
# Recursion

## Exercises

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

# Exercise 1

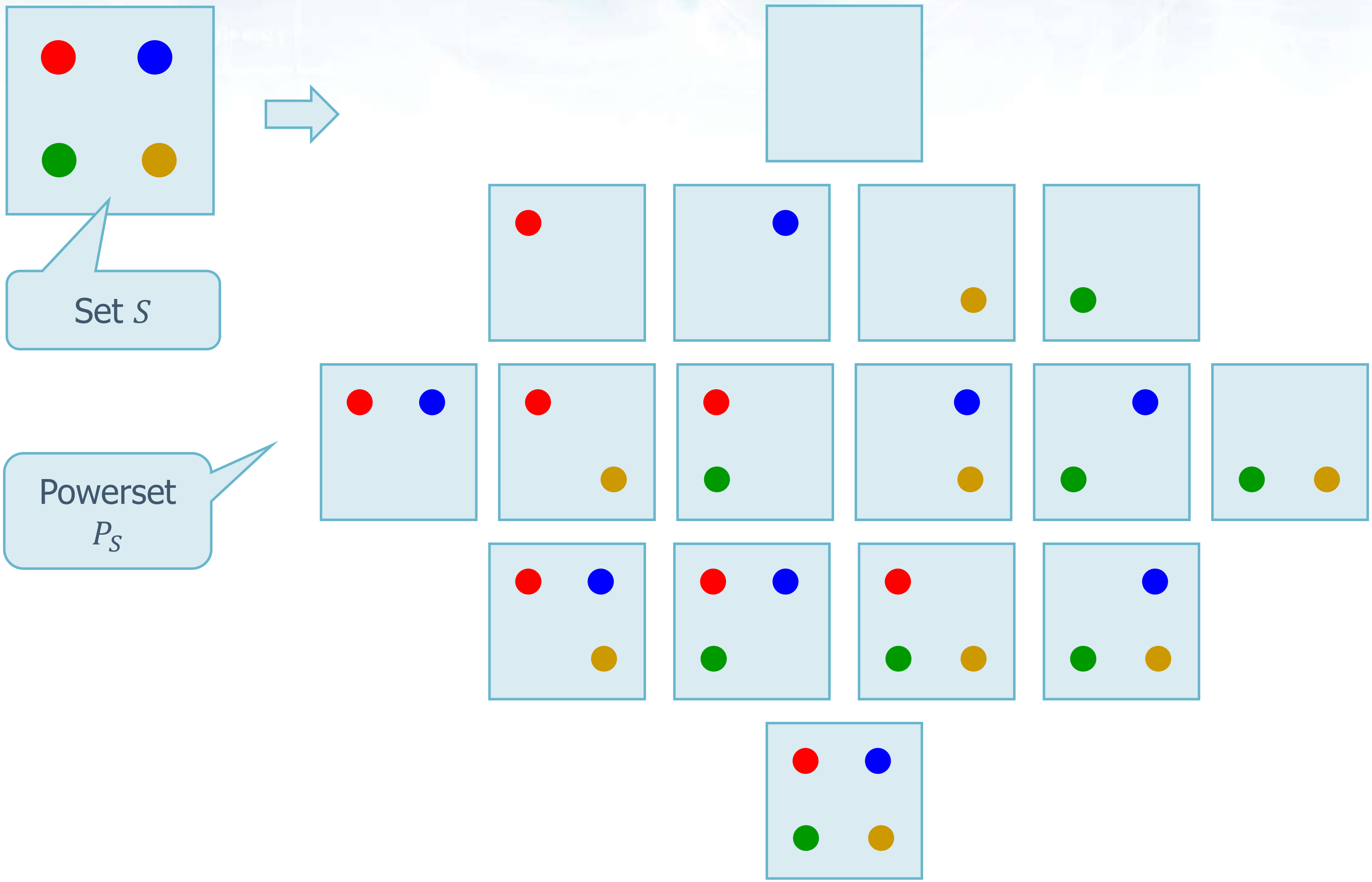
- The Powerset
  - Given a set $S$, its powerset $P_S$ is the set of all subsets of $S$, including the set $S$ itself and the empty set $\emptyset$
  - Example

$n = |S|$

$$S = \{1, 2, 3, 4\}$$
$$n = 4$$
$$P_S = \{\emptyset, 1, 2, 3, 4, 12, 13, 14, 23, 24, 34, 123, 124, 134, 234, 1234\}$$

- Problem
  - Given a set $S$ displays its powerset $P_S$

# Example



## Solution

- The powerset $P_S$ can be computed using 3 different models
  - Arrangements with repetitions
  - Simple combinations
    - Re-activating the procedure k times
  - Simple combinations
    - Adopting a divide and conquer strategy

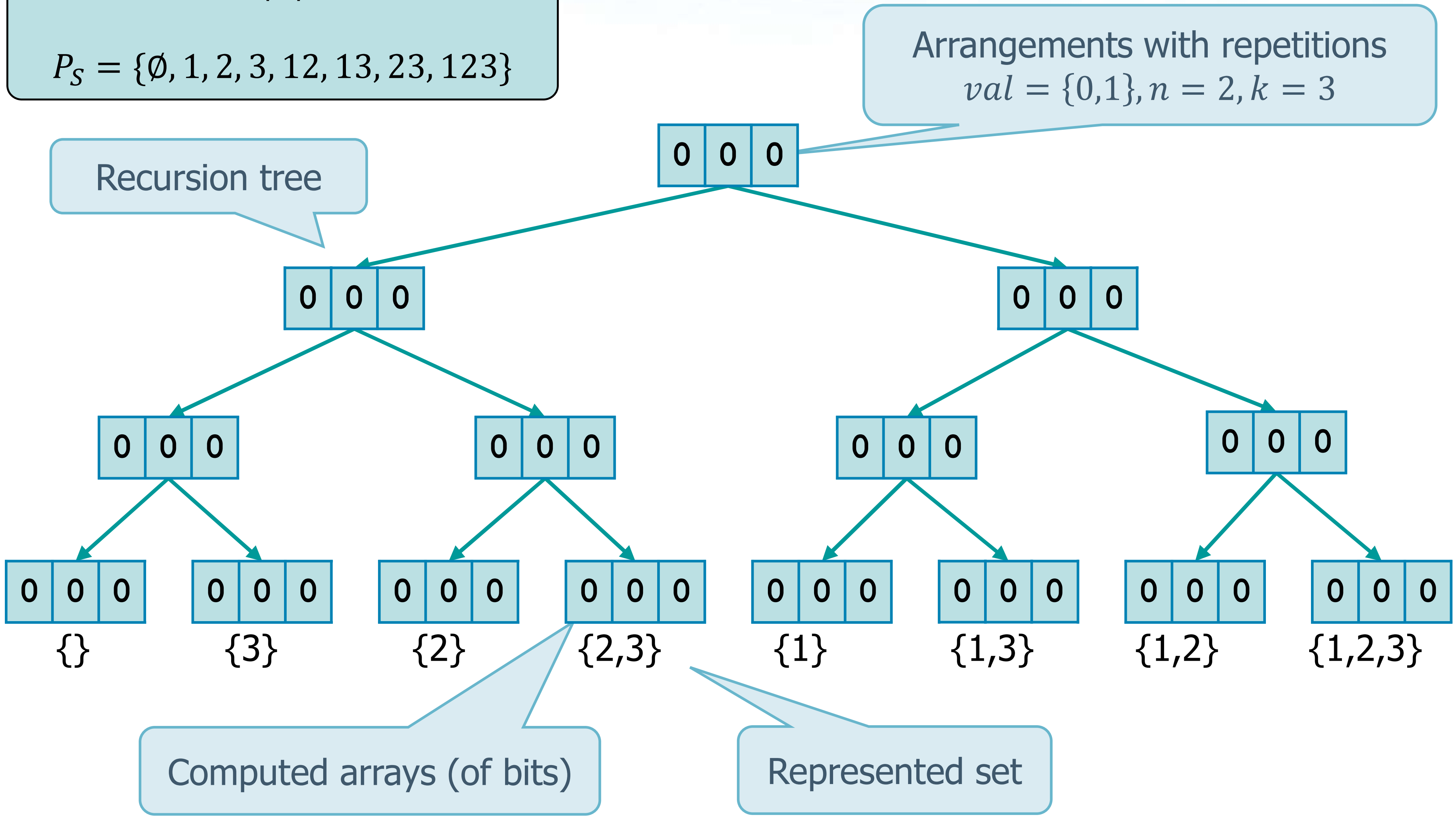
# Solution 1

- With the arrangements with repetition model the core idea is the following one
  - Each one of the $|S|$ objects of the set are paired with a binary digit
    - If the value of this digit is **0** the object is **not** inserted in the powerset
    - If the value of this digit is **1** the object **is** inserted in the powerset
  - Thus we have to arrange two values (0 and 1) on $\mathrm{n} = |S|$ positions
    - The computed array will tell which elements have to be selected within the powerset

# Solution 1

$$S = \{1, 2, 3\}$$
$$n = |S| = 3$$

$$P_S = \{\emptyset, 1, 2, 3, 12, 13, 23, 123\}$$



# Solution 1

- Each subset is represented by the sol array having k elements
  - Each element represent the set of possible choices, thus 0 and 1 (thus, n = 2 in the arrangements with repetition scheme)
  - The for loop is replaced by 2 explicit assignments
  - If
    - sol[pos]=0 if the pos-th object doesn't belong to the subset
    - sol[pos]=1 if the pos-th object belongs to the subset
  - 0 and 1 may appear several times in the same solution

## Solution 1

As arrangements with repetitions with the cycle substituted by two explicit calls

```
int powerset_1 (int *val, int *sol,
                int k, int count, int pos) {
  int j;
  if (pos >= k) {
    printf("{ \t");
    for (j=0; j<k; j++)
      if (sol[j]!=0)
        printf("%d \t", val[j]);
      printf("} \n");
    return count+1;
  }

  sol[pos] = 0;
  count = powerset_1(val,sol,k,count,pos+1);
  sol[pos] = 1;
  count = powerset_1(val,sol,k,count,pos+1);
  return count;
}
```

Termination condition

Iteration on 2 choices substituted by 2 explicit calls

0: No object pos in powerset

1: object pos in powerset

Recur on pos+1

# Solution 2

- Given the set S, we have to select k object from it varying k from 0 to n
  - We select 0 object, then we select 1 object (all possibility of 1 object), then we select 2 objects (all possibile pairs), etc.
  - Order does not matter (the powerset 123, 132, 312, etc., are equivalent)
- Thus the core idea is the following
  - Use simple combinations of $|S|$ distinct objects of class $k$, with incresing values of $k$ ($k = 0, \dots, |S|$)
  - In this case the recursive function generates the desired set (not an array of bits previously generated)

# Solution 2

- We must
  - Union of the empty set and
  - The powerset of size $1, 2, 3, \ldots, k$
- To compute the powerset, we use simple combinations of $k$ elements taken by groups of $n$

$$P_s = \{\emptyset\} \cup \bigcup_{n\_1}^{k} \binom{k}{n}$$

- A wrapper function takes care of the union of empty set (not generated as a combination) and of iterating the recursive call to the function computing combinations

## Solution 2

Wrapper

```
int powerset_2 (int *val, int *sol, int n){
  int count, k;

  count = 0;
  for (k=1; k<=n; k++){
     count += powerset_2_r (val,sol,n,k,0,0);
  }

  return count;
}
```

Empty set

Initially start = 0 (initial choice)

Initially pos = 0 (recursion level)

Iteration on recursive calls (simple combinations)

# Solution 2

Simple combination

```
int powerset_2_r (int *val, int *sol,
                  int n, int k, int start, int pos) {
  int count = 0, i;

  if (pos >= k){
    printf("{ ");
    for (i=0; i<k; i++)
      printf("%d ", sol[i]);
    printf(" }\n");
    return 1;
  }
  for (i=start; i<n; i++){
    sol[pos] = val[i];
    count += powerset_2_r(val,sol,n,k,i+1,pos+1);
  }
  return count;
}
```

Print-out desired solution (not an array of bits)

## Solution 3

- Simple combinations can be used to generate a powerset of $k$ objects extracted from the set $S$
  - Instead of re-calling simple combinations over and over again with increasing value of $k$ we may use a divide and conquer approach
  - The divide and conquer approach is based on the following formulation

Terminal case:
empty set

$$if\ k = 0\ then\ P_{S_k} = \{\emptyset\}$$

$$if\ k > 0\ then\ P_{S_k} = \{P_{S_k} \cup S_k\} \cup \{P_{S_{k-1}}\}$$

Recursive case:
powerset for $k-1$ elements union either the $k$-th element $S_k$ or the empty set

## Solution 3

- In the simple combinations function
  - We generate 2 distinct recursive branches
    - The first one include the current element in the solution
    - The second does not include it
- In sol we directly store the element, not a flag to indicate its presence/absence
- The value of index start is used to exclude symmetrical solutions
- The return value count represents the total number of sets

# Solution 3

```
int powerset_3(int *val, int *sol,
               int k, int start, int count, int pos) {
  int i;
  if (start >= k) {
    for (i=0; i<pos; i++)
      printf("%d ", sol[i]);
    printf("\n");
    return count+1;
  }
  for (i=start; i<k; i++) {
    sol[pos] = val[i];
    count = powerset_3(val,sol,k,i+1,count,pos+1);
  }
  count = powerset_3(val,sol,k,k,count,pos);
  return count;
}
```

For all elements from start onwards

Add $S_k$ and recur

Do not add $S_k$ and recur

# Exercise 2

- Partition of a set
  - Given a set $S$ of $|S|$ elements, a collection $S = \{S_i\}$ of non empty blocks forms a partition only iff both the following conditions hold
    - Blocks are pairwise disjoint
    - The union of those blocks is $S$

$$\forall S_i, S_j \in S \quad with\ i \neq j \quad then\ S_i \cap S_j = \varnothing$$
$$S = \cup_i S_i$$

We use
$n$ to indicate the number of elements in $S$ (i.e., $|S|$ )
$k$ to indicate the number of blocks we want in our partitions

# Exercise 2

- The number of blocks $k$ ranges
  - From 1, i.e., the block coincides with the set $S$
  - To $n$, i.e., each block contains only 1 element of $S$

$$\forall S_i, S_j \in S \quad with\ i \neq j \quad then\ S_i \cap S_j = \varnothing$$
$$S = \cup_i S_i$$

- Problem
  - Given a set $S$ find its partition
    - Subproblem A: With a specific number of block $k$
    - Subproblem B: With all possible blocks $k$

# Example

- Given the set $S = \{ A, B, C, D \}$
  generate all possibile partitions with $1, 2, 3, 4$ blocks

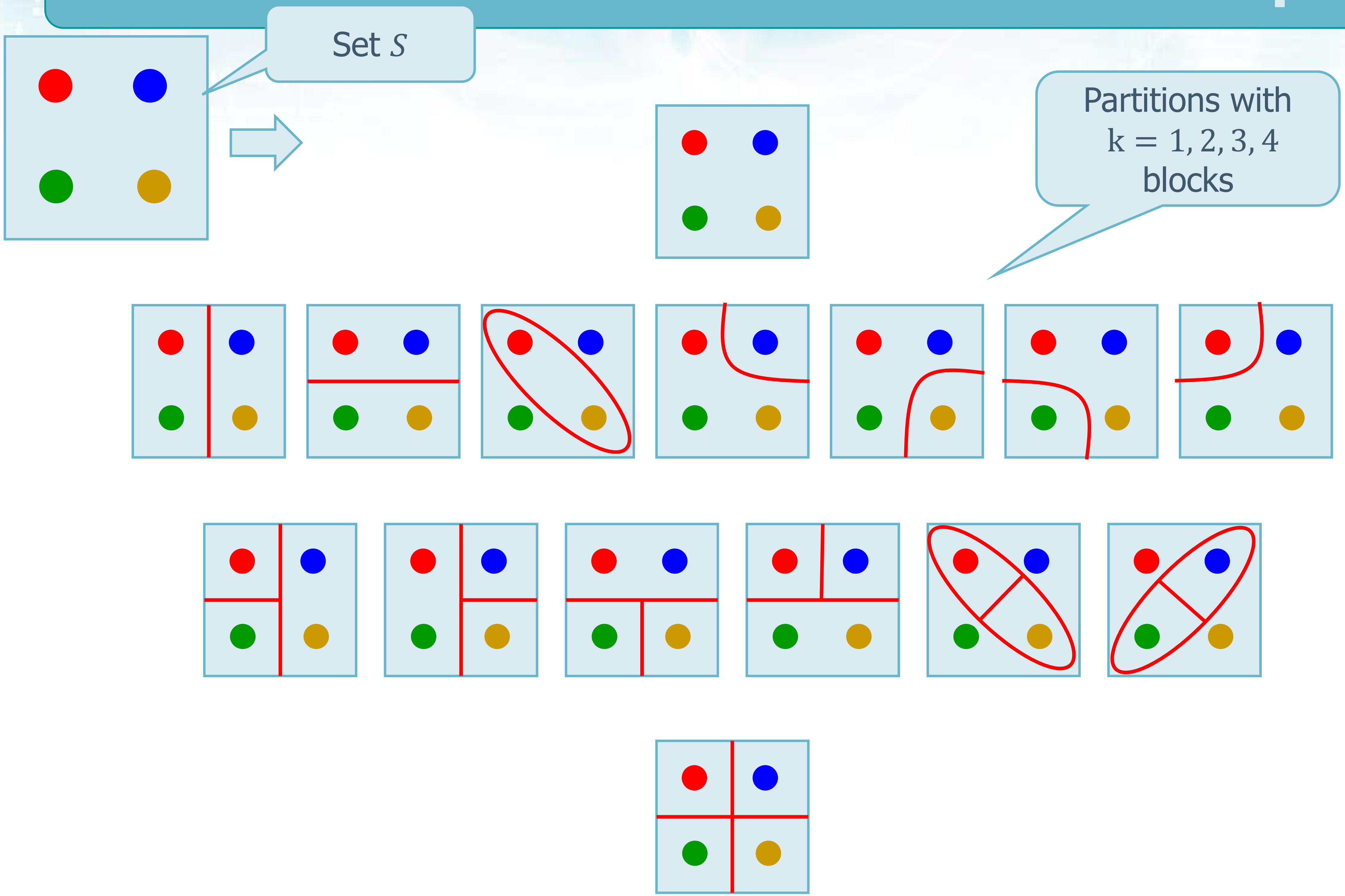
| $k = 1$ | $k = 2$ | $k = 3$ | $k = 4$ |
|---|---|---|---|
| 1 partition | 7 partitions | 6 partitions | 1 partition |
| {A, B, C, D} | {A, C}, {B, D}<br>{A, B}, {C, D}<br>{A, D}, {B, C}<br>{A, B, C}, {D}<br>{A, B, D}, {C}<br>{A, C, D}, {B}<br>{A}, {B, C, D} | {A, B}, {C}, {D}<br>{A, C}, {B}, {D}<br>{A}, {B ,C}, {D}<br>{A, D}, {B}, {C}<br>{A}, {B, D}, {C}<br>{A}, {B}, {C, D} | {A}, {B}, {C}, {D} |

partition

block

{A, B, C}, {D} AND {D}, {C, B, A} are equivalent.
The order of the blocks and of the elements within each block doesn't matter

# Example



## Solution

- To represent a partitions we can
  - Given the element, identify its block
  - Given the block, list its elements
- The first approach is simpler, as it works on an array of integers and not on lists

$$S = \{ A, B, C, D \}$$
$$k = 4\ blocks$$

$$Partition_1 = \{A, B, C, D\}$$
$$Partition_2 = \{A, D\}, \{B\}, \{C\}$$
$$Partition_3 = \{A, B\}, \{C, D\}$$

| A | B | C | D |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 2 | 0 |
| 0 | 0 | 1 | 1 |

A, B $\in$ partition 0

C, D $\in$ partition 1

# Solution

- Given the set $S$ of cardinality $n = |S|$, it is possibile to find
  - All partitions in exactly $k$ blocks, where $k$ is a constant value
    - This problem can be solved with arrangements with repetitions
  - All partitions in all blocks, i.e., with $k$ ranges between 1 and $k$
    - This problem can be solved with arrangements with repetitions re-called for every value of $k$ or with the Er's algorithm (1987)

We present only on the first problem

## Solution

- To find all partitions in exactly $k$ blocks, we can use arrangements with repetitions
  - This is a generalization of the powerset problem (solution 1)
  - Instead of arranging only two values ($0$ and $1$) on $n$ positions we arrange $k$ values
  - Each value is (from $0$ to $k - 1$) will indicate the partition

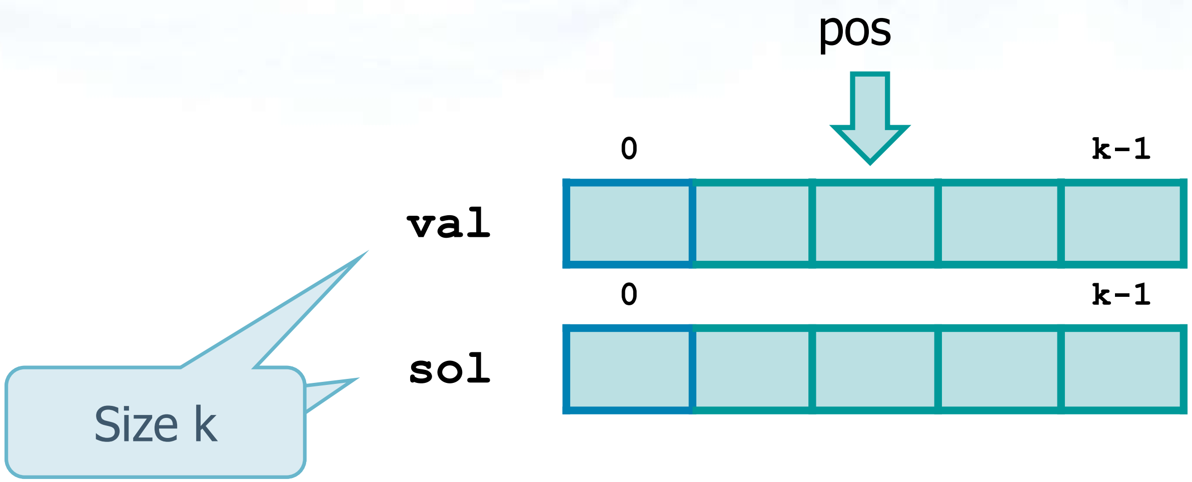
# Solution

- Limitations
  - We generate all numbers base $k$, thus we generate duplicates
    - For example
      - With $S = \{A, B, C, D\}$ and $k = 2$ blocks, we generate not only $\{A, B, C\}, \{D\}$ but also $\{A, B, C\}, \{D\}$
      - With $S = \{A, B, C, D\}$ and $k = 3$ blocks we generate not only $\{A, B\}, \{C\}, \{D\}$ but also $\{A, B\}, \{D\}, \{C\}$ and $\{C\}, \{A, B\}, \{D\}$ and $\{D\}, \{A, B\}, \{C\}$ and $\{C\}, \{D\}, \{A, B\}$ and $\{D\}, \{C\}, \{A, B\}$
  - We only check not to have empty blocks
    - For example if we want to have $k = 3$ blocks, we chack not to have an empy block otherwise we sould have $k = 3$ block not 3

## Solution

- The number of objects stored in array val is n
  - The number of decisions to take is **n**, thus array **sol** contains **n** cells
  - The number of possible choices for each object is the number of blocks, that ranges from **1** to **k**
  - Each block is identified by an index **i** in the range from **0** to **k-1**
  - **sol[pos]** contains the index i of the block to which the current object of index pos belongs

# Solution




Don't forget to check for NULL

```
val = malloc (k*sizeof(int));
sol = malloc (k*sizeof(int));
```

# Solution

```
void arr_rep(int *val, int *sol,
             int n, int k, int pos) {
  int i, j, t, ok=1, *occ;

  if (pos >= n) {
    check_and_display(sol,n,k);
  }

  for (i=0; i<k; i++) {
    sol[pos] = i;
    arr_rep(val,sol,n,k,pos+1);
  }

  return;
}
```

Occurrence check

Recur:
Simple arrangements

# Solution

```
void check_and_display(int *sol, int n, int k) {
  int i, j, end, *occ;

  occ = calloc (k, sizeof (int));
  if (occ == NULL) { ... }
  for (j=0; j<n; j++) occ[sol[j]]++;
  for (end=j=0; j<k && end==0; j++)
    if (occ[j]==0) end = 1;
  free (occ);
  if (end==1) return;
  fprintf (stdout, "Partition: ");
  for (i=0; i<k; i++) {
    printf("{ ");
    for (j=0; j<n; j++)
      if (solution[j]==i) printf("%d ", value[j]);
    printf("}  ");
  }
  printf("\n");
  return;
}
```

Block occurrence array

Occurrence computation

Occurrence check

Discard solution with an empty block

Print solution

## Consideration

- The total number of partitions of a set $S$ of $n$ objects is given by Bell's numbers
  - Bell's number are defined by the following recurrence equation

$$B_0 = 1$$

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

$$B_0 = 1, B_1 = 1, B_2 = 2, B_3 = 5, B_4 = 15, B_5 = 52, \ldots$$

  - Their search space is not modelled in terms of combinatorics