’

PRIORITY QUEUES AND SORTING

ADS1, S2023

PRIORITY QUEUES

Stack

Last in
First out

Queue

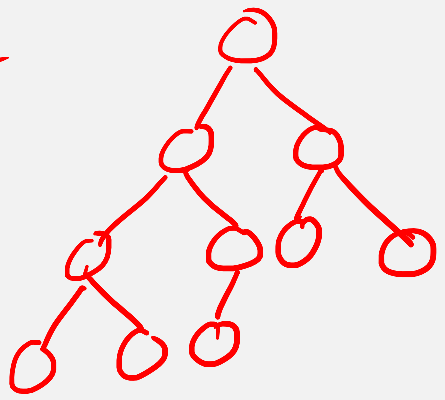
First in
First out

Priority queue

Elements out
in some priority
order

HEAPS

→ Complete binary tree



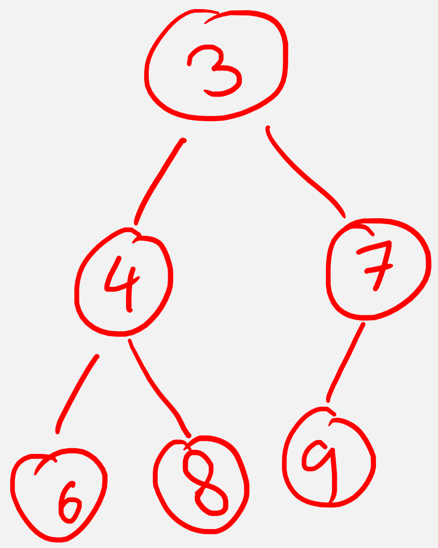
→ Each element is
$\leq$ its children

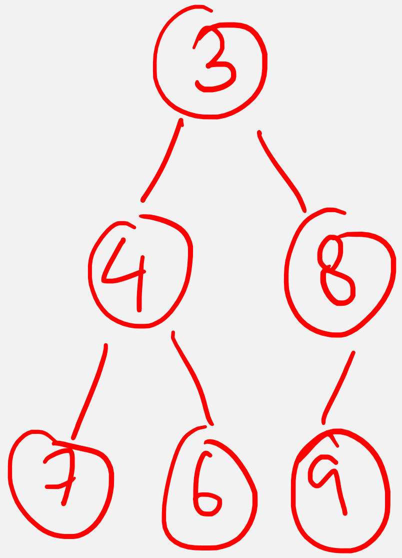
↘ min-heap

OPERATIONS ON A HEAP

- addElement
- removeMin
- findMin

EXAMPLES



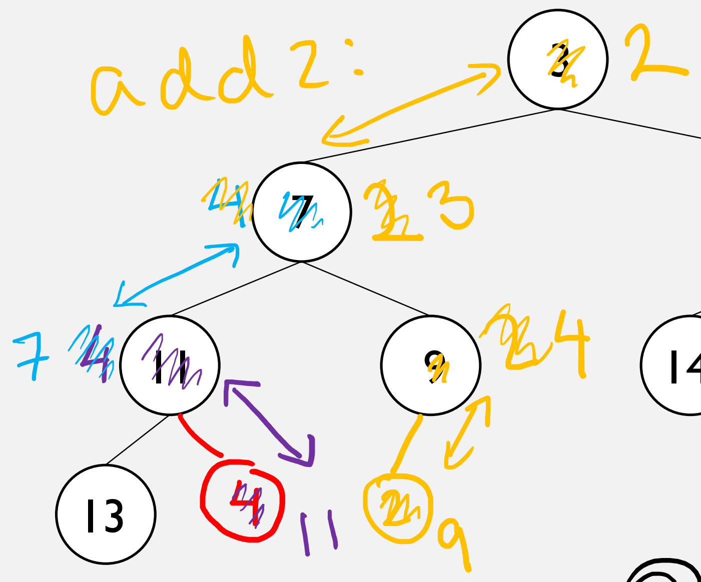


contains the same data

ADDING AN ELEMENT TO A HEAP

add 4:

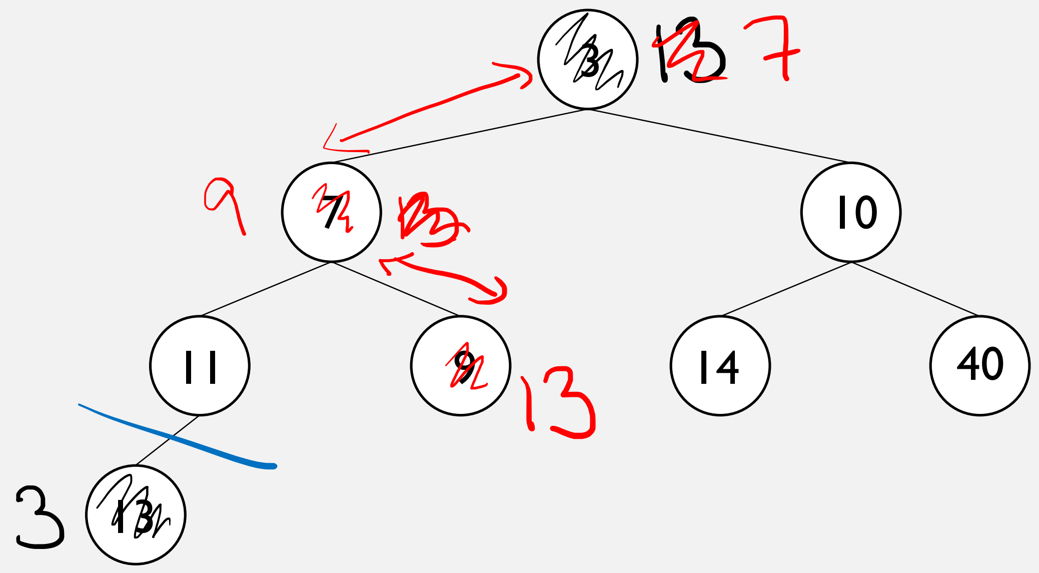
add 2:



add element as next leaf
while <parent:
    swap

$O(\log n)$

REMOVING THE MINIMUM ELEMENT



Swap root and last leaf

remove last leaf from heap

while any children smaller:

swap with smallest

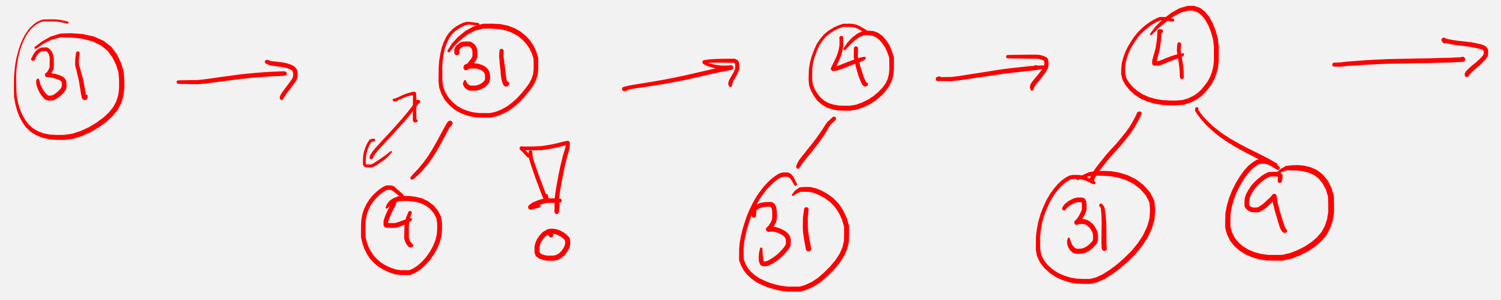
HEAPSORT

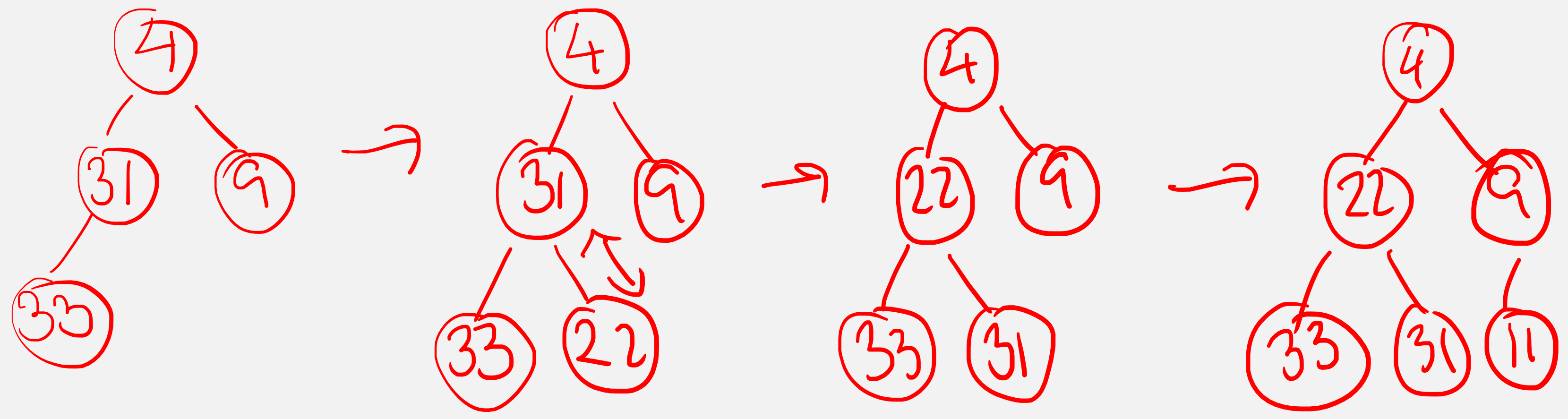
HEAPSORT

Sort this:

31	4	9	33	22	11

turn into heap:

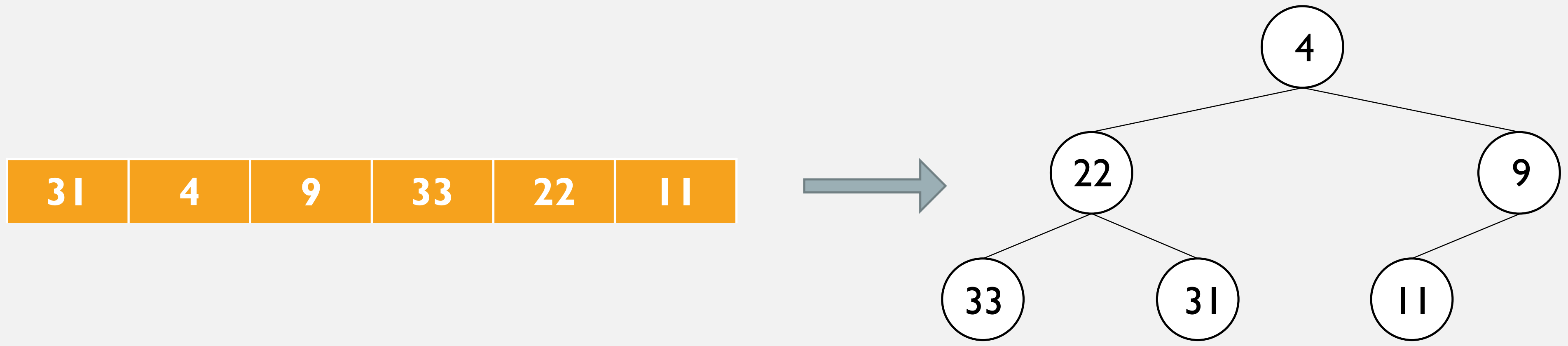




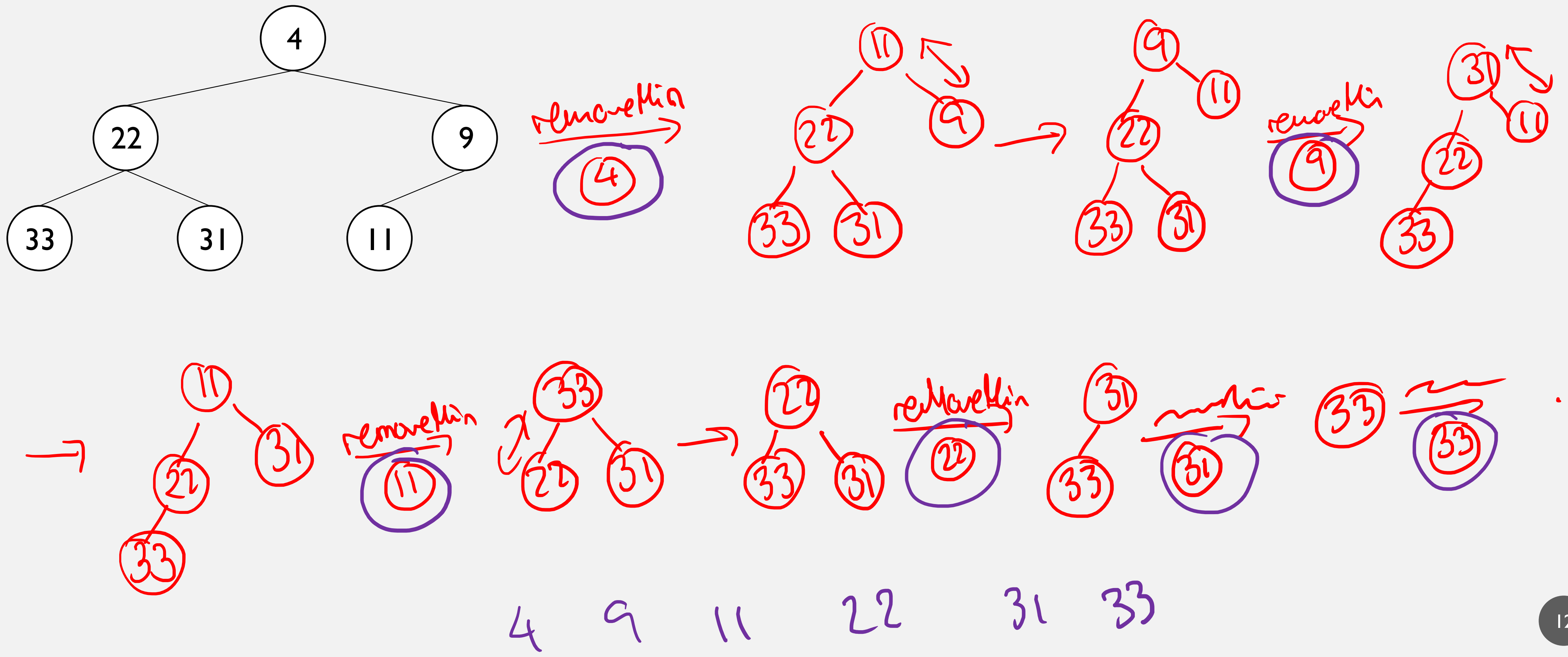
WHAT IS THE TIME COMPLEXITY OF BUILDING A HEAP?

$O(n \log n)$

HEAPSORT



HEAPSORT

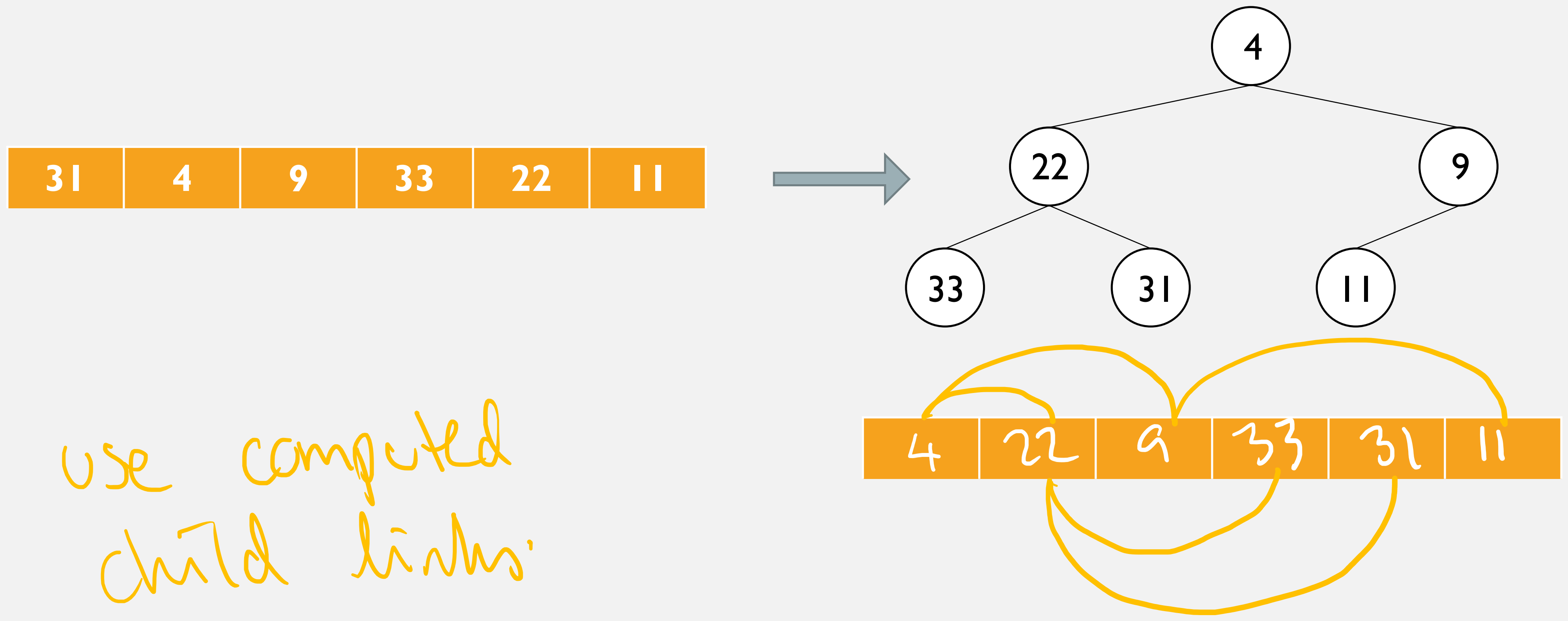


4
22
9
33
31
11
removeMin
removeMin
removeMin
removeMin
removeMin
4 9 11 22 31 33

TIME COMPLEXITY OF HEAPSORT?

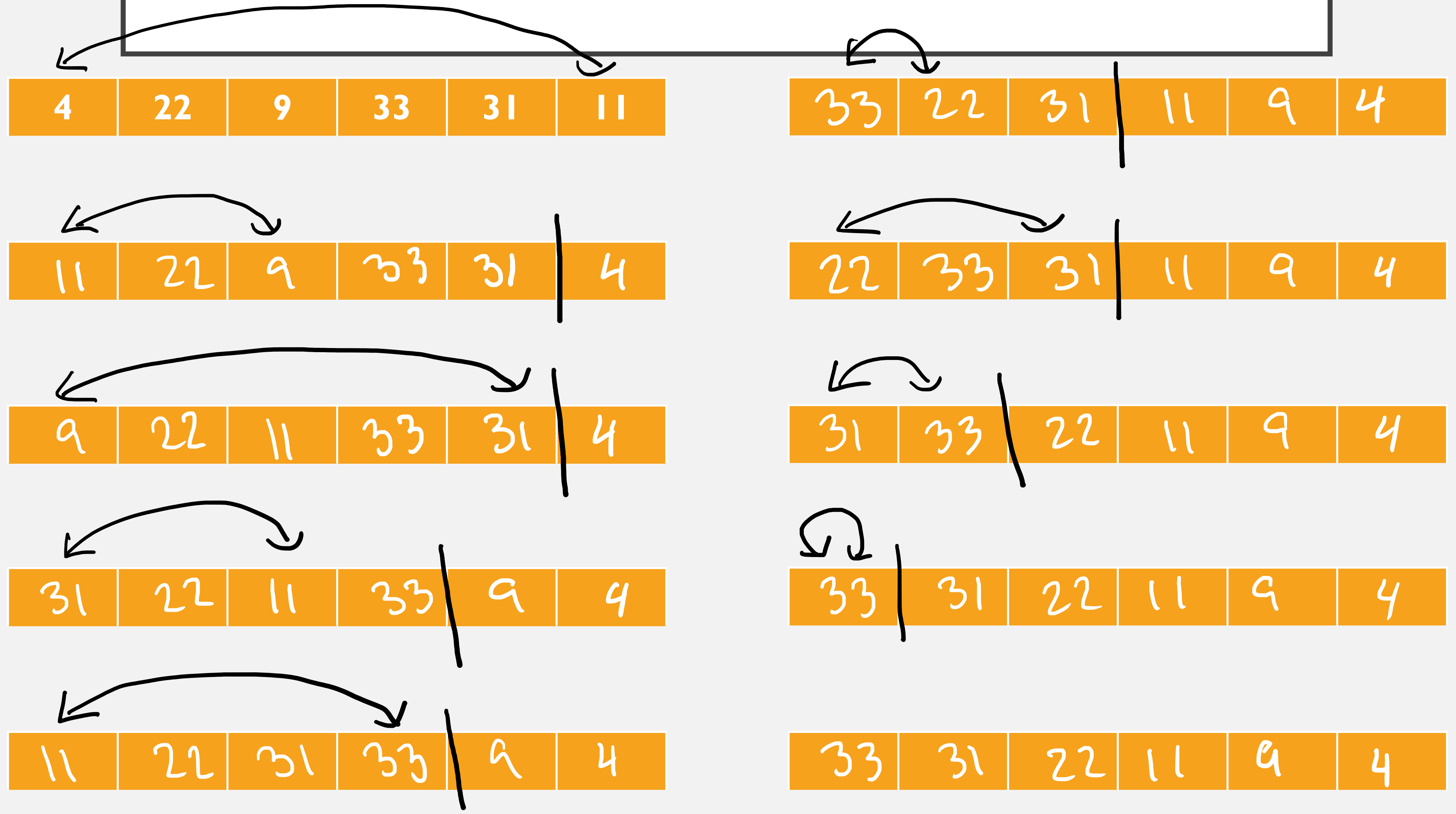
$O(n \log n)$

REPRESENTING HEAPS



use computed
child links

HEAPSORT IN MEMORY



HEAPSORT IN MEMORY

array	31	4	9	33	22	11
↓						
heap	4	22	9	33	31	11
↓						
sorted array	33	31	22	11	9	4

HEAPSORT PSEUDOCODE

```
to sort array A:
    build min-heap from A
    while heap not empty:
        removeMin(A)
    read array right-to-left
```

OVERVIEW OF SORTING ALGORITHMS

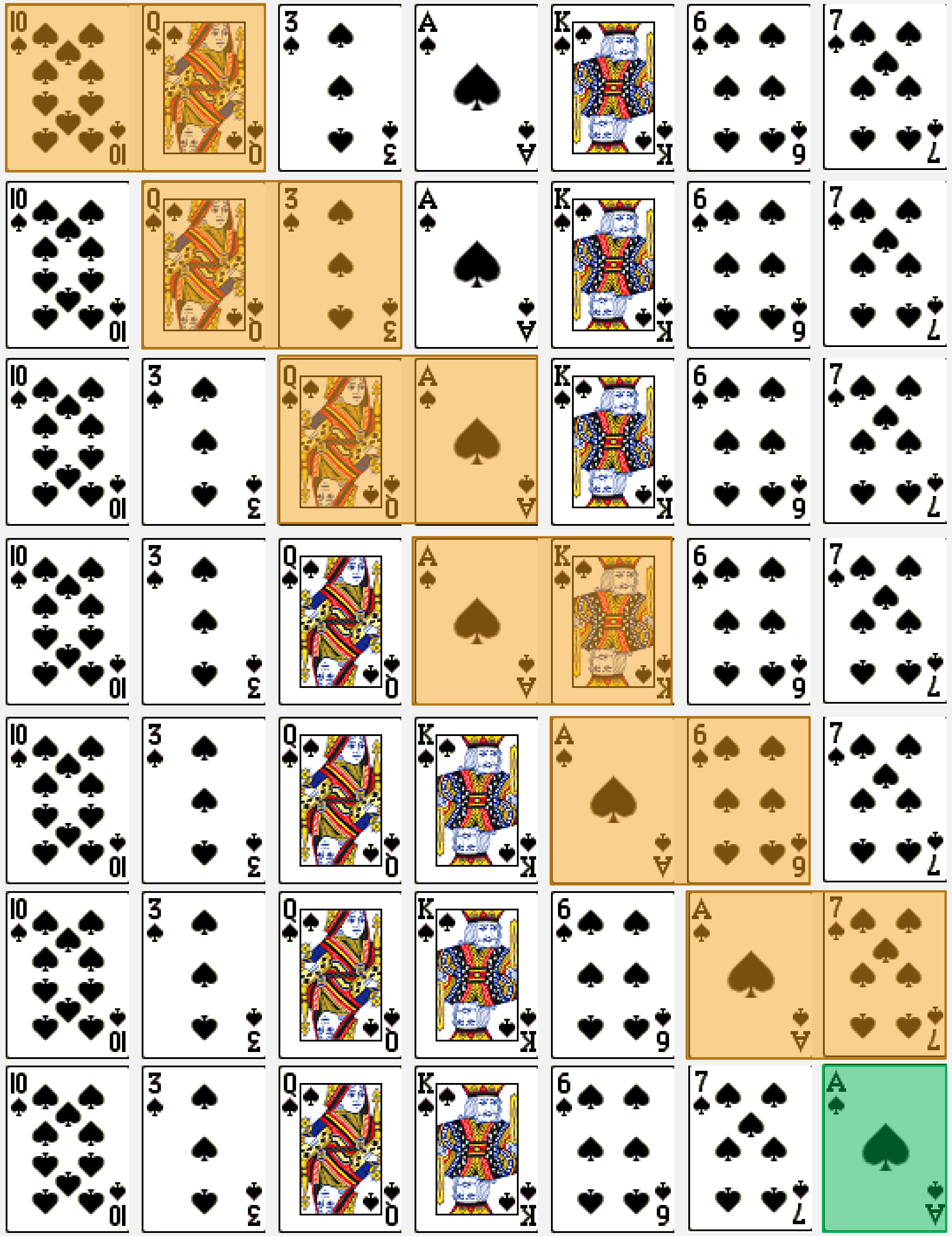
OVERVIEW OF SORTING ALGORITHMS

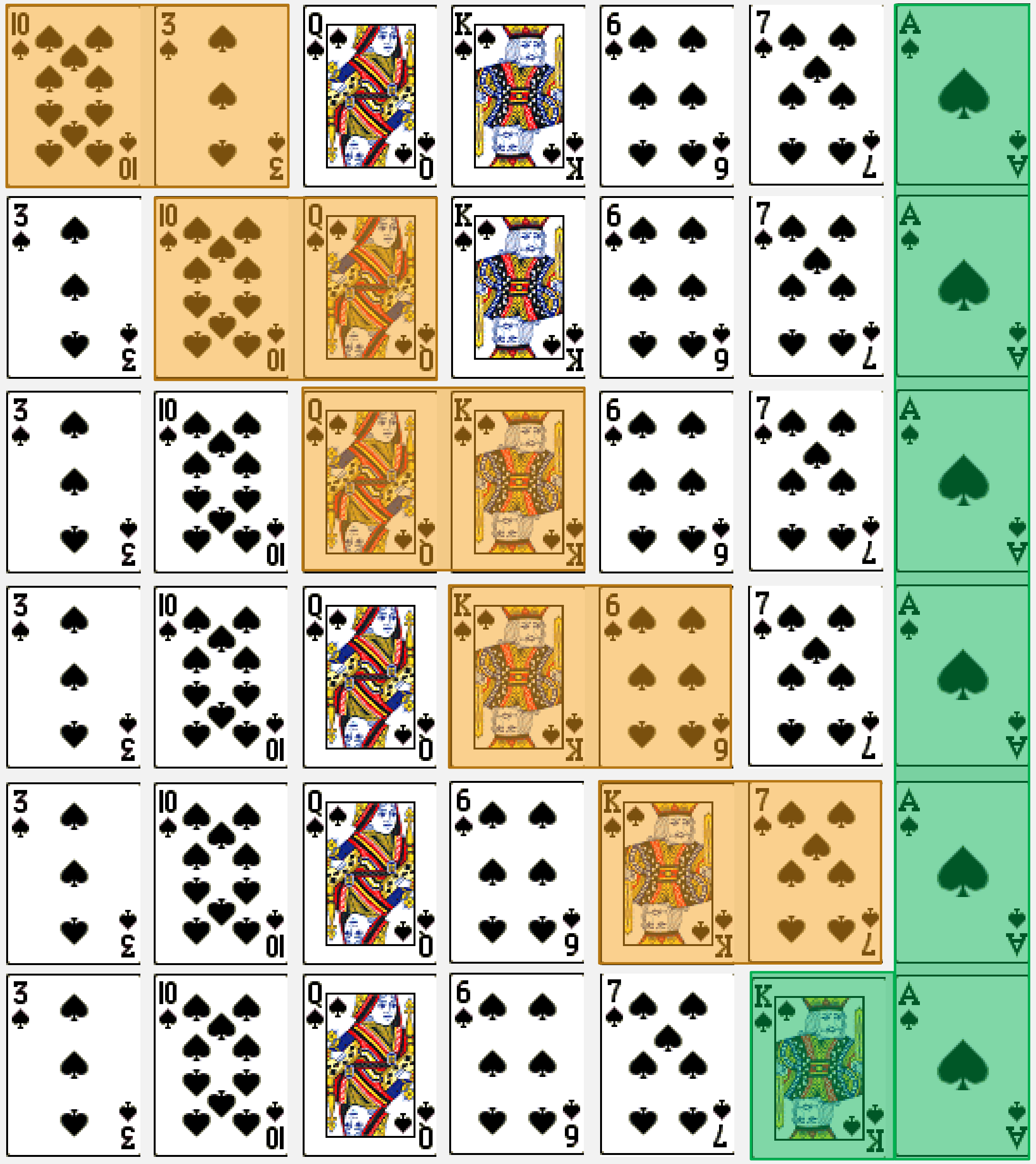
	Best case	Average case	Worst case	Space complexity	Adaptive?	Stable?
BubbleSort	n	n^2	n^2	1	✓	✓
InsertionSort	n	n^2	n^2	1	✓	✓
HeapSort	$n \log n$	$n \log n$	$n \log n$	1	%	%
MergeSort	$n \log n$	$n \log n$	$n \log n$	n	%	✓
QuickSort	$n \log n$	$n \log n$	n^2	$\log n$	✓	%
BucketSort	n	n	n^2	$n+k$	%	✓

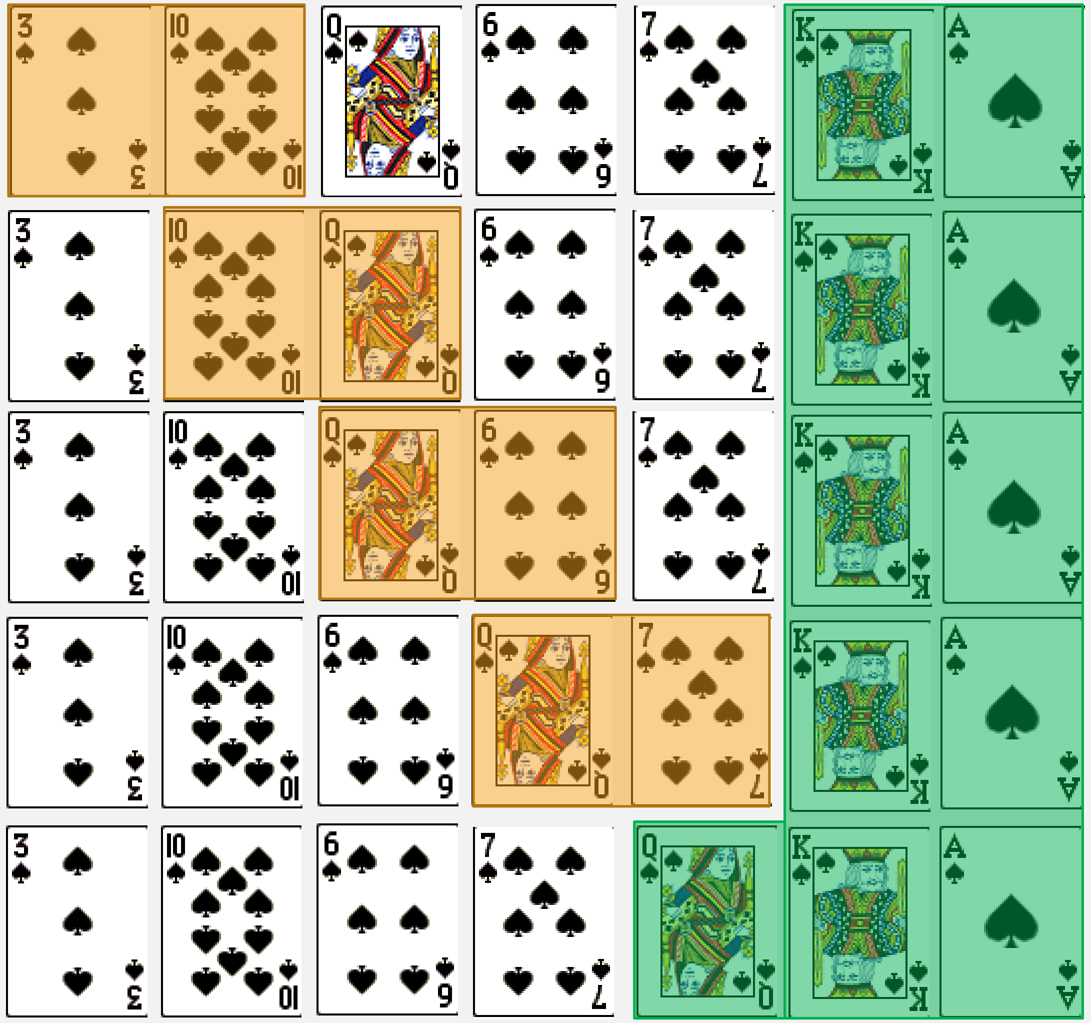
BUBBLESORT

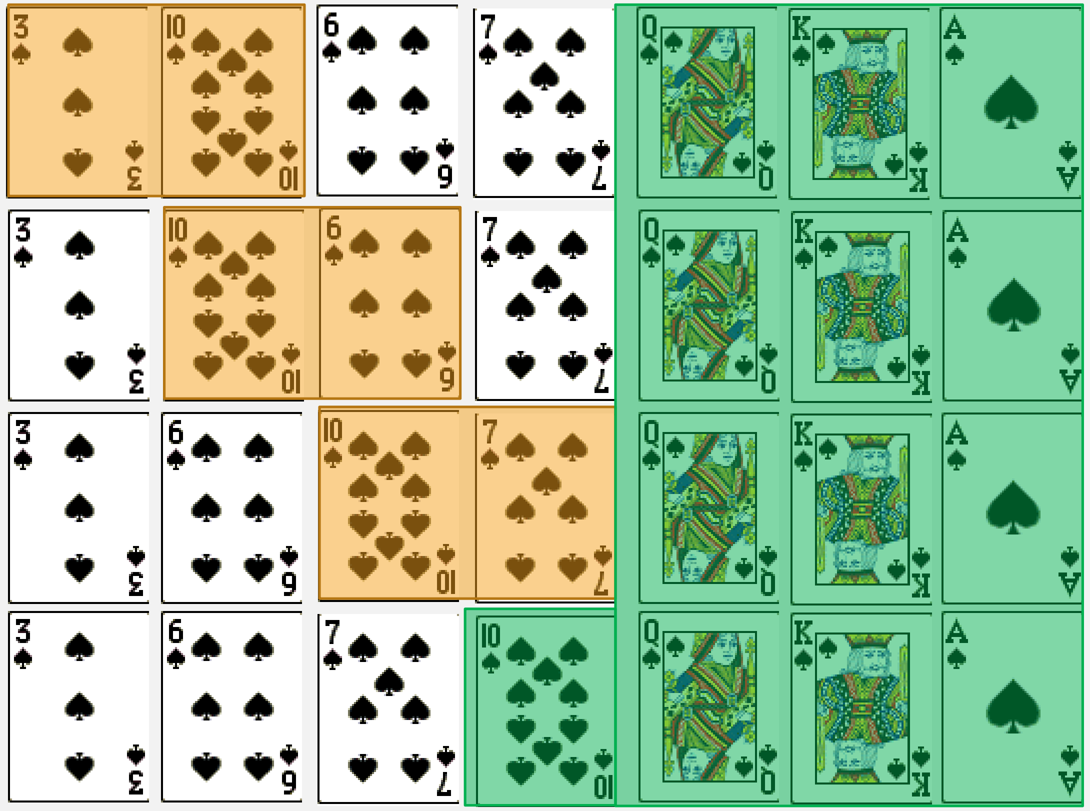
BUBBLESORT

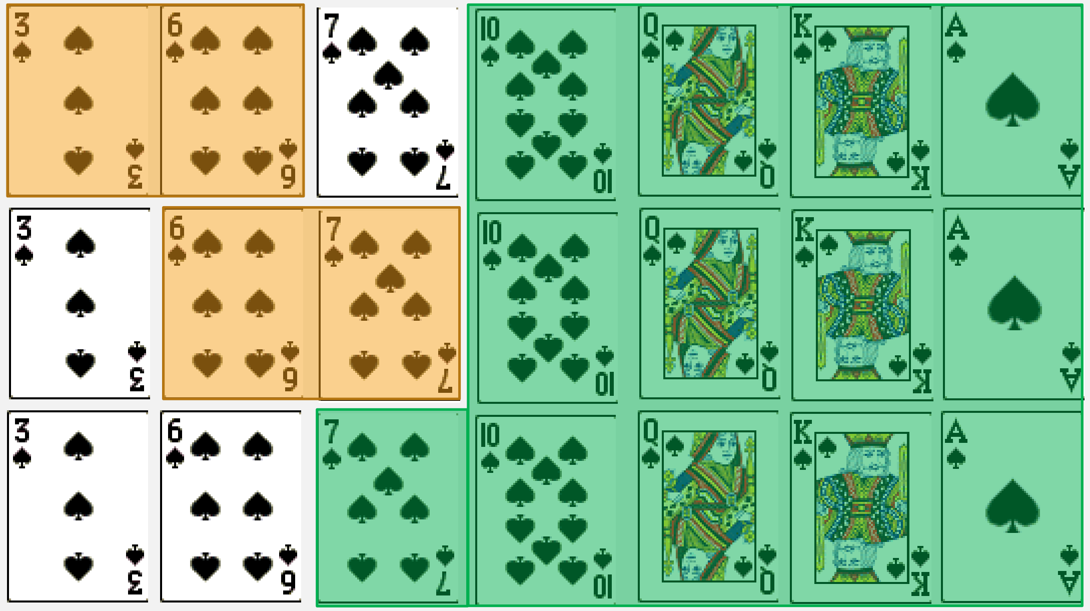
```
BubbleSort(list):
    repeat length(list) times:
        for all elements in list:
            if list[i] > list[i+1]:
                swap list[i] with list[i+1]
    return list
```

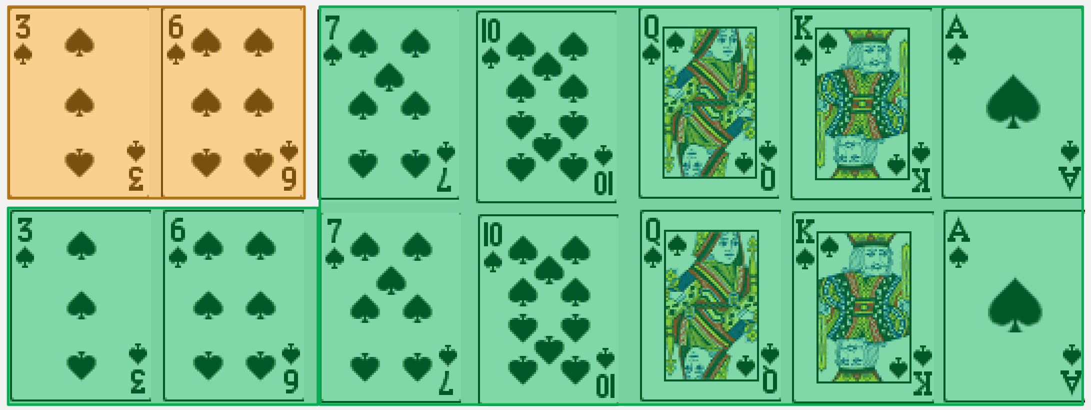








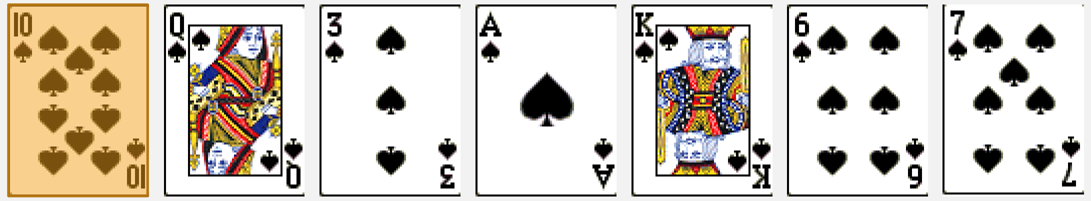


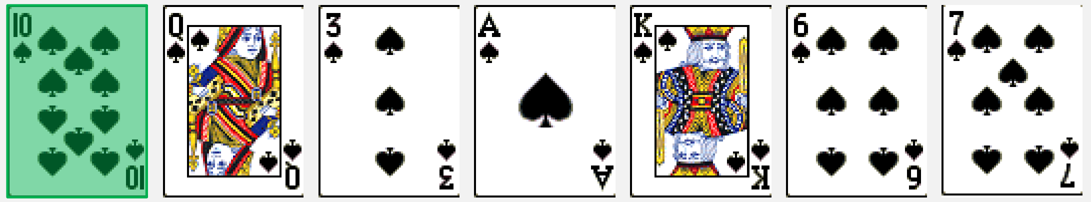


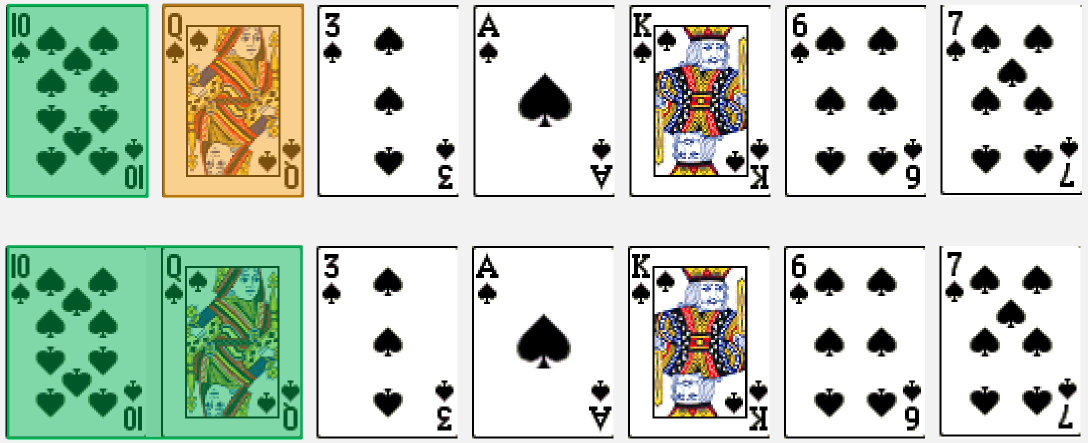
INSERTIONSORT

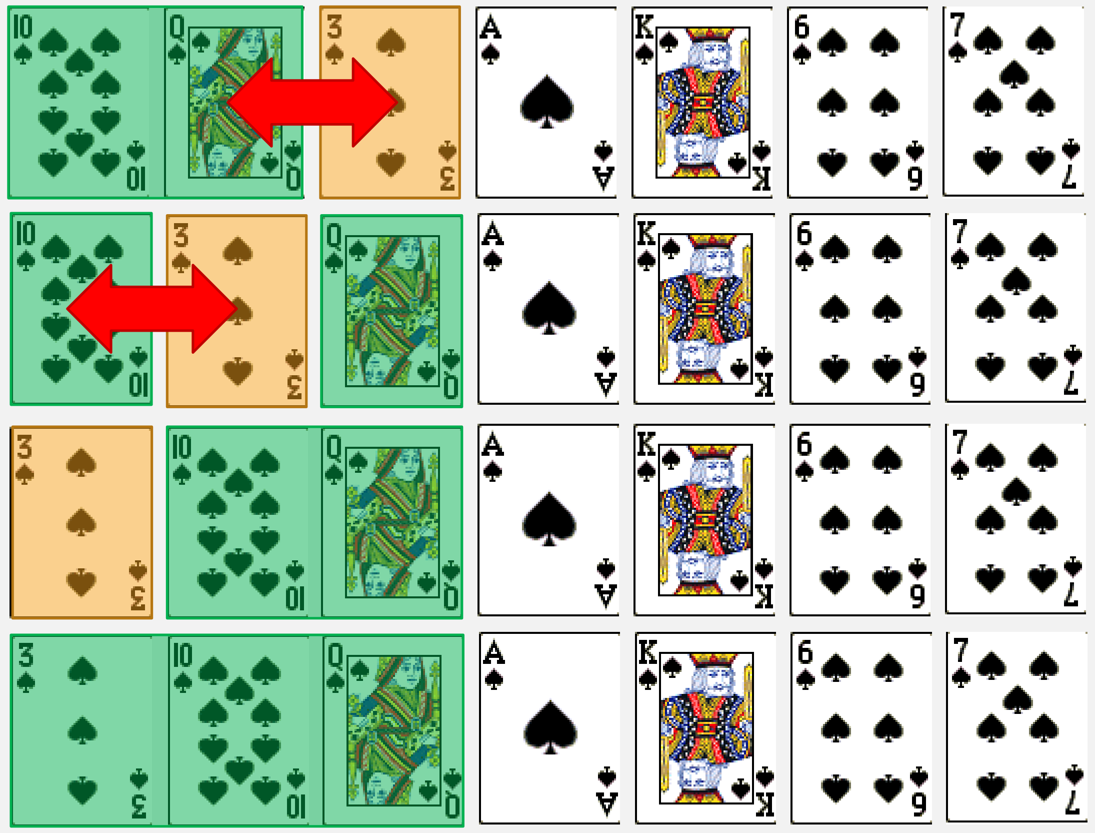
INSERTIONSORT

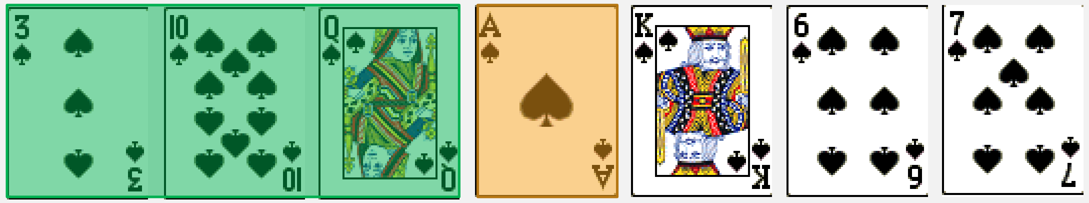
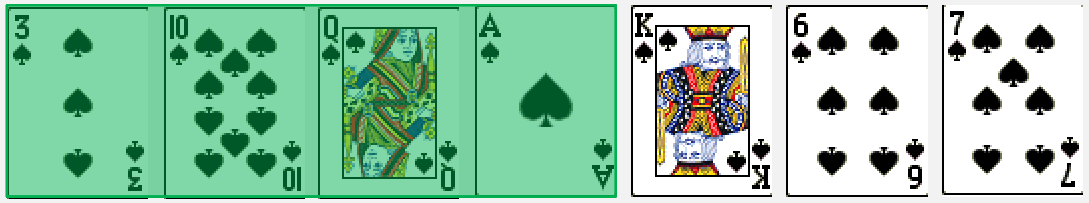
```
InsertionSort(list):
    if it is the first element, it is already a sorted sublist
    repeat until list is sorted:
        pick next element
        compare with all elements in the sorted sublist
        shift all elements in the sorted sublist that is
            greater than the value to be sorted
        insert the value
```

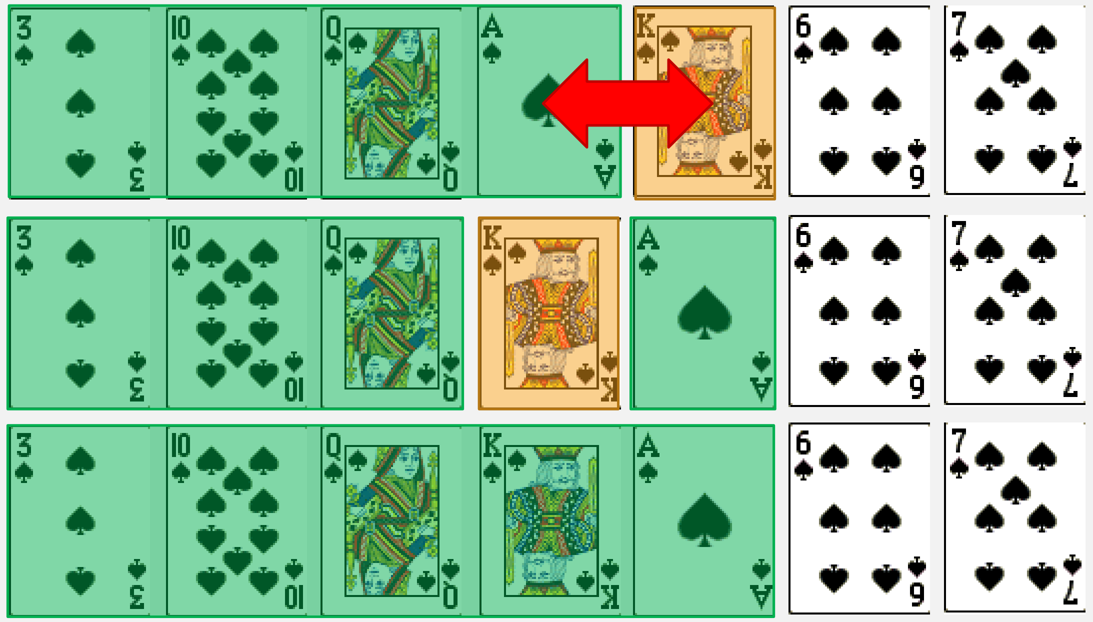


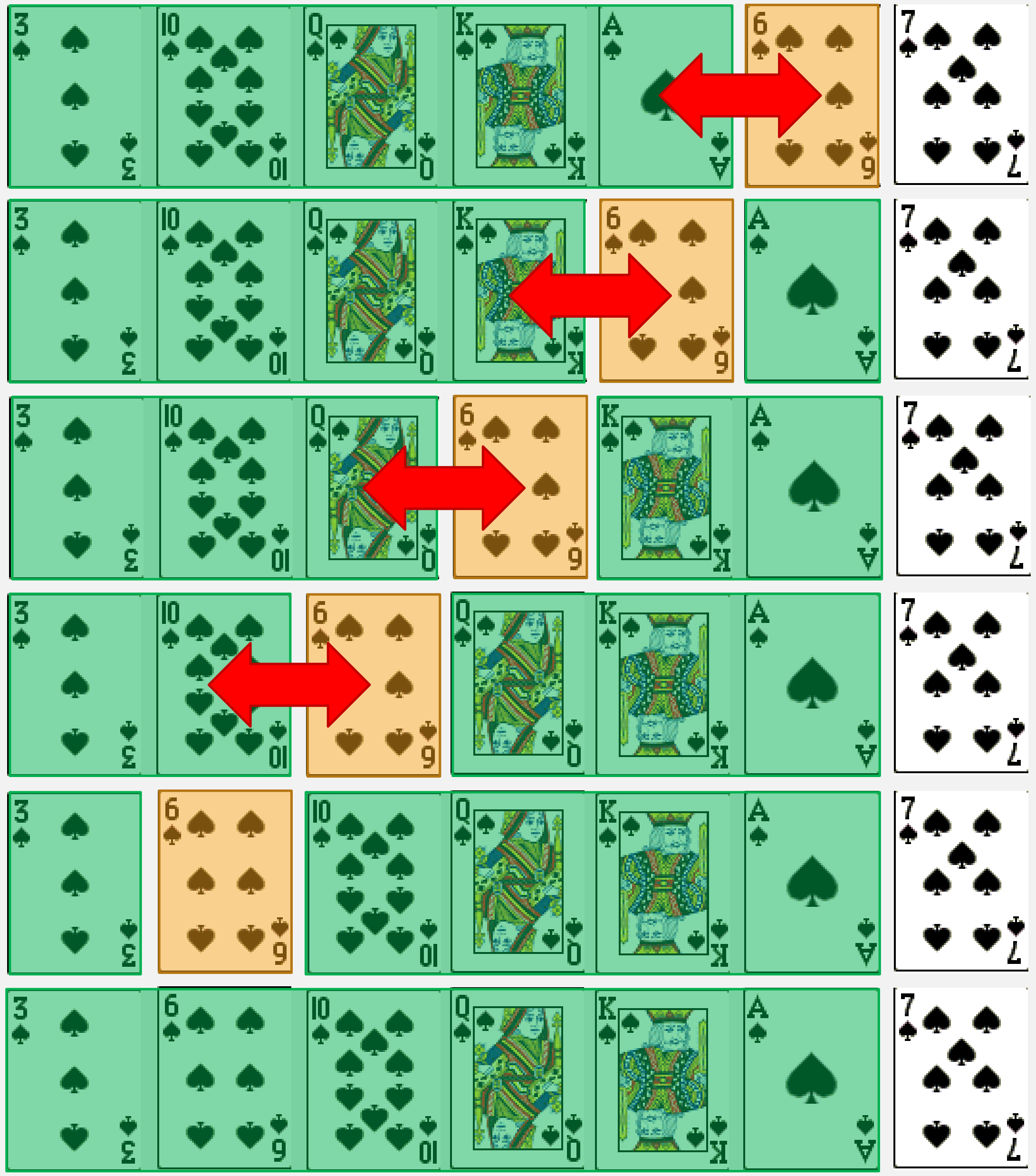


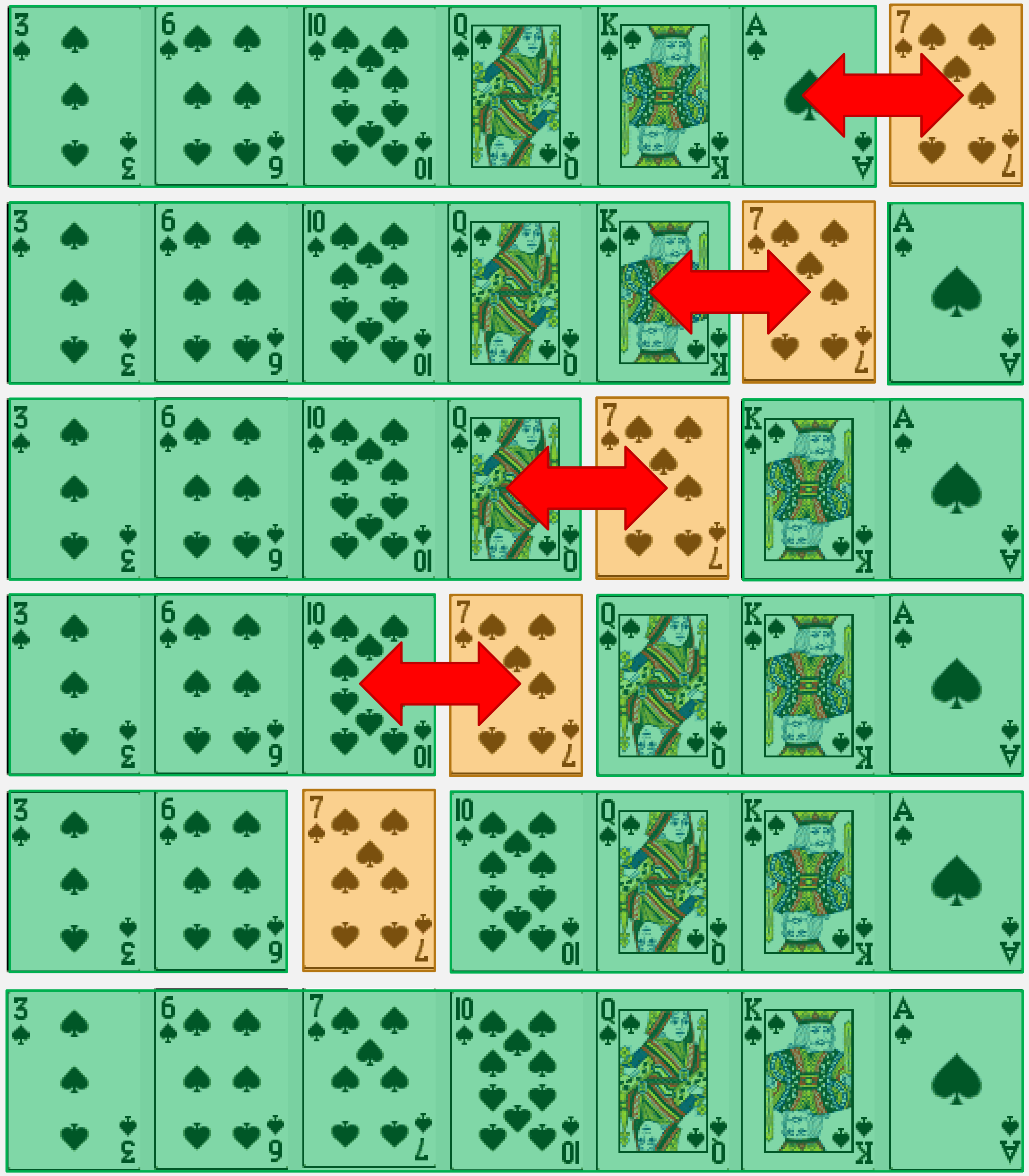








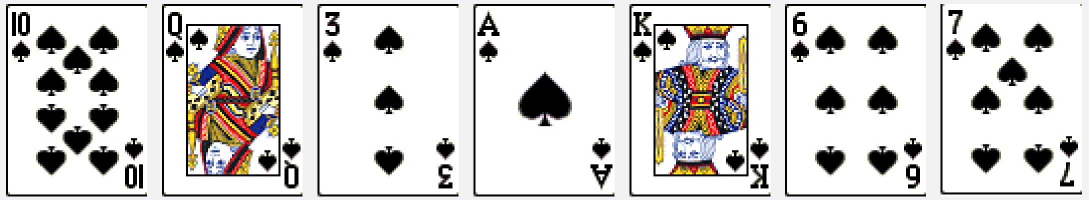


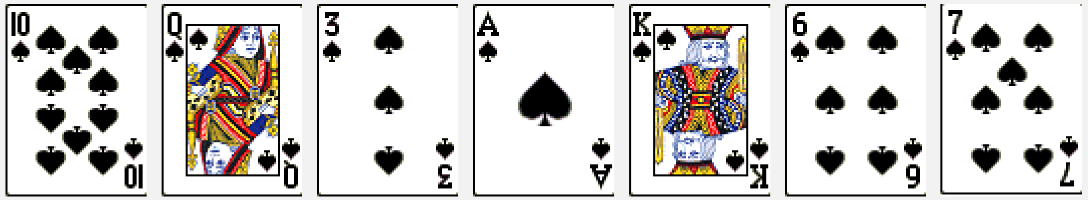
MERGESORT

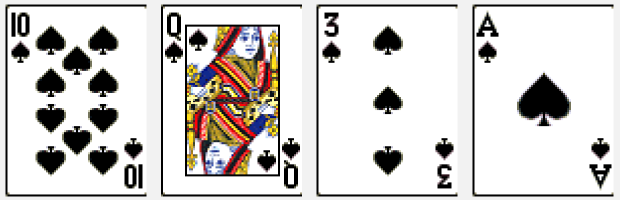
MERGESORT

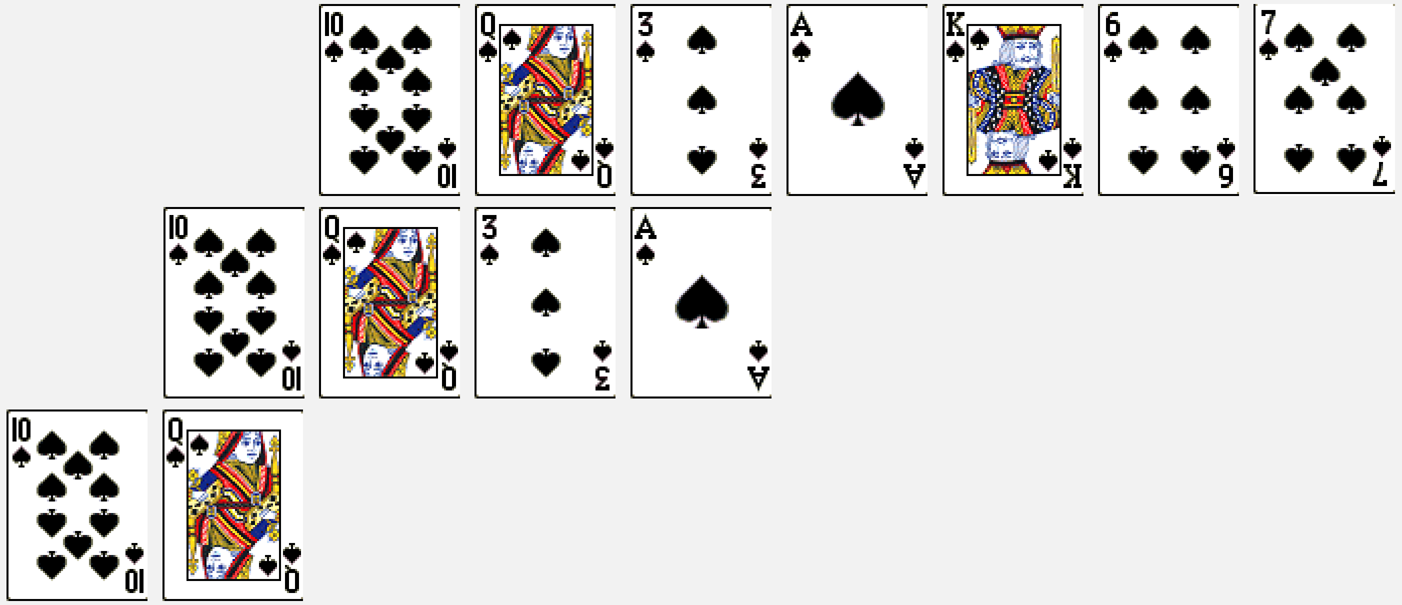
```
MergeSort(list, p, r):
    if p < r:
        q = floor((p + r) / 2)
        MergeSort(list, p, q)
        MergeSort(list, q + 1, r)
        Merge(list, p, q, r)


call MergeSort(list, 1, length(list))
```

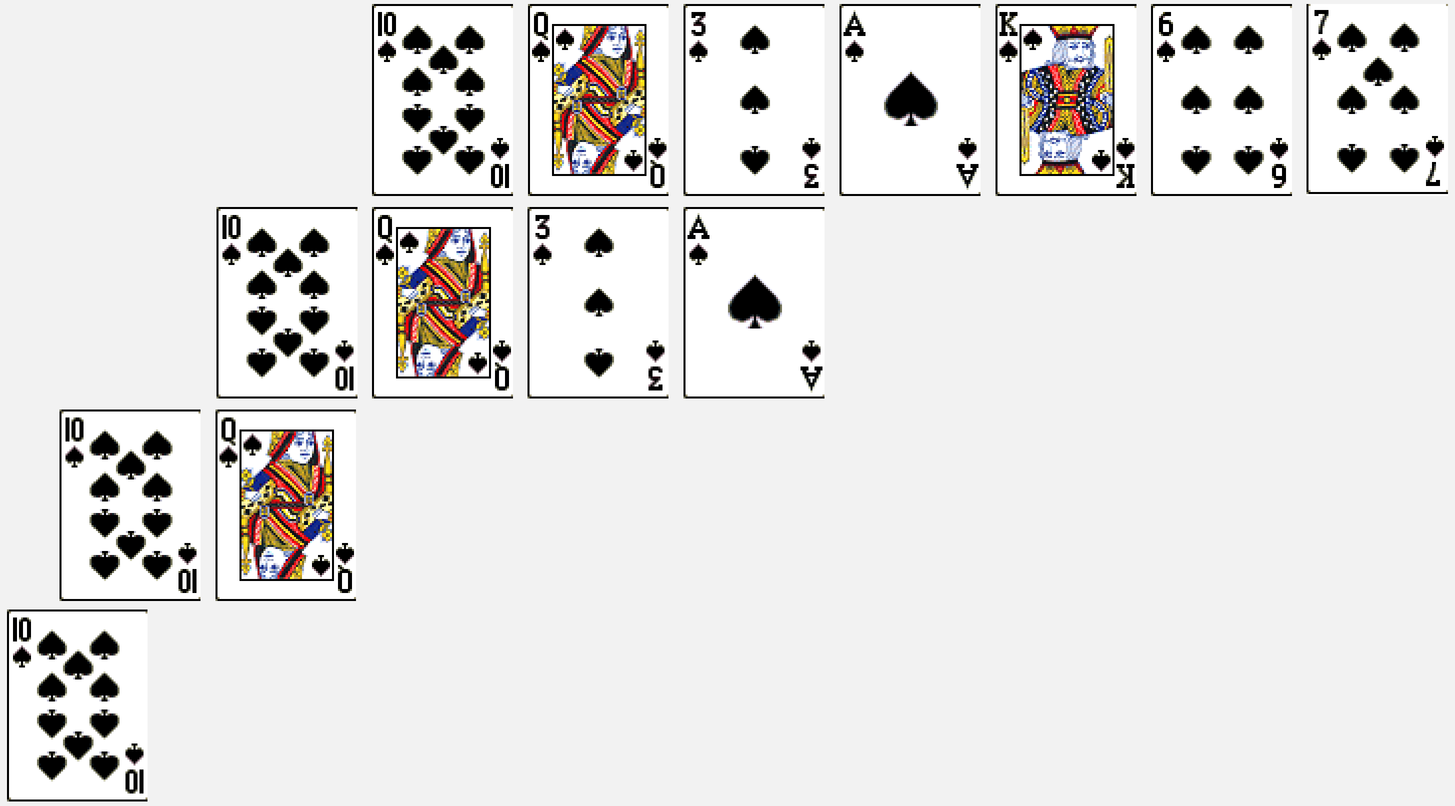
```
Merge(list, p, q, r):
    let L be [list[1], …, list[q], ∞]
    let R be [list[q + 1], …, list[end], ∞]
    i = 1
    j = 1
    while L[i] < ∞ and R[j] < ∞:
        if L[i] ≤ R[j]:
            list[k] = L[i]
            i = i + 1
        else:
            list[k] = R[j]
            j = j + 1
```

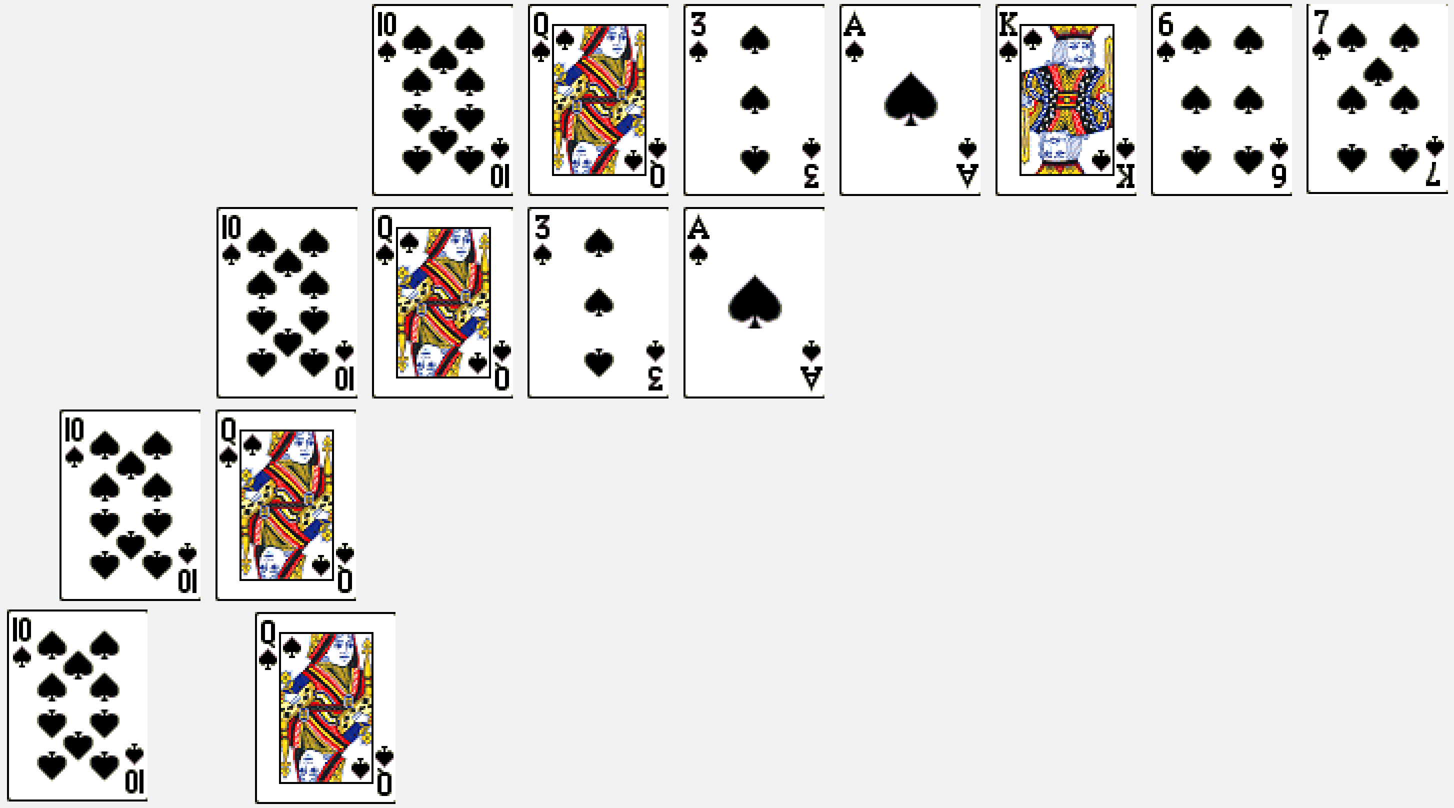


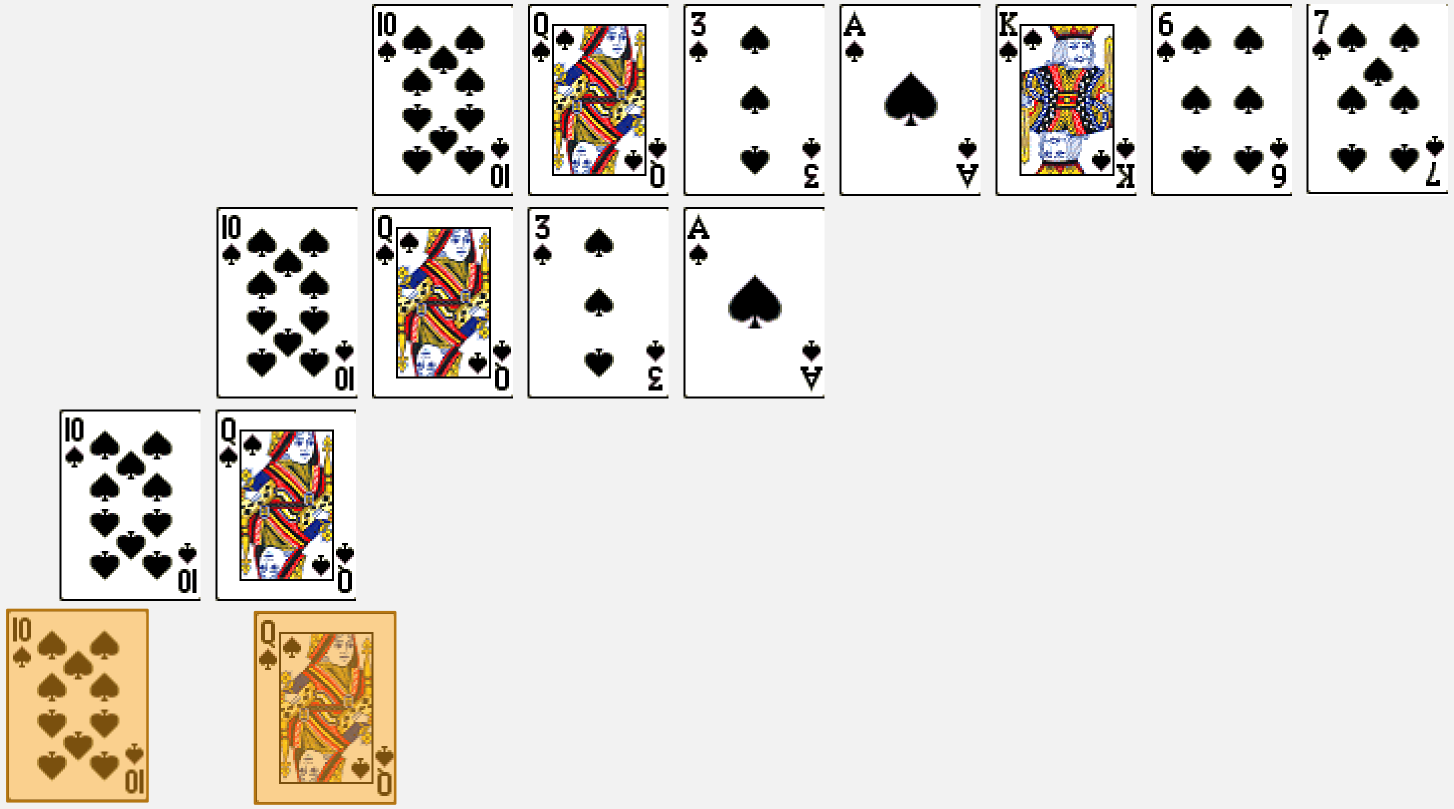


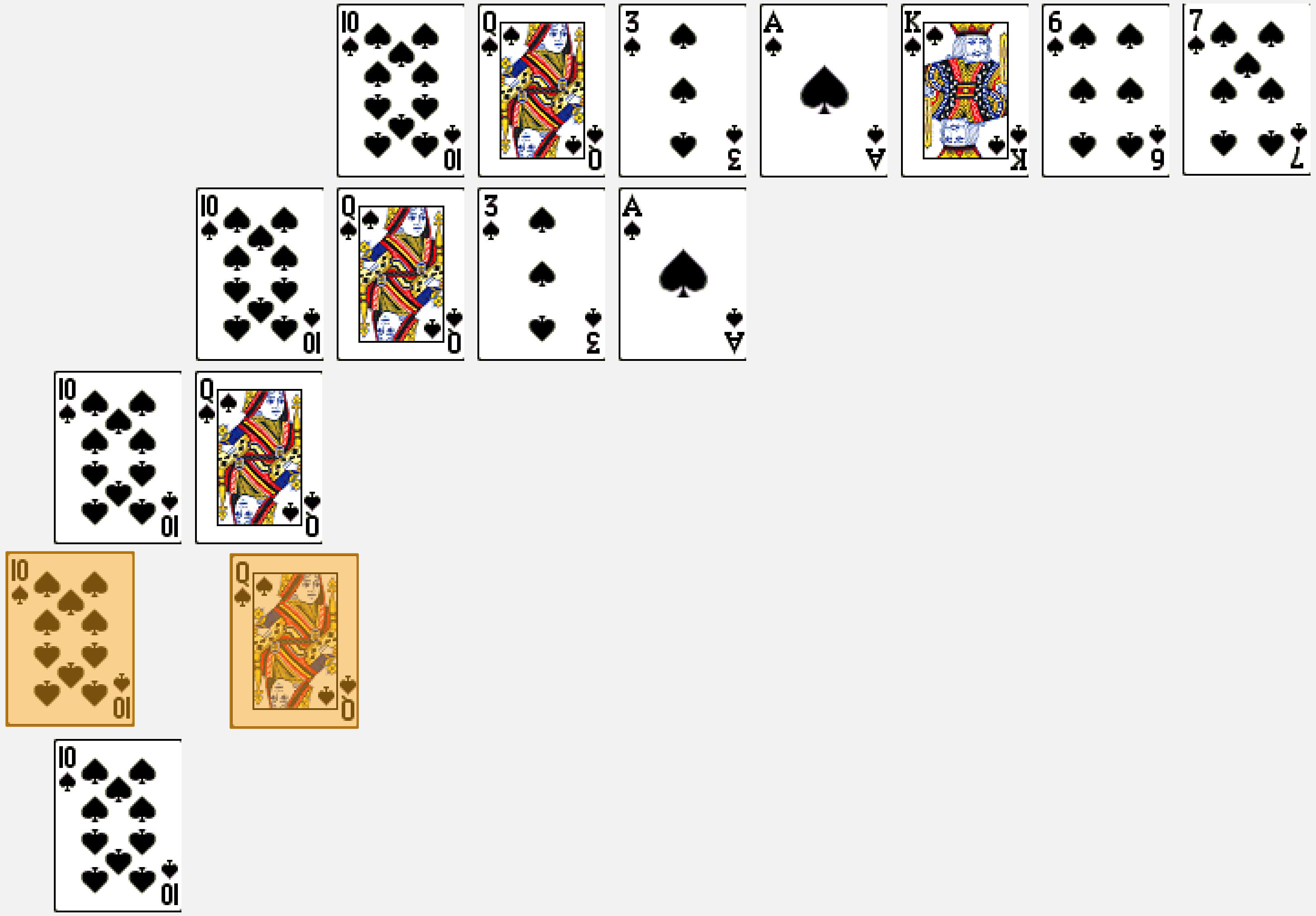


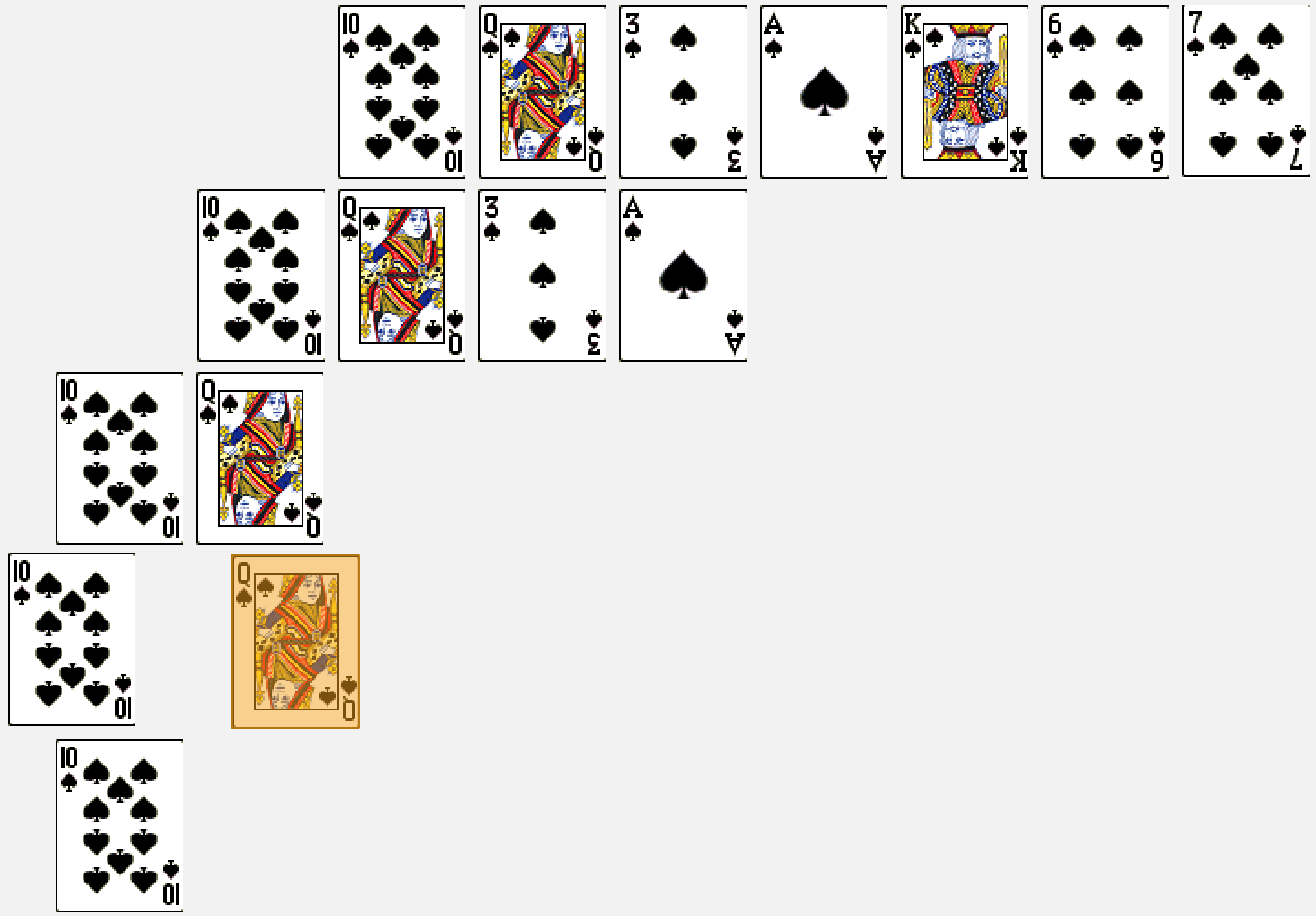


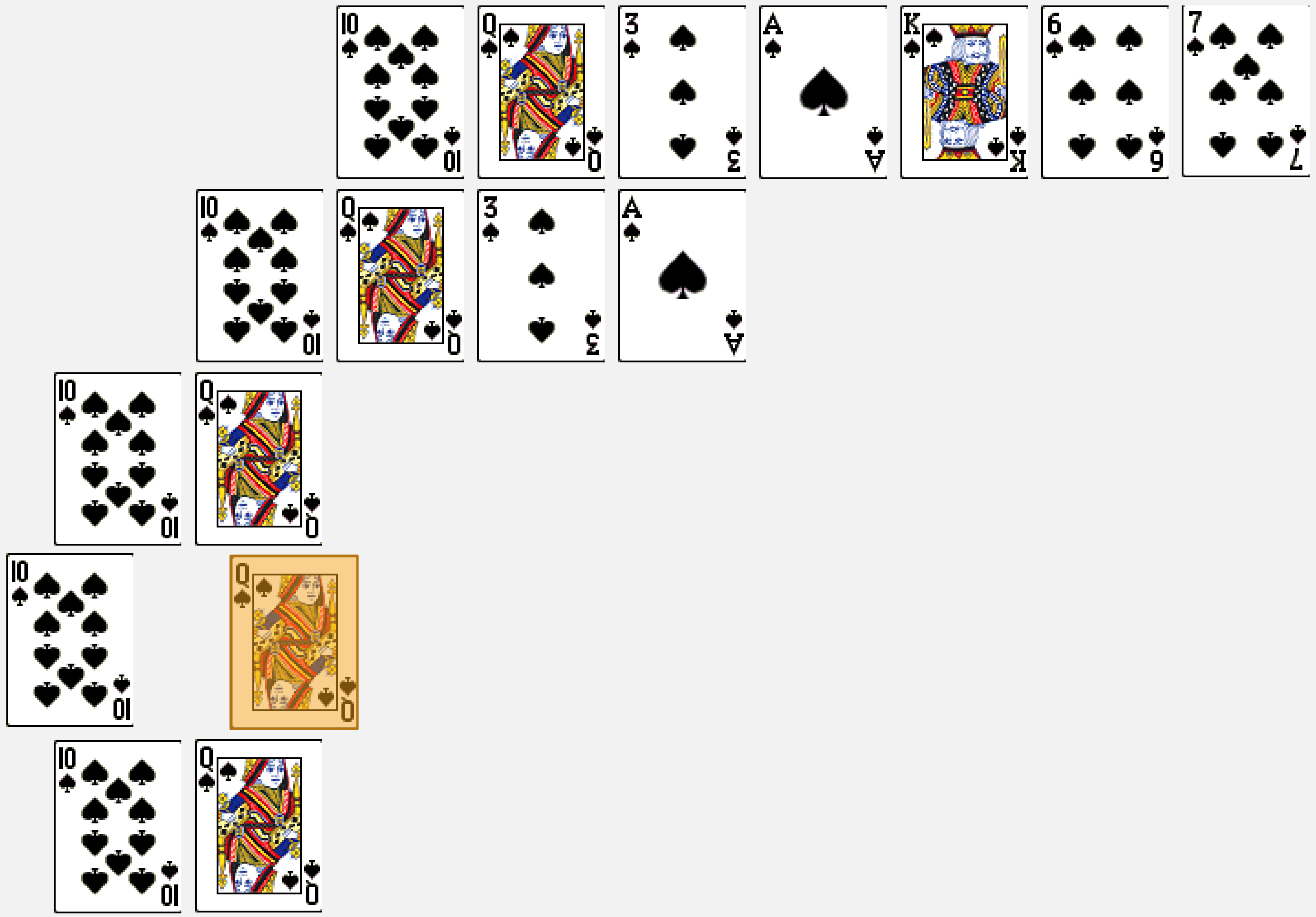


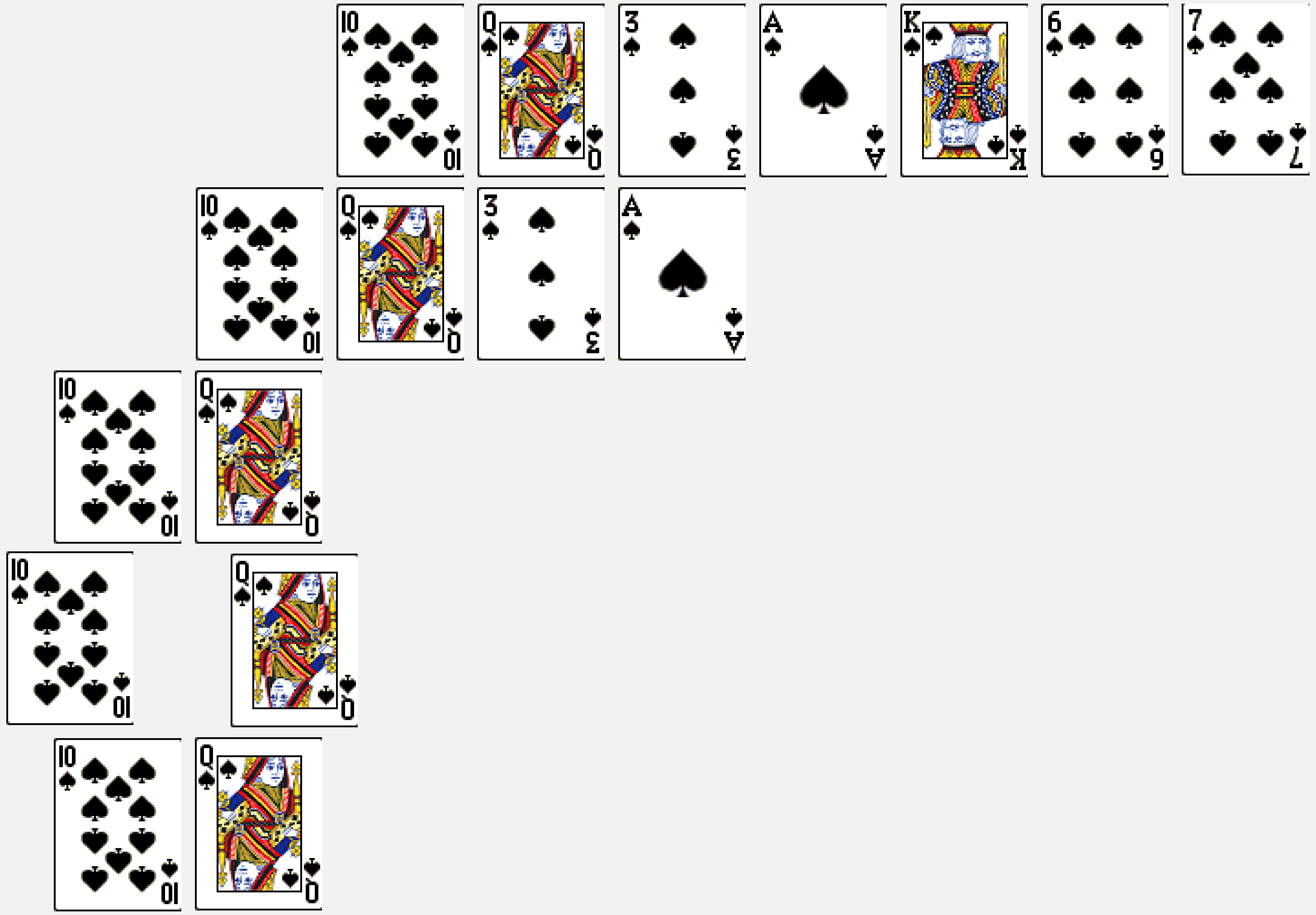


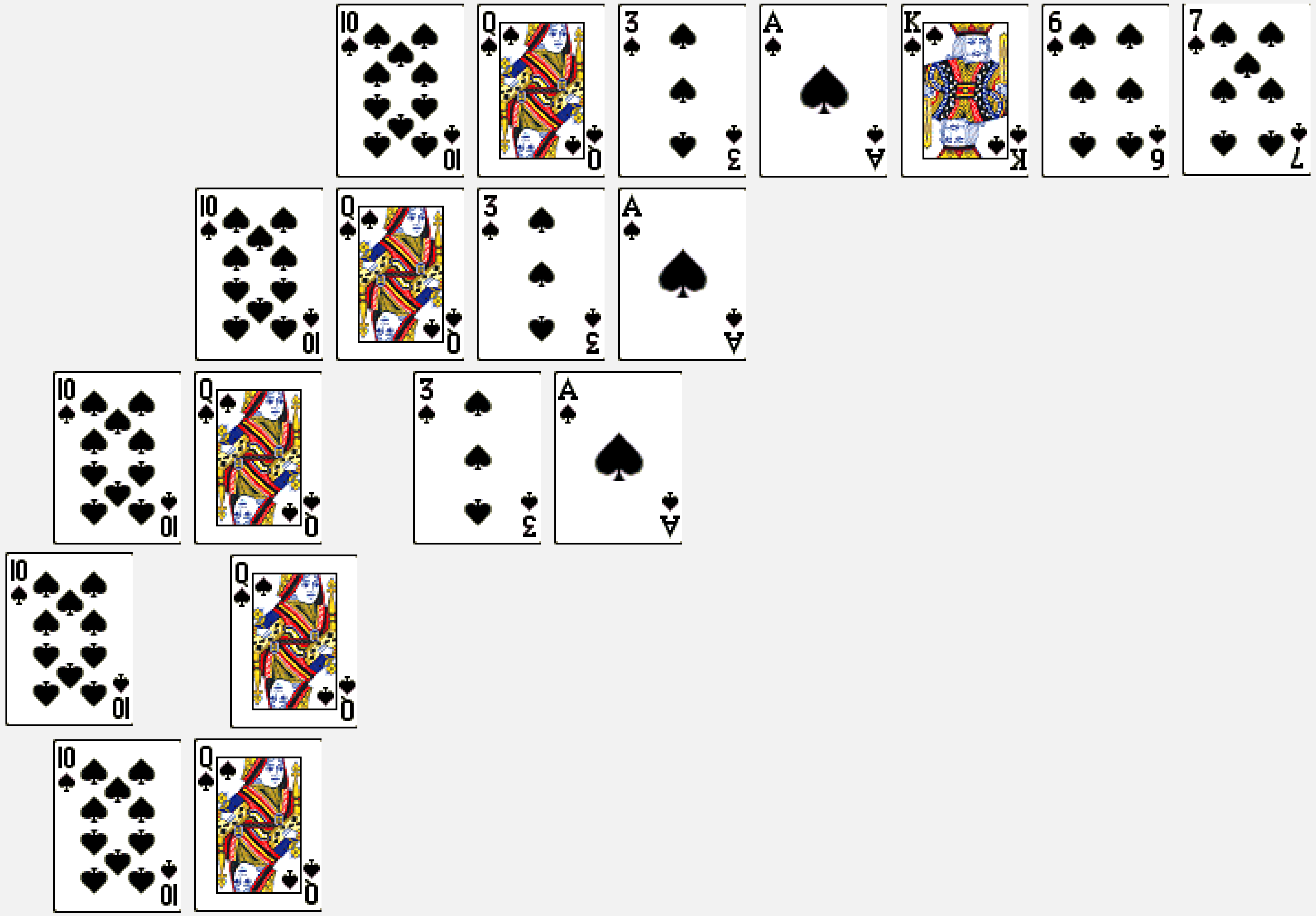


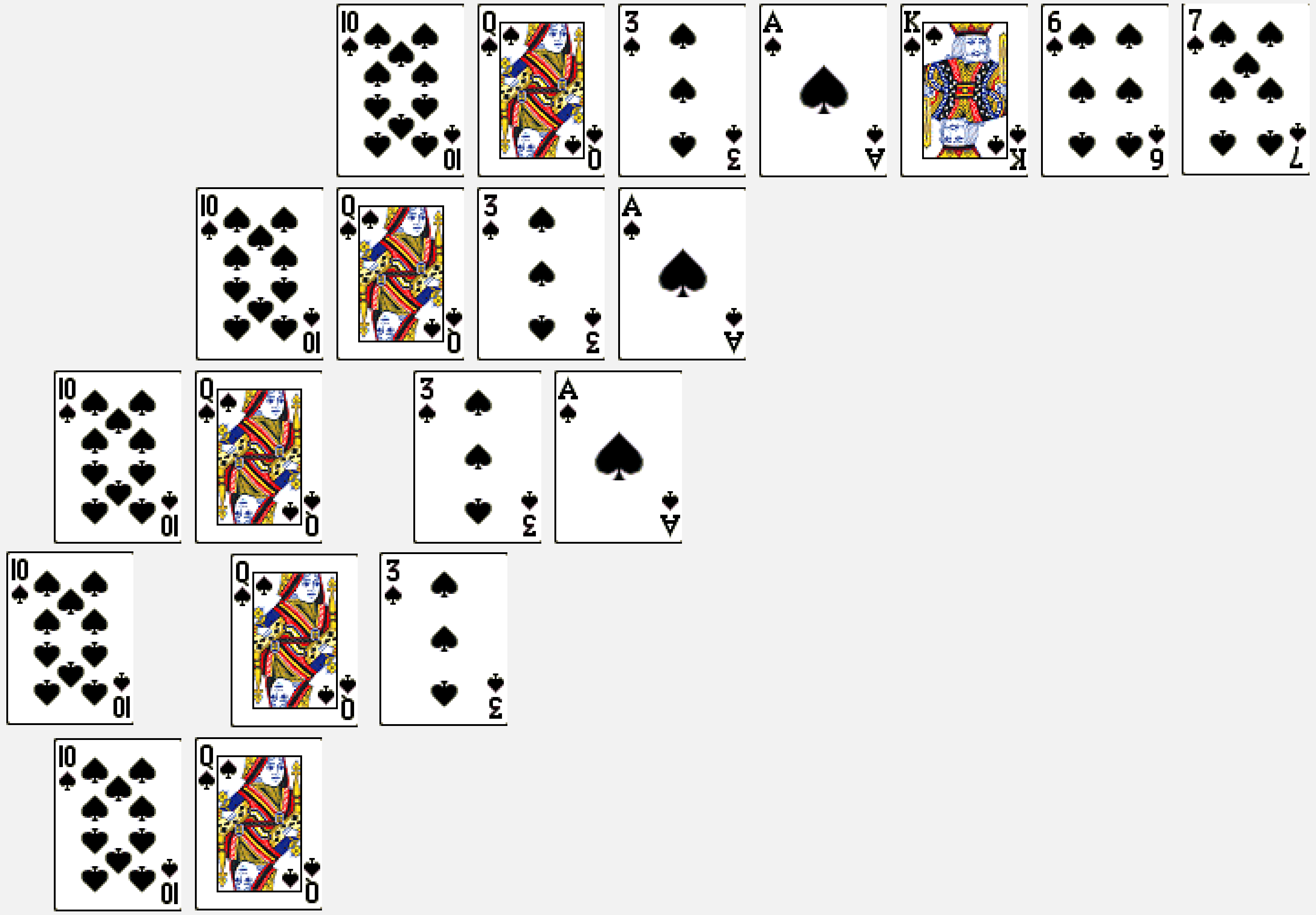


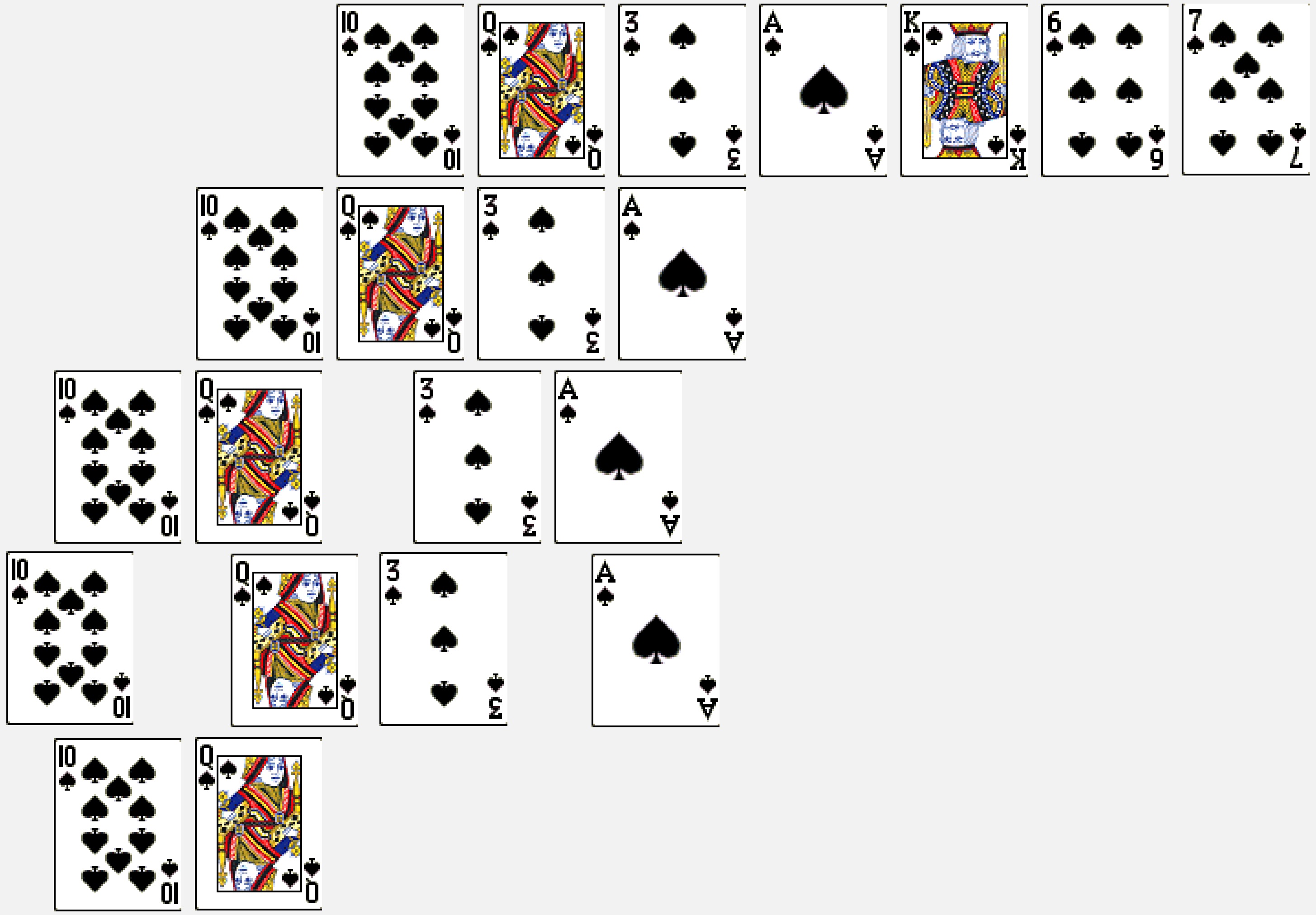


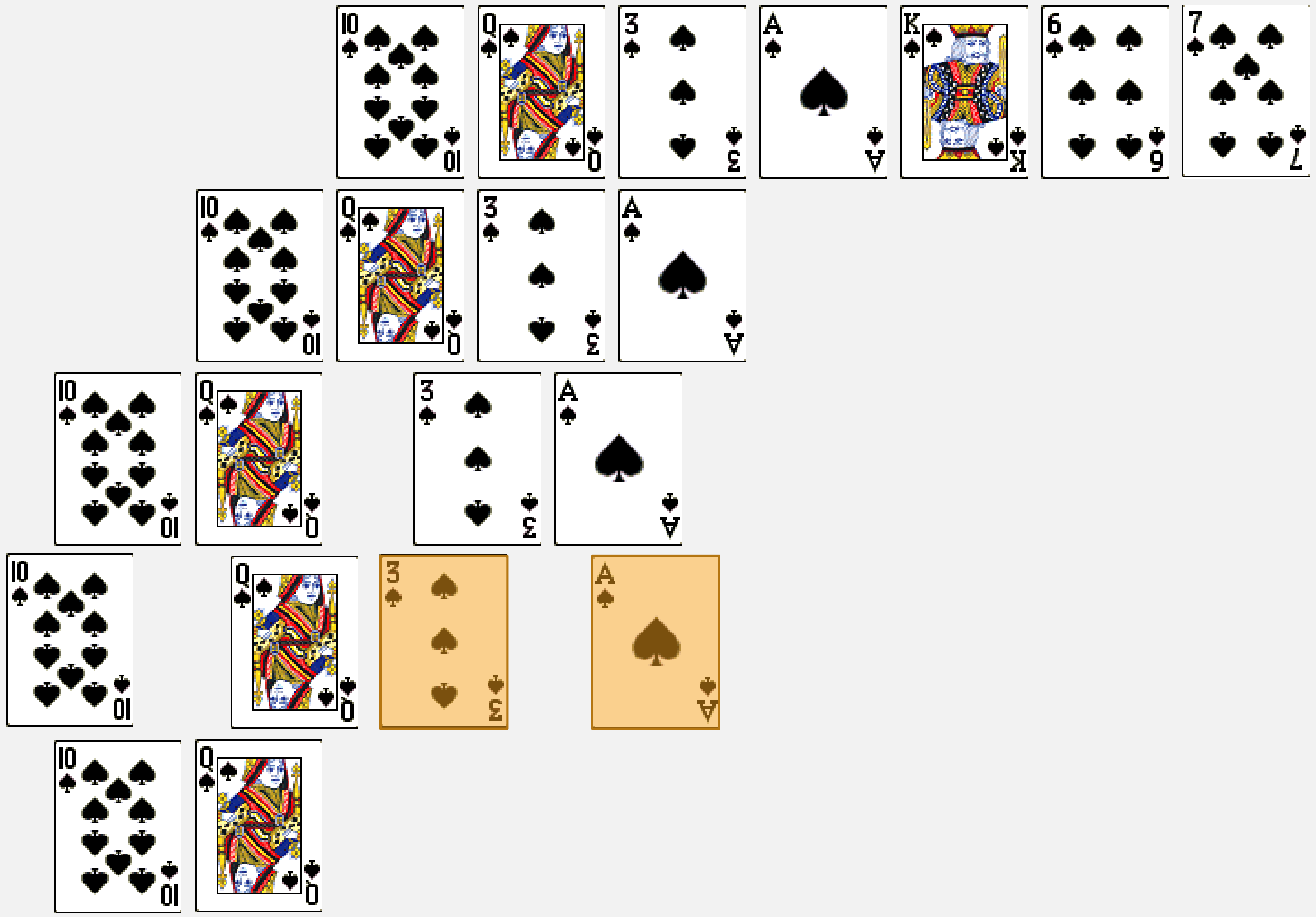


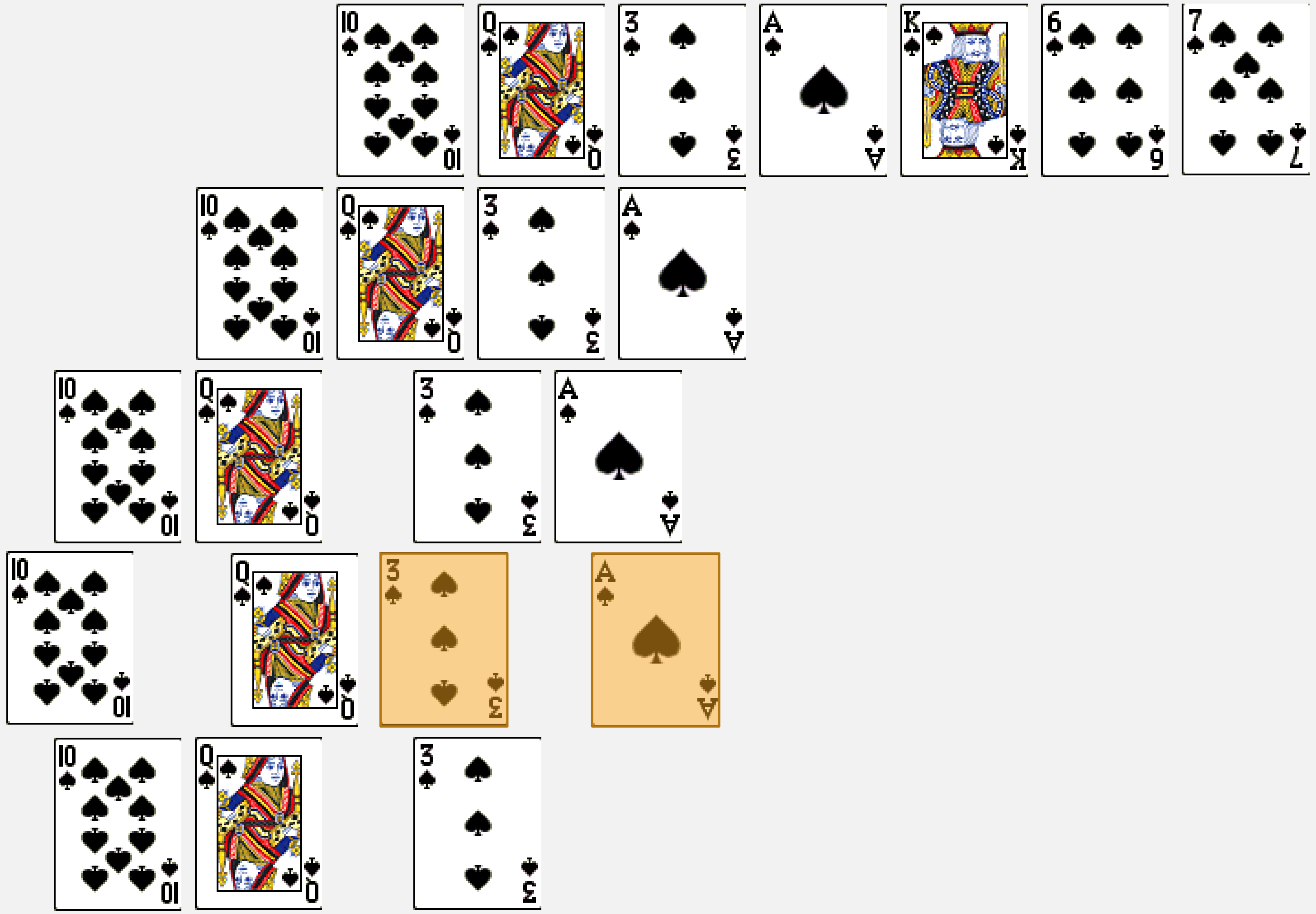


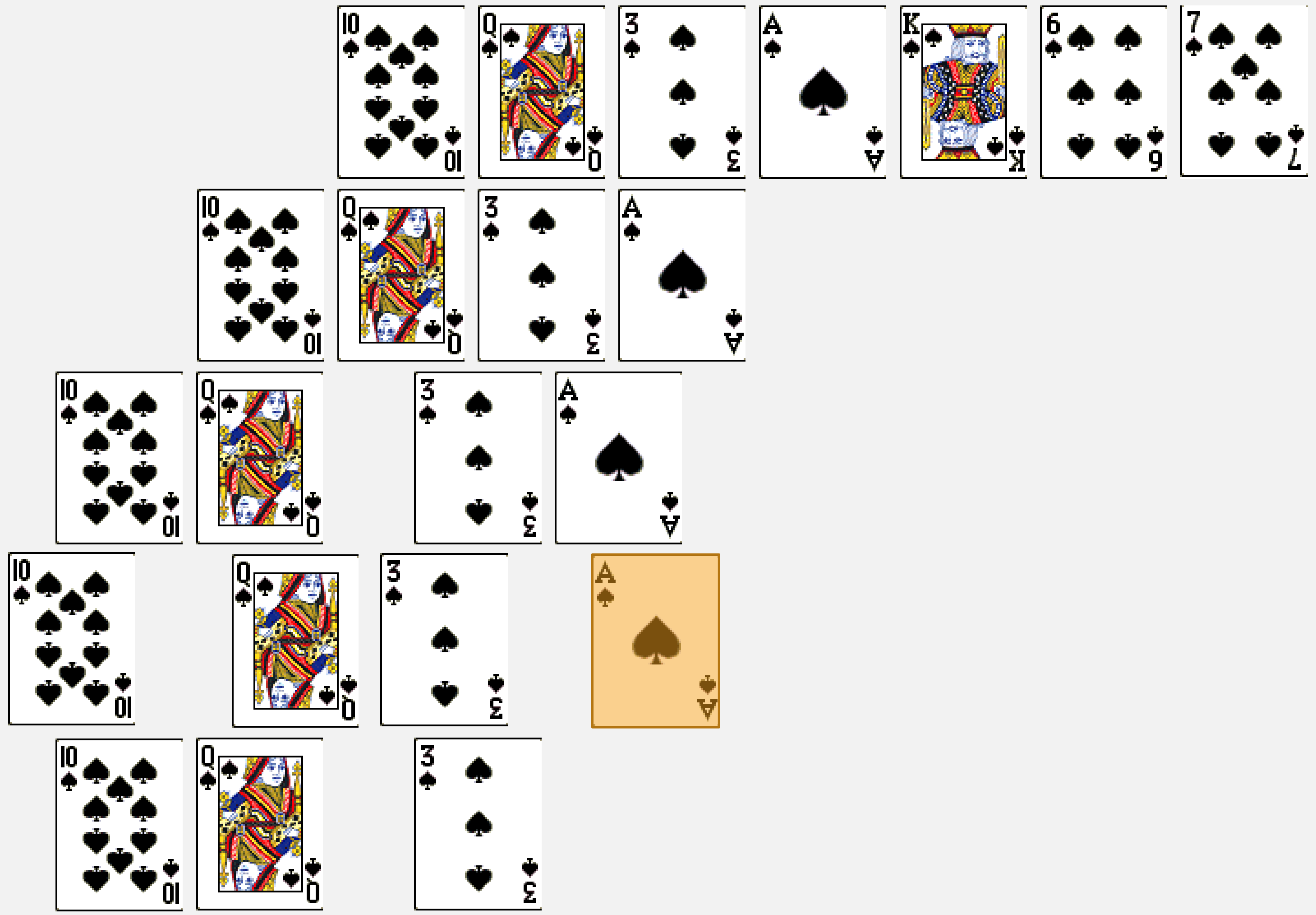


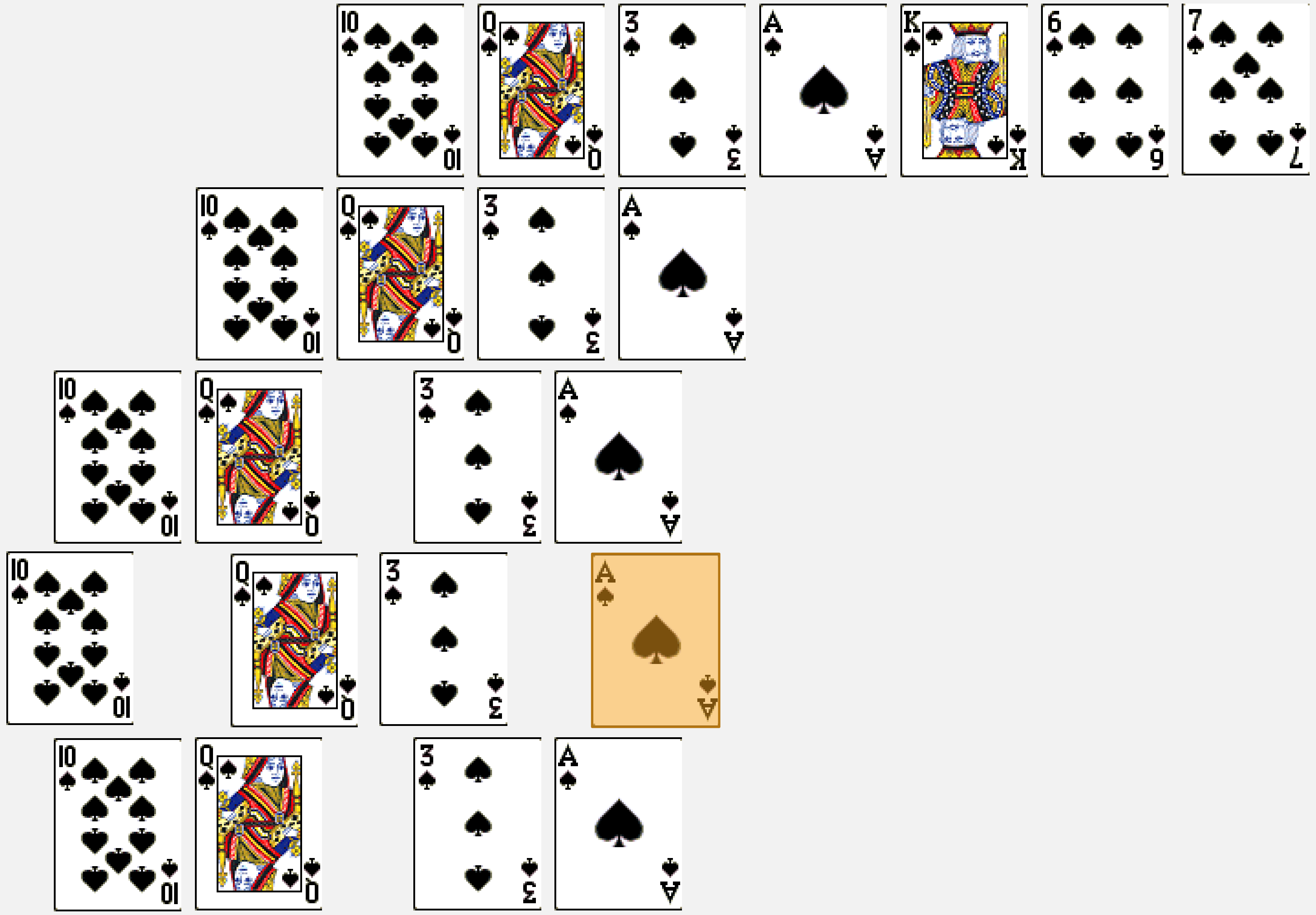


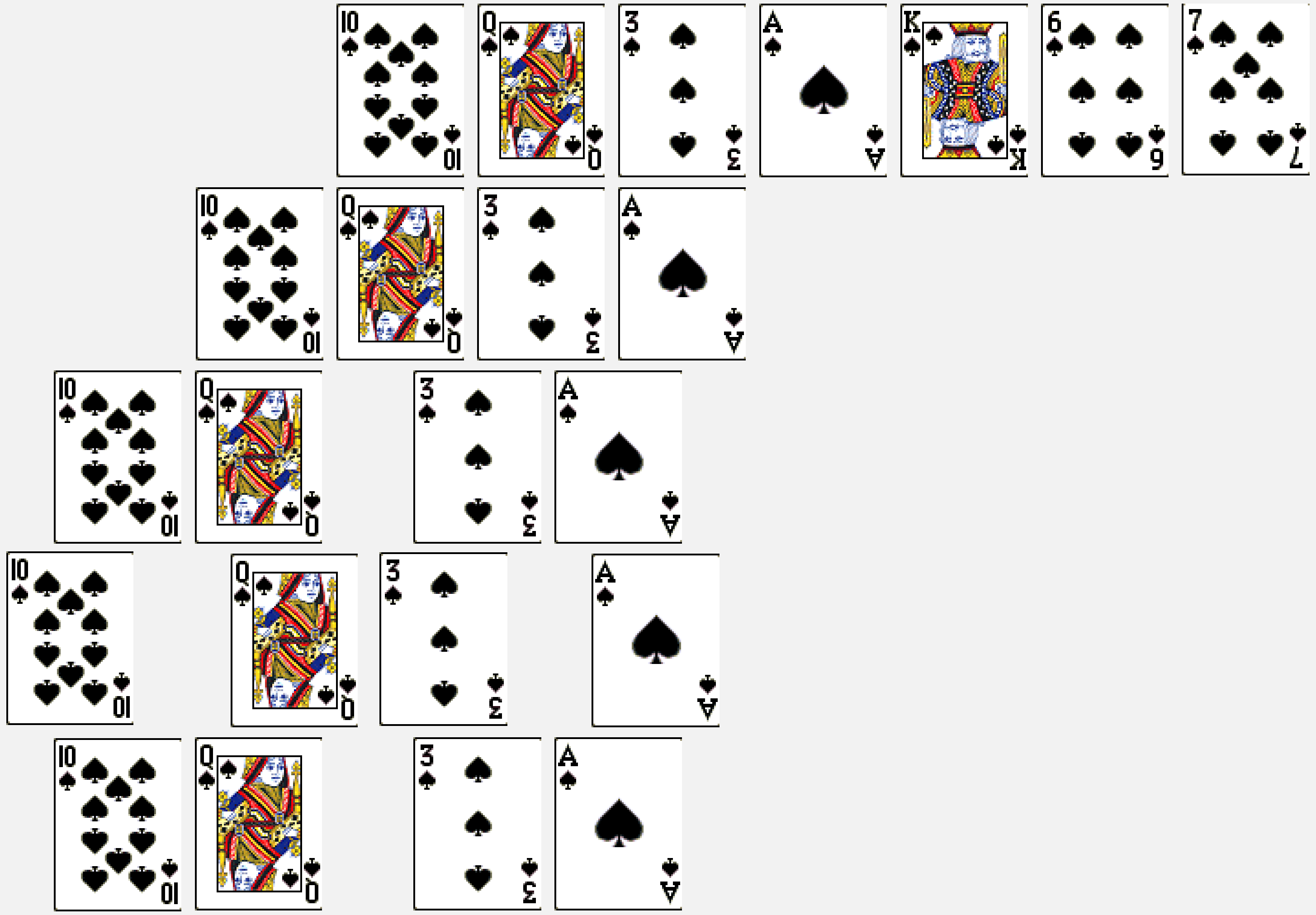


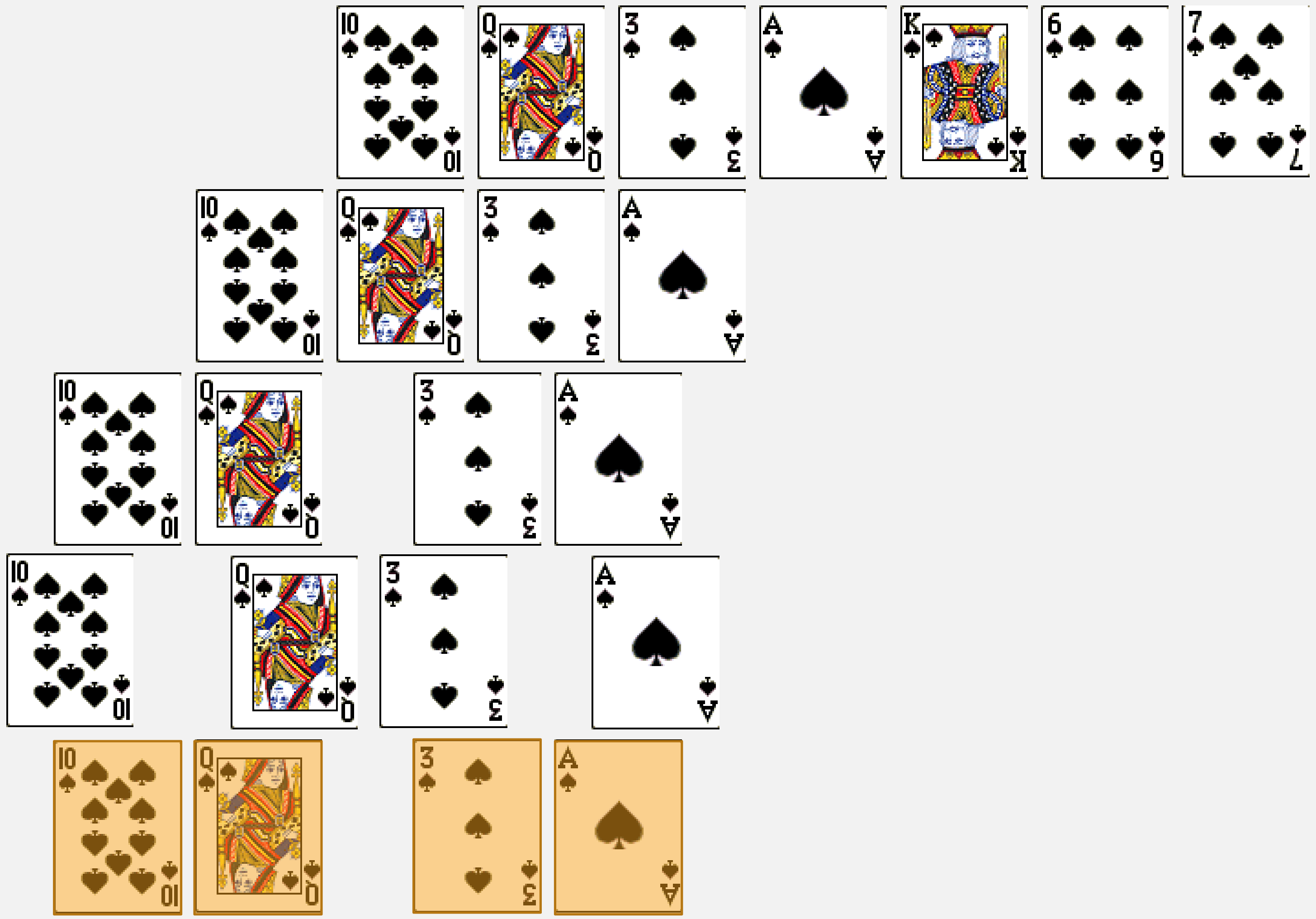


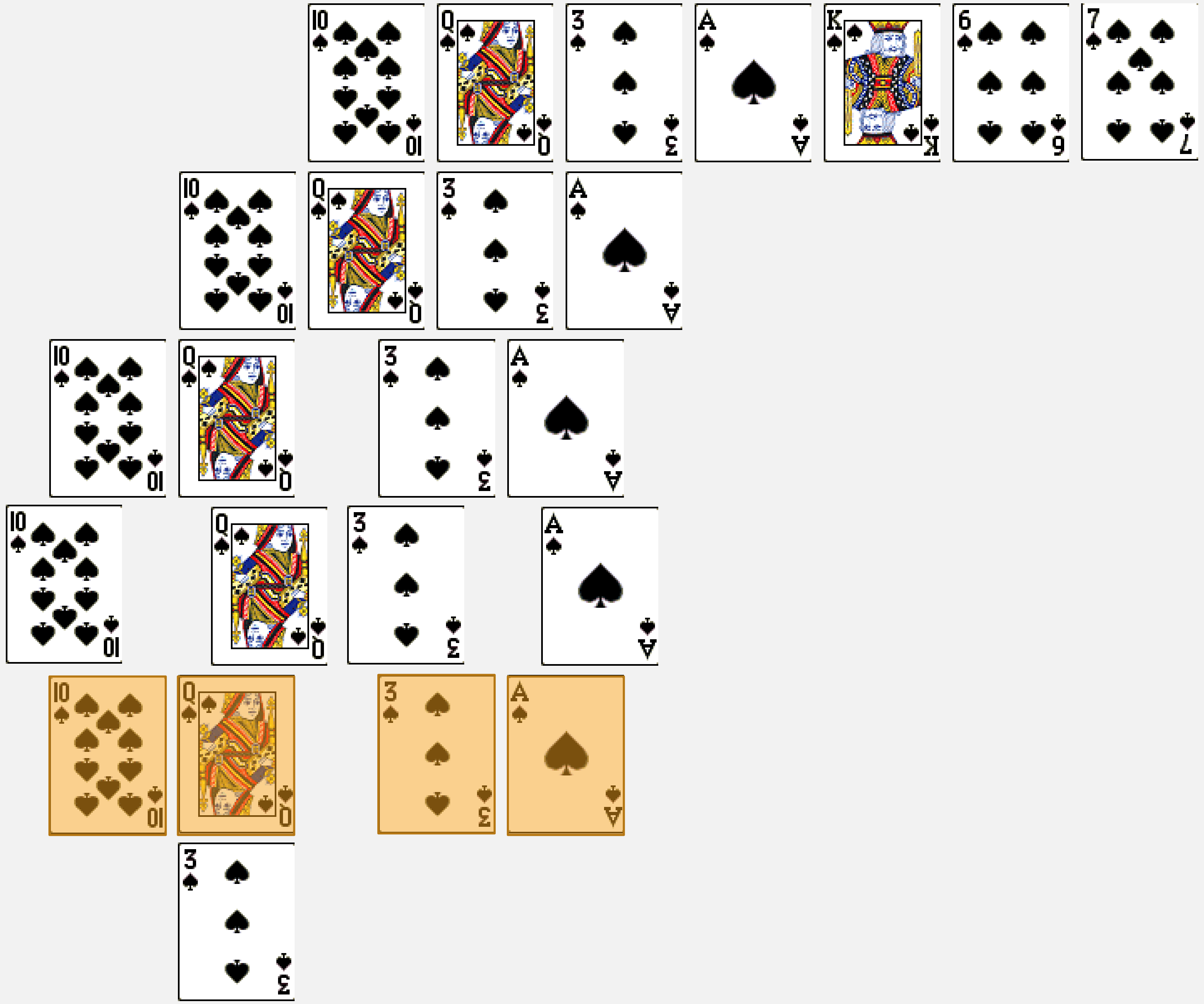


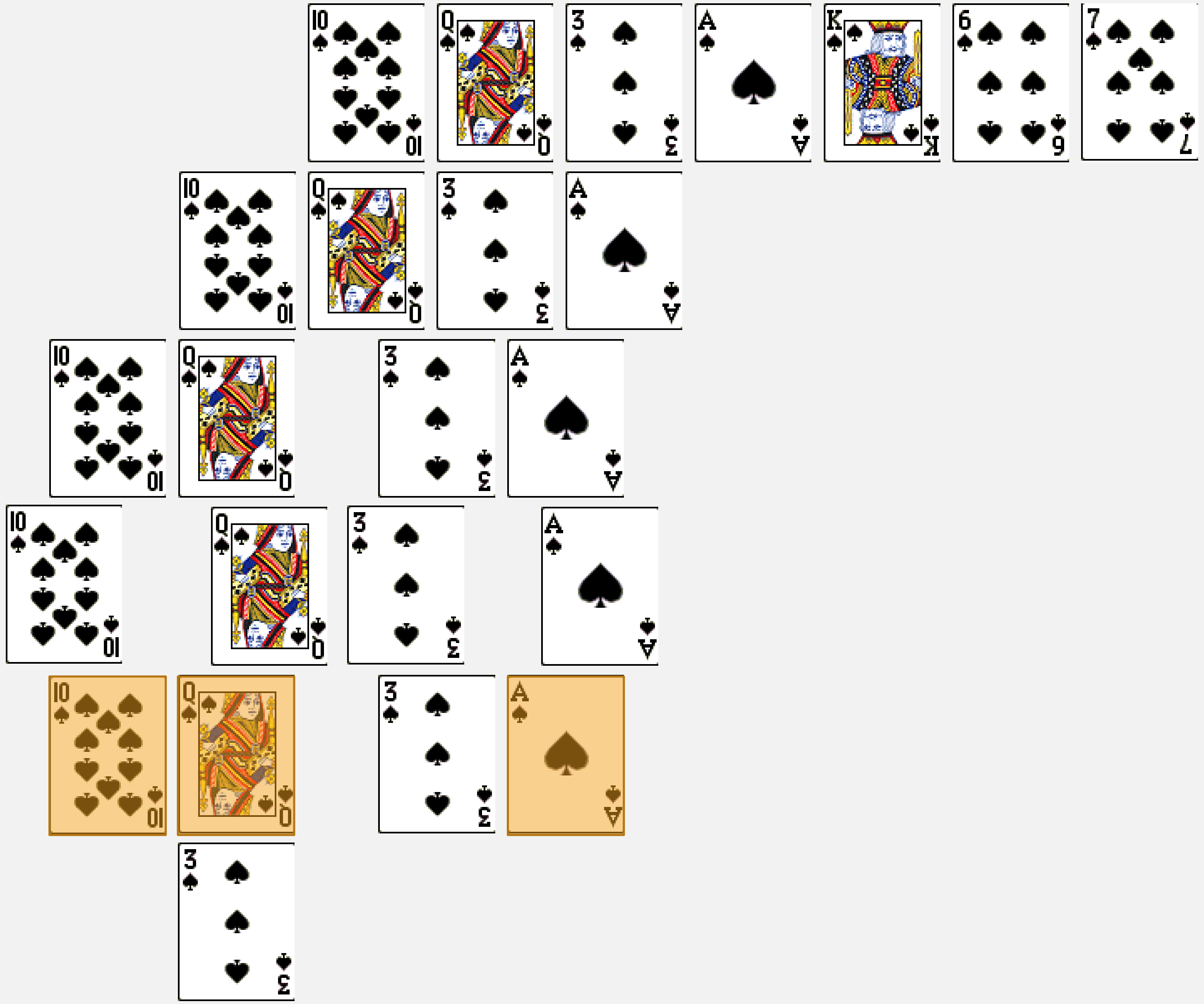


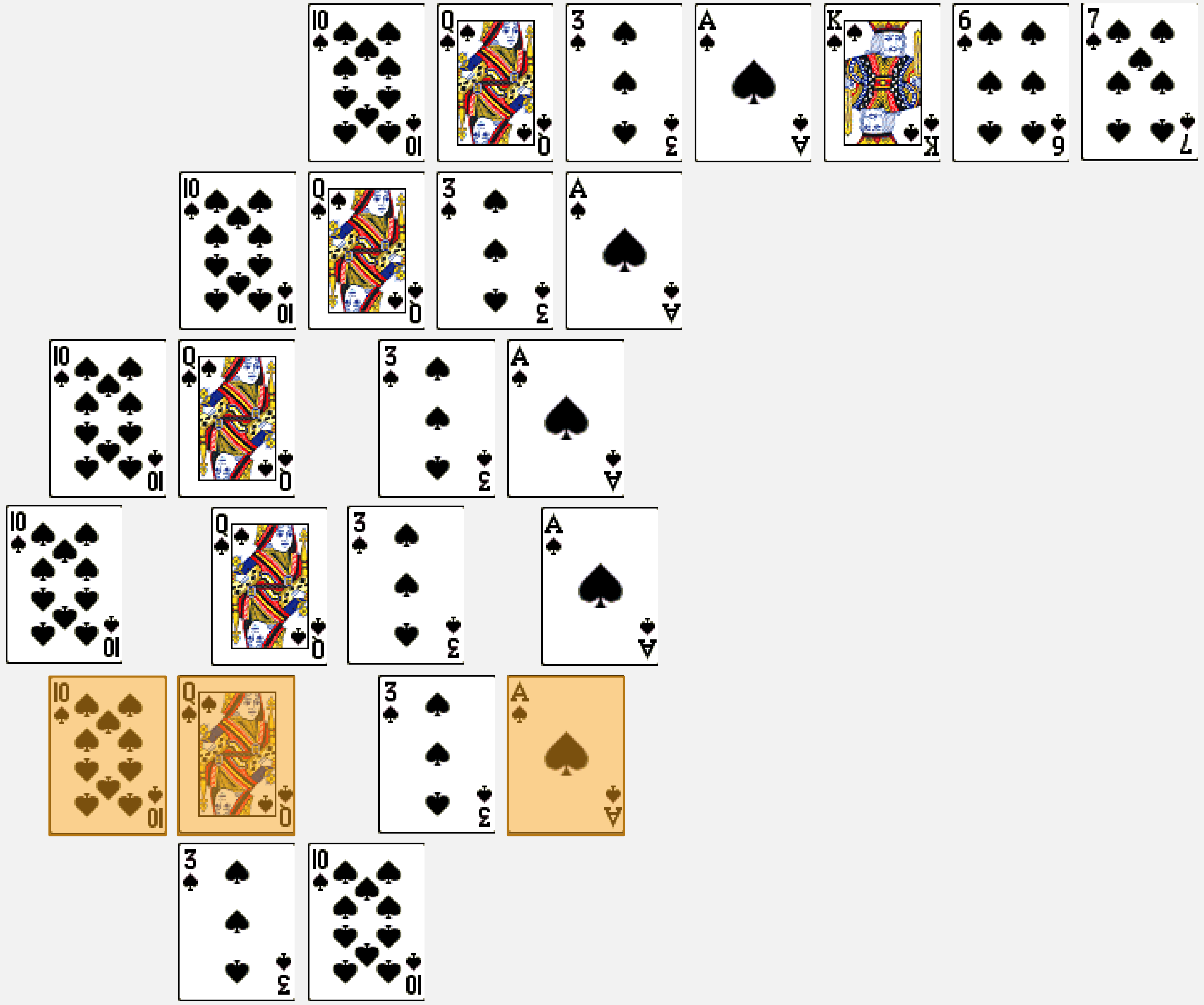


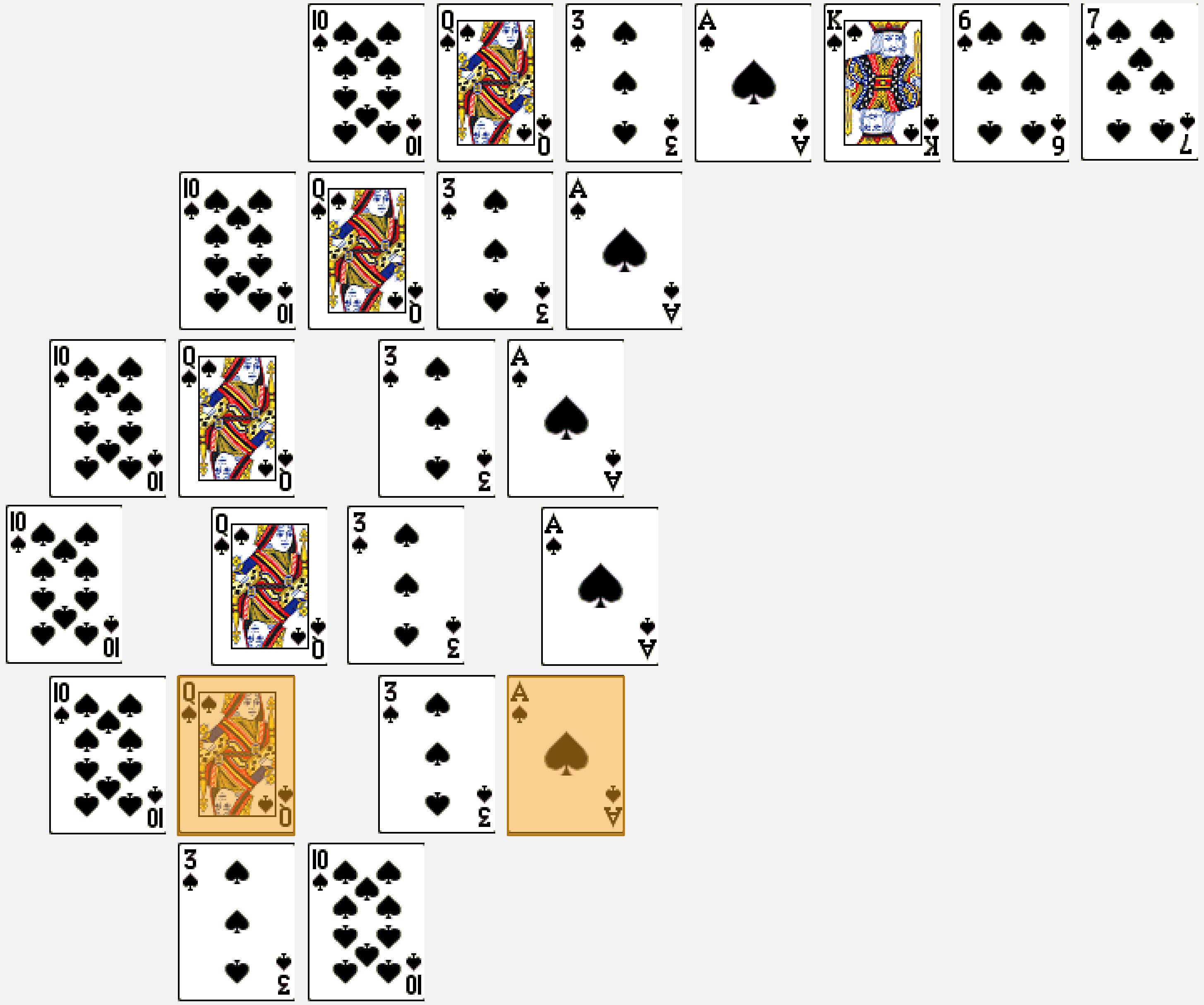


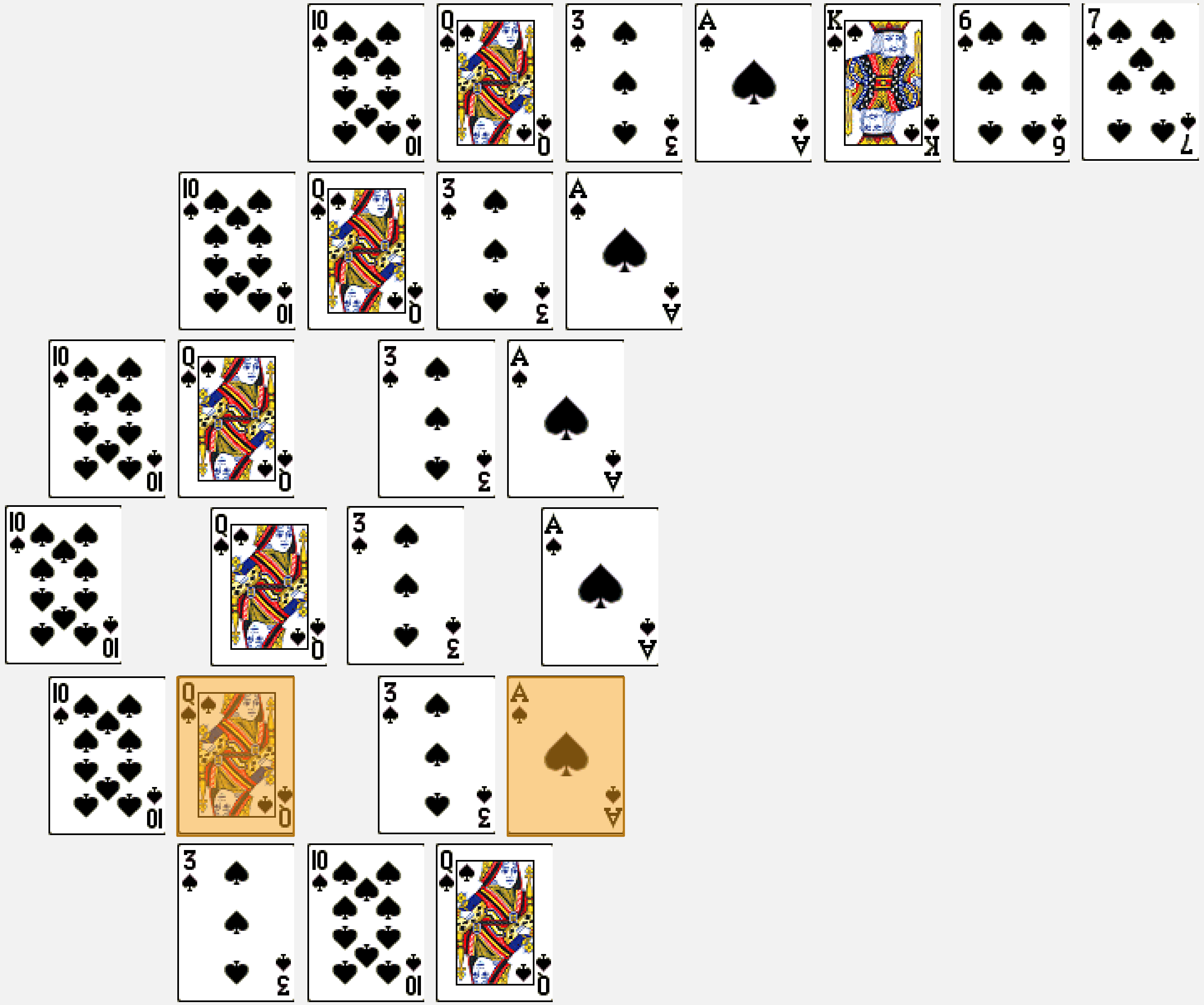


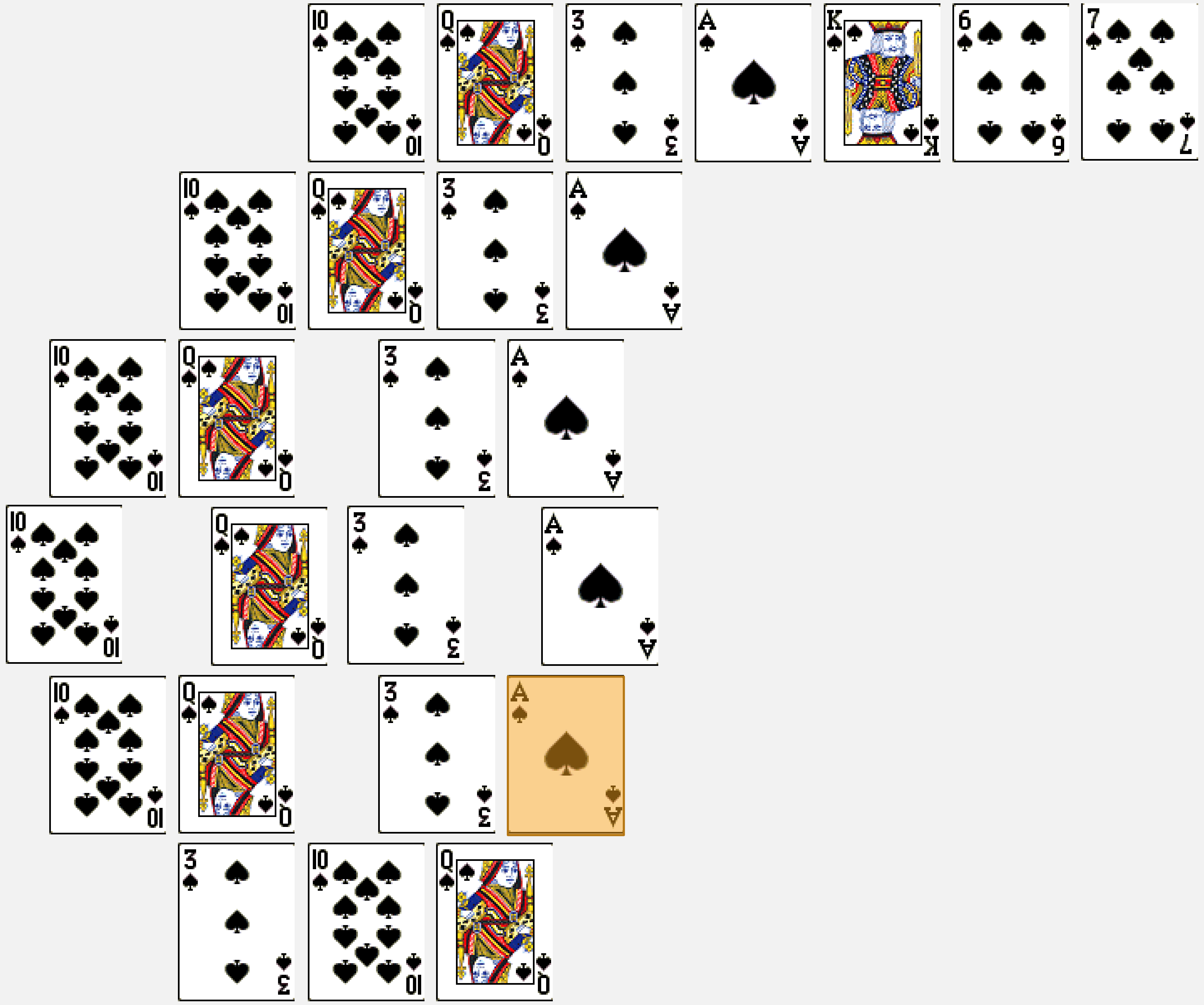


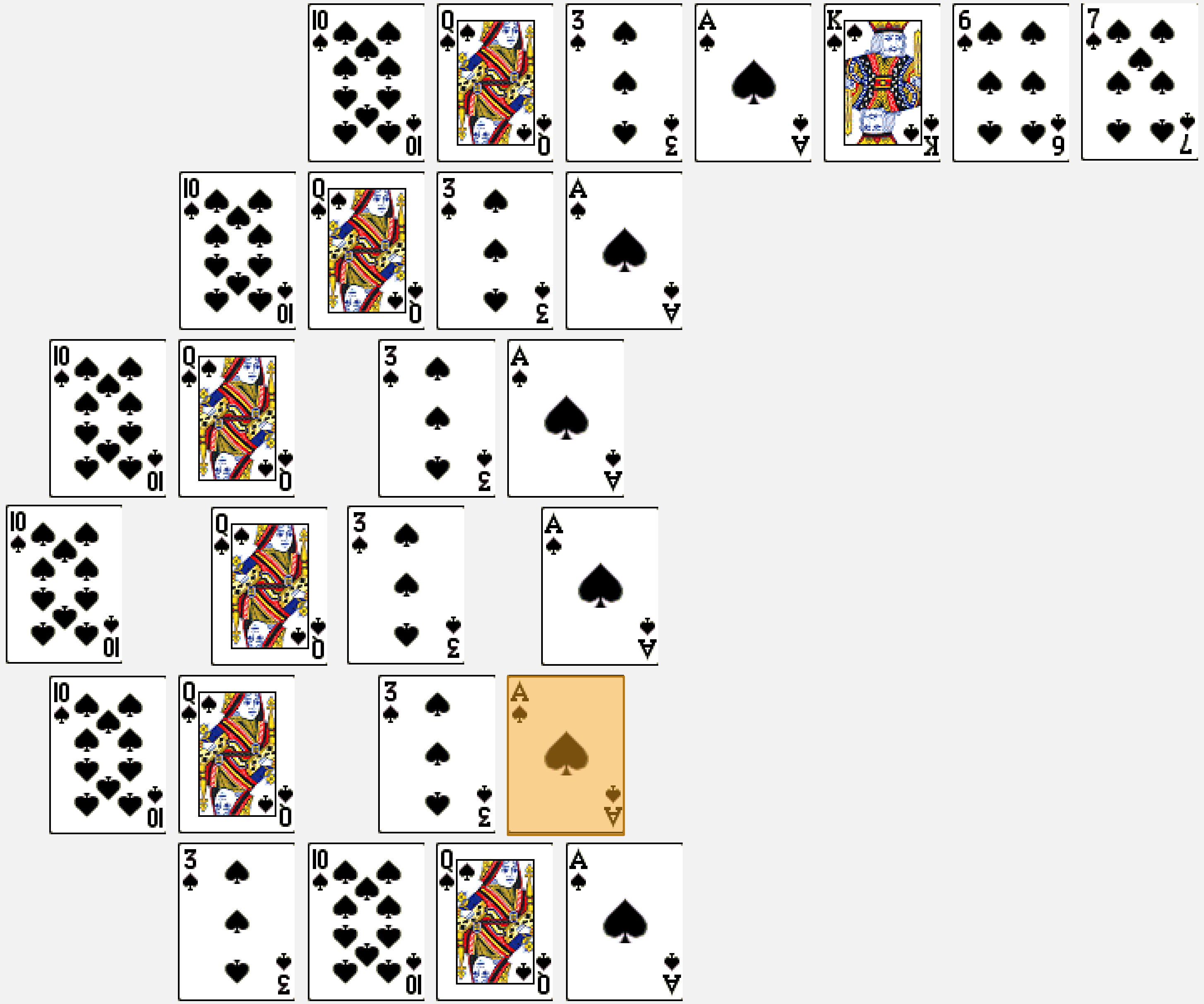


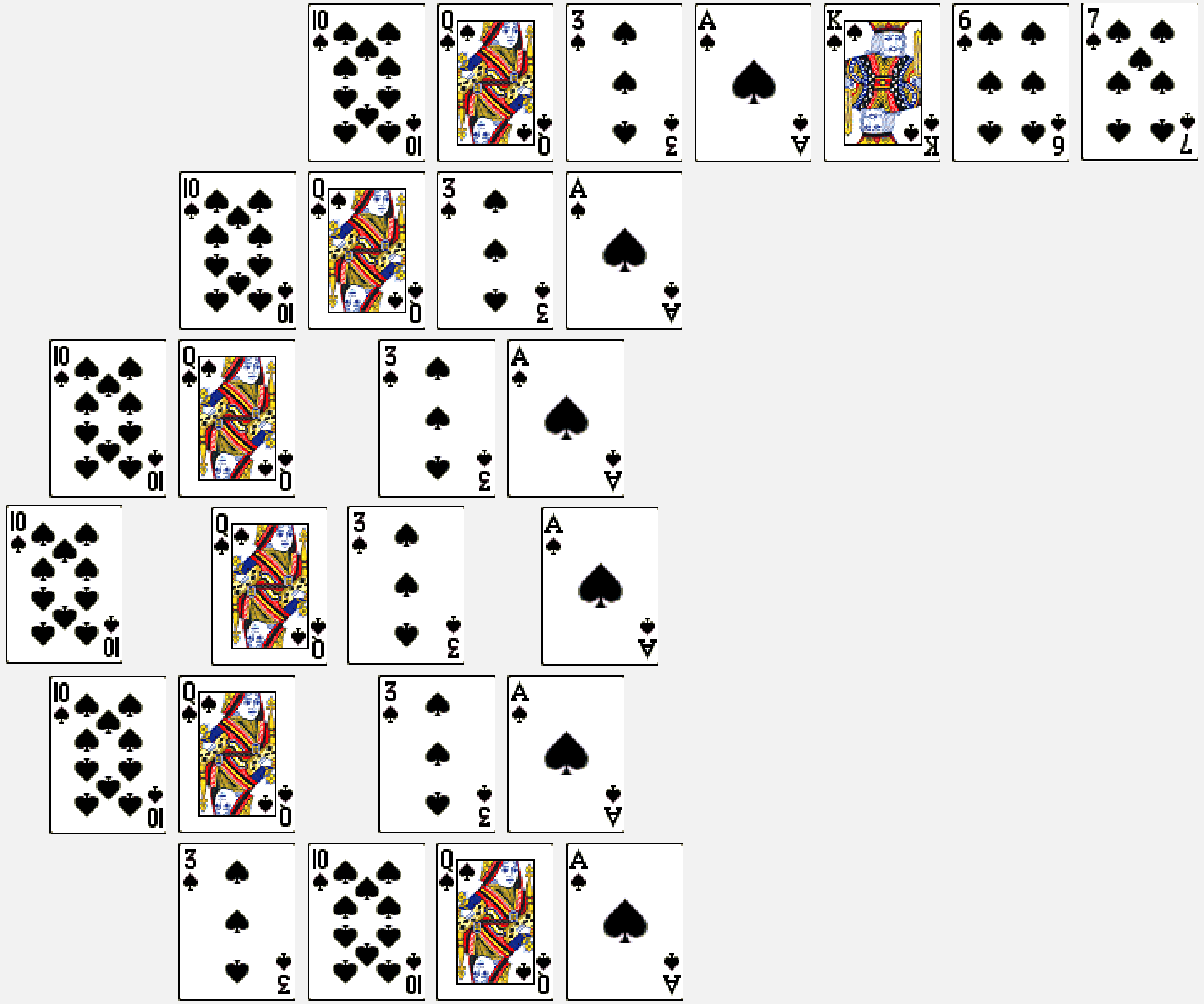


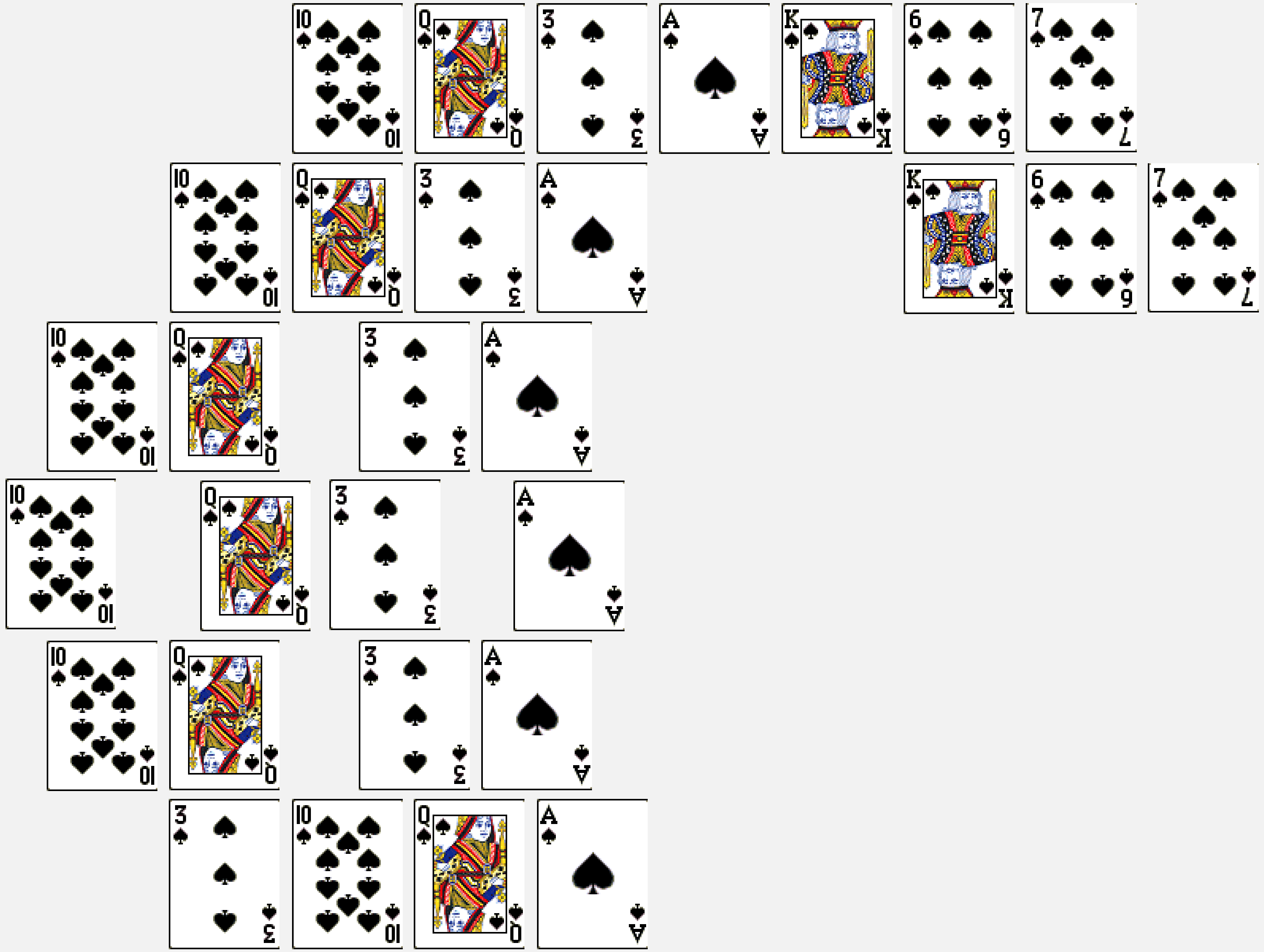


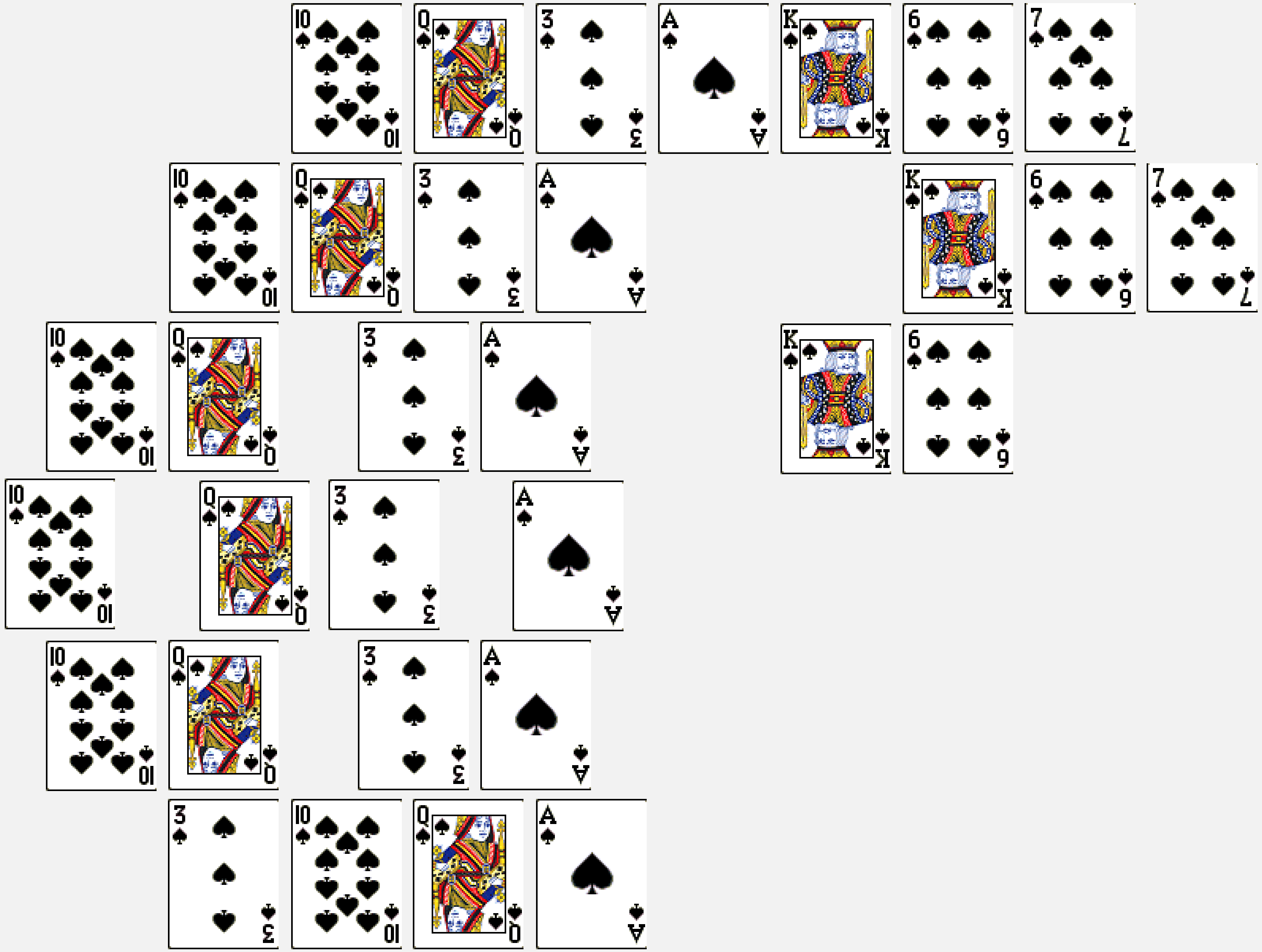


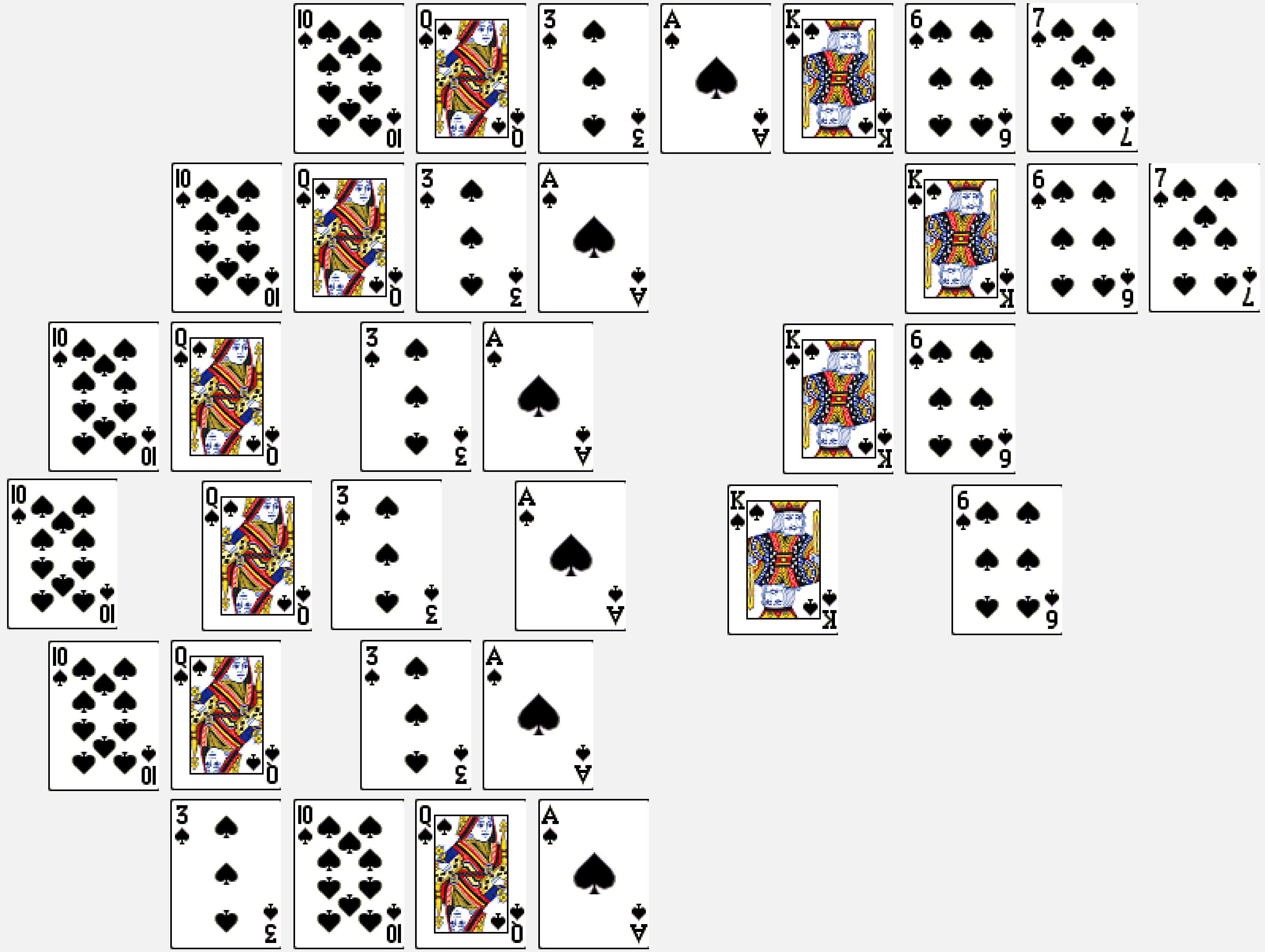


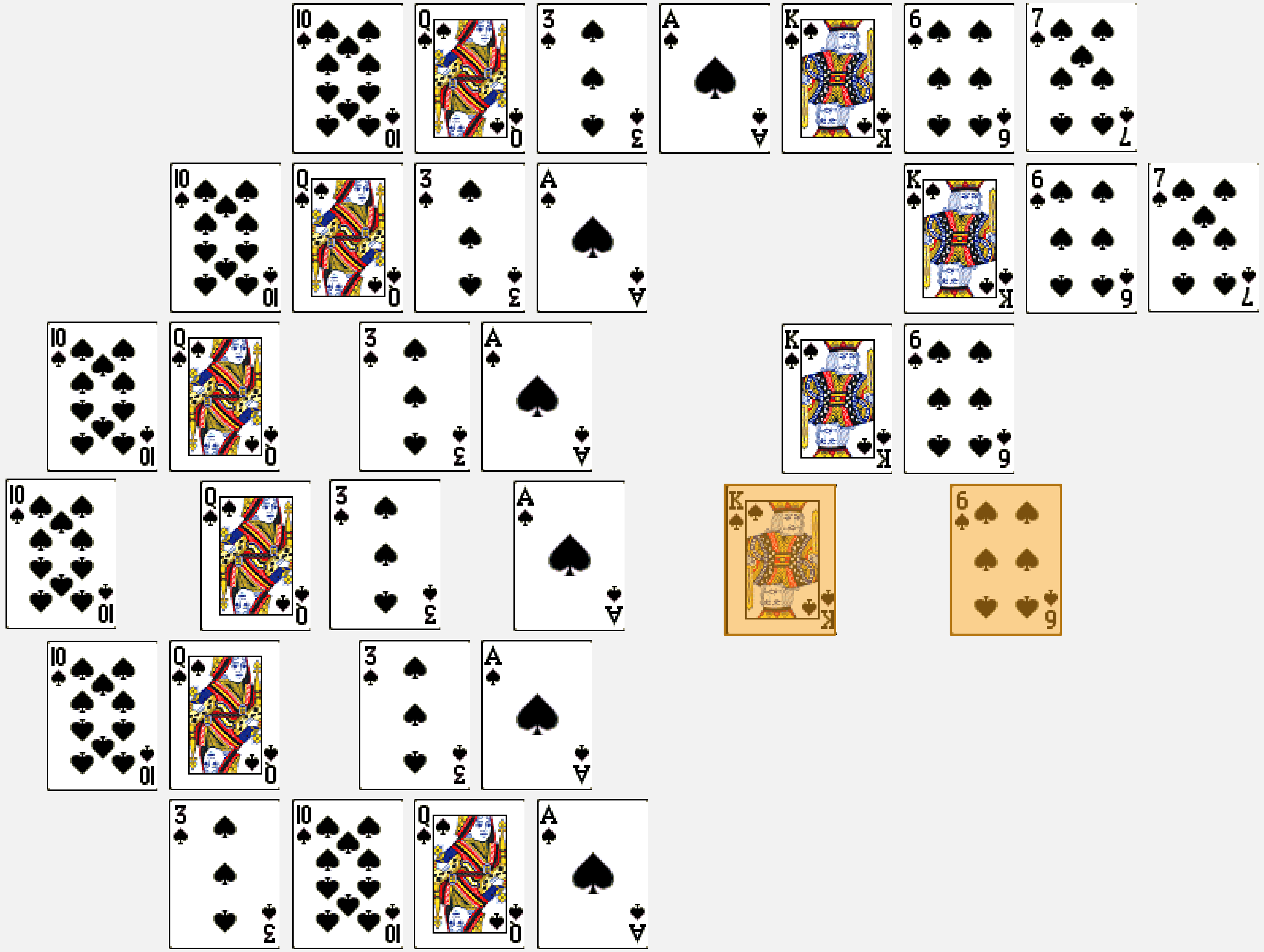


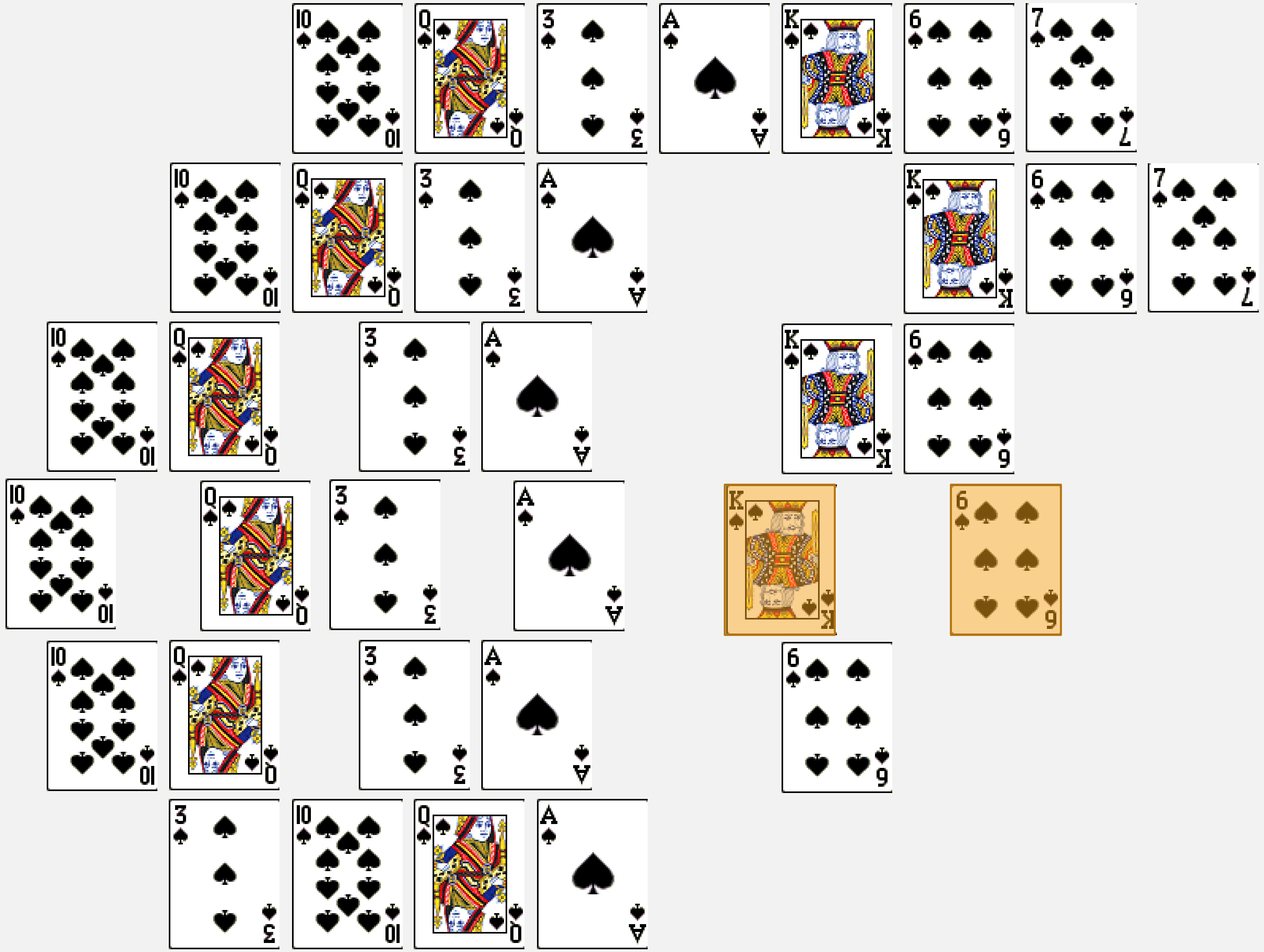


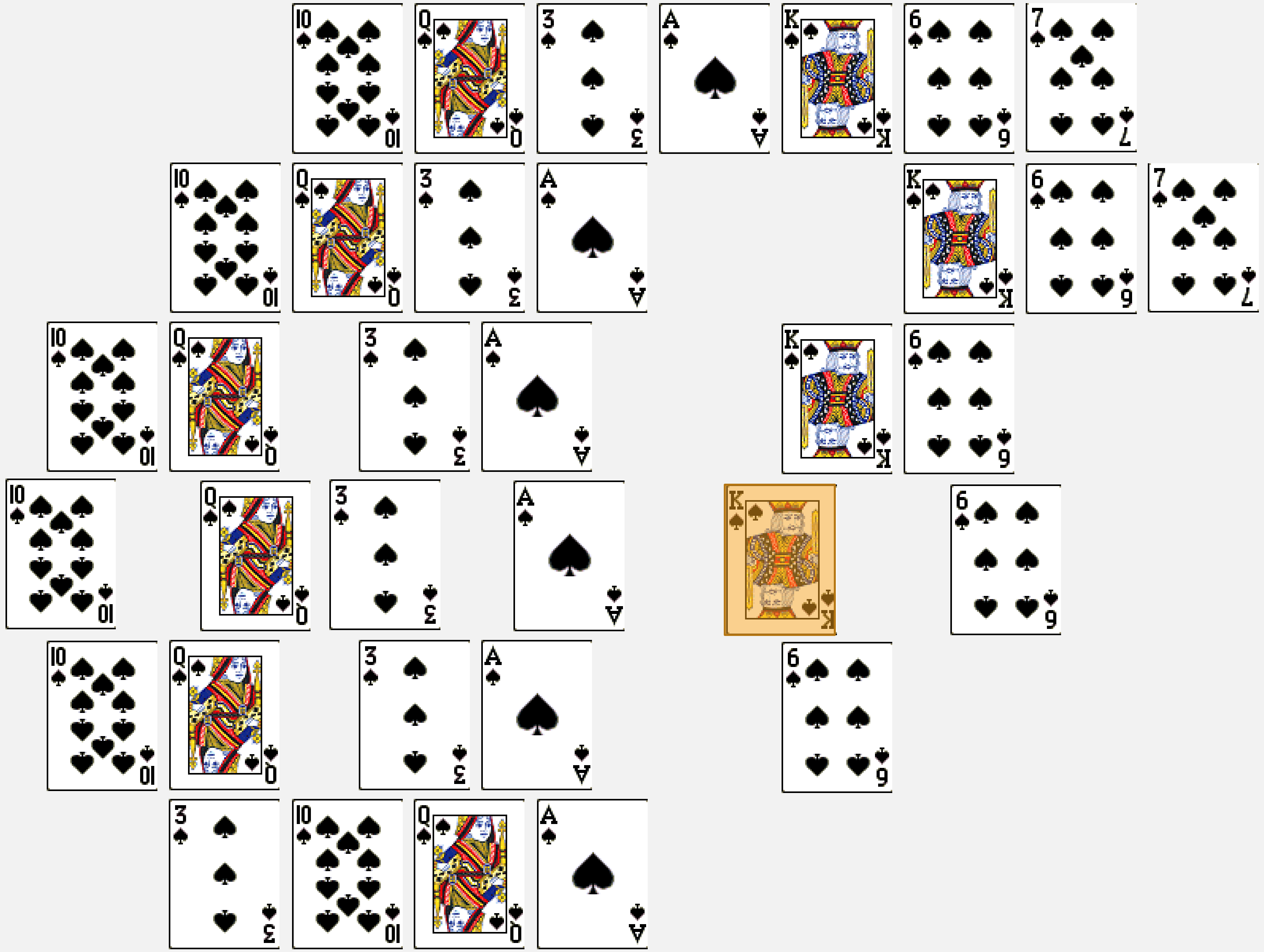


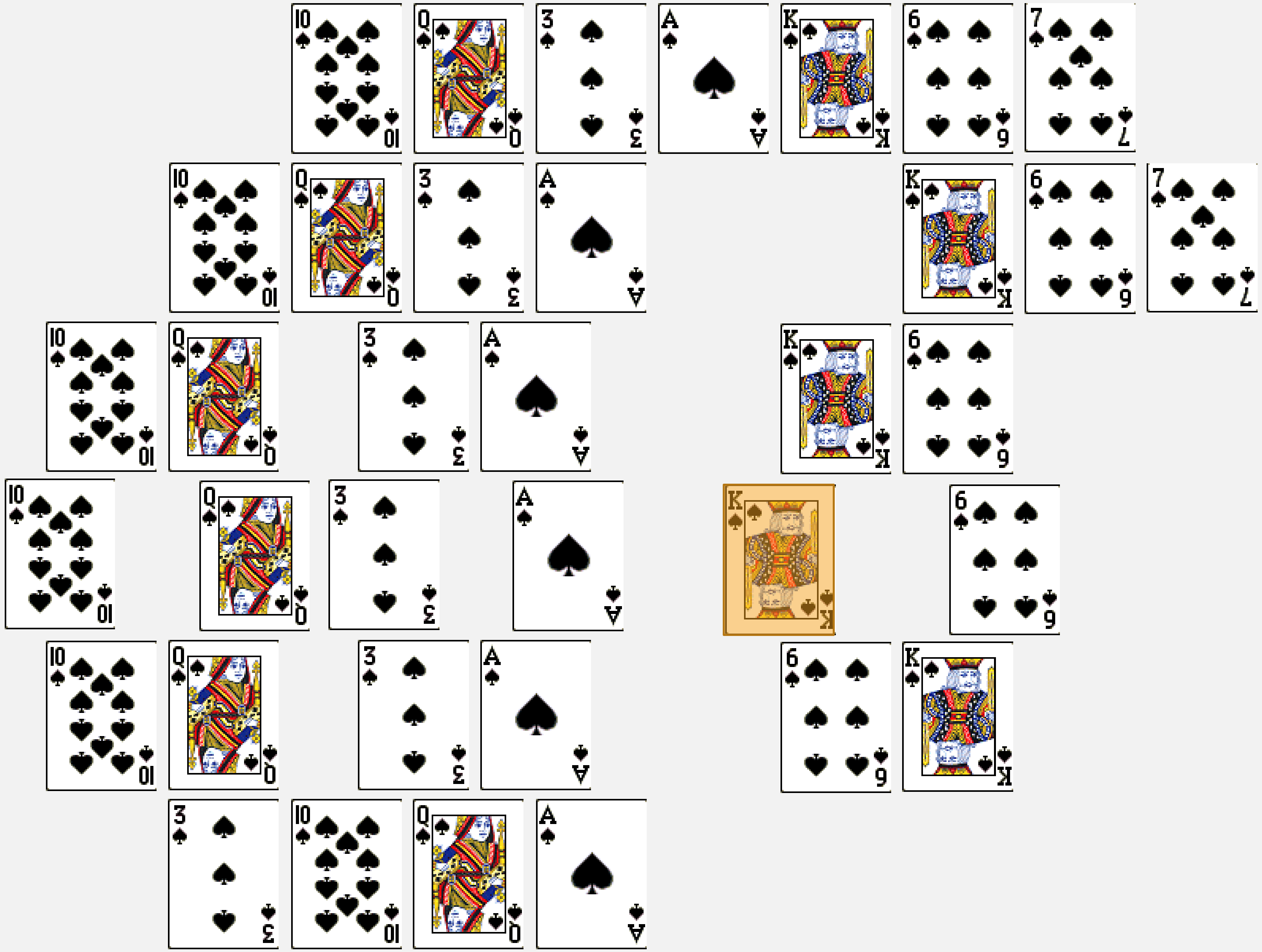


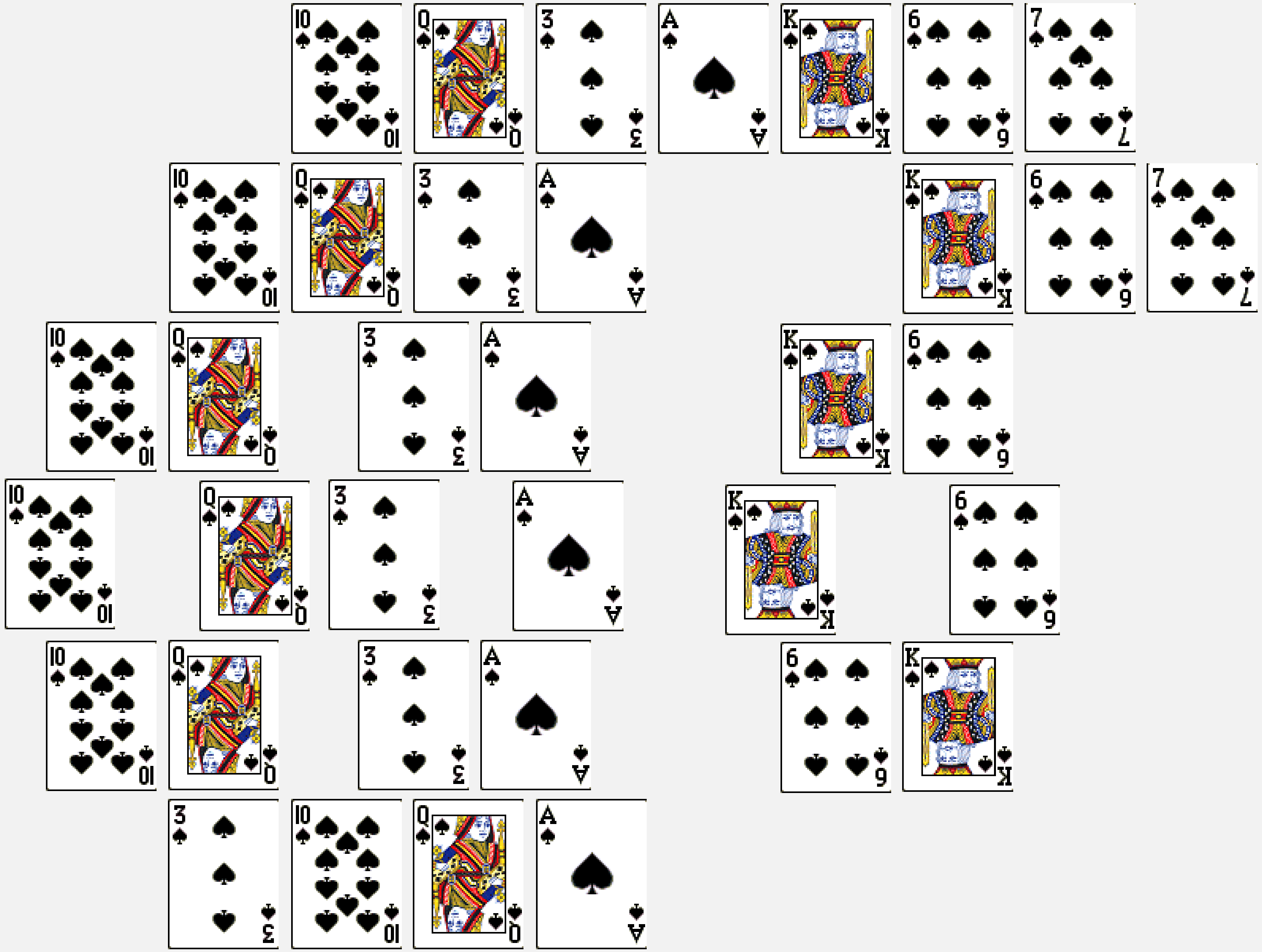


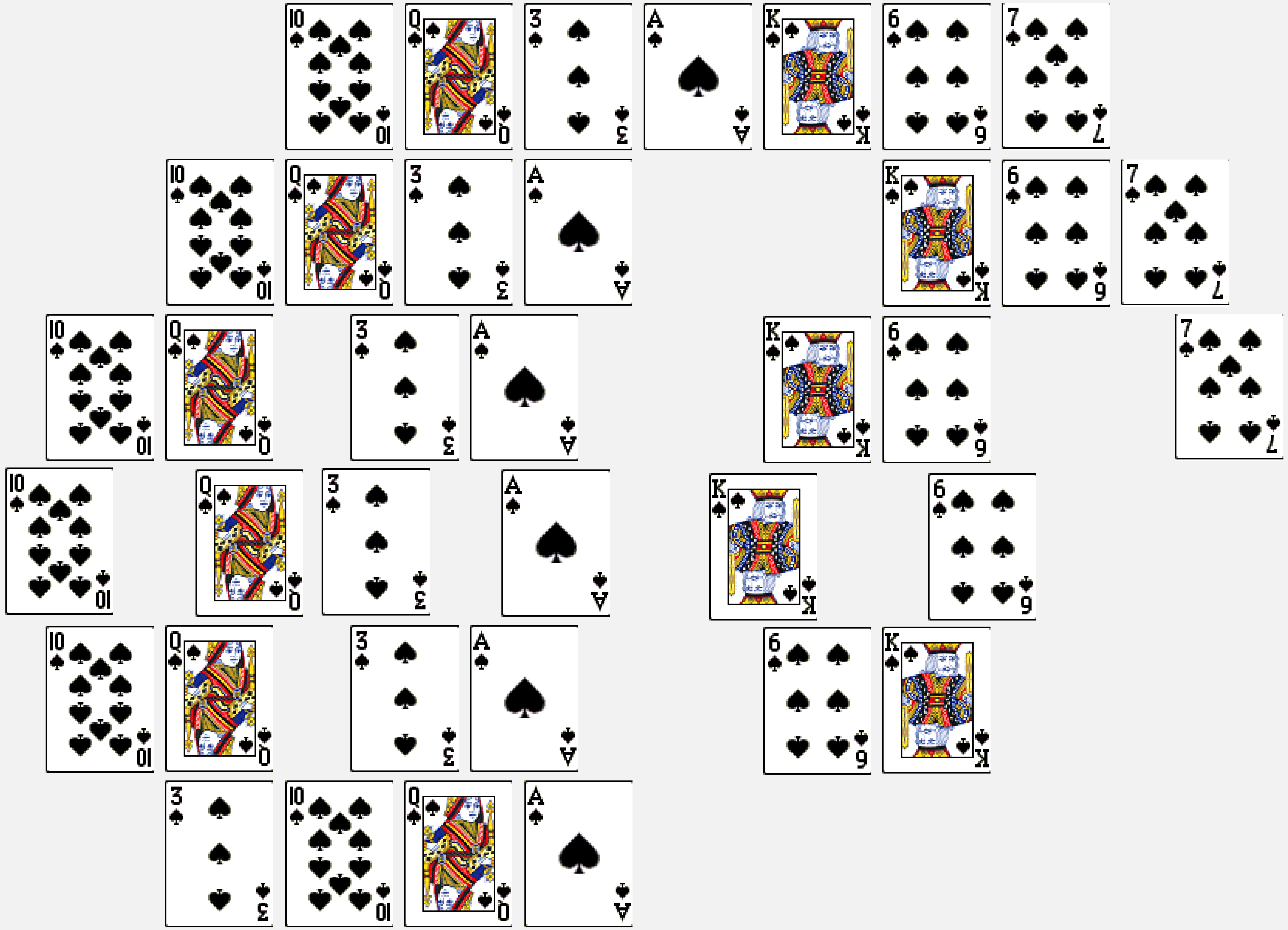


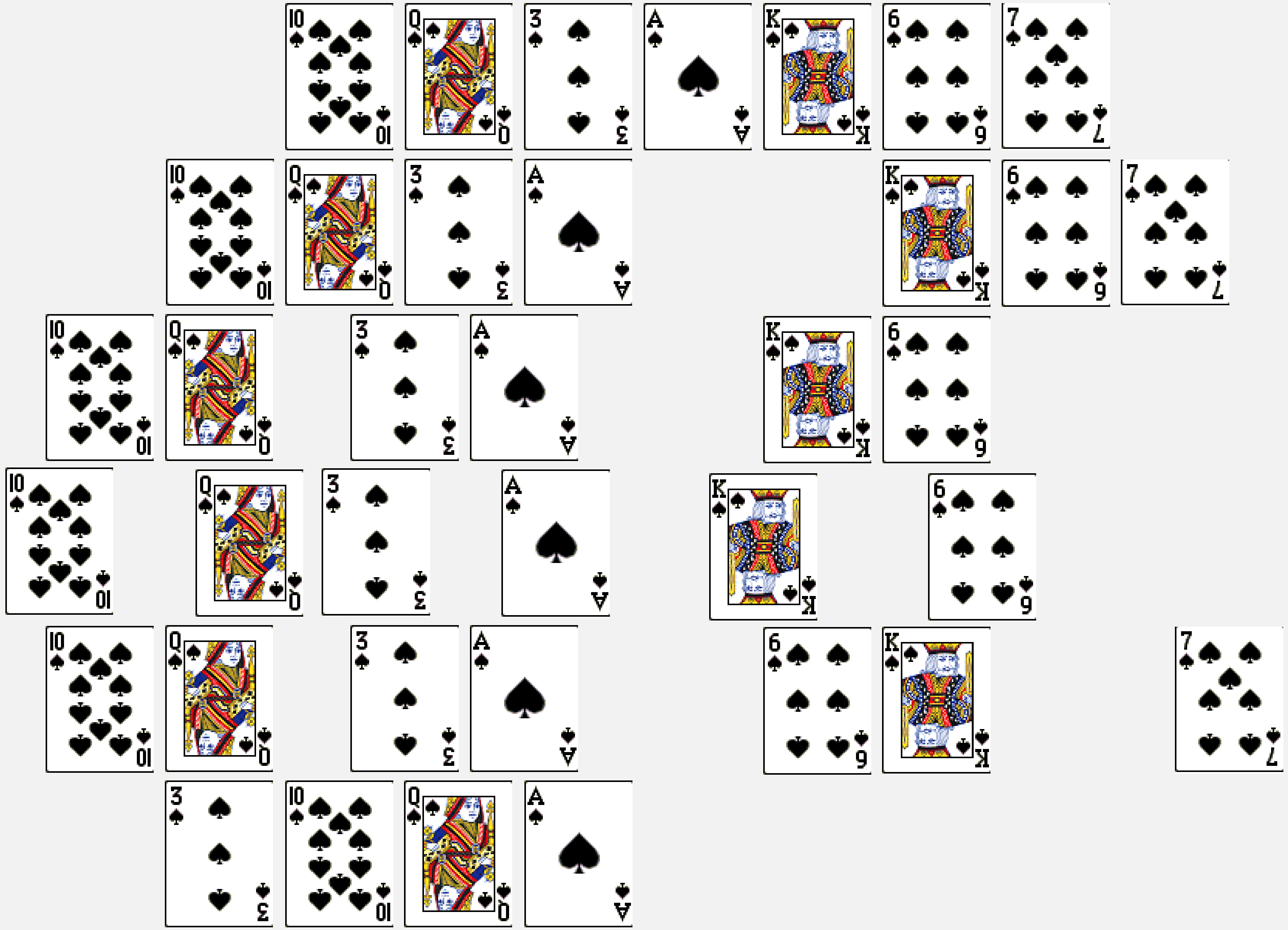


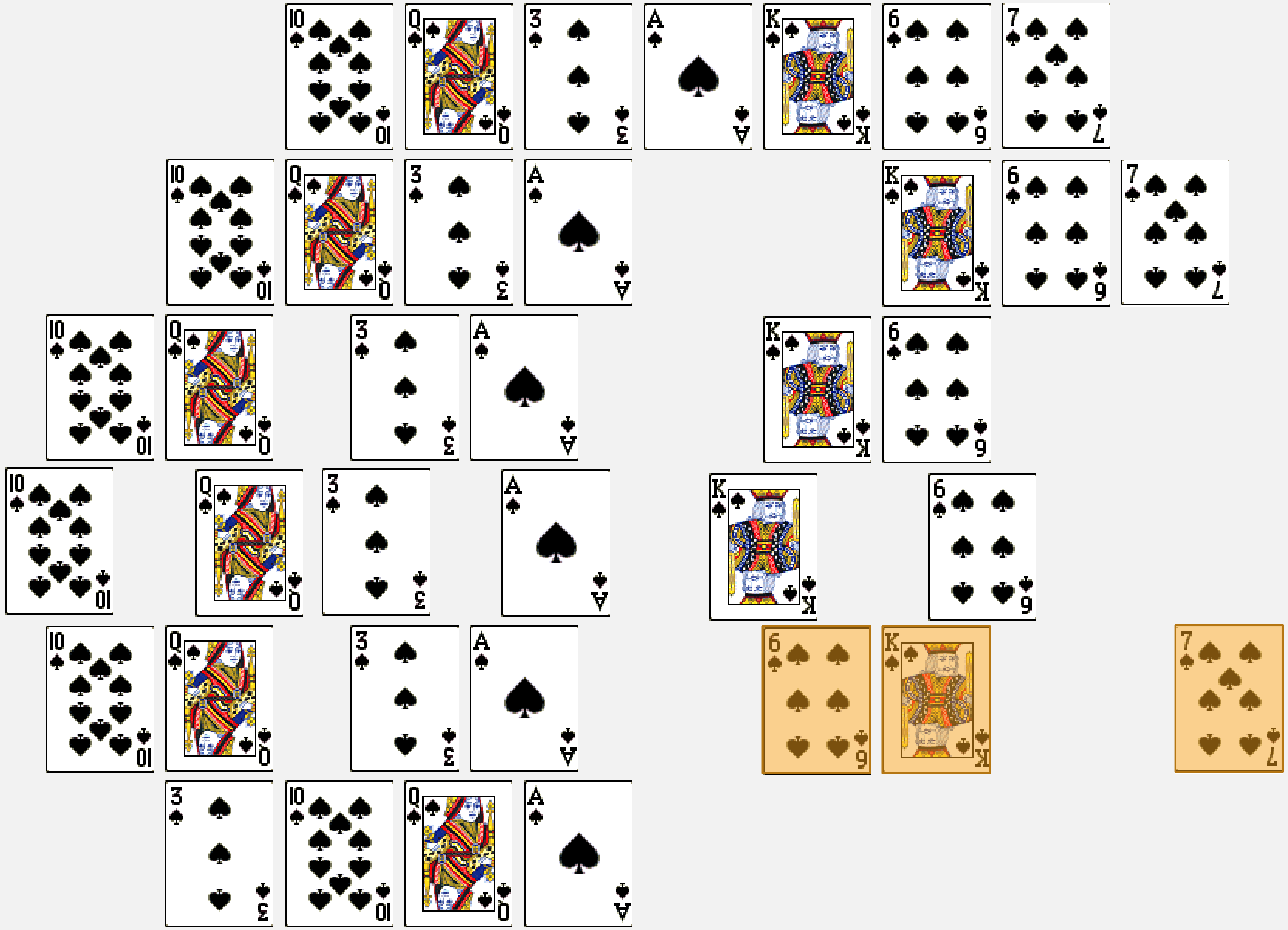


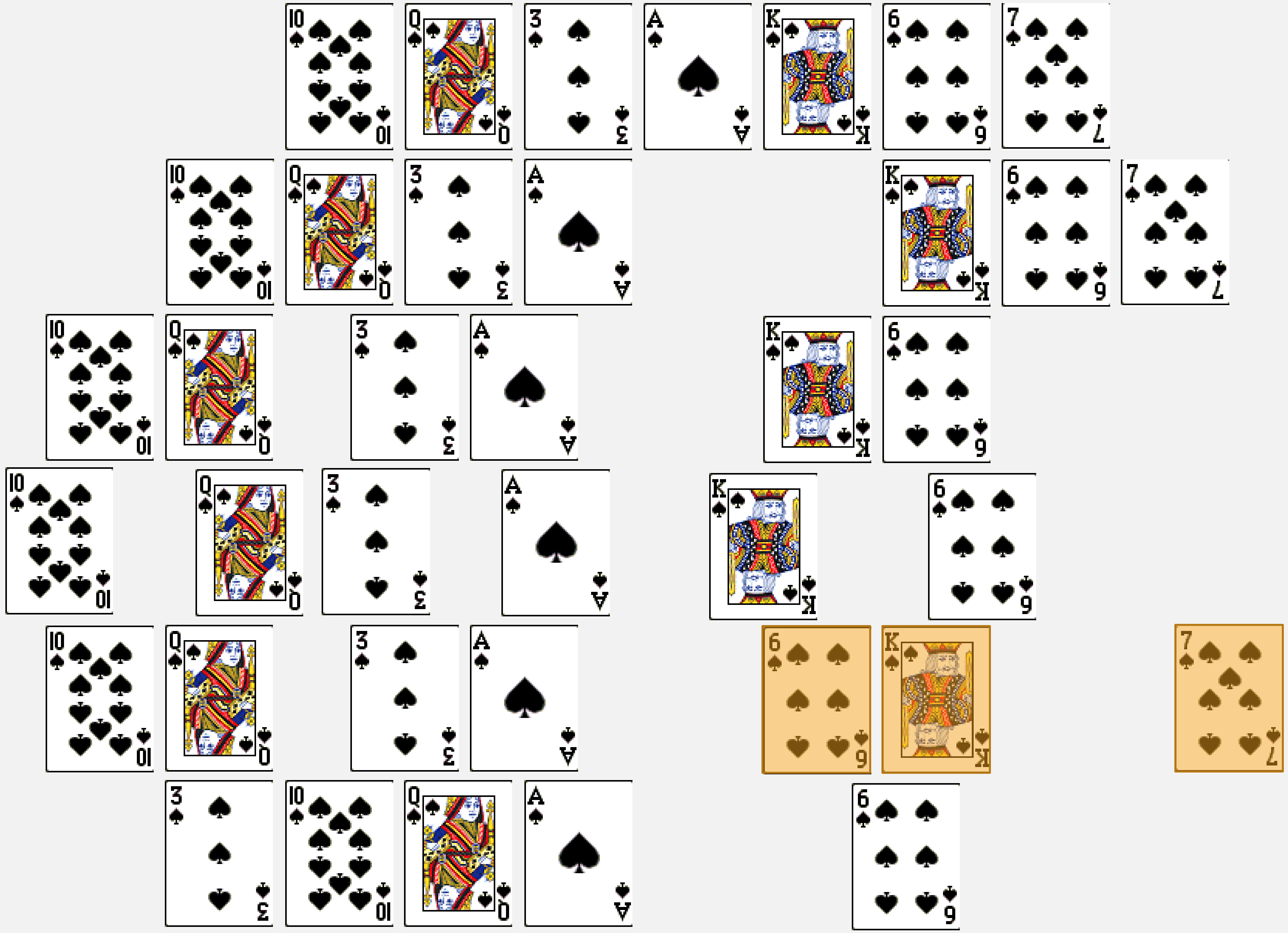


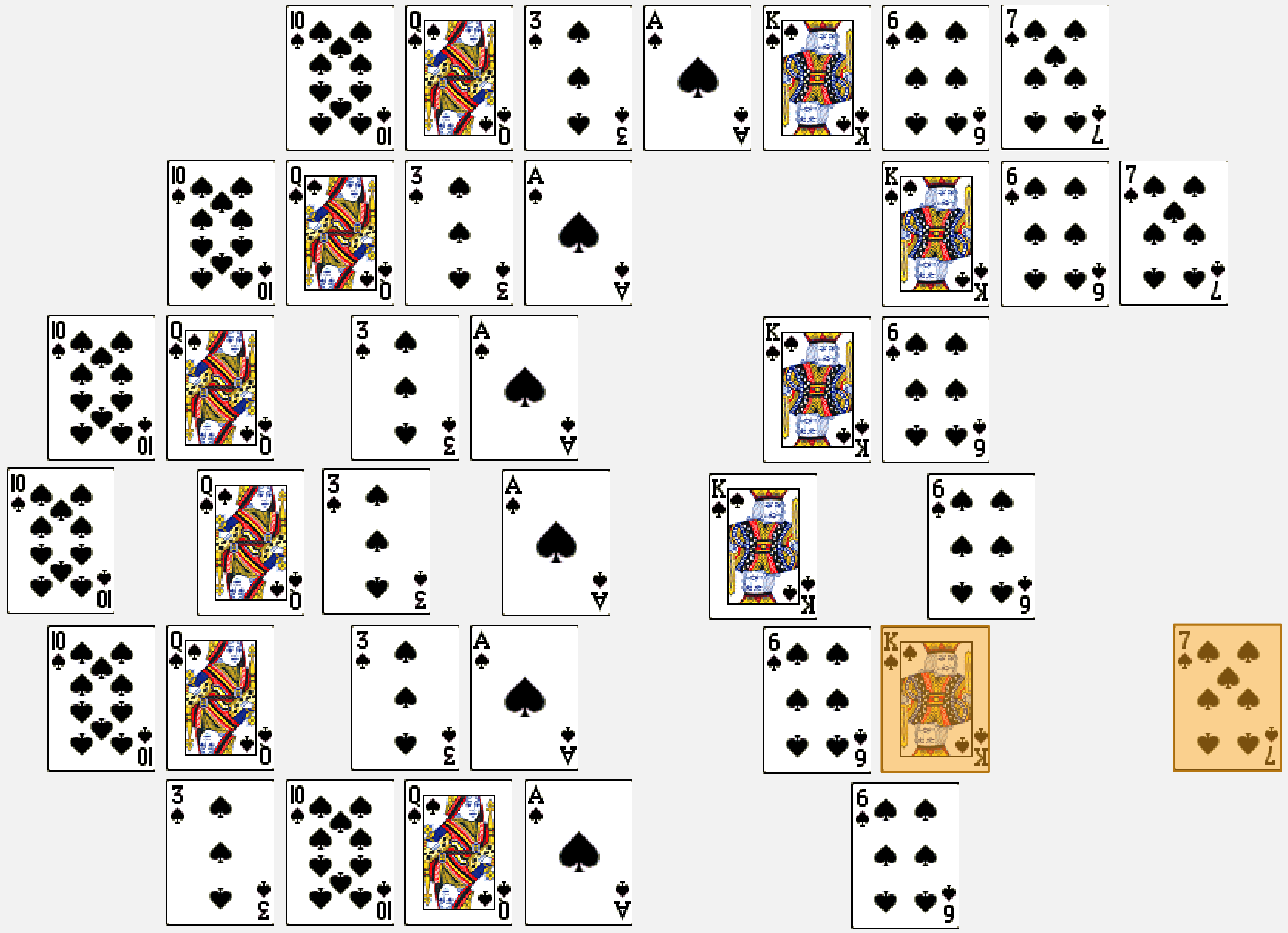


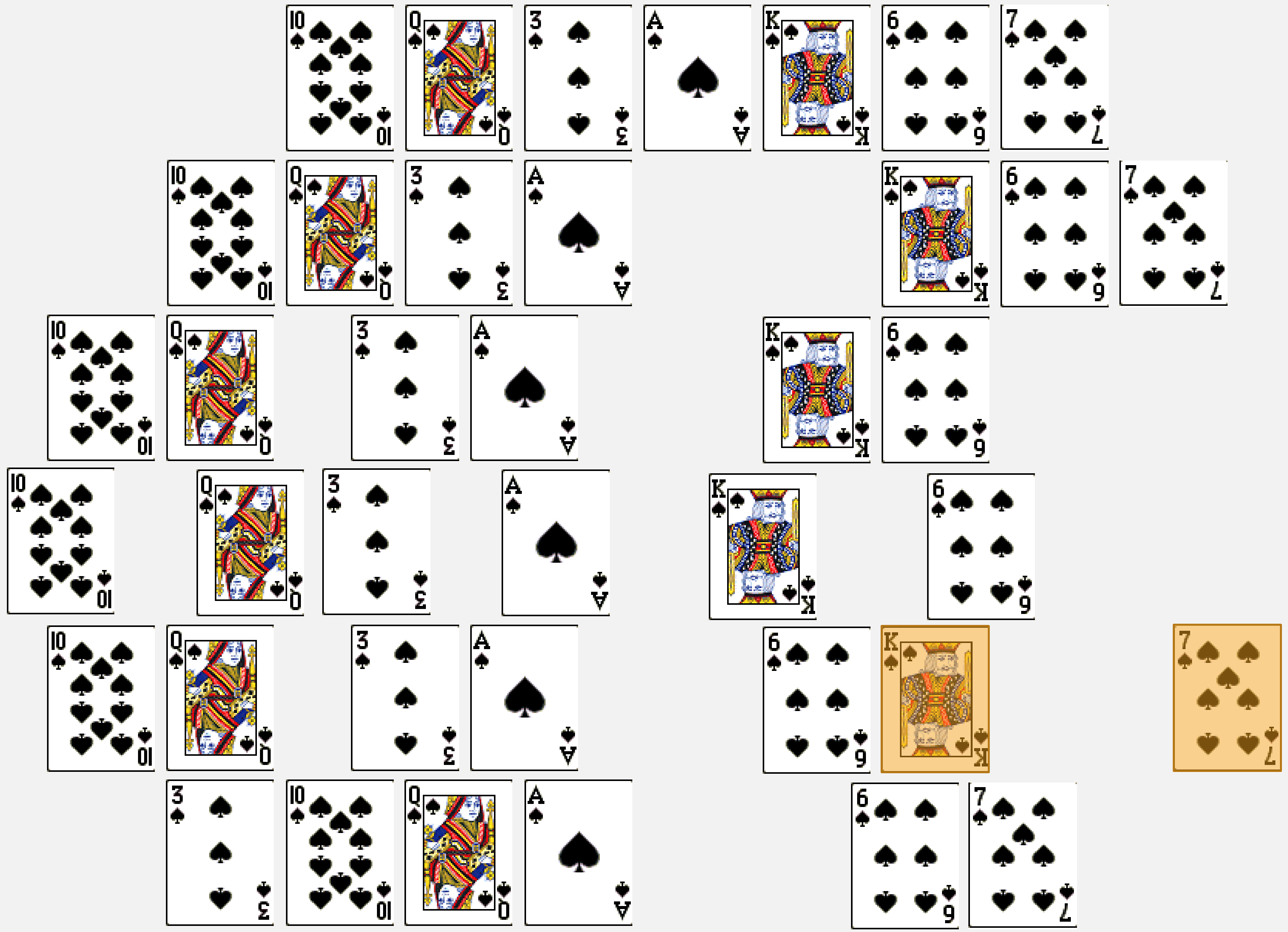


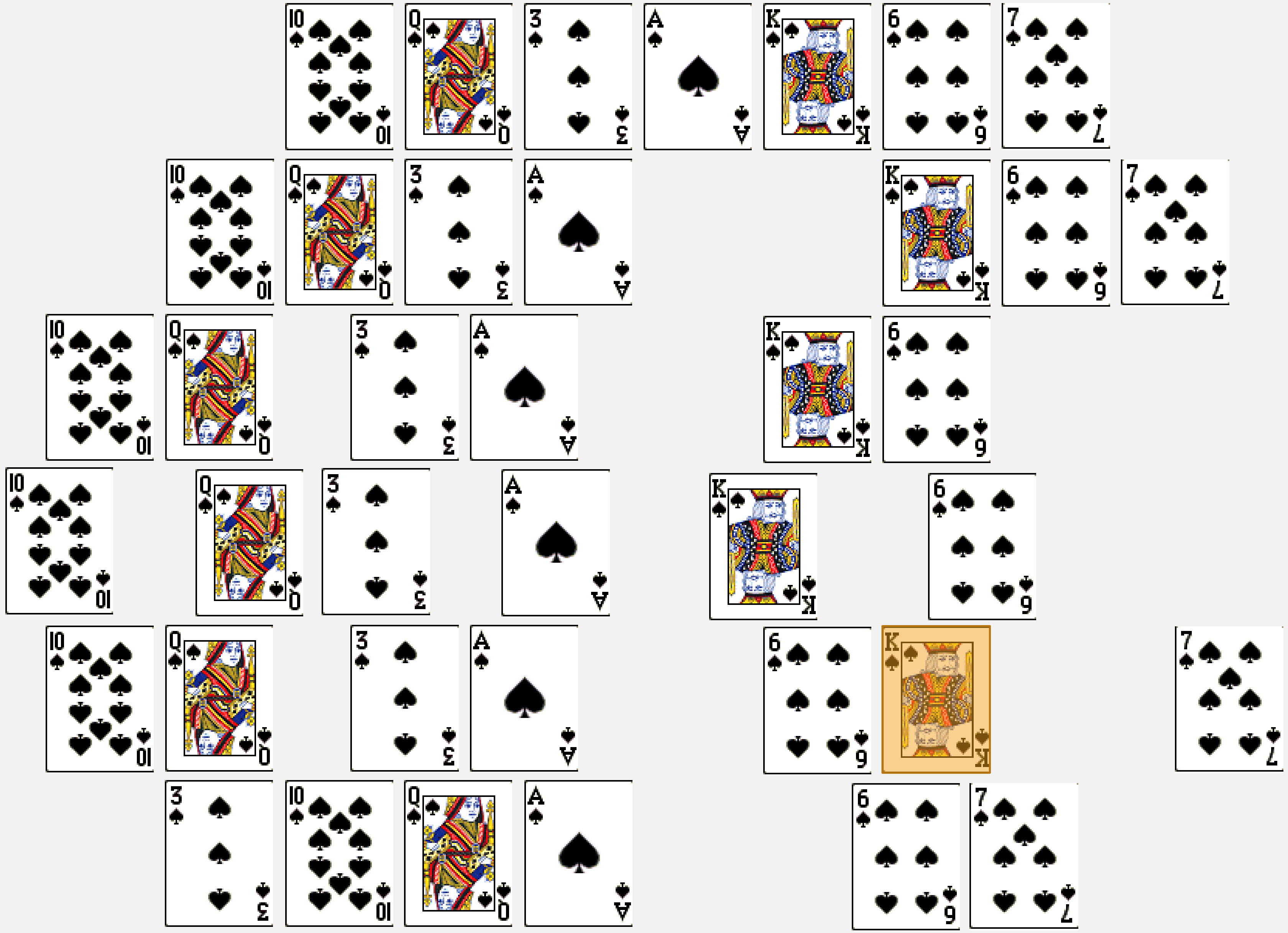


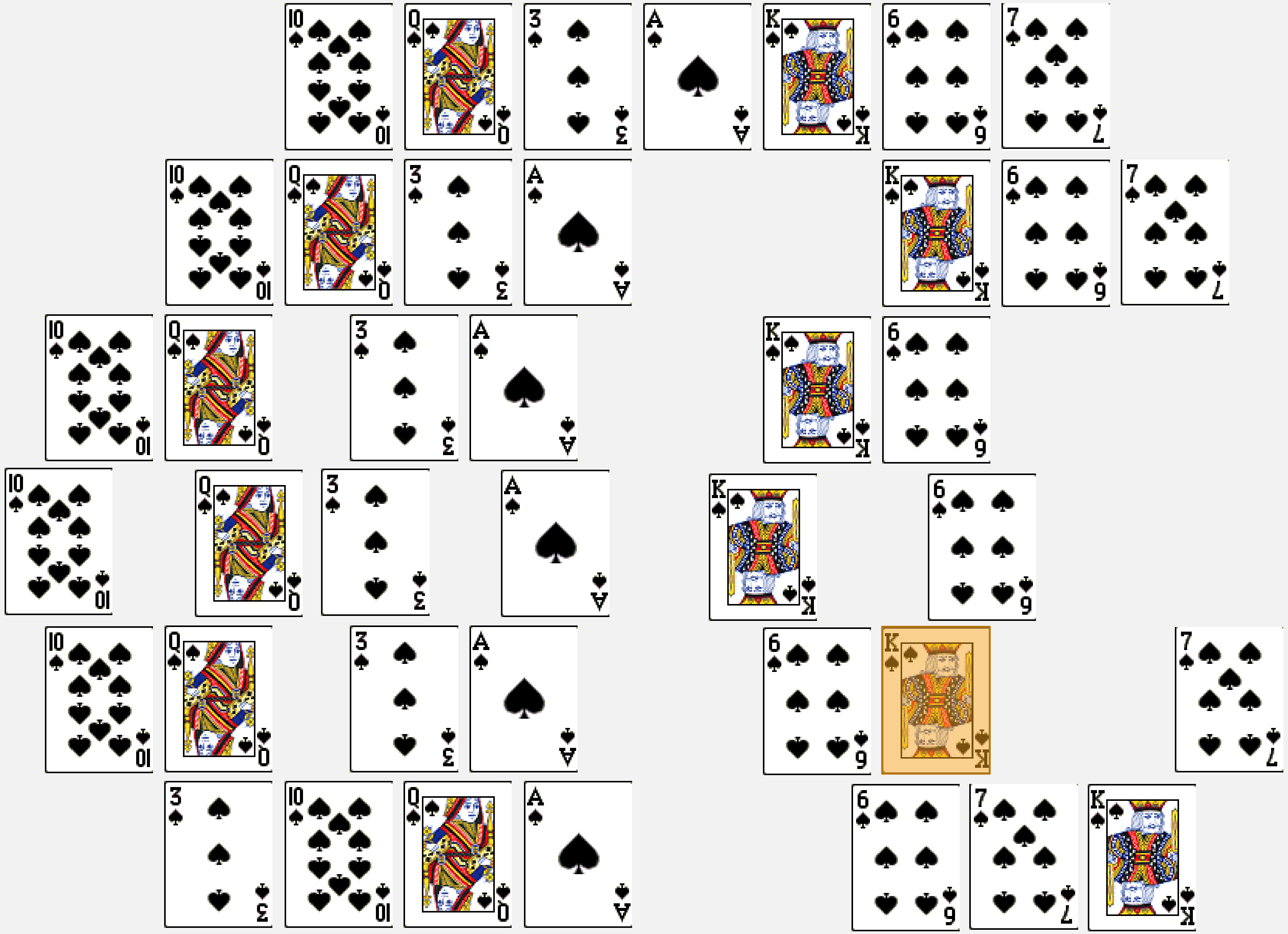


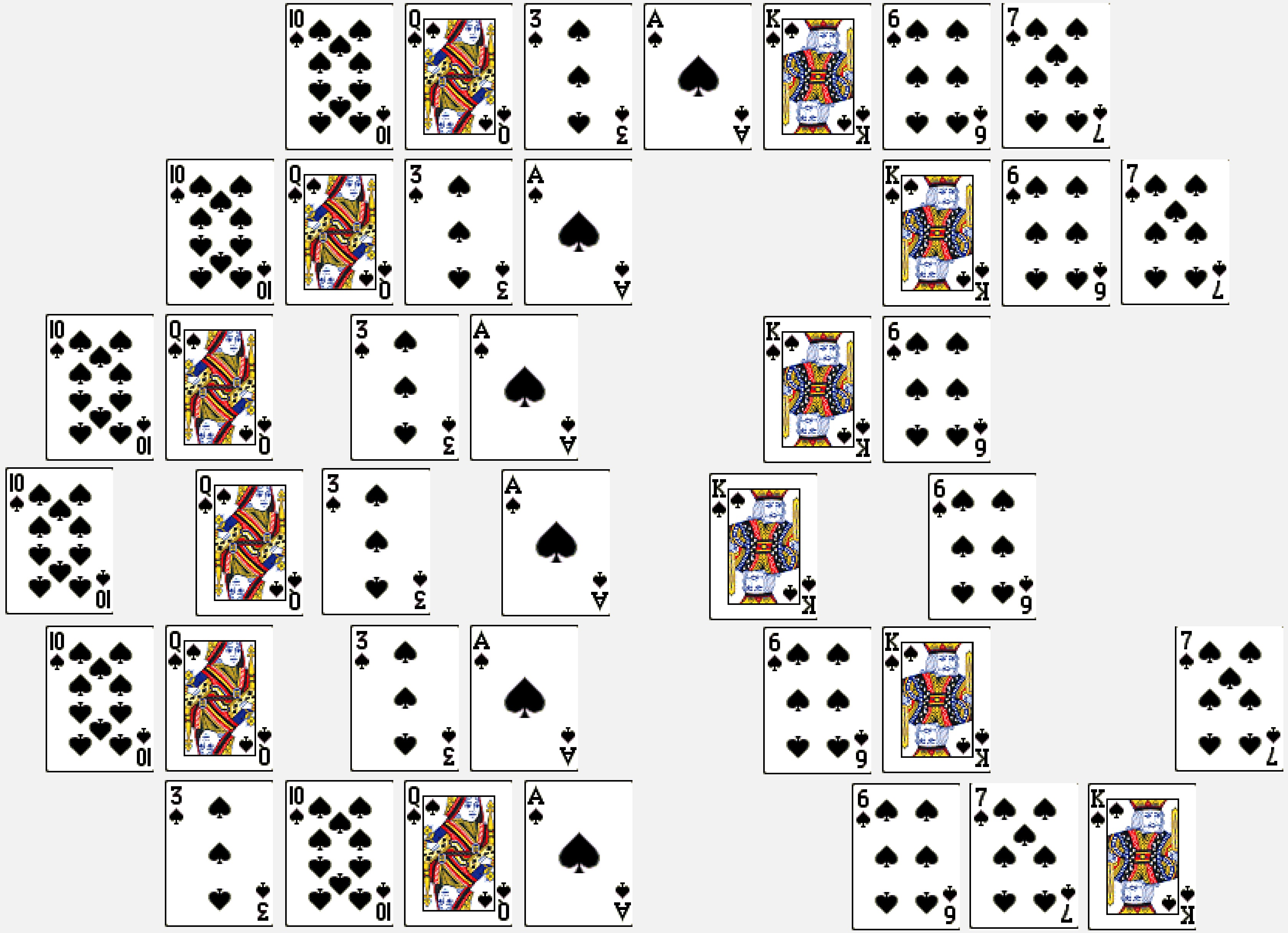


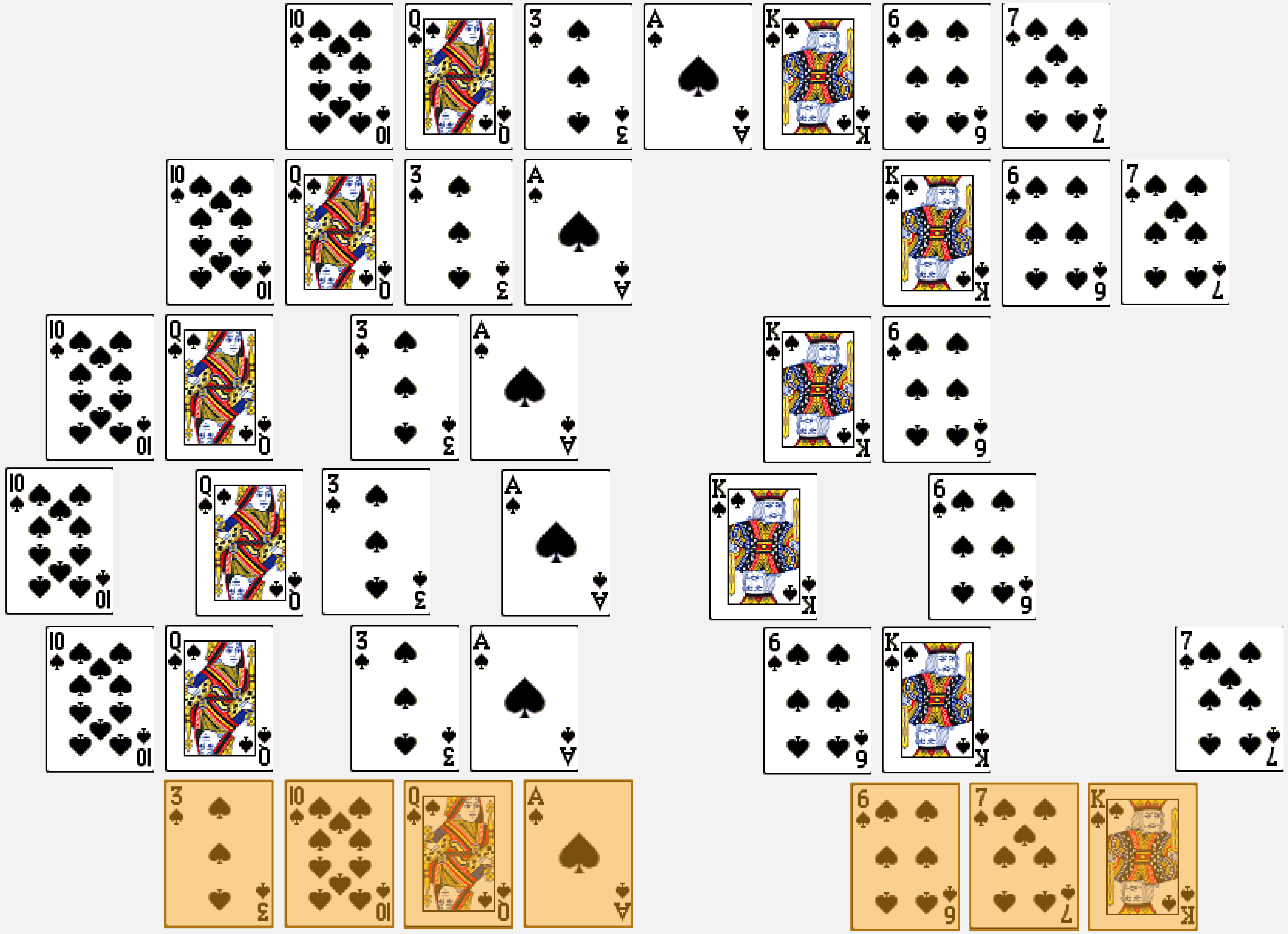


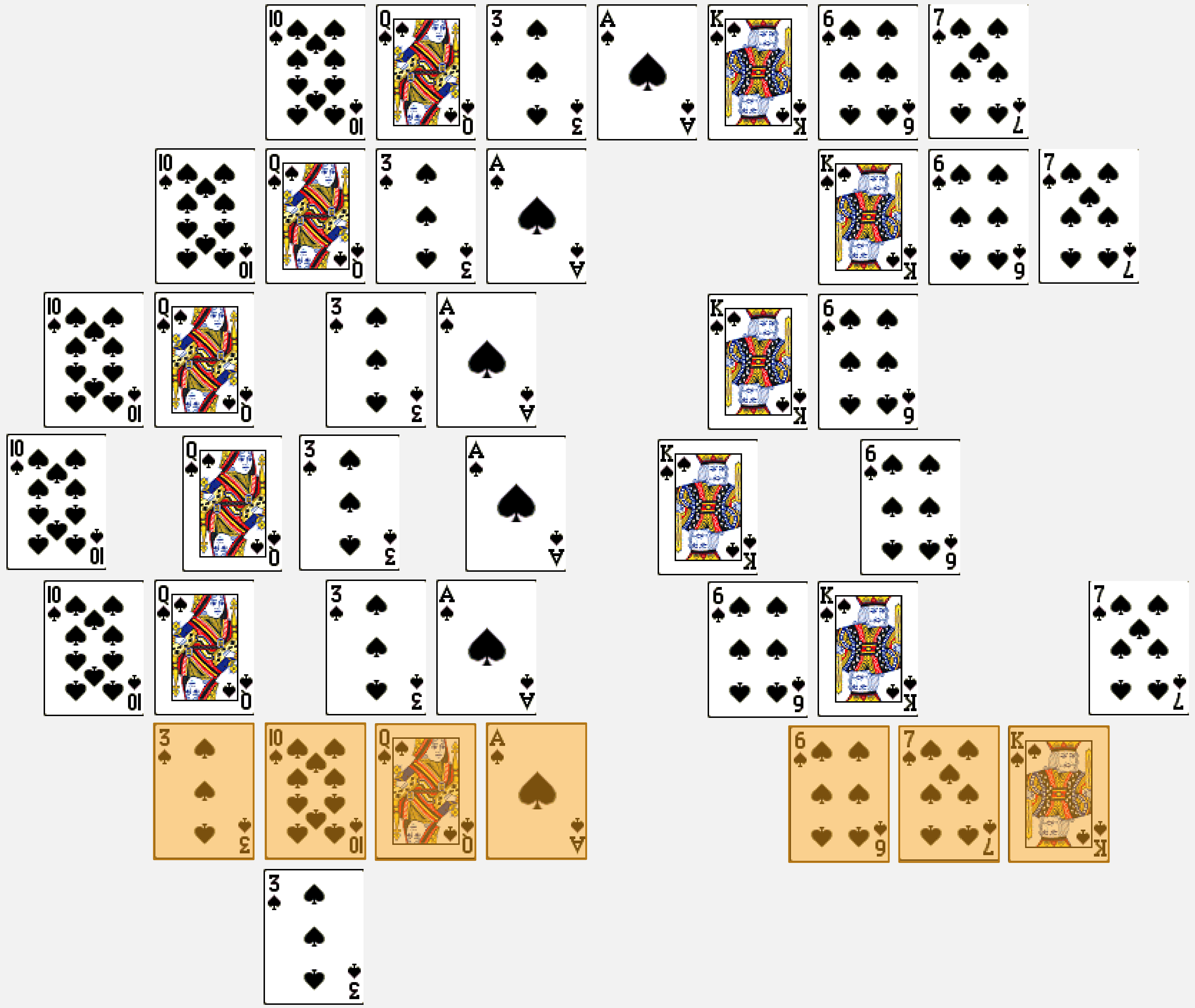


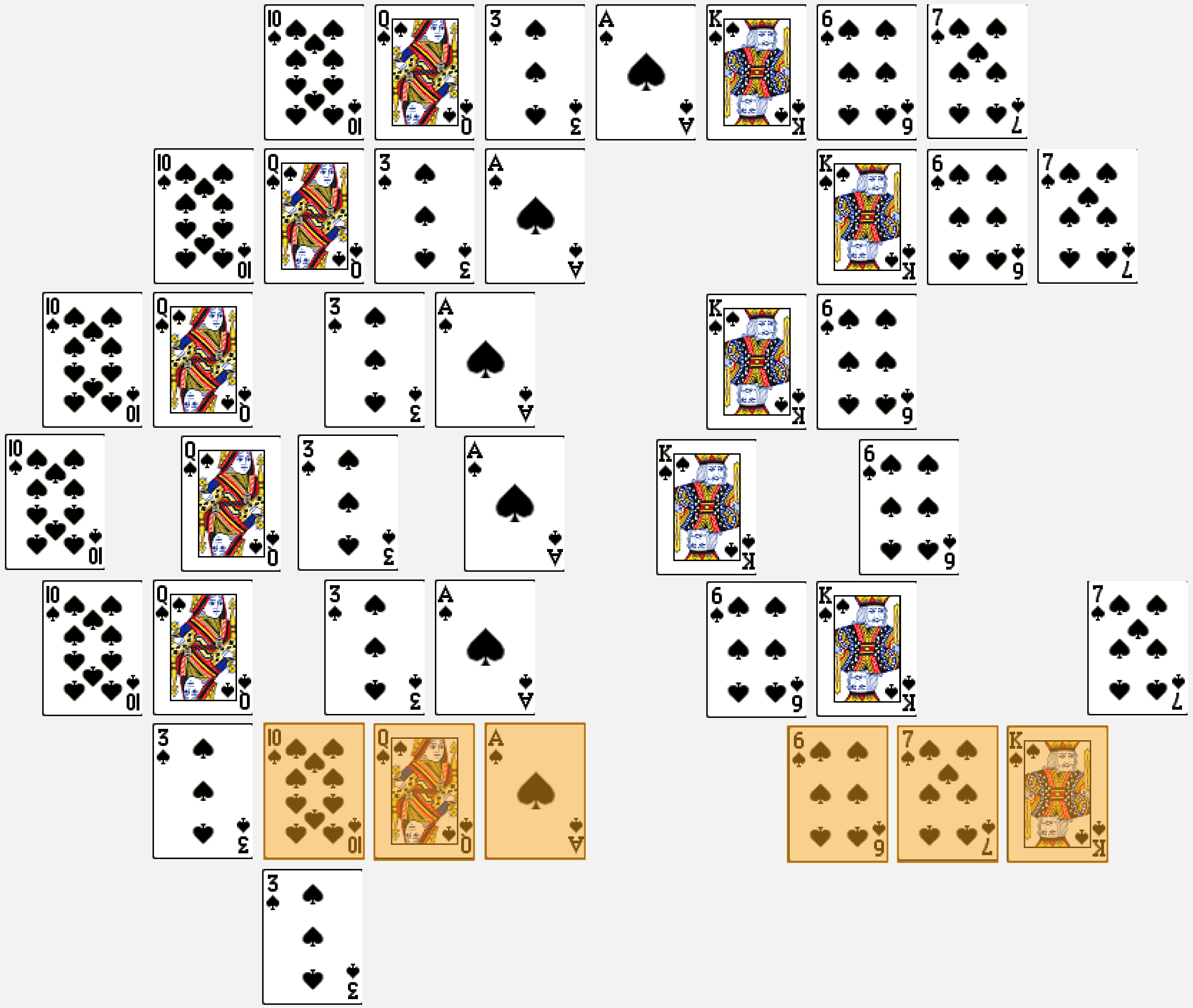


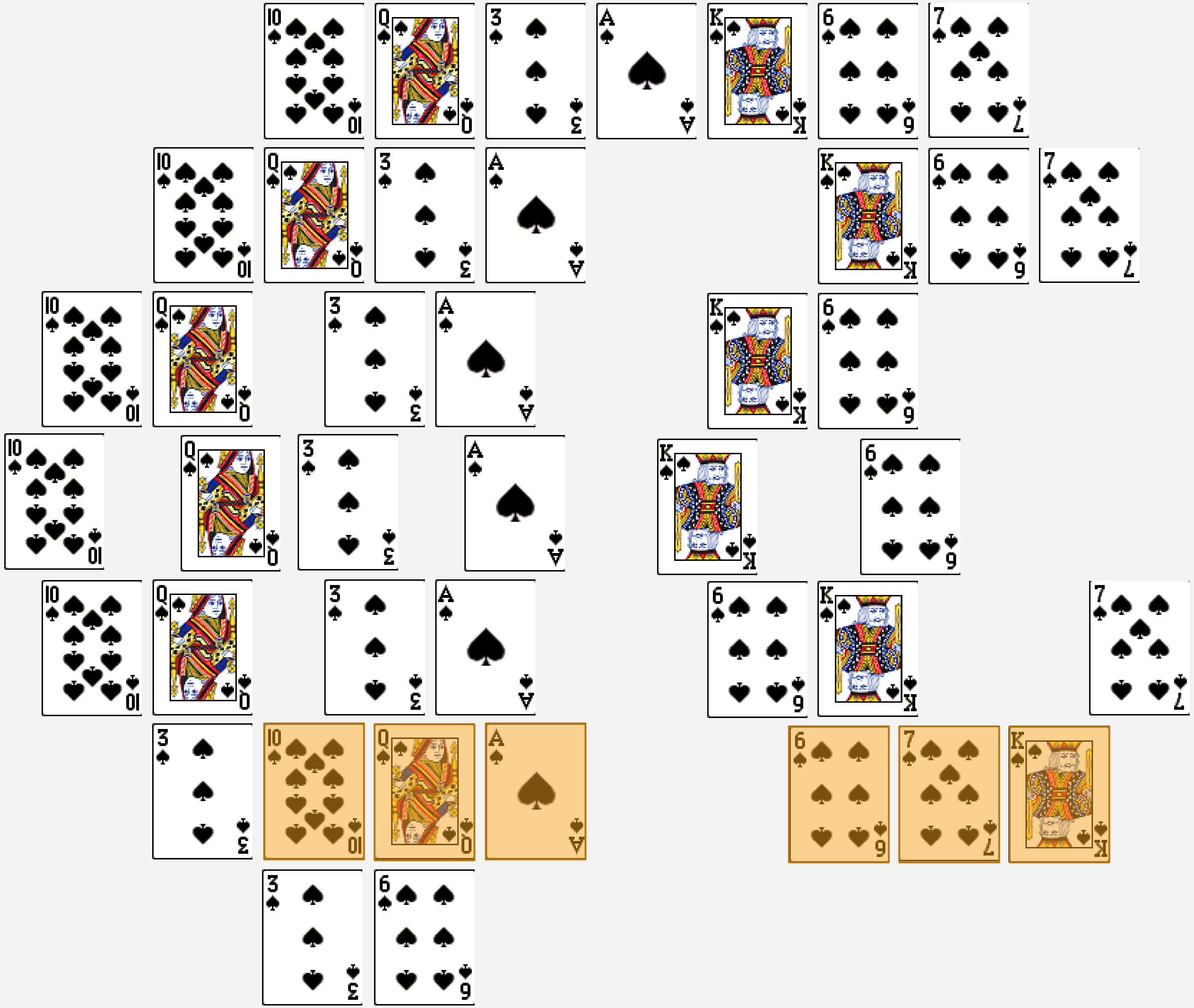


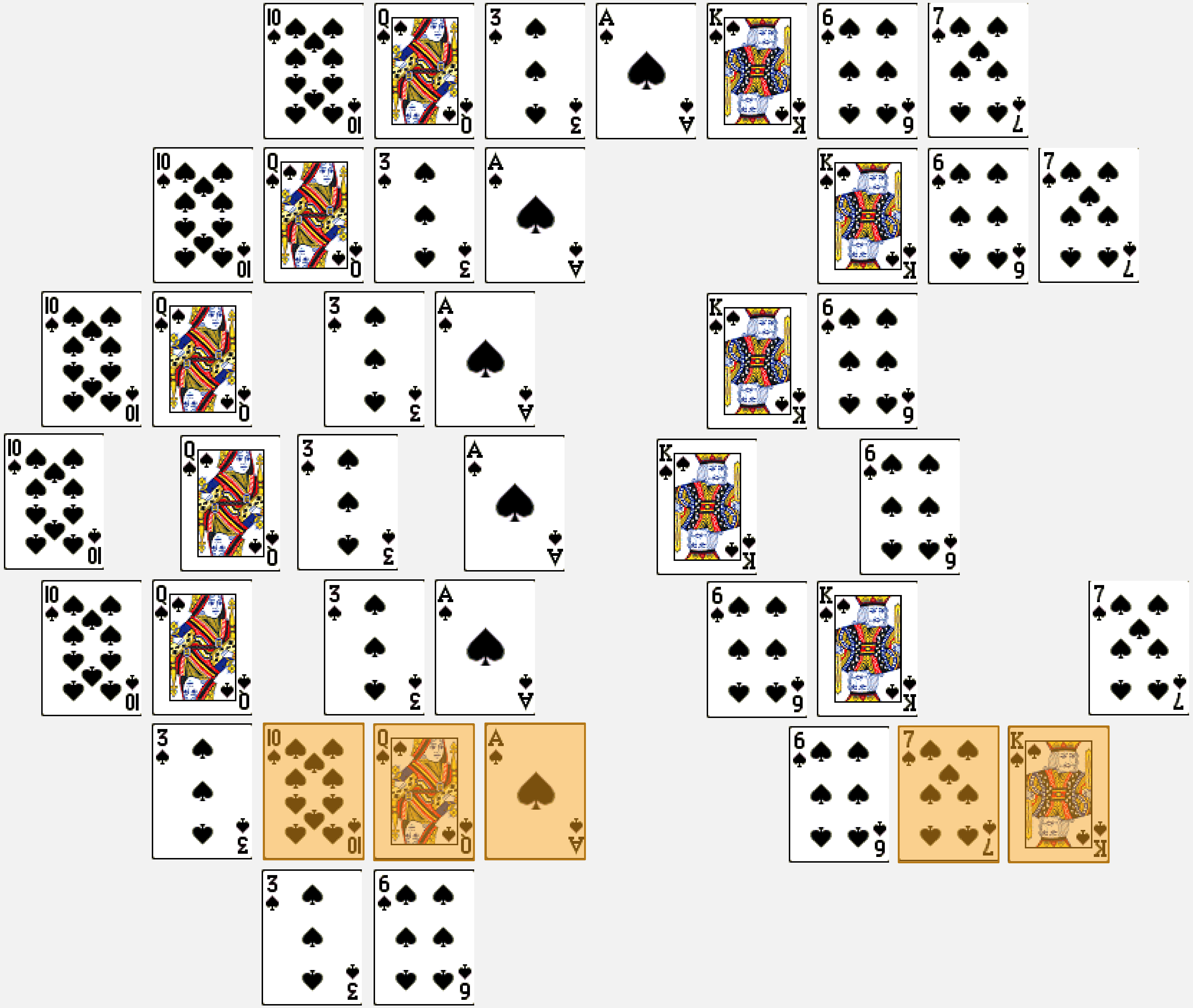


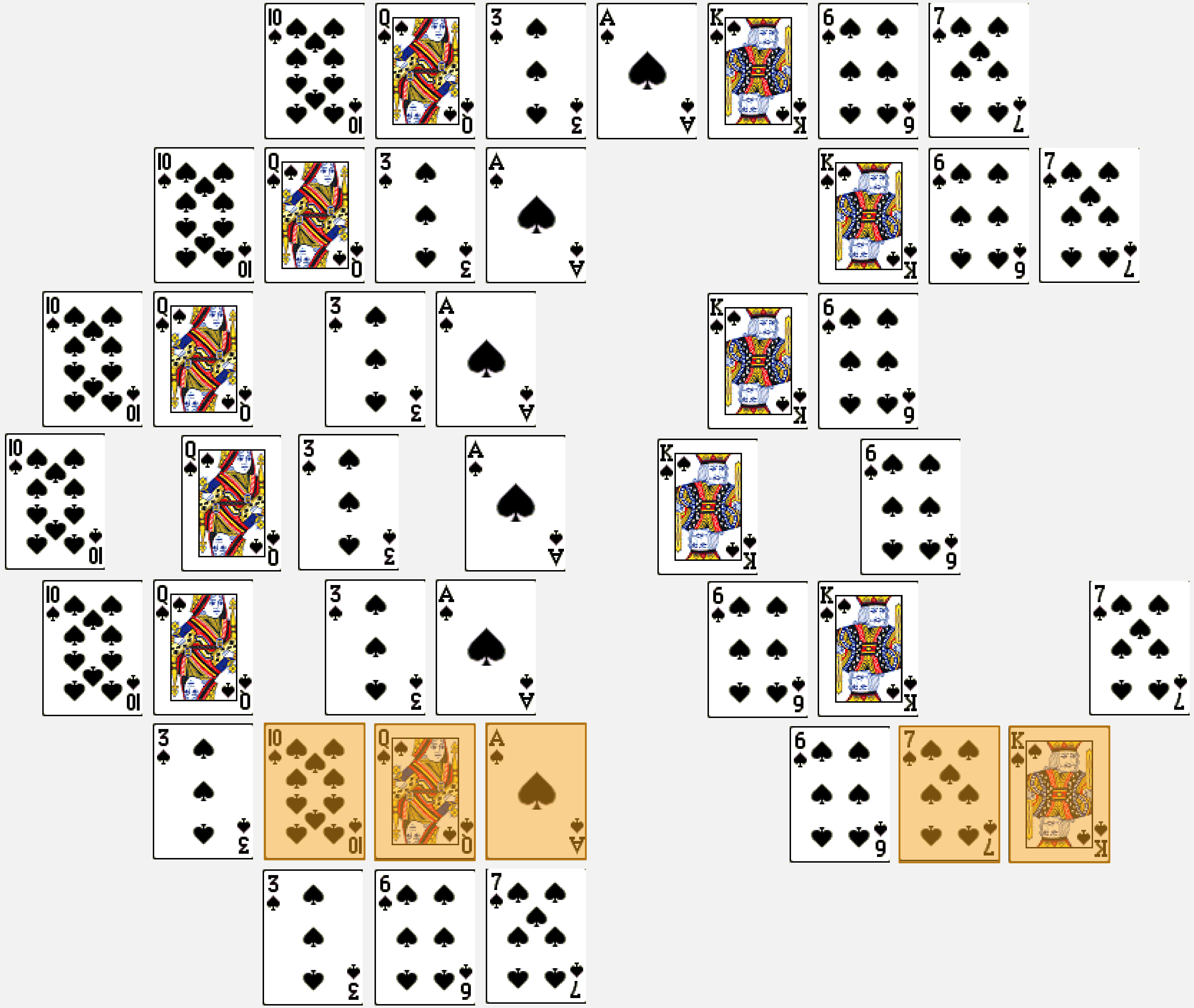


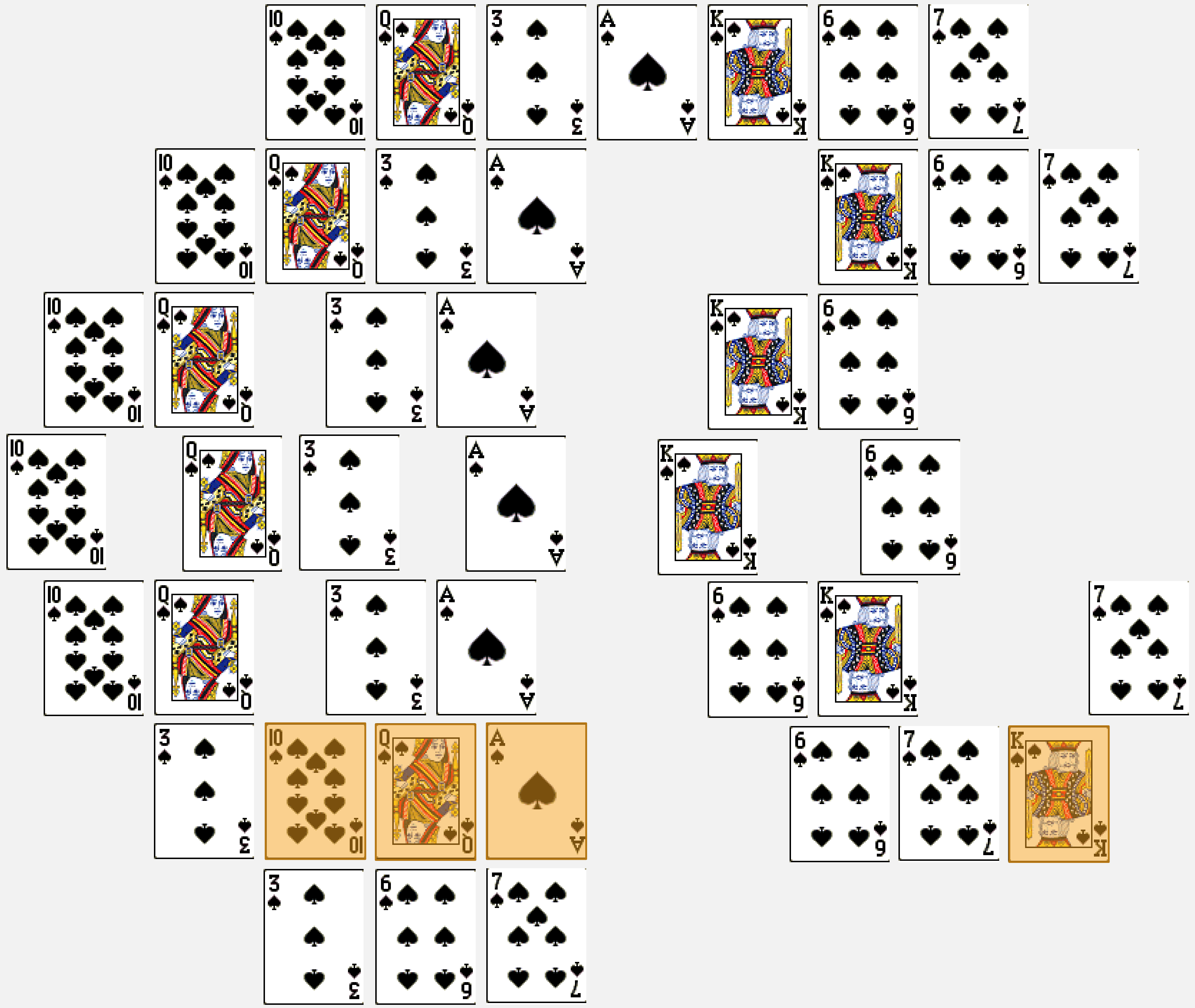


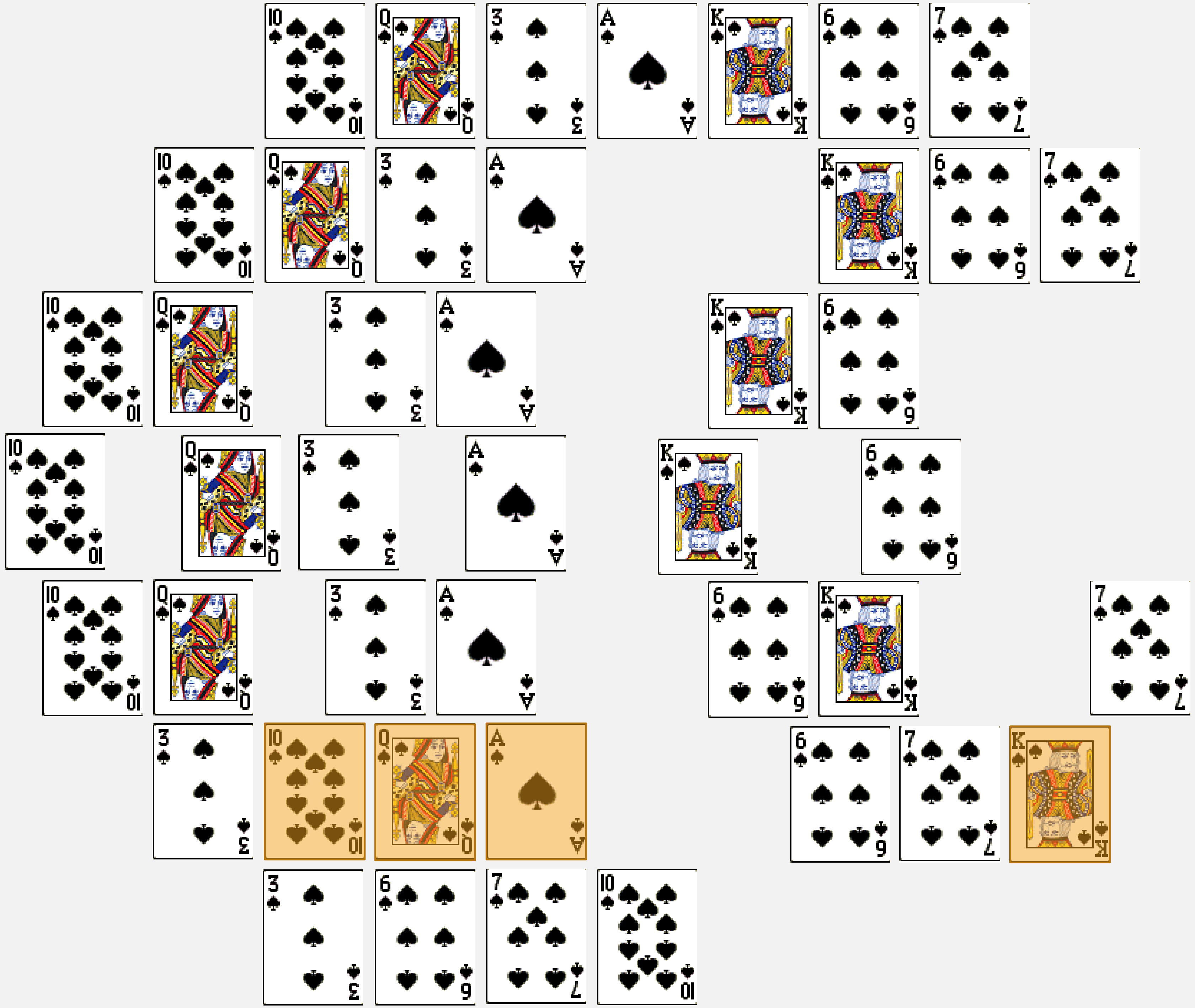


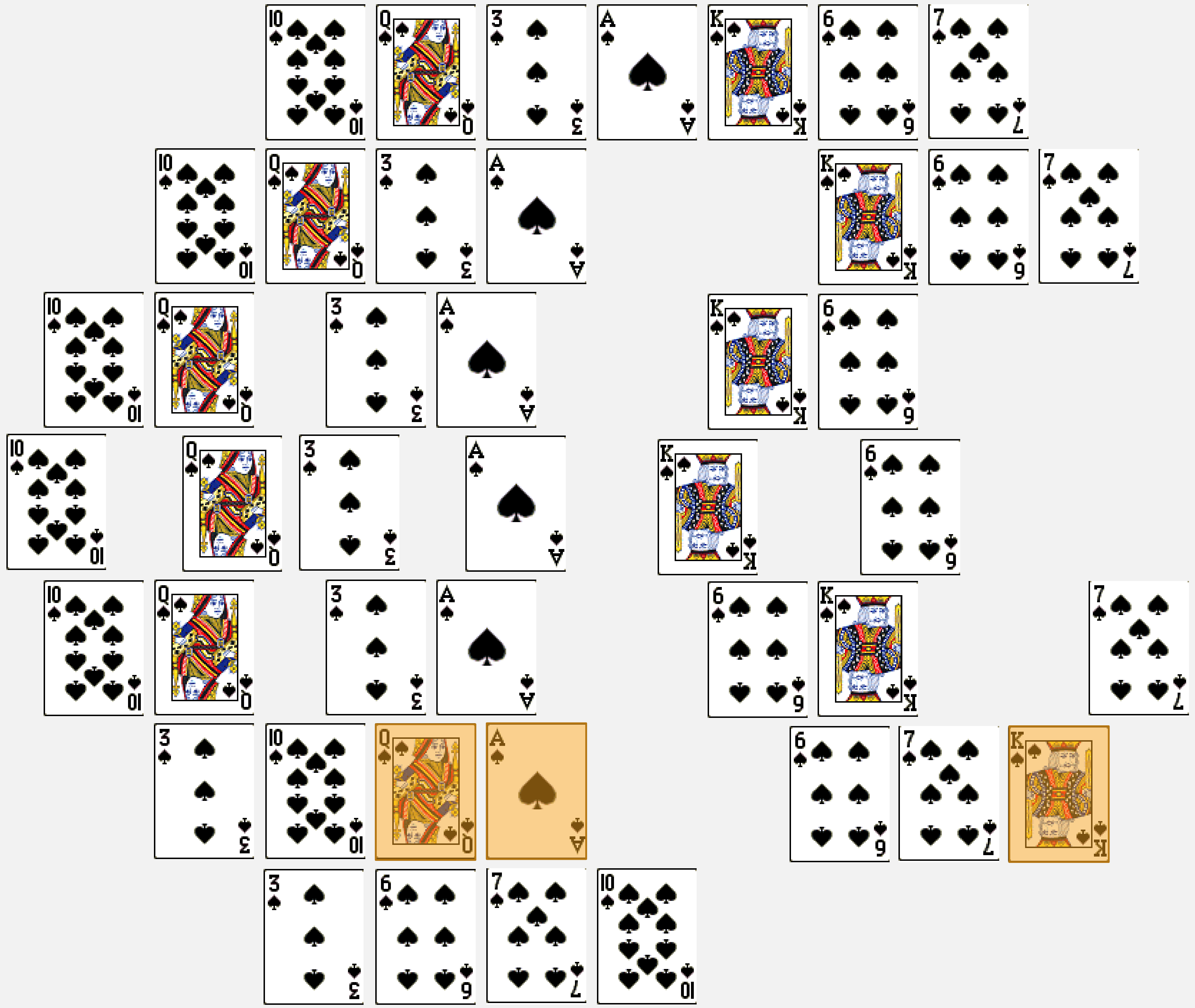


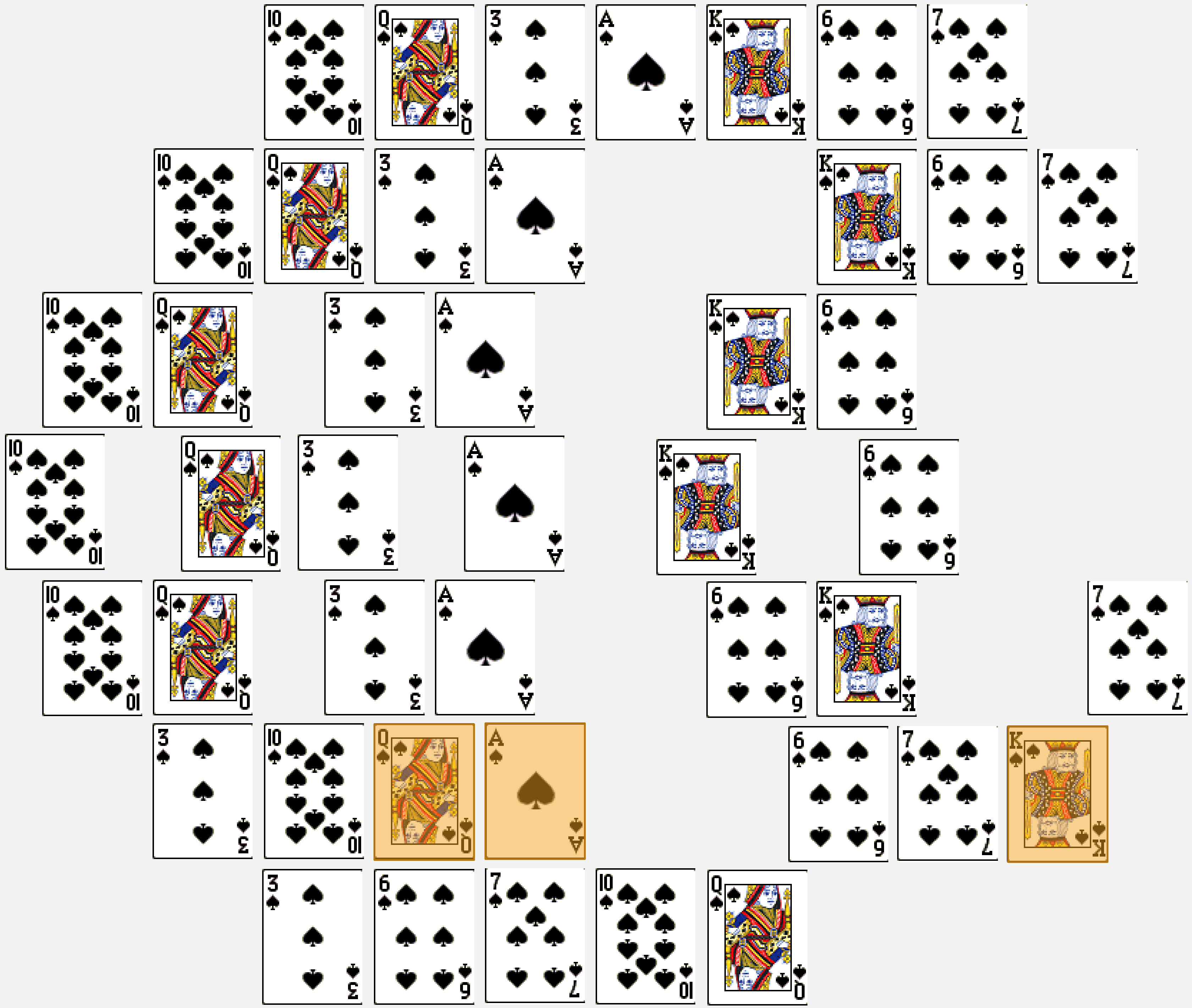


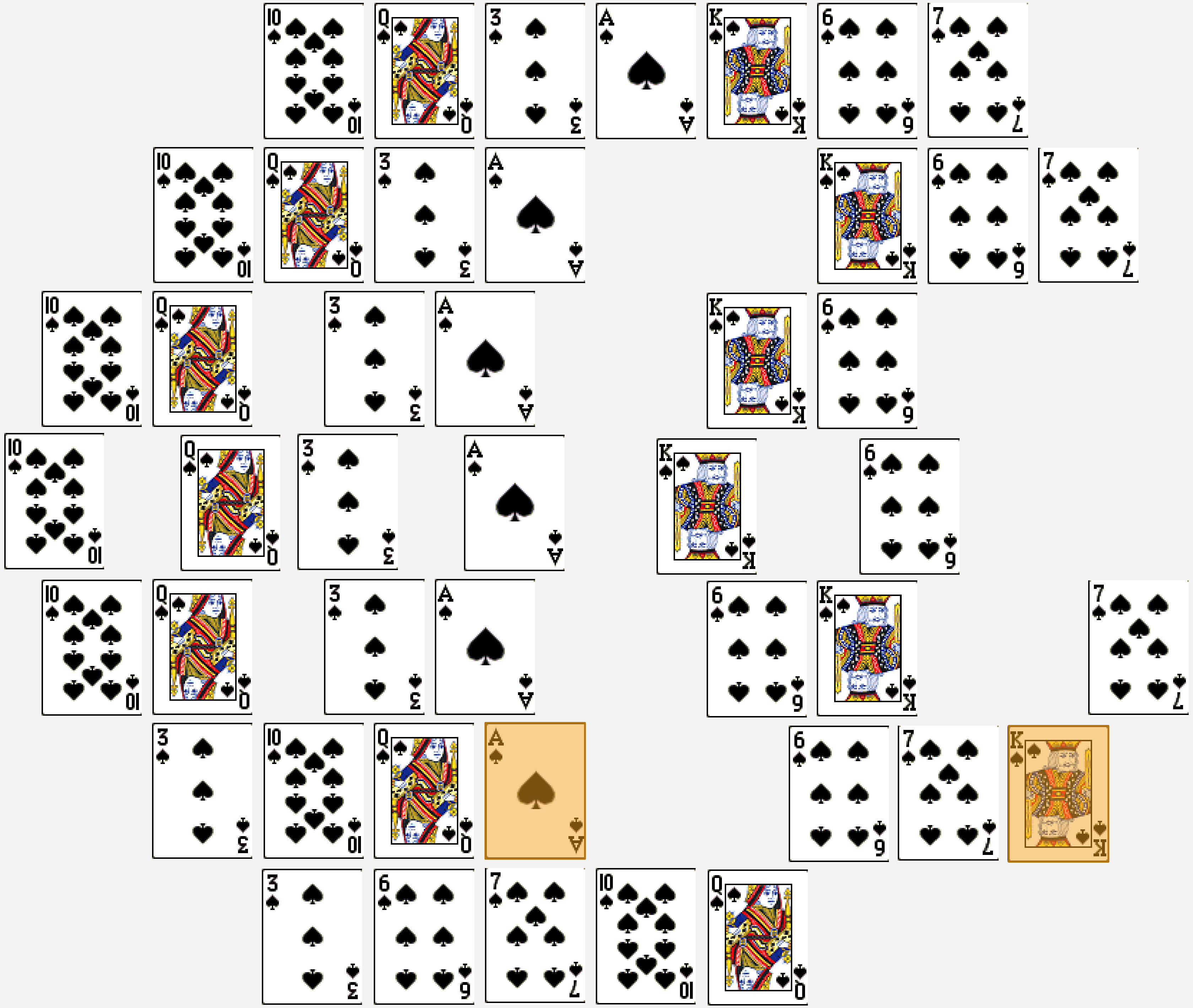


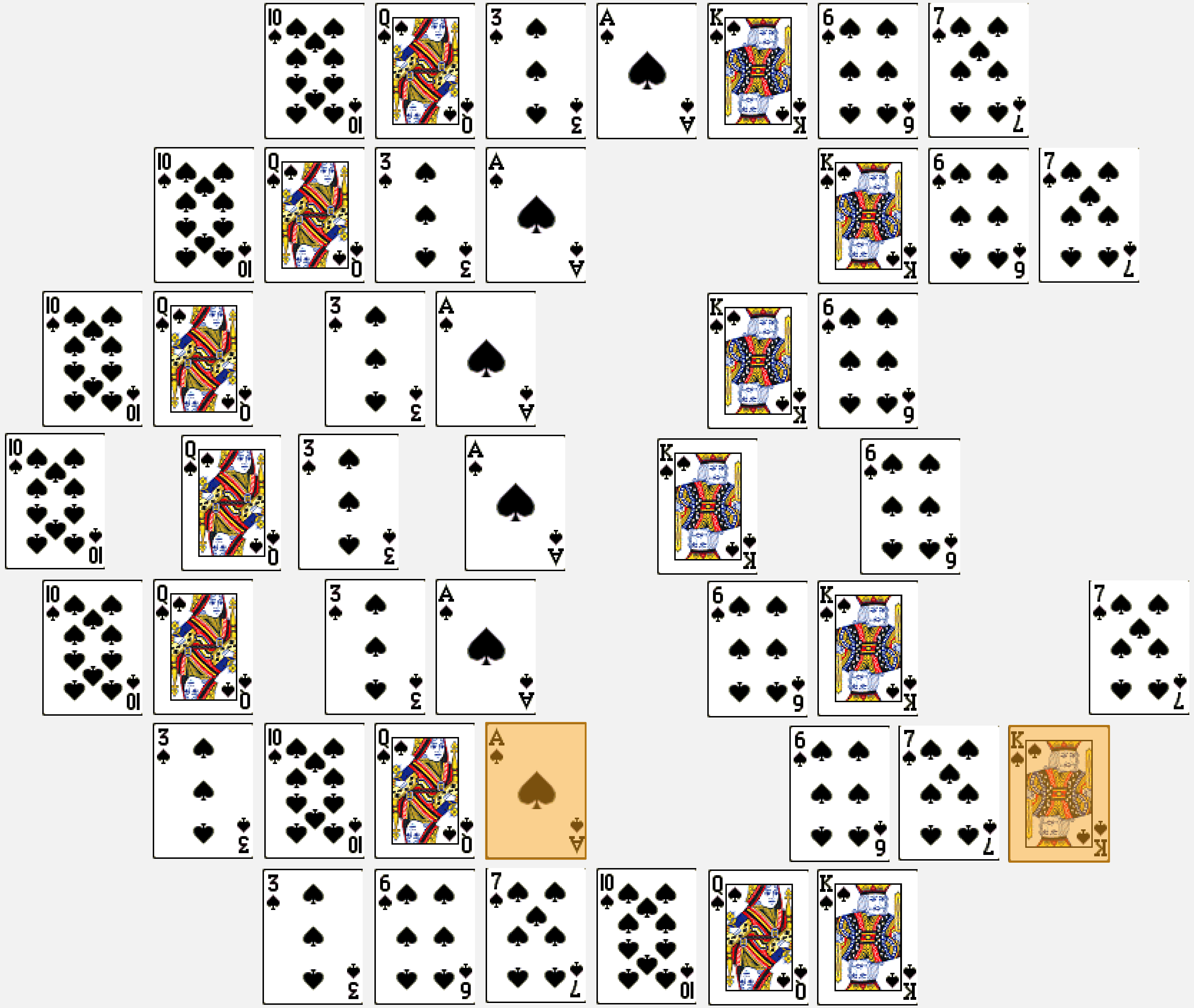


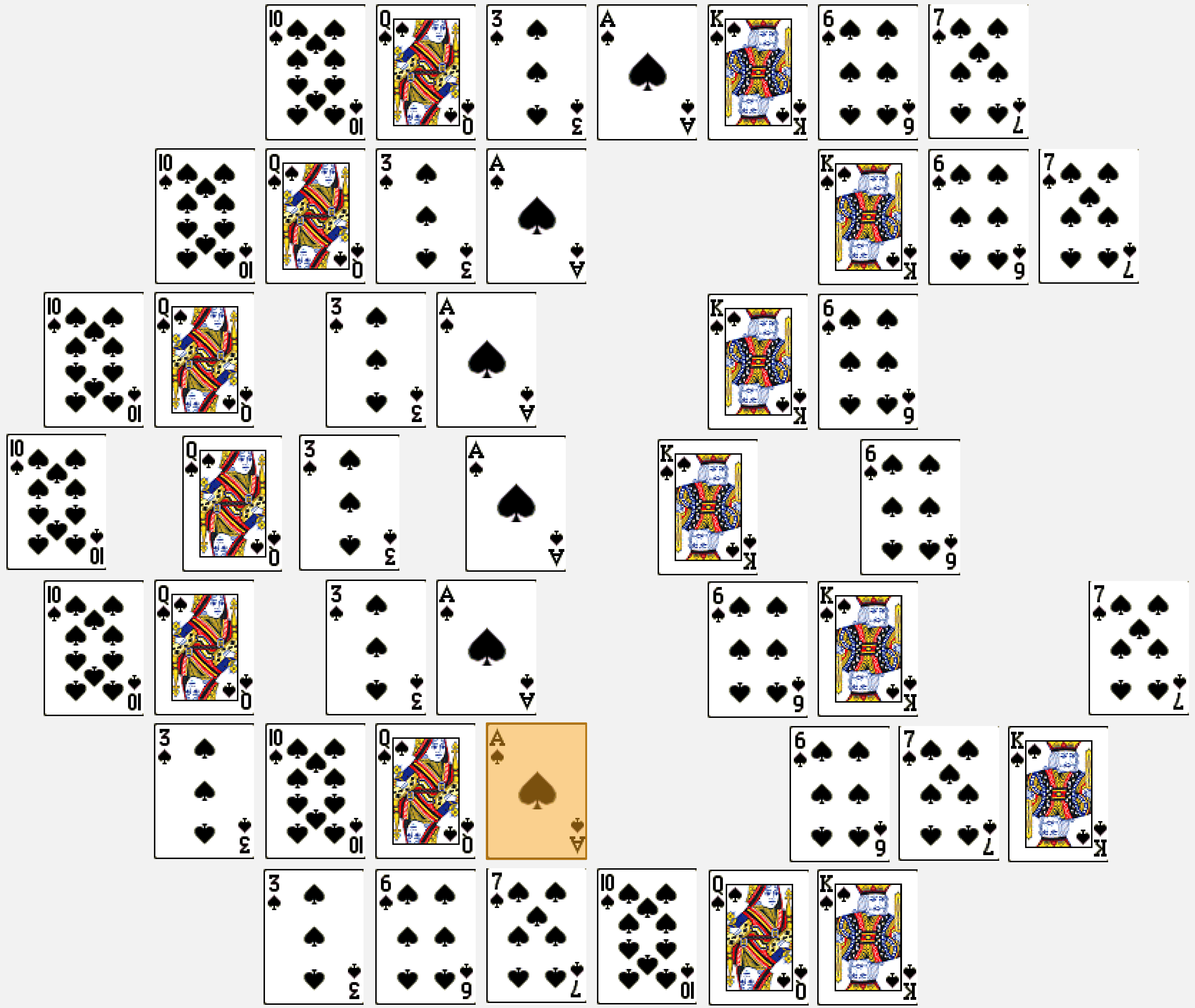


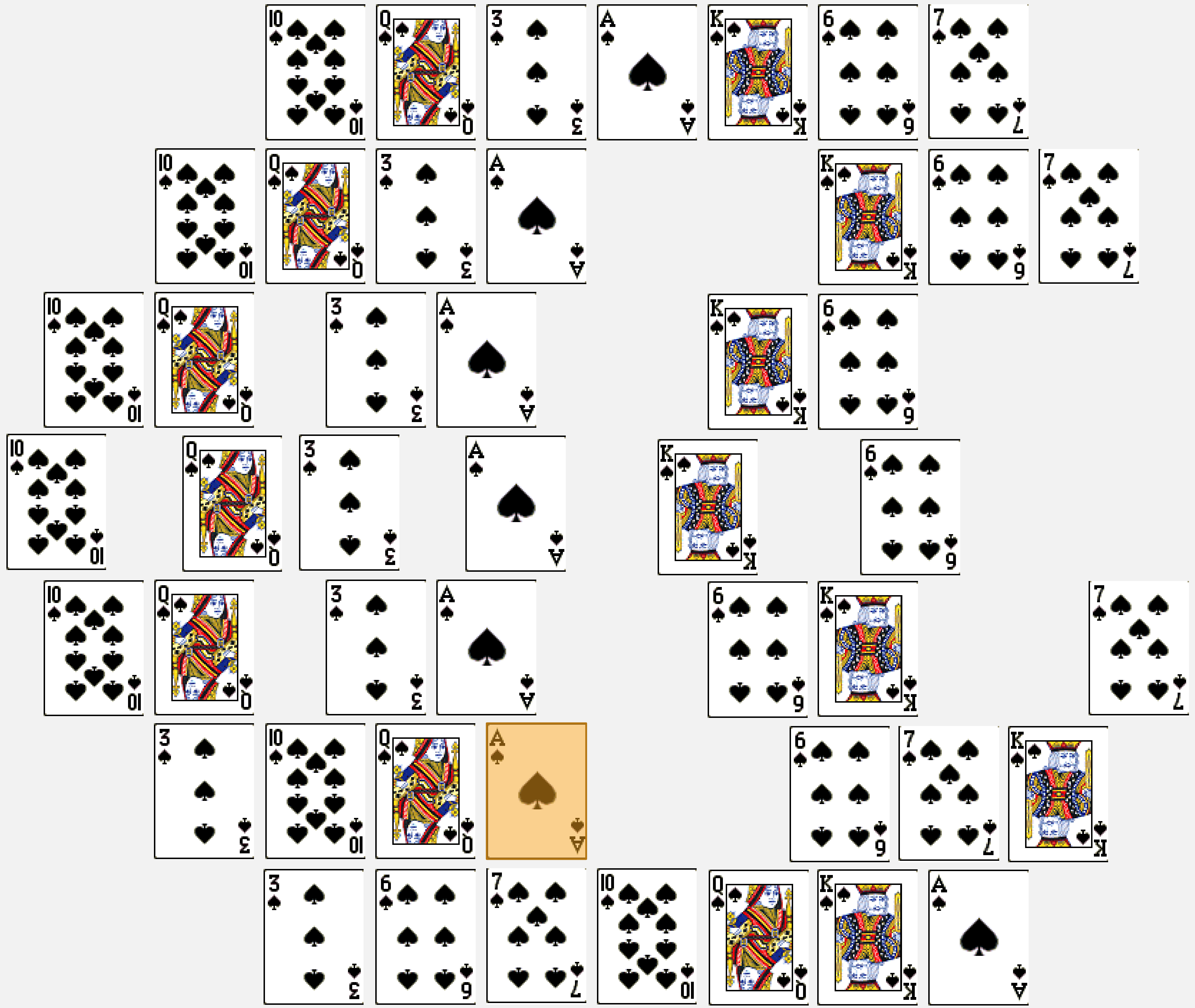


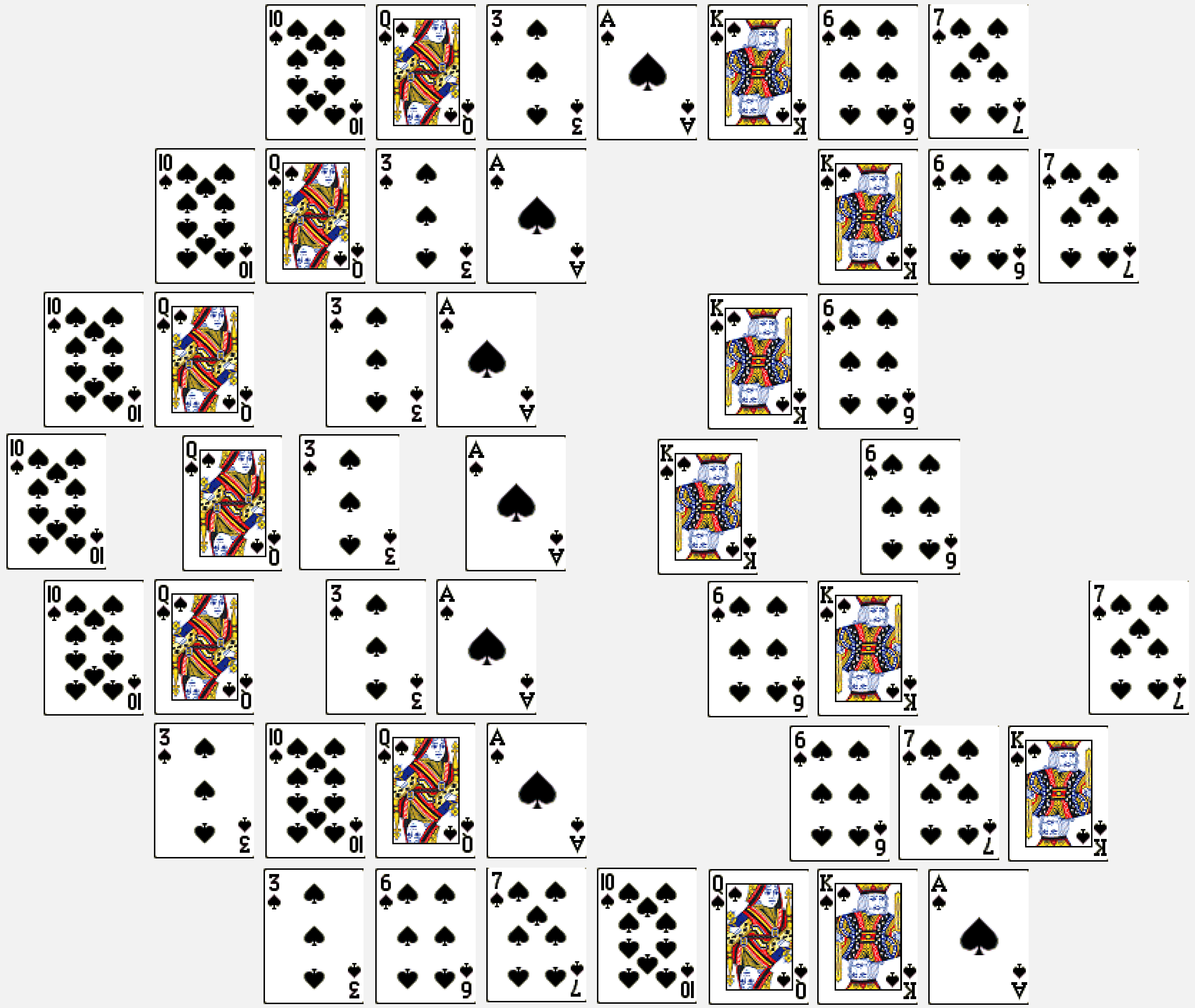


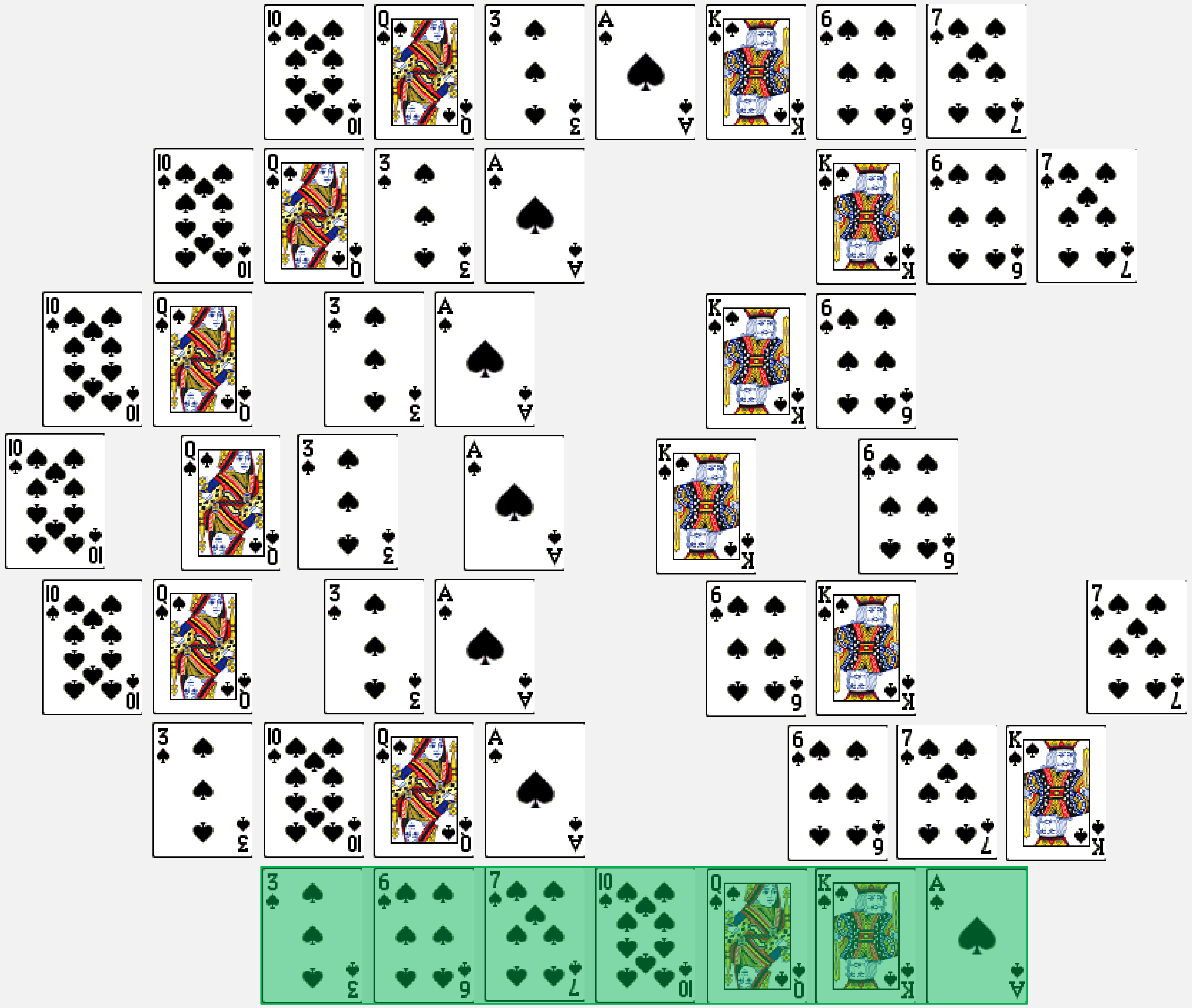






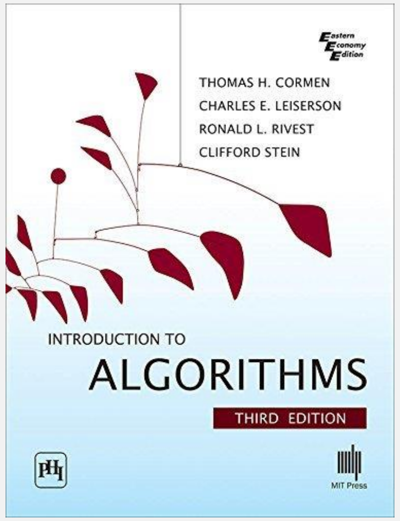






QUICKSORT

QUICKSORT

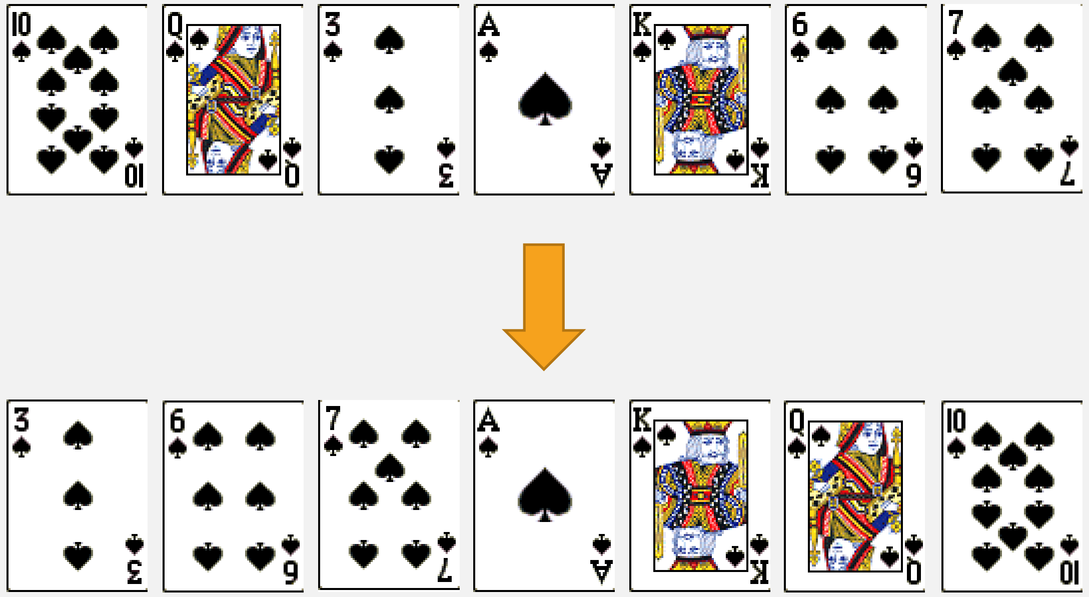


```
QuickSort(list, p, r):
    if p < r:
        q = Partition(list, p, r)
        QuickSort(list, p, q - 1)
        QuickSort(list, q + 1, r)
```

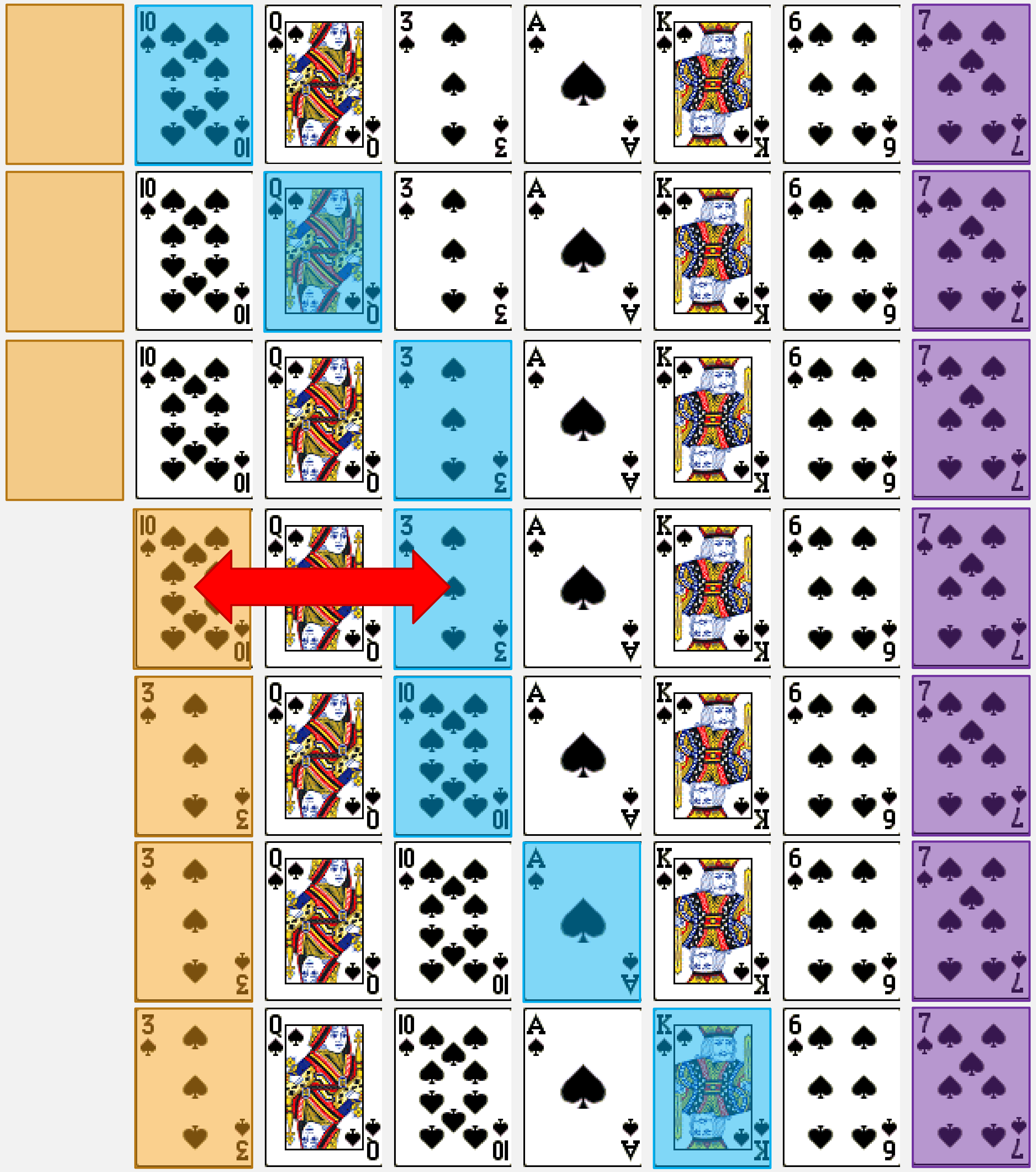
```
Partition(list, p, r):
    x = list[r]
    i = p – 1
    for j = p to r – 1:
        if list[j] <= x:
            i = i + 1
            swap A[i] and A[j]
    swap A[i + 1] and A[r]
    return i + 1
```

```
call QuickSort(A, 1, length(list))
```

PARTITION



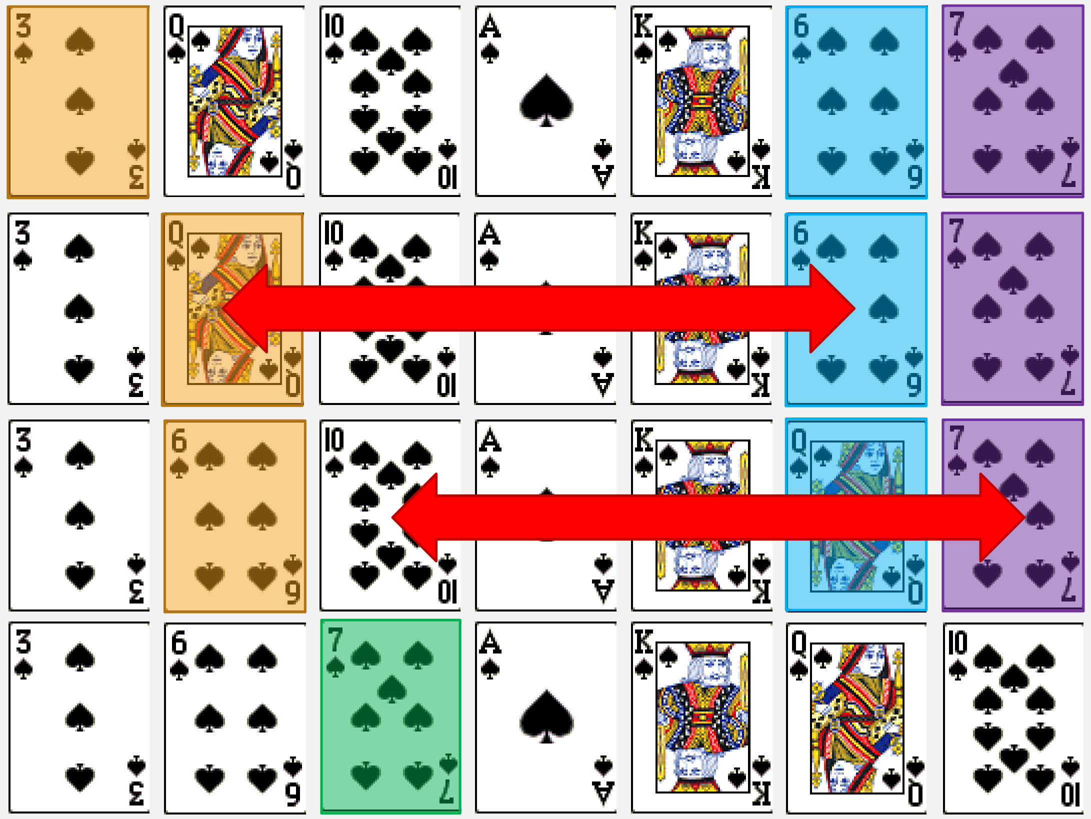
```
Partition(list, p, r):
    x = list[r]
    i = p – 1
    for j = p to r – 1:
        if list[j] <= x:
            i = i + 1
            swap A[i] and A[j]
    swap A[i + 1] and A[r]
    return i + 1
```

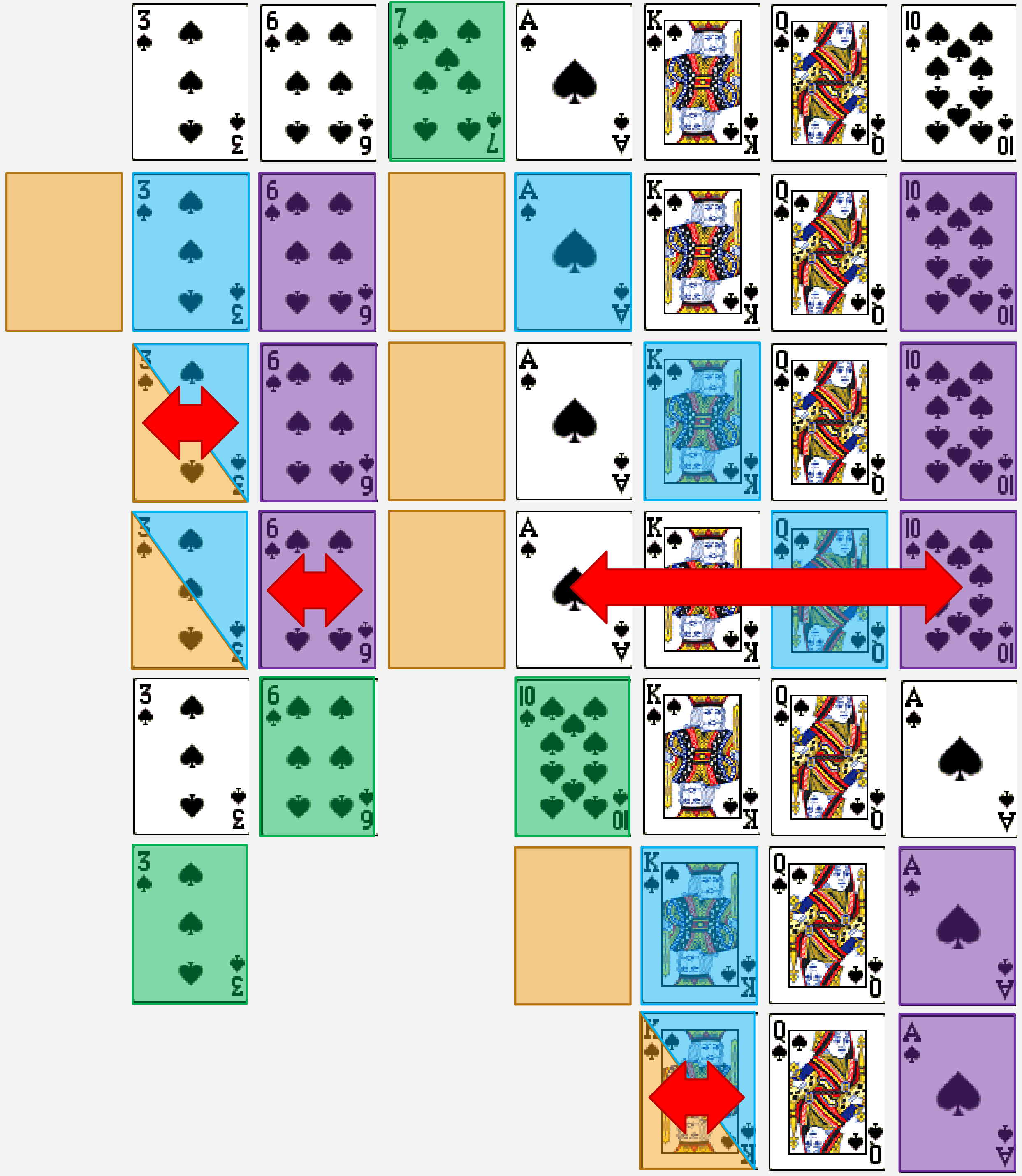


i

j

pivot

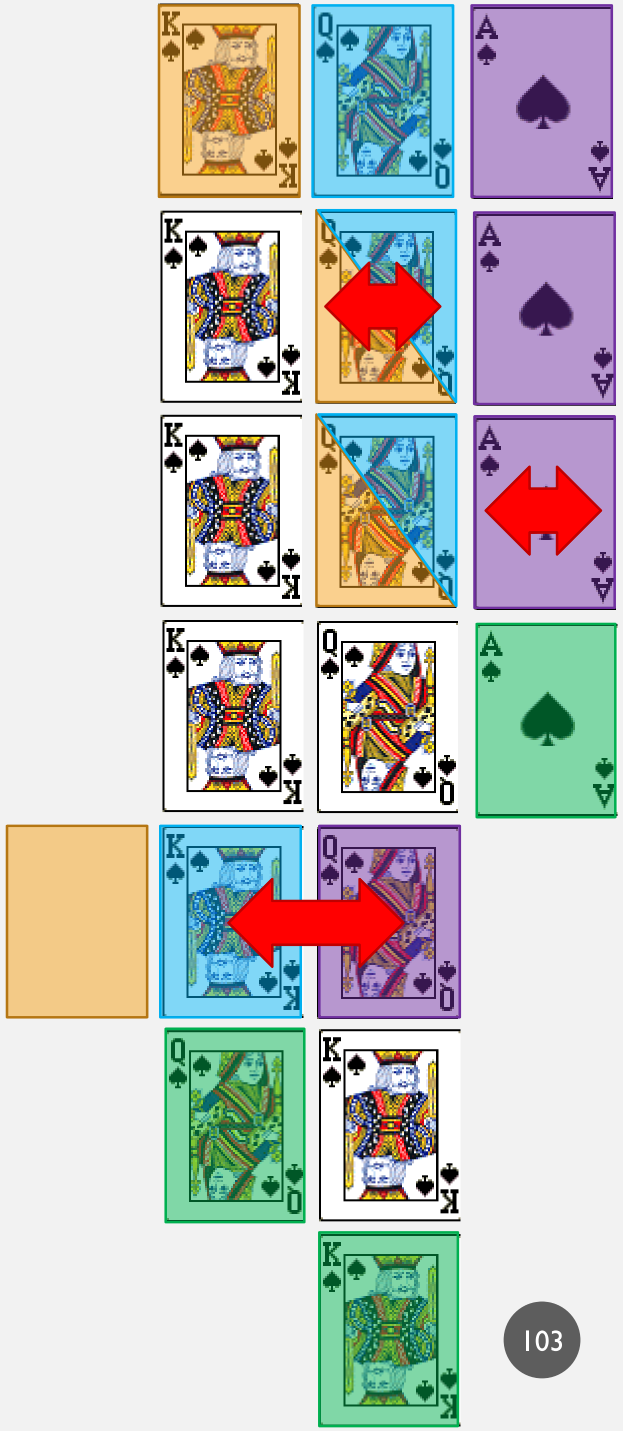




i

j

pivot



BUCKETSORT

BUCKETSORT

Assume your numbers evenly distributed across an interval

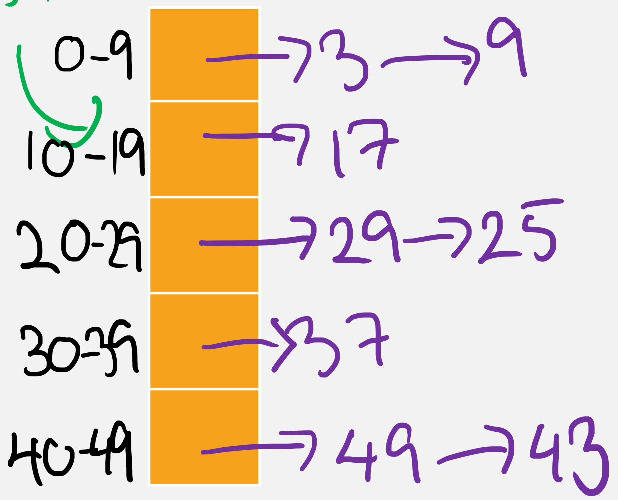
$$\underbrace{0 \text{ to } M}_{\text{can be generalized}}$$

29	25	3	49	9	37	17	43

0 – 49

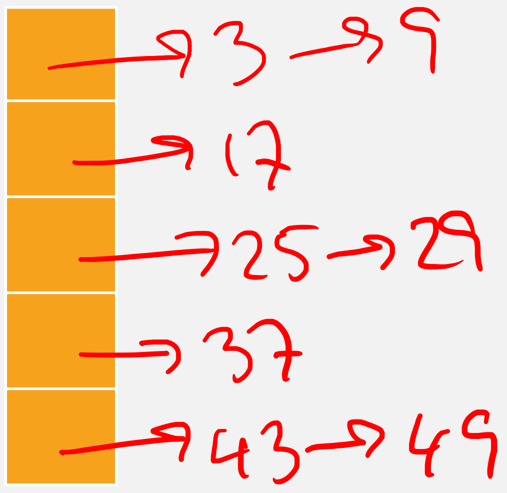
(1) make k buckets (k=5)

(2) fill numbers in buckets in $O(n)$ time



(3) sort each bucket using InsertionSort in $O\left(k\left(\frac{n}{k}\right)^2\right)$

$= O\left(\frac{n^2}{k}\right)$



(4) read each bucket and concatenate $O(k)$

3	9	17	25	29	37	43	49

$$O\left(n + \frac{n^2}{k} + k\right)$$

TIME COMPLEXITY

$$O\left(n + \frac{n^2}{k} + k\right)$$

let $k = O(n)$

$$O\left(n + \frac{n^2}{n} + n\right) = O(n + n + n) = O(n)$$

PSEUDOCODE

```
BucketSort(list, k):
    buckets = array of k empty lists
    M = maximum value in list
    for i = 0 to length(list):
        insert list[i] into buckets[floor(k*list[i]/(M+1))]
    for j = 0 to k:
        InsertionSort(buckets[j])
    return concatenation of buckets
```