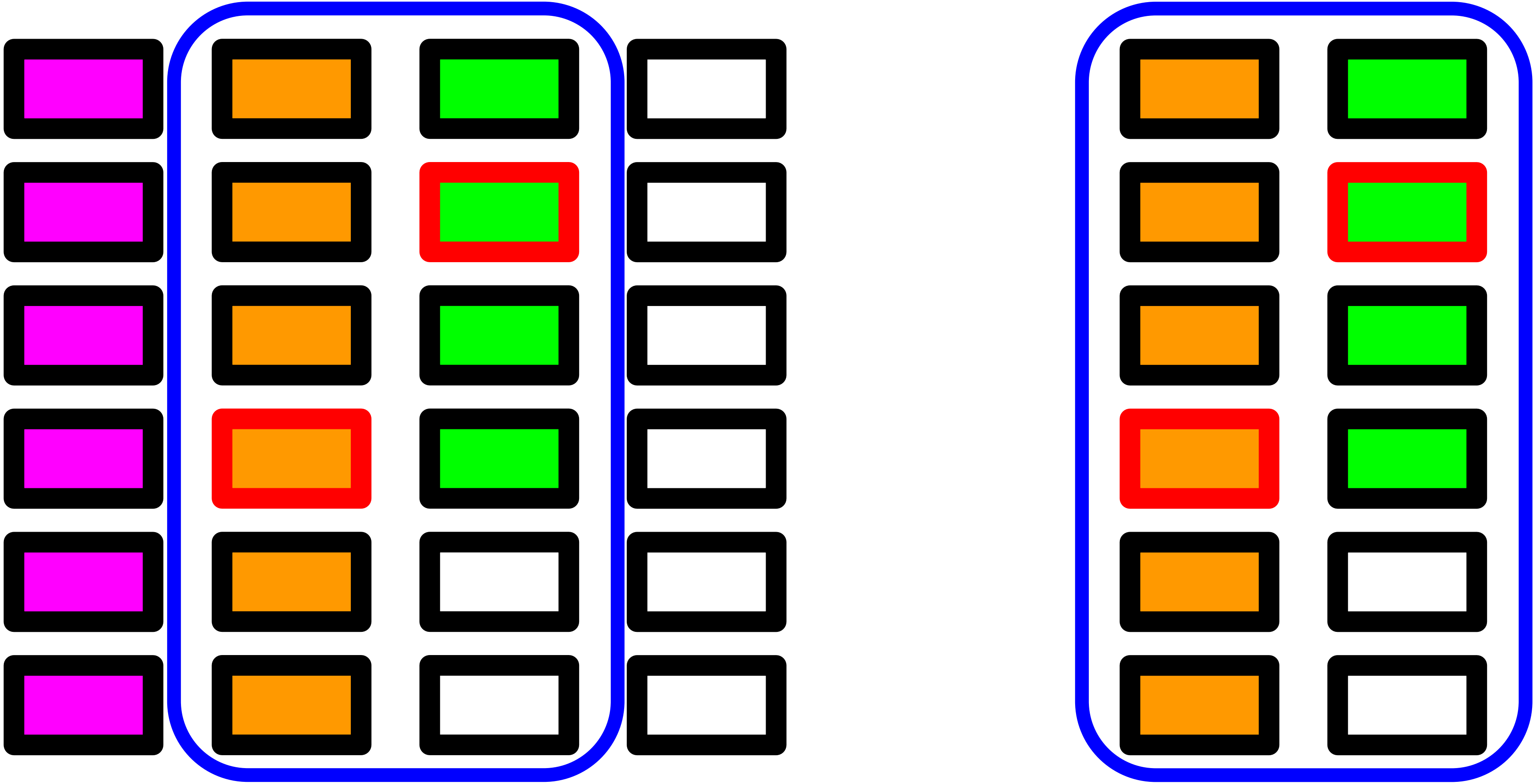


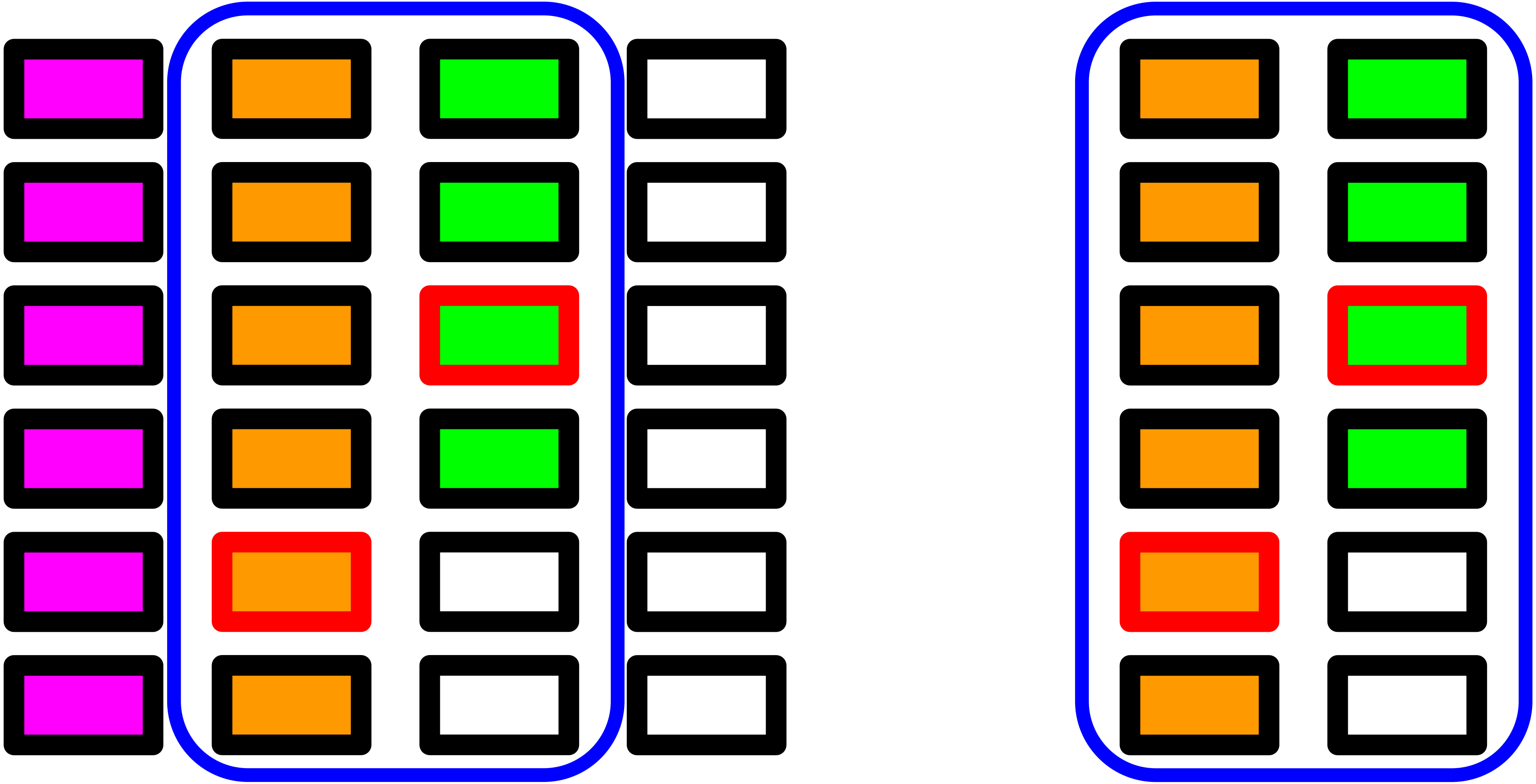


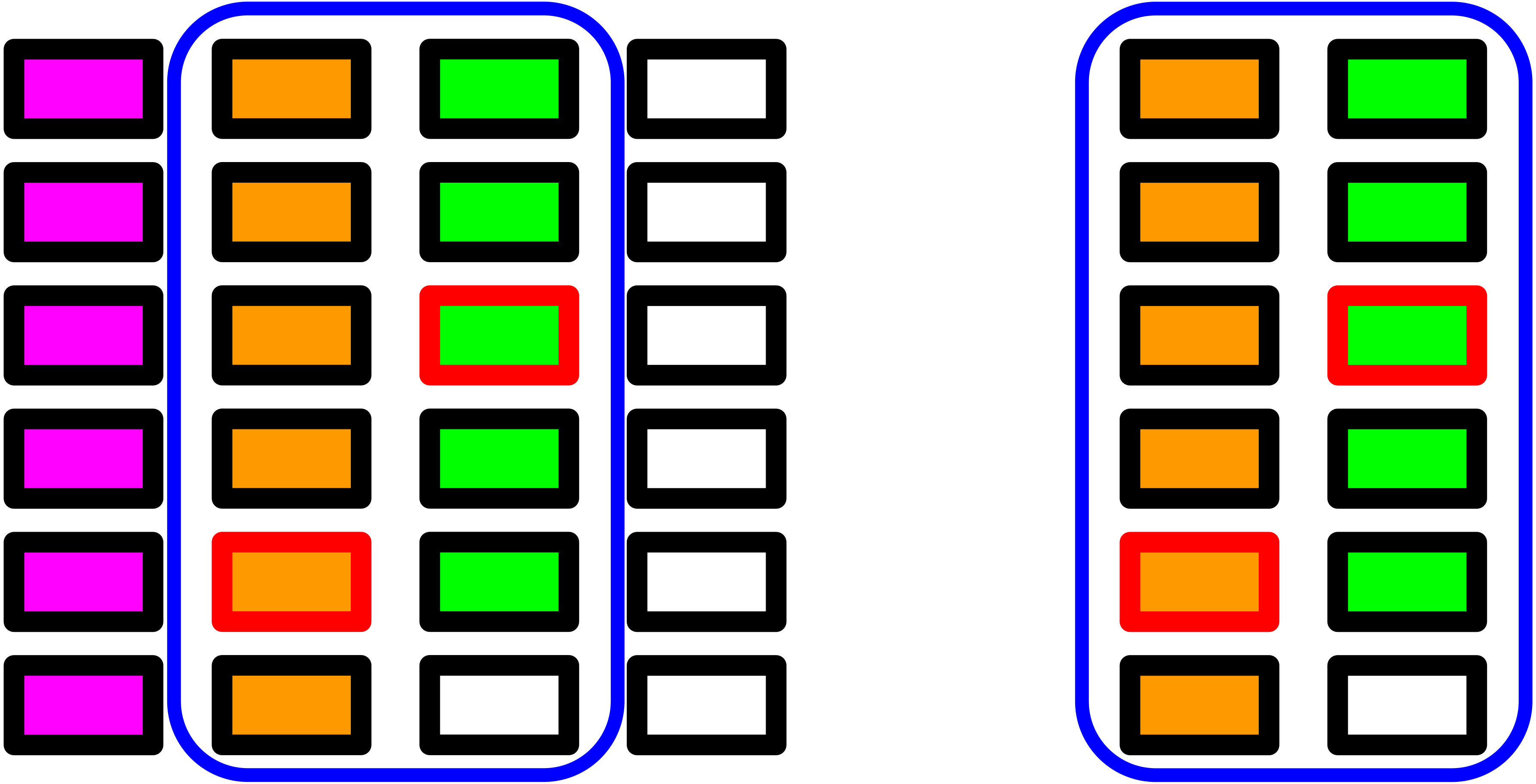


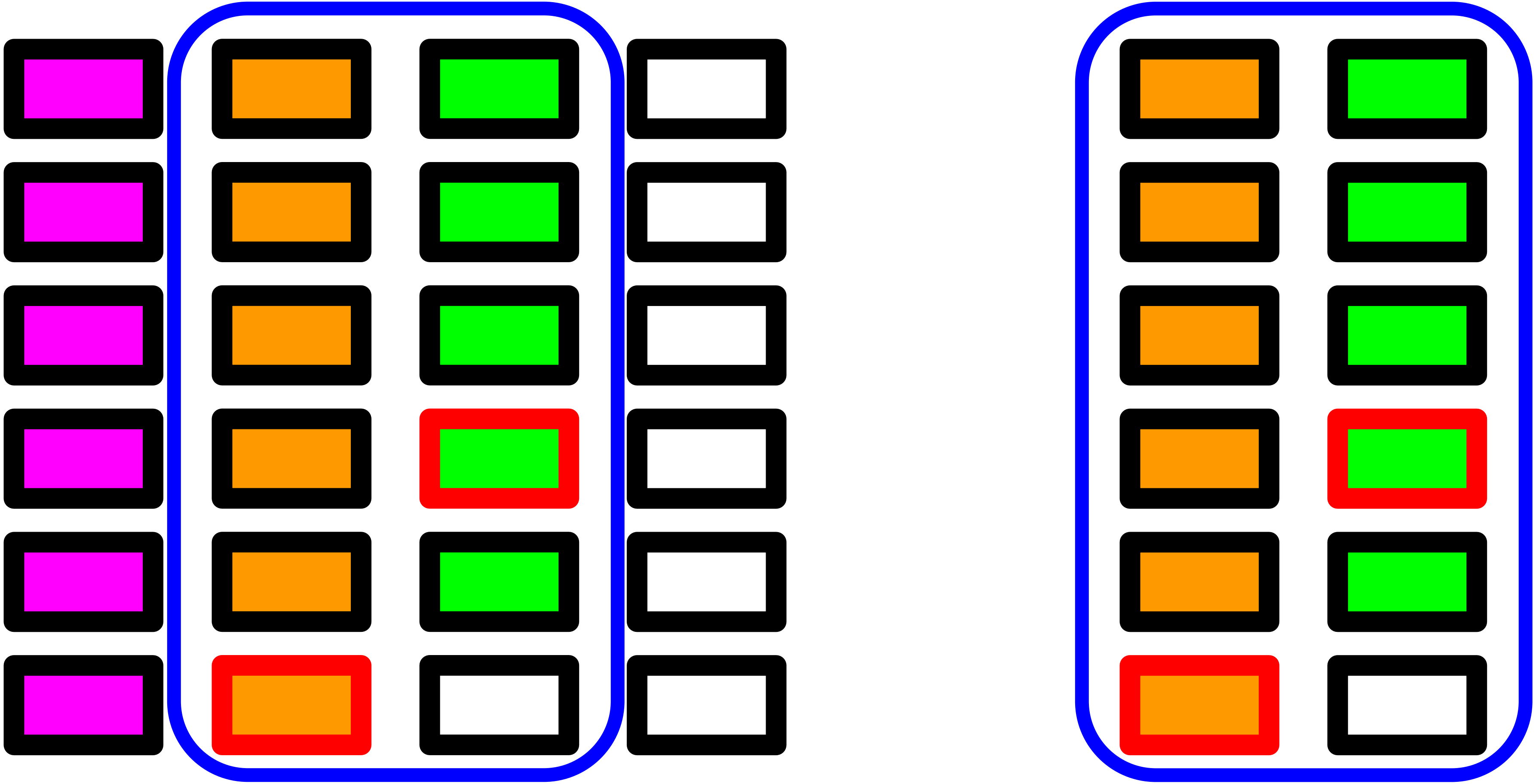


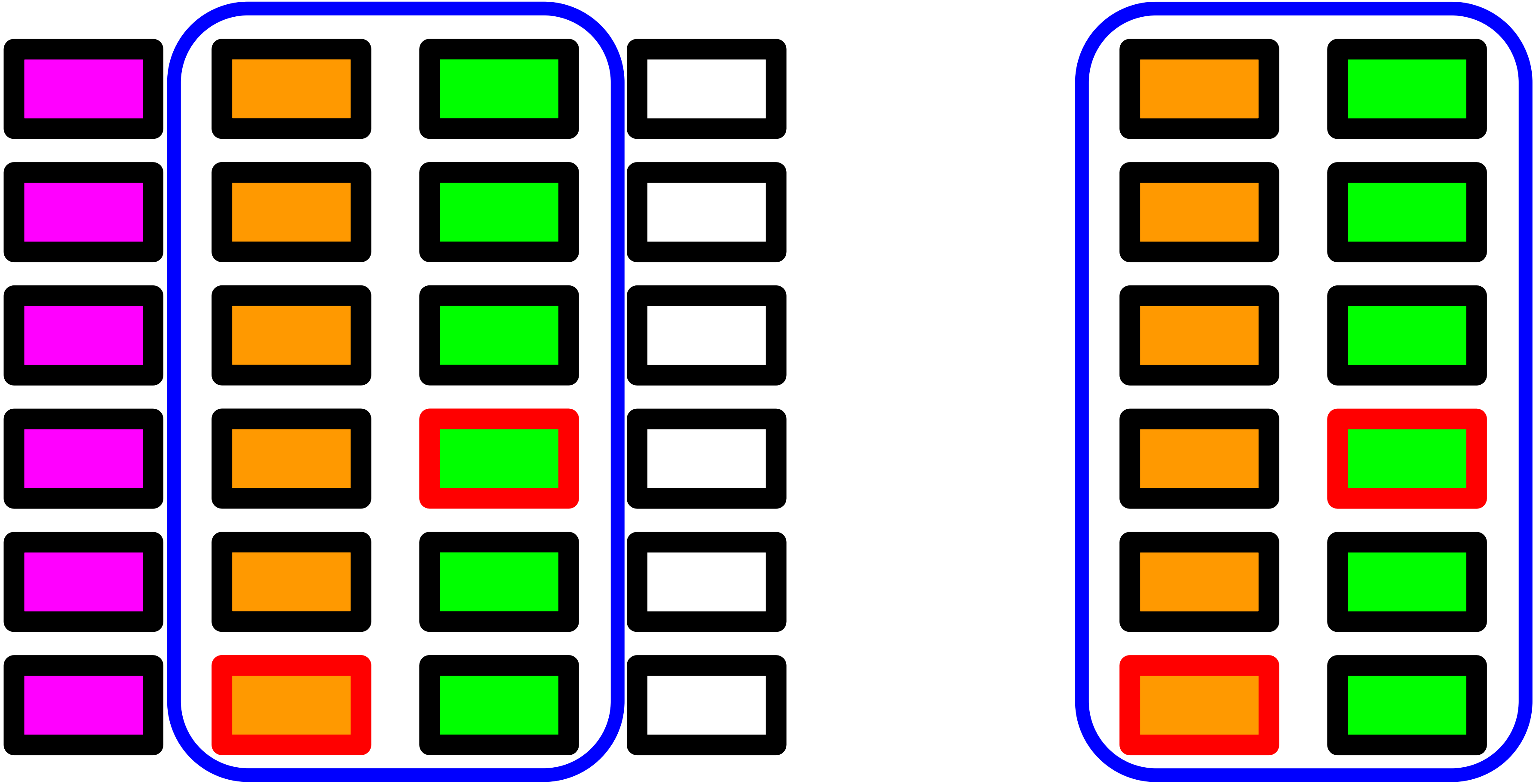


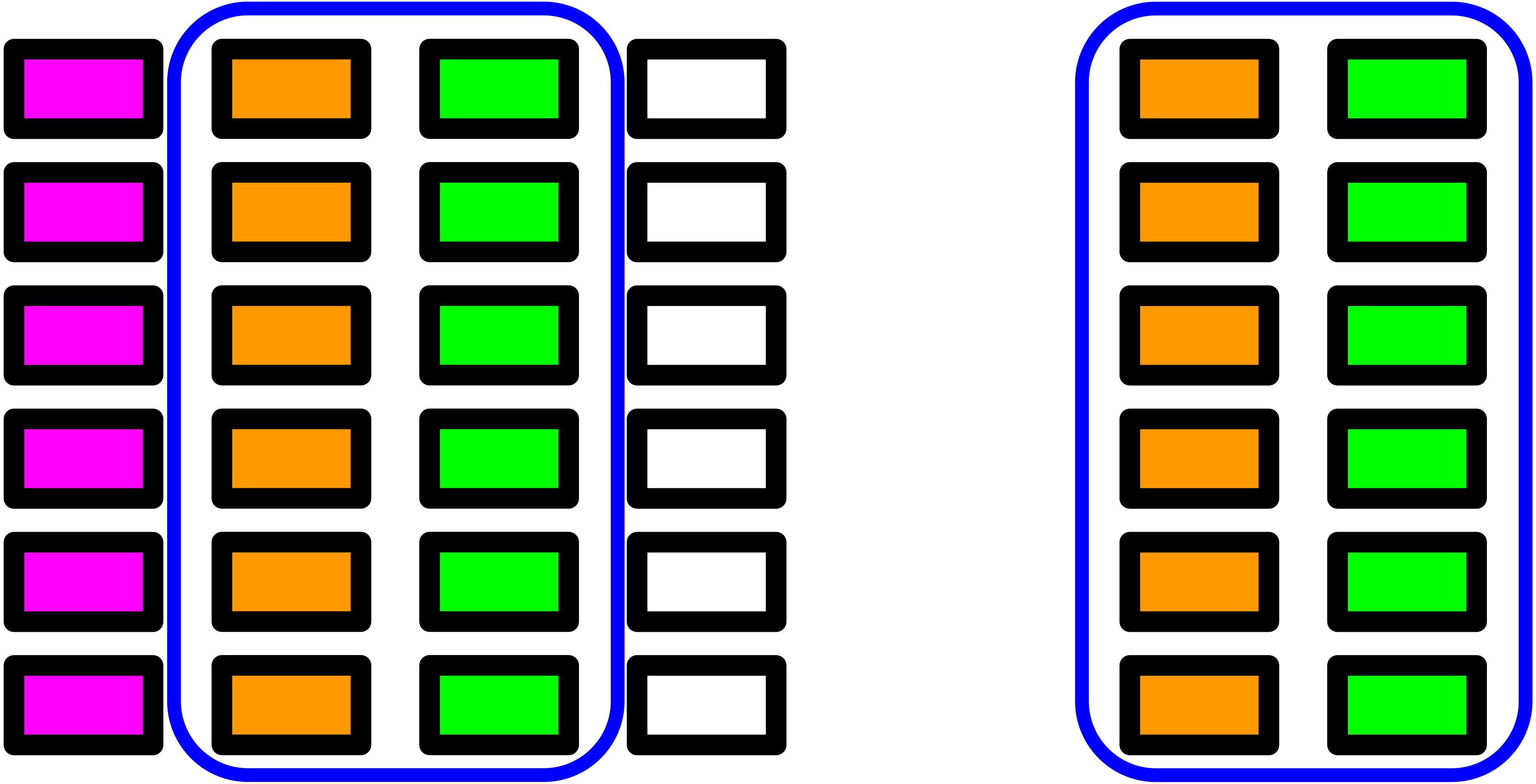


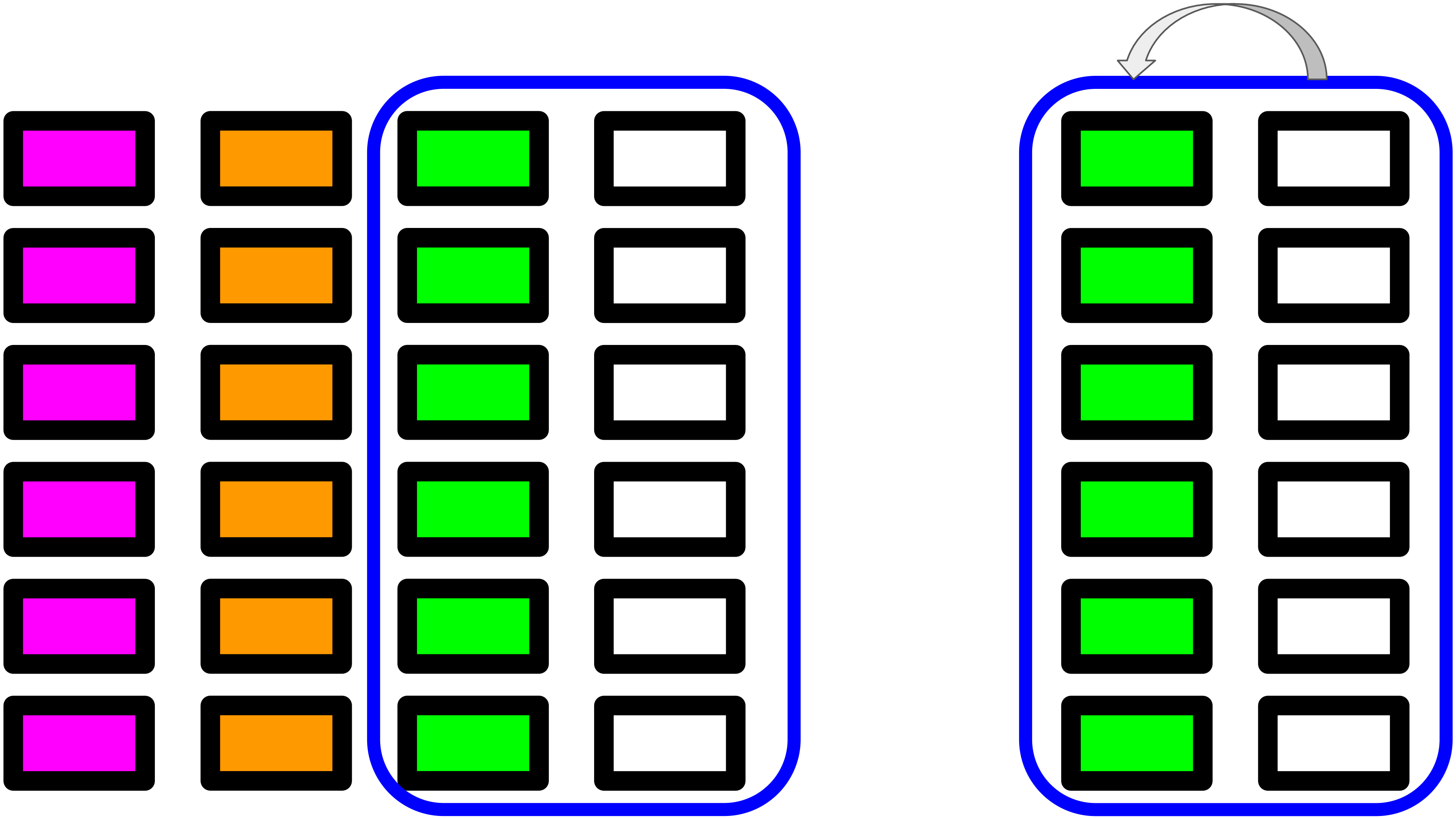




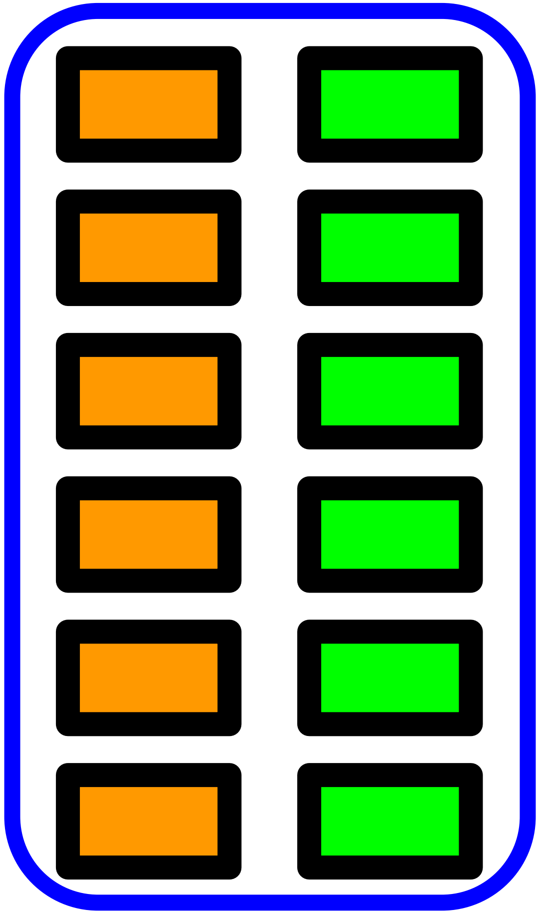


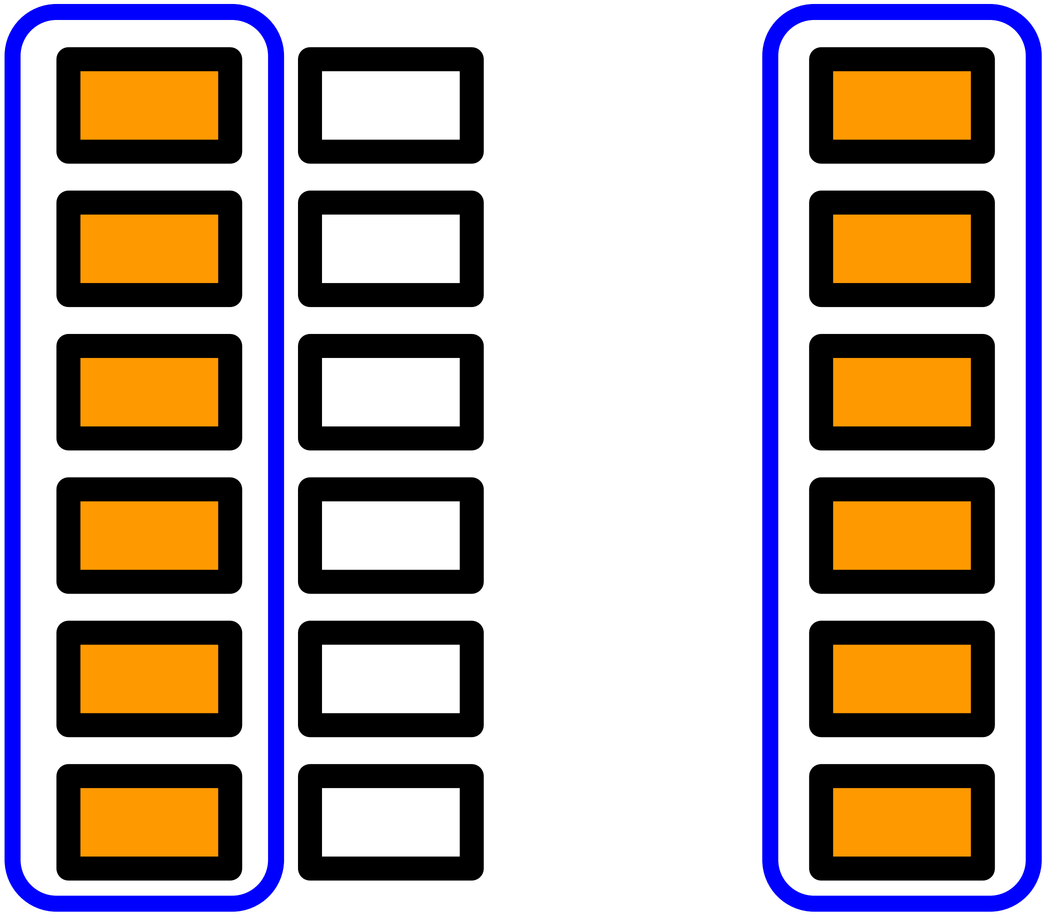


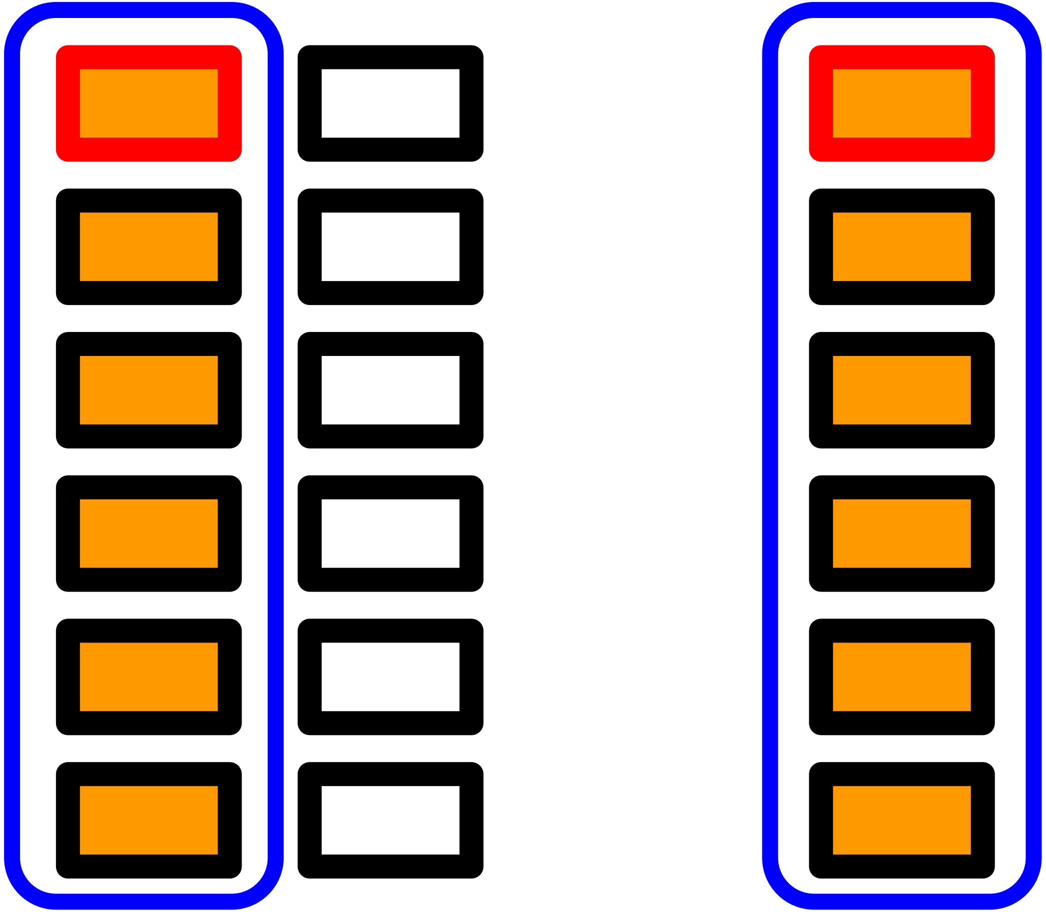


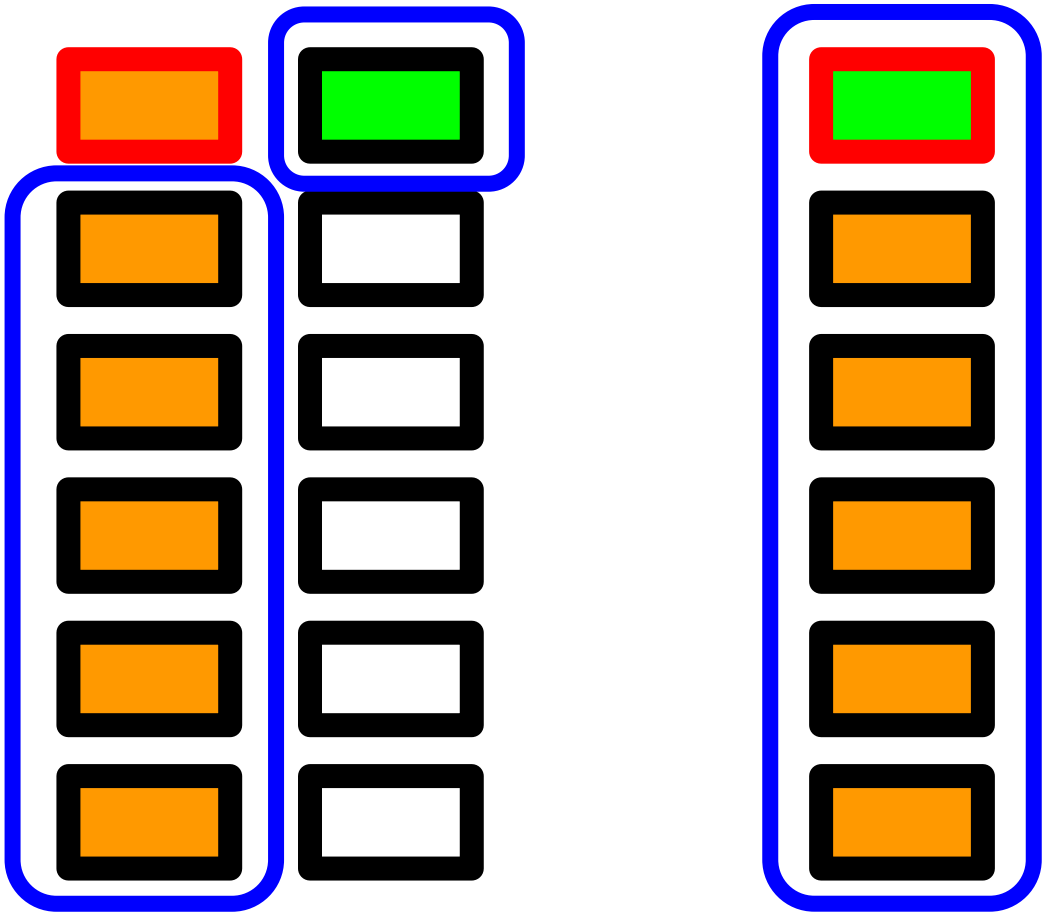


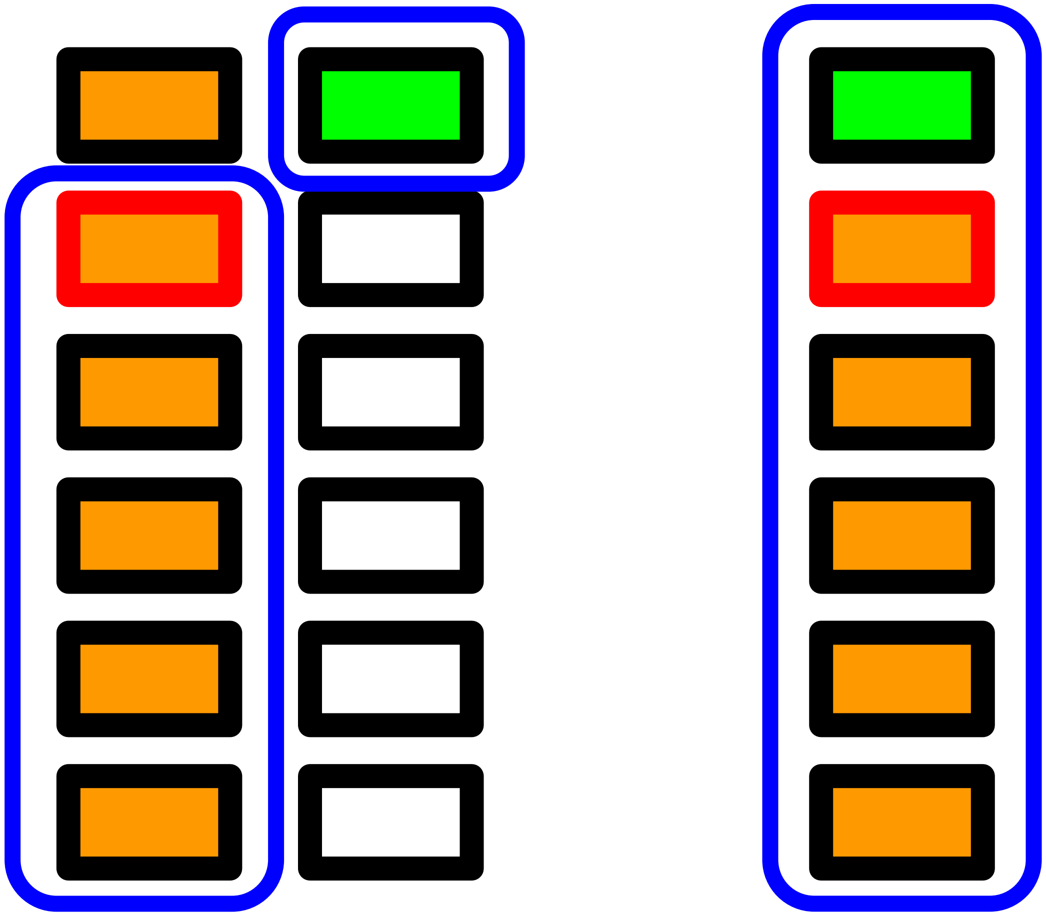
Saves space!
Improves space complexity

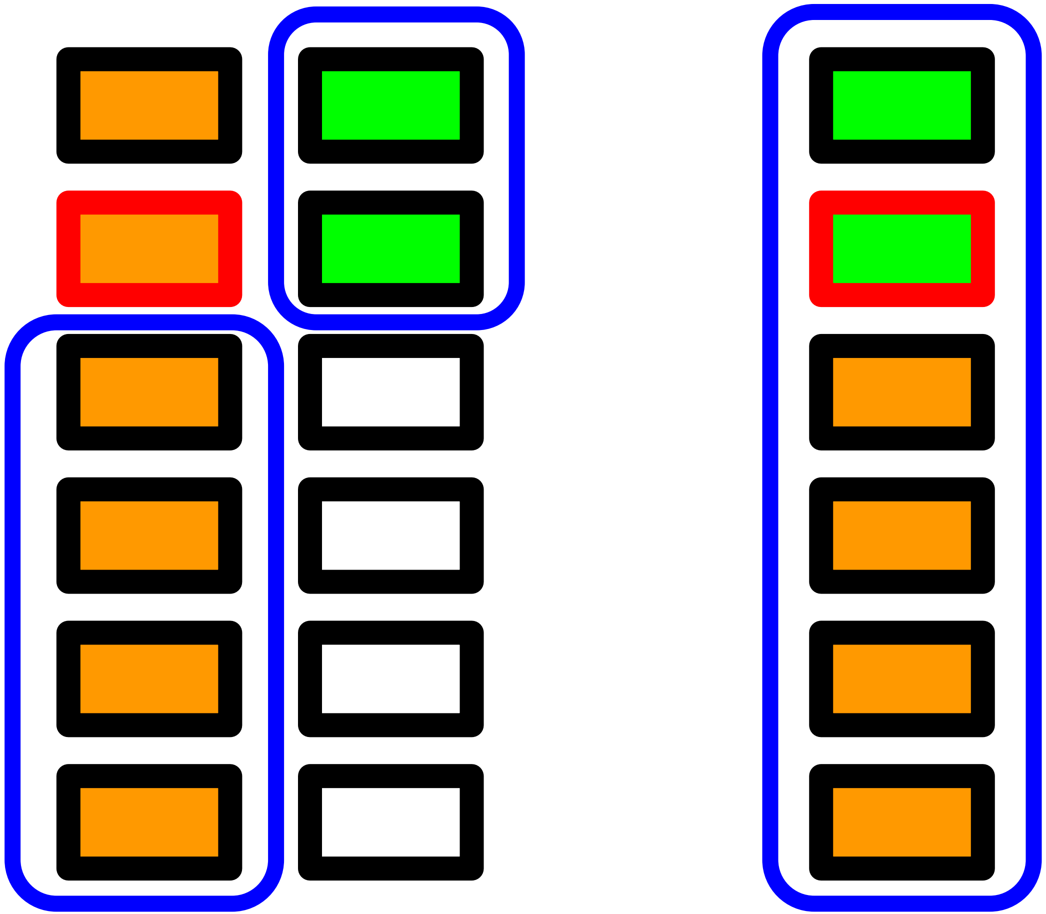


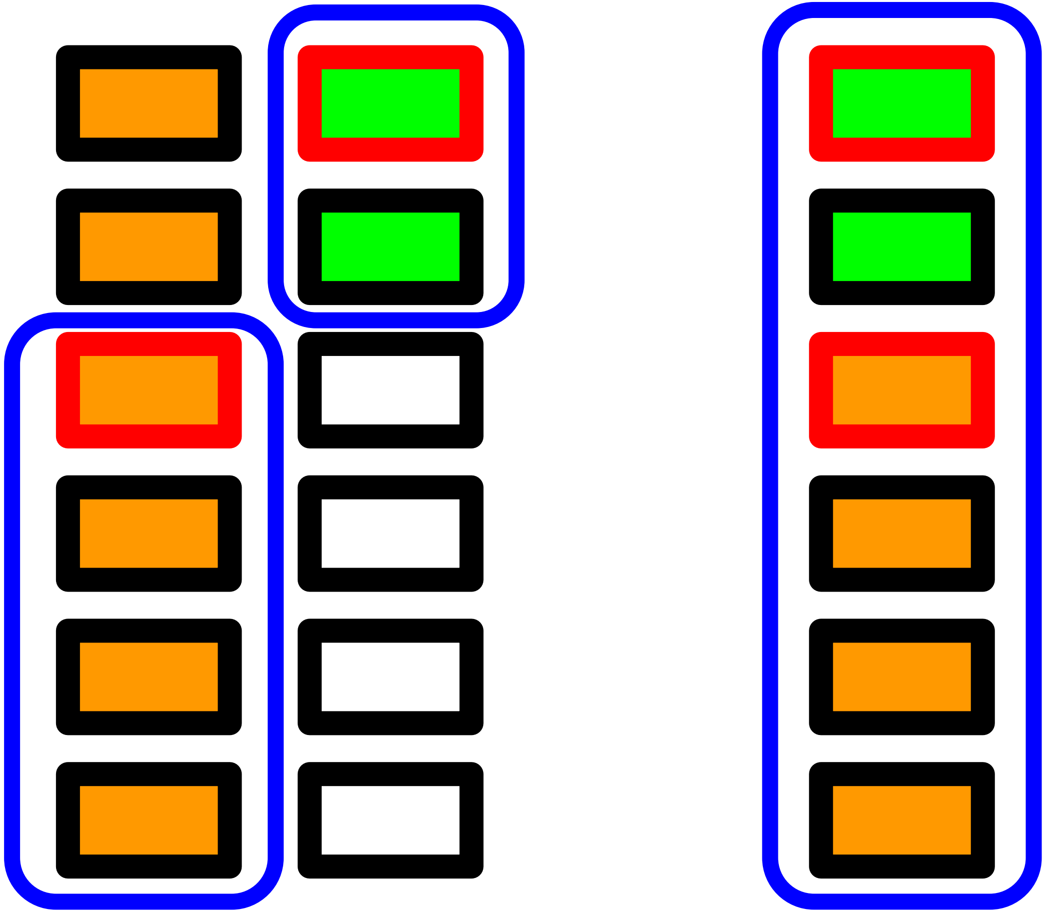


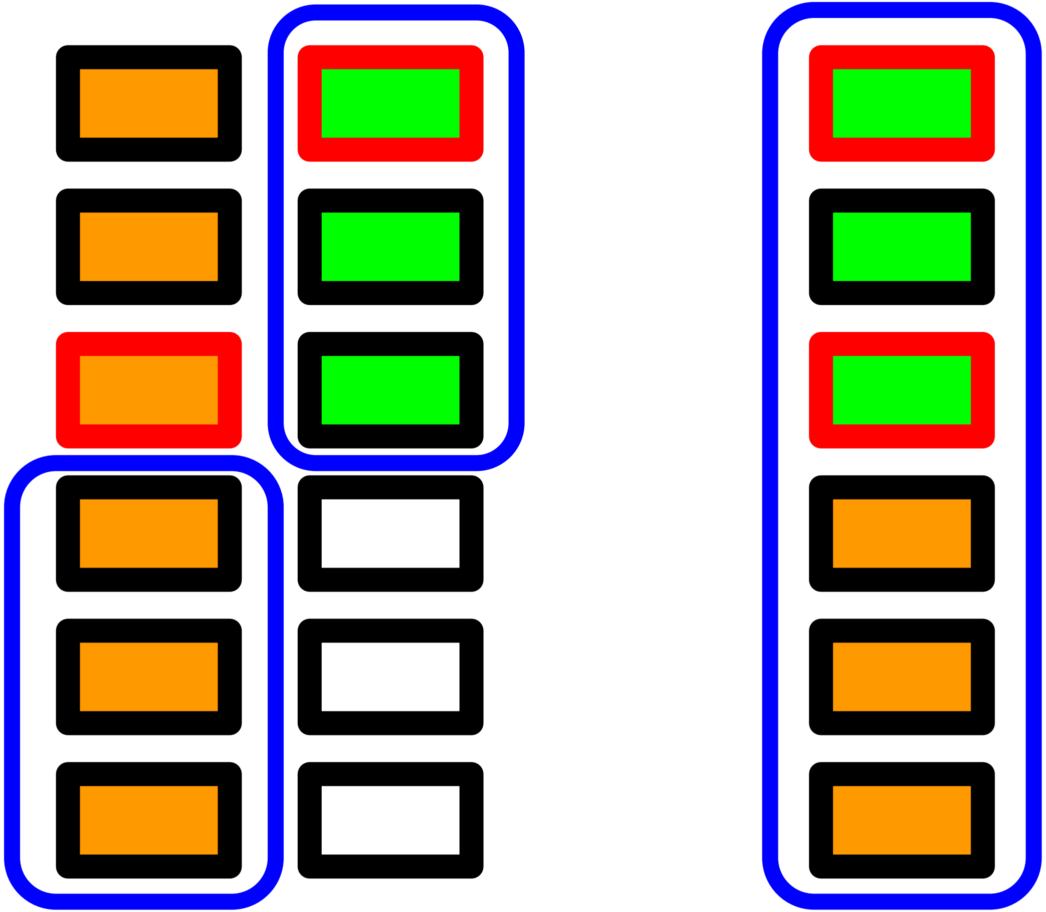


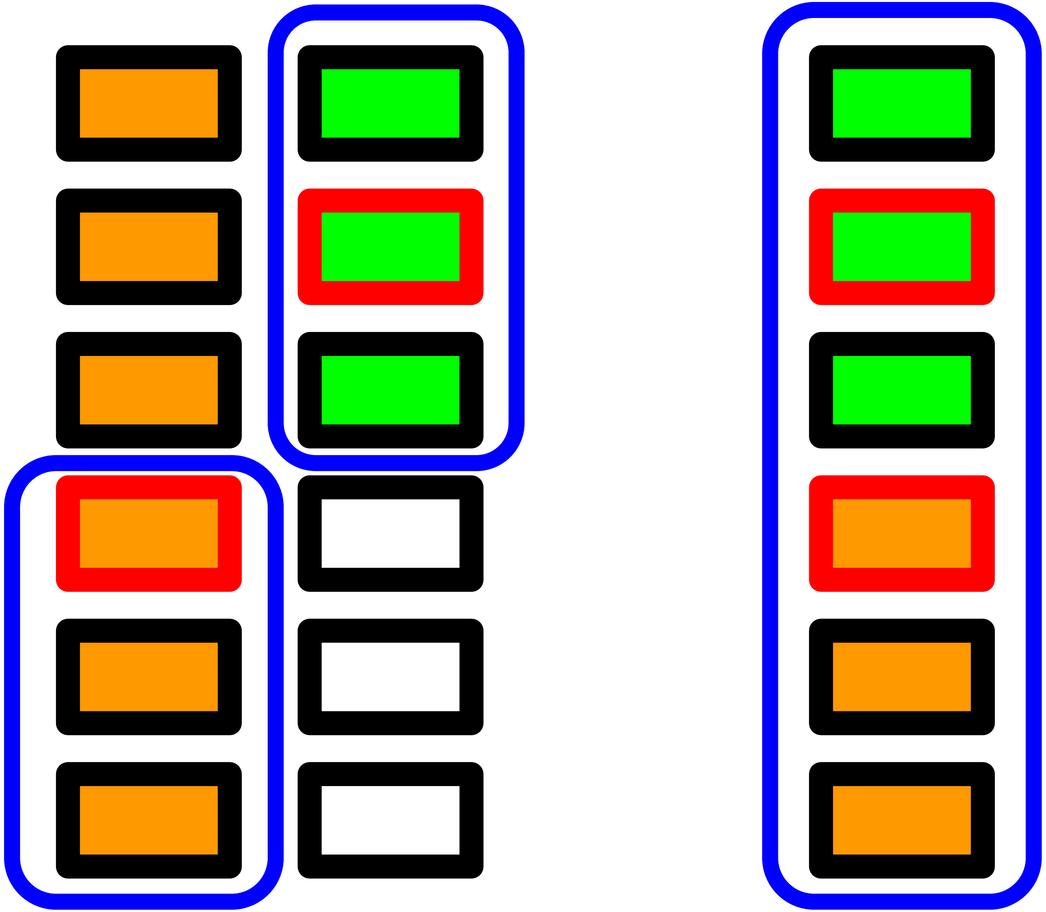


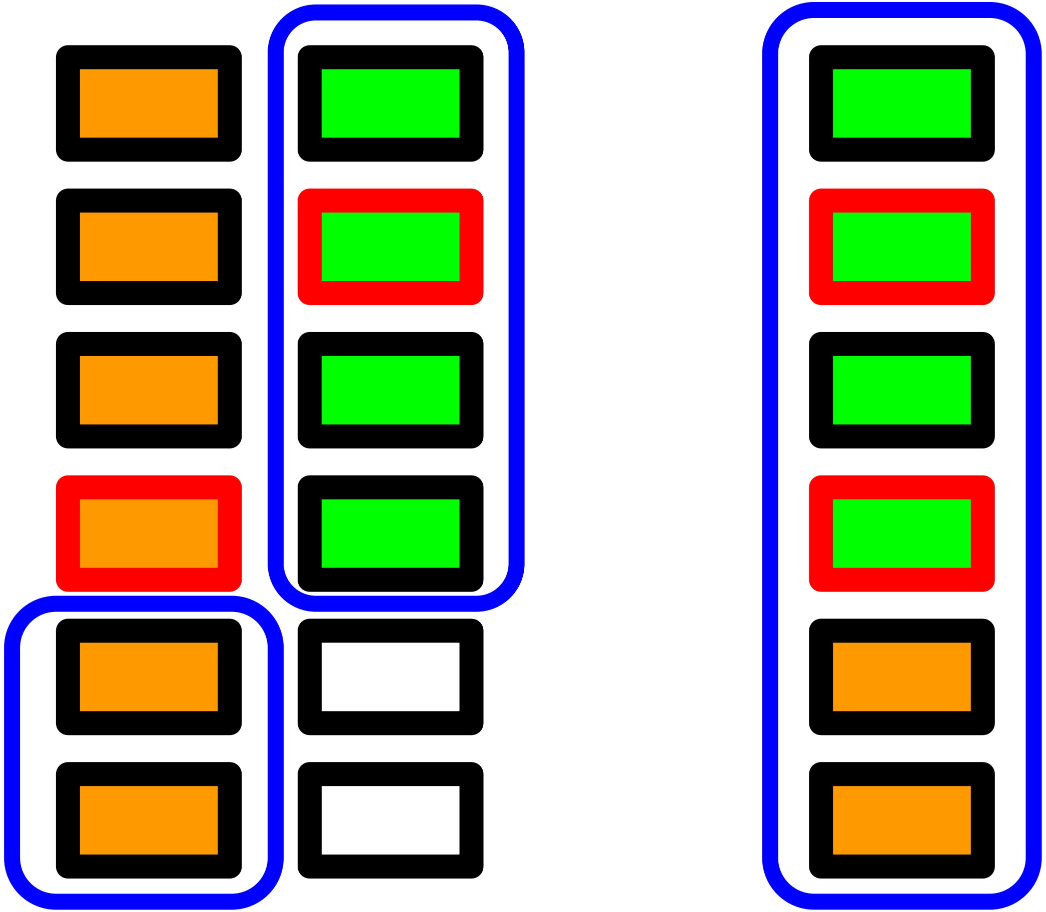


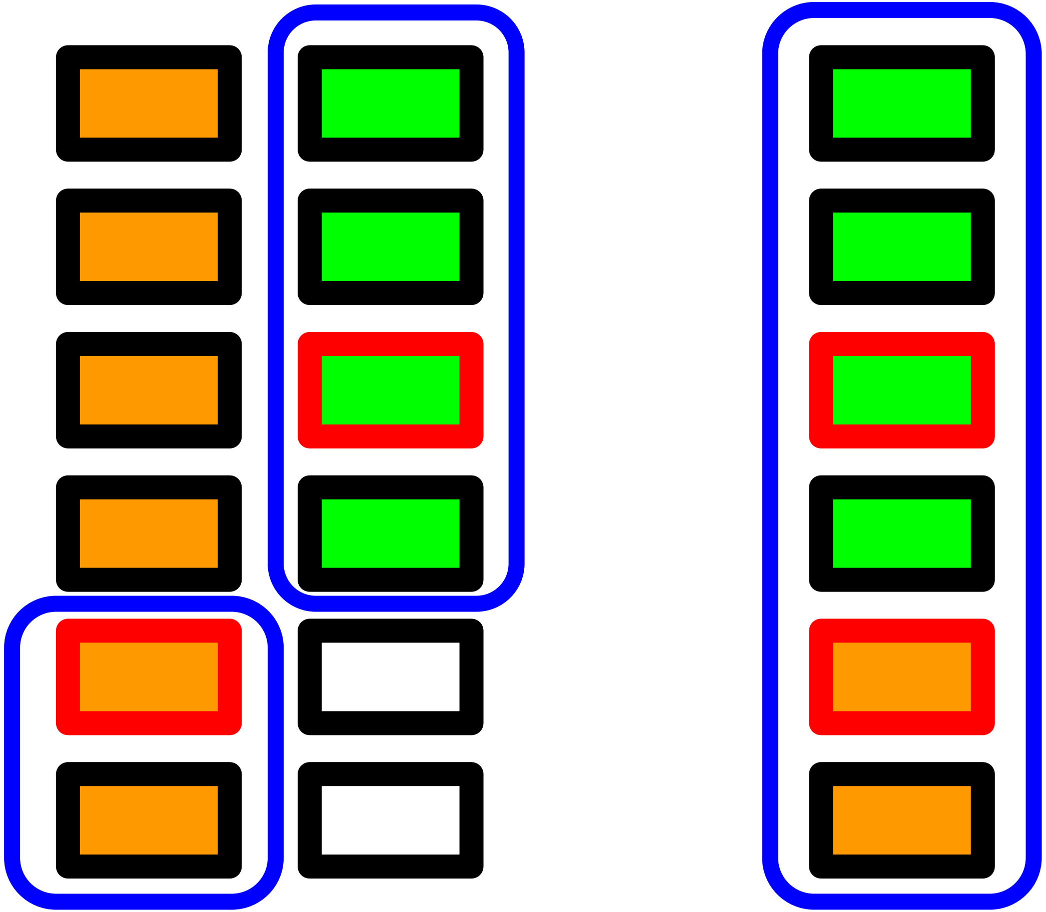


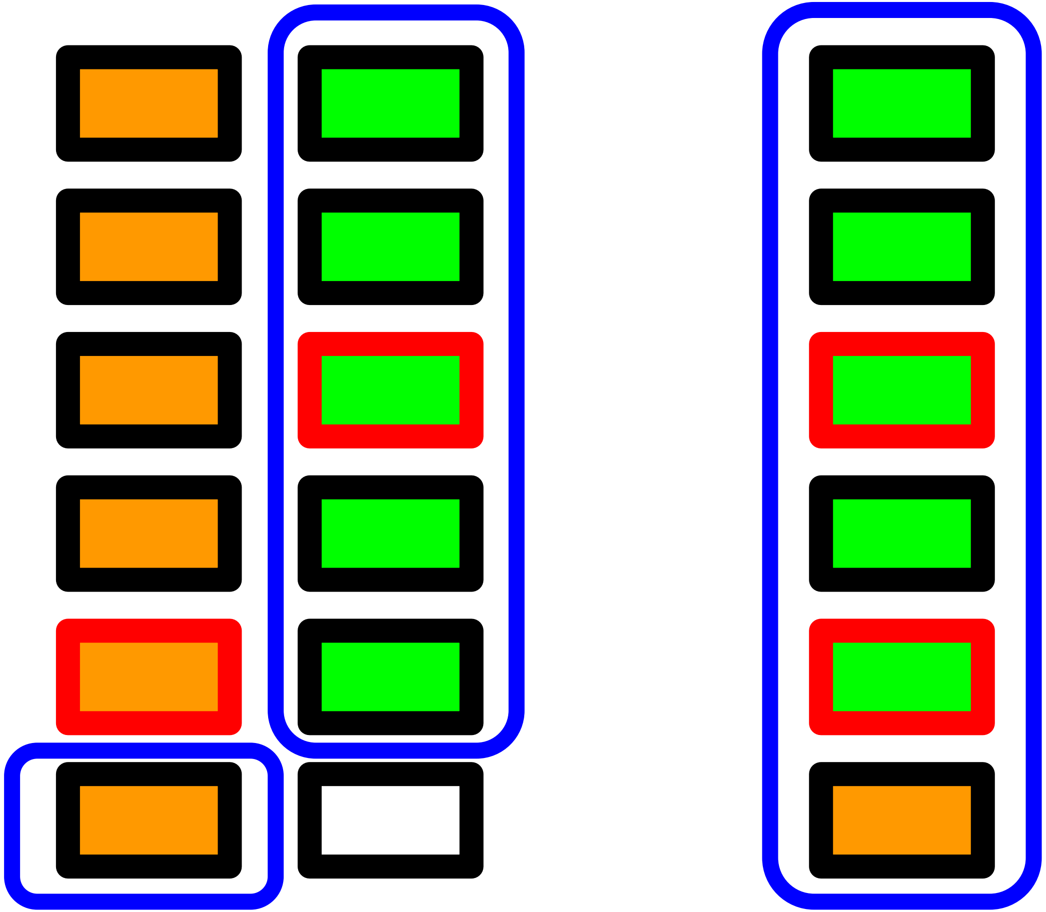


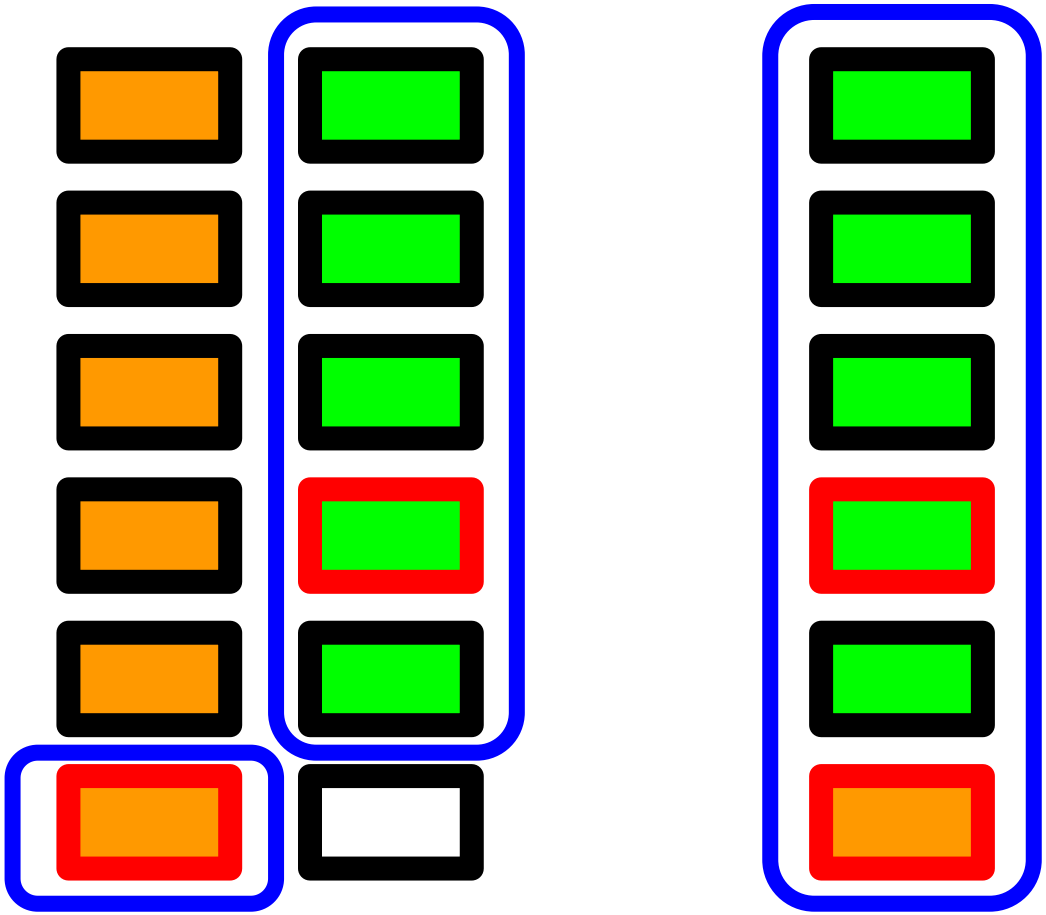


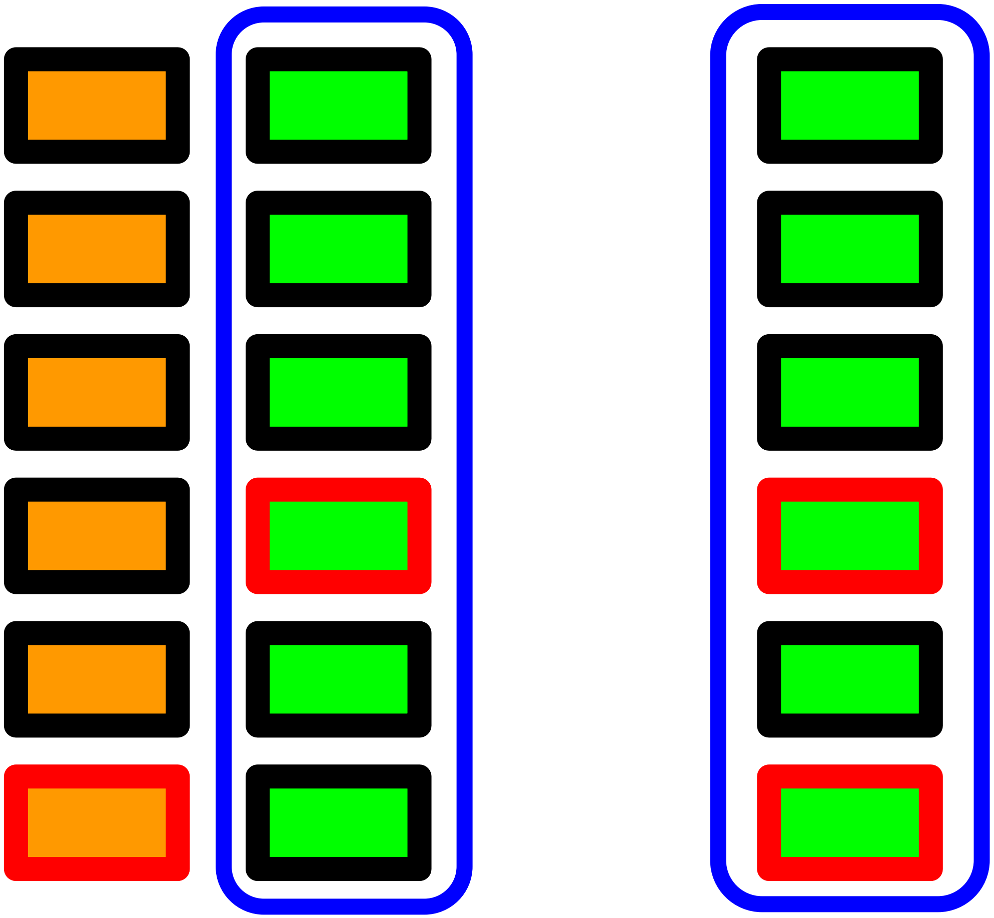




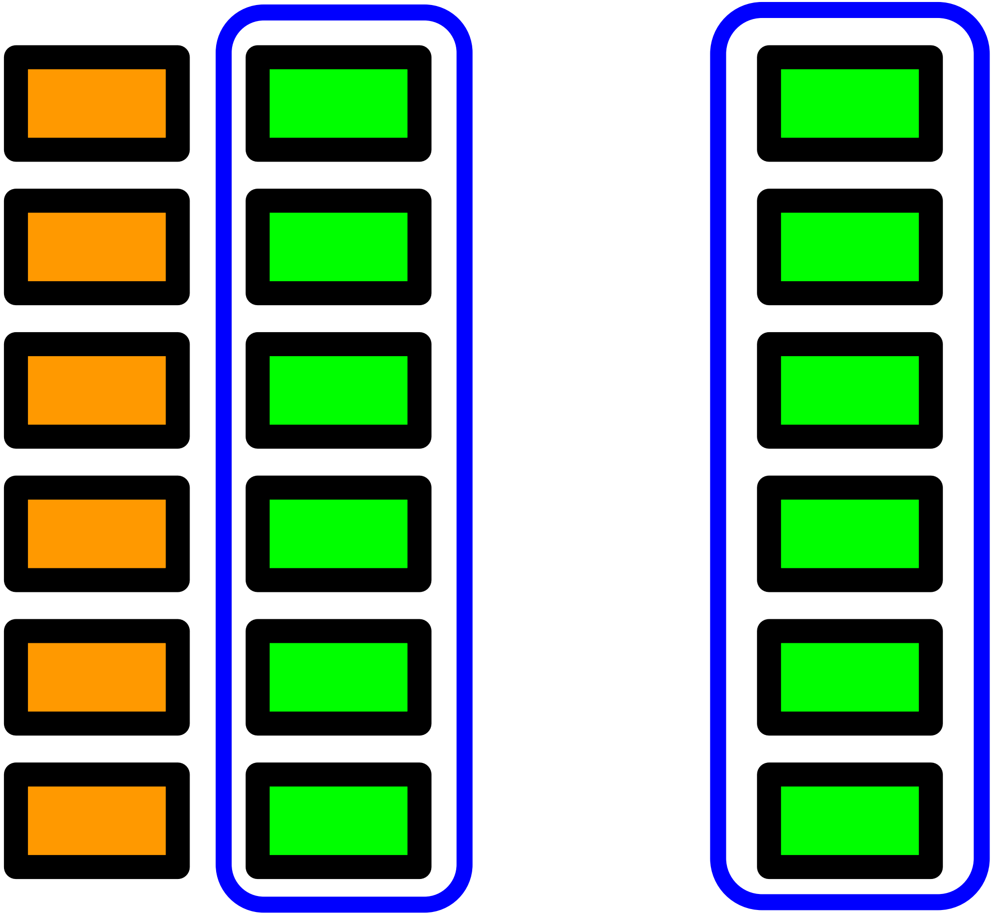








Saves space!
Improves space complexity



Final Approach

```
int knapsack(const Item *items_begin, const Item *items_end,
             const int capacity) {
  int *f = calloc(capacity + 1, sizeof(int));
  memset(f, 0, (capacity + 1) * sizeof(int));
  for (const Item *item = items_begin; item != items_end; ++item) {
    for (int i = item->weight; i <= capacity; ++i) {
      f[i] = max(f[i], f[i - item->weight] + item->profit);
    }
  }
  int ans = f[capacity];
  return ans;
}
```

	0	1	2
w_i	4	6	8
p_i	7	6	9

Looking closely

```
int knapsack(const Item *items_begin, const Item *items_end,
             const int capacity) {
  int *f = calloc(capacity + 1, sizeof(int));
  memset(f, 0, (capacity + 1) * sizeof(int));
  for (const Item *item = items_begin; item != items_end; ++item) {
    for (int i = item->weight; i <= capacity; ++i) {
      f[i] = max(f[i], f[i - item->weight] + item->profit);
    }
  }
  int ans = f[capacity];
  return ans;
}
```

$$P(C) = \max_{j=0}^{n-1} \overbrace{\{P(C - w_j) + p_j\}}^{\text{Choose } j^{th} \text{ element}}$$

P(14)	P(13)	P(12)	P(11)	P(10)	P(9)	P(8)	P(7)	P(6)	P(5)	P(4)	P(3)	P(2)	P(1)	P(0)