Recursion

Combinatorics

Paolo Camurati and Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Definition

- Combinatorics is a topic of the course in Mathematical Methods for Engineering
- Combinatorics
  - **Count** on how many subsets of a given set a property holds
  - **Determines** in how many ways the elements of a same group may be associated according to predefined rules
- In problem-solving we need to enumerate the ways, not only to count them

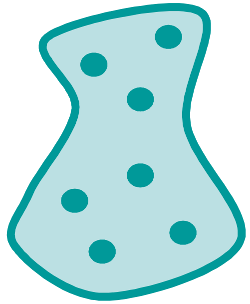
Model

- The search space may modelled as
  - Addition and multiplication principles
  - Simple **arrangements**
  - **Arrangements** with repetitions
  - Simple **permutations**
  - **Permutations** with repetition
  - Simple **combinations**
  - **Combinations** with repetitions
  - Powerset
  - Partitions

We are going to analyze an implementation frame/scheme for each one of these models

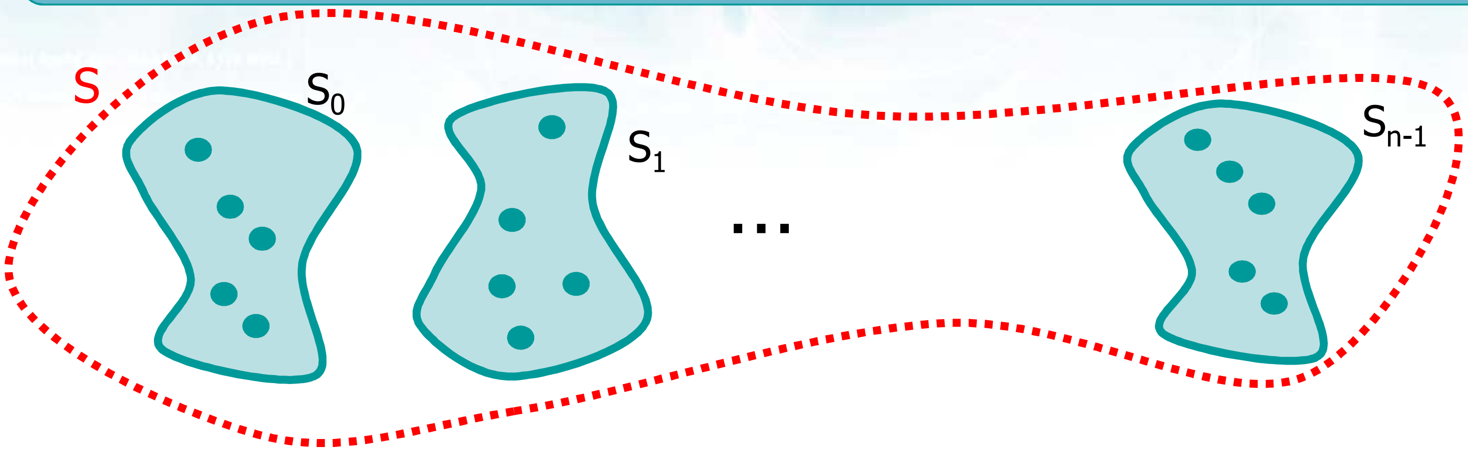
Grouping criteria

- Given a group S of n elements, we can select k objects keeping into account



  - Unicity
    - Are all elements in group S distinct?
    - Is thus S a set? Or is it a multiset?
  - Ordering
    - No matter a reordering, are 2 configurations the same?
  - Repetitions
    - May the same object of a group be used several times within the same grouping?

Basic principle: Addition

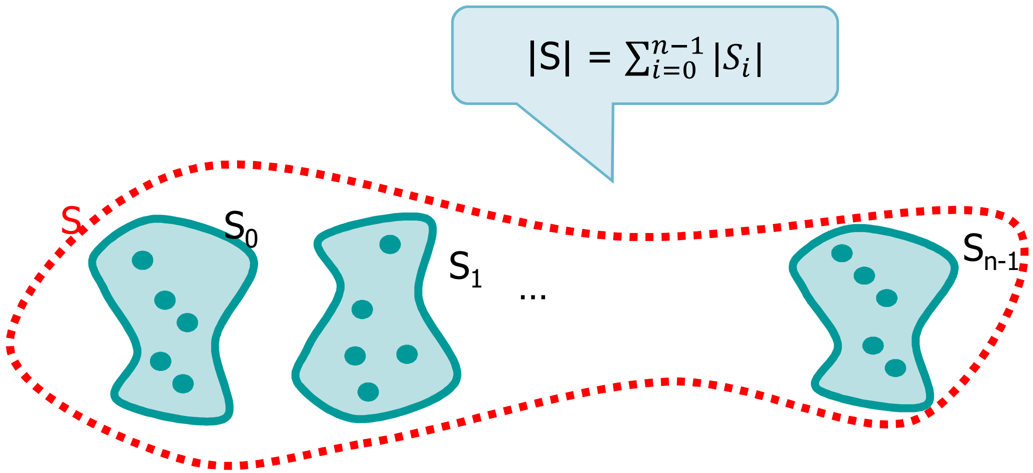


- If a set S of objects is partitioned in pair-wise disjoint subsets $\{S_0, \ldots, S_{n-1}\}$ such that
  - $S = S_0 \cup S_1 \cup \ldots S_{n-1}$

  and
  - $\forall\ i \neq j\ ,\ S_i \cap S_j = \phi$

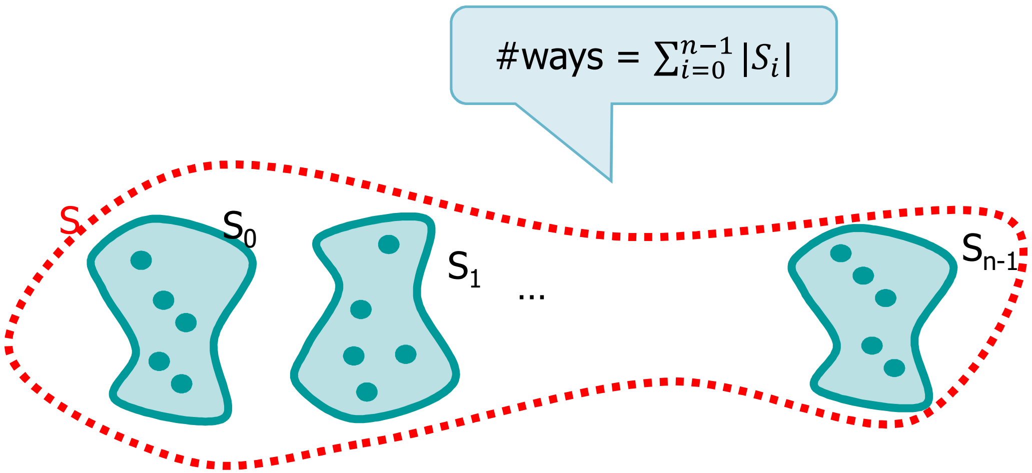
Basic principle: Addition

- Definition
  - The number of objects in S may be determined adding the number of objects of each of the sets $\{S_0, \ldots, S_{n-1}\}$
    - $|S| = \sum_{i=0}^{n-1} |S_i|$



Basic principle: Addition

- Alternative definition
  - If an object can be selected in $|S_0|$ ways from S_0, in $|S_1|$ ways from S_1, …, in $|S_{n-1}|$ ways from S_{n-1}
  - Then, selecting an object from any of the n groups may be performed in a number of ways equal to
    - $\#ways = \sum_{i=0}^{n-1} |S_i|$



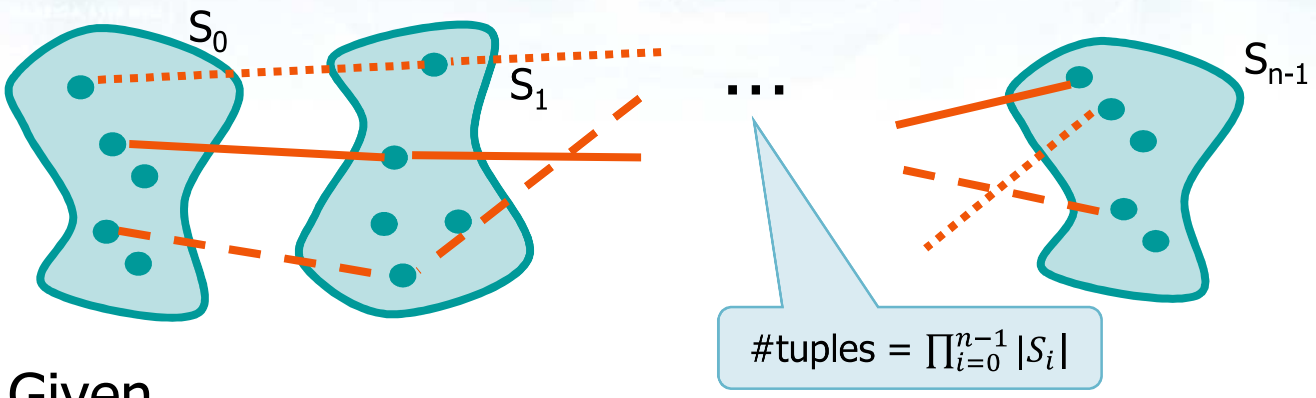
Example

- In an university there are
  - 4 Computer Science courses

  and
  - 5 Mathematics courses
- A student can select just one course
- In how many ways can a student choose?

- Solution
  - Disjoint sets ⇒
  - Model: Principle of addition
  - Number of choices = 4 + 5 = 9

Basic principle: Multiplication



- Given
  - n sets S_i ($0 \le i < n$), each one of cardinality $|S_i|$
  - The number of ordered t-uples (s_0 … s_{n-1}) with
    - $s_0 \in S_0$ … $s_{n-1} \in S_{n-1}$

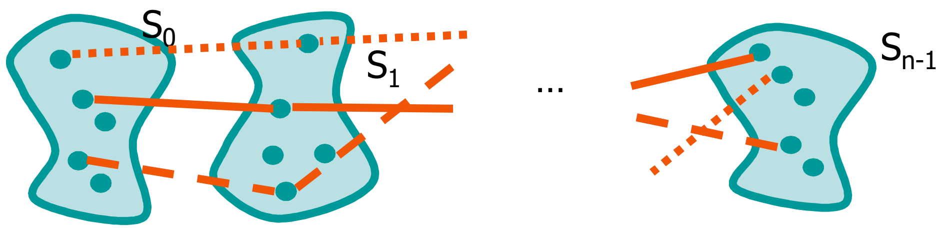
  is
    - #tuples = $\prod_{i=0}^{n-1} |S_i|$

Basic principle: Multiplication

- Alternative definition
  - If an object can be selected in $|S_0|$ ways from S_0, in $|S_1|$ ways from S_1, …, in $|S_{n-1}|$ ways from S_{n-1}
  - Then, the choice of a t-uple of objects (s_0 … s_{n-1}) can be done in
    - #tuples $= n_0 \cdot n_1 \cdot n_2 \cdot \dots \cdot n_{n-1} = \prod_{i=0}^{n-1} |n_i|$

    ways

#tuples = $\prod_{i=0}^{n-1} |n_i|$



Example

- In a restaurant a menu is served made of
  - Appetizers, 2 overall
  - First course, 3 overall
  - Second course, 2 overall
- Any customer can choose 1 appetizer, 1 first course, and 1 second course
- Problem
  - How many different menus can the restaurant offer?
  - How are these menu composed?

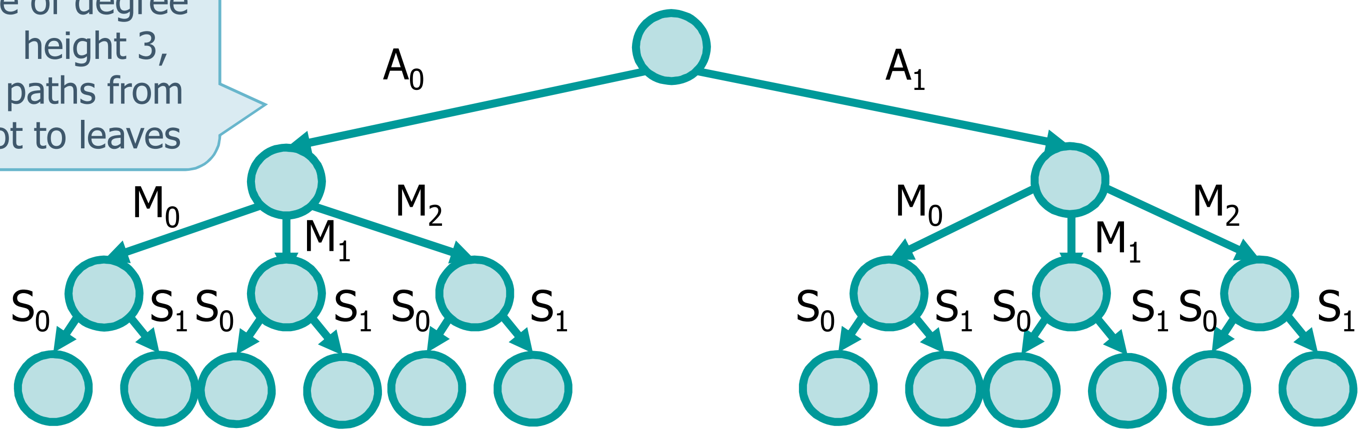
We want to count the number of solution and generate those solutions

Solution

2 appetizers (A_0, A_1)
3 main courses (M_0, M_1, M_2)
2 second courses (S_0, S_1)
(n=k=3)

- Model
  - Principle of multiplication
  - #menus = 2 x 3 x 2 = 12
    - menus = { (A_0,M_0,S_0), (A_0,M_0,S_1), (A_0,M_1,S_0), (A_0,M_1,S_1), (A_0,M_2,S_0), (A_0,M_2,S_1),$(A_1,M0,S_0)$, (A_1,M_0,S_1), (A_1,M_1,S_0), (A_1,M_1,S_1), (A_1,M_2,S_0), (A_1,M_2,S_1) }

Tree of degree 3, height 3, 12 paths from root to leaves



Solution

- Choices are made in sequence
  - They are represented by a tree
  - The number of choices
    - Is fixed for a level
    - Varies from level to level
  - Nodes have a number of children that varies according to the level
    - Each one of the children is one of the choices at that level
    - The maximum number of children determines the degree of the tree
  - The tree's height is **n** (the number of groups)

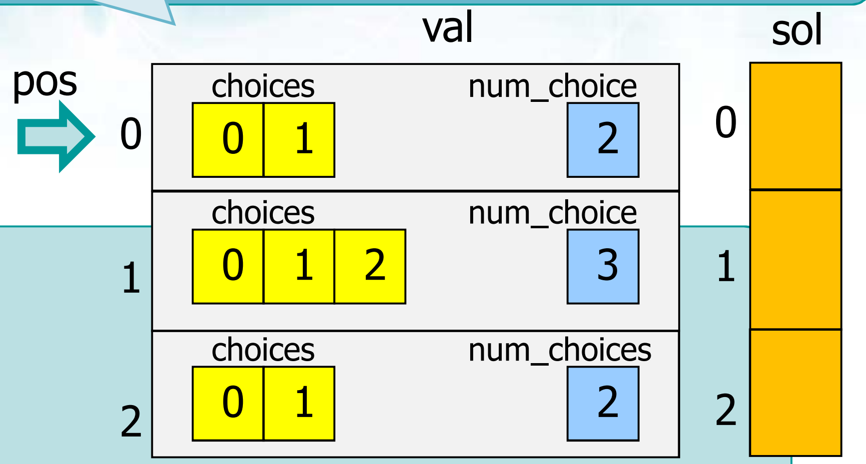
Solution

- Given the recursion tree, solutions are the labels of the edges along each path from root to node
  - The goal is to enumerate all solutions, searching their space
    - All solutions are valid
  - Each new recursive call increases the size of the solution
    - The total number of recursive calls along each path is equal to n
  - Termination
    - Size of current solution equals final desired size n

Implementation

Referring to the example

The check for NULL is missing




```
typedef struct {
  int *choices;
  int num_choice;
} Level;

val = malloc(n*sizeof(Level));

for (i=0; i<n; i++)
  val[i].choices =
    malloc(val[i].n_choice*sizeof(int));

sol = malloc(n*sizeof(int));
```

Implementation

- As far as the data-base is concerned
  - There is a 1:1 matching between choices and a (possibly non contiguous) subset of integers
  - Possible choices are stored in array **val** of size **n** containing structures of type **Level**
    - Each structure contains
      - An integer field **num_choice** for the number of choices at that level
      - An array ***choices** of **num_choice** integers storing the available choices
  - A solution is represented as an array **sol** of **n** elements that stores the choices at each step

Implementation

- As far as the recursive function is concerned
  - At each step index **pos** indicates the size of the partial solution
    - If **pos>=n** a solution has been found
  - The recursive step iterates on possible choices for the current value of **pos**
    - The contents of **sol[pos]** is taken from **val[pos].choices[i]** extending each time the solution's size by 1 and recurs on the pos+1-th choice
  - Variable **count** is the integer return value for the recursive function and counts the number of solutions

pos is the recursion level (level)

Implementation

```
int mult_princ (Level *val, int *sol,
                int n, int count, int pos) {
  int i;

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

Termination condition

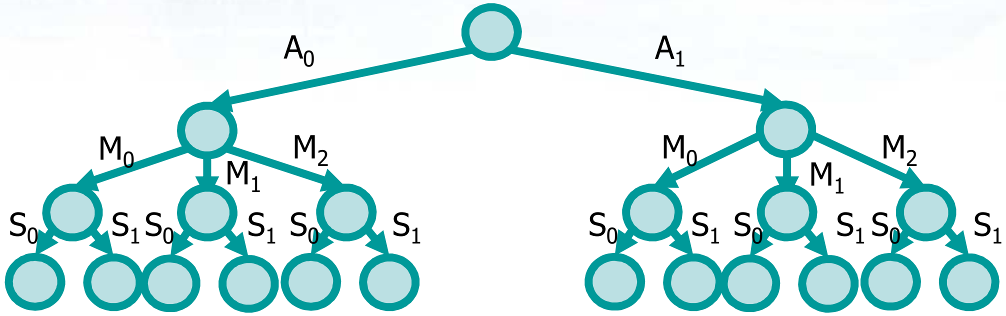
Iteration on n choices

Choose

Recur

Passing pos+1 does not modify pos at this recursione level

Implementation




```
int mult_princ (...) {
  int i;
  if (pos >= n) {
    print ...
    return count+1;
  }
  for (i=0; i<val[pos].num_choice; i++) {
    sol[pos] = val[pos].choices[i];
    count = mult_princ (...);
  }
  return count;
}
```

Simple arrangements

Simple means no repetitions

Distinct means it is a set

- A simple arrangement $D_{n,k}$ of n distinct objects of class k is an ordered subset composed by k out of n objects ($0 \leq k \leq n$)

Class k means size k (set taken k by k)

Order matters

- The number of simple arrangements of n objects k by k is
  - $D_{n,k} = n \cdot (n-1) \cdot \ldots\ldots \cdot (n-k+1) = \frac{n!}{(n-k)!}$

I select an object out of n, then I select an object out of the n-1 remaining, etc.

Simple arrangements

- Note that
  - In simple arrangements objects are
    - Distinct $\Rightarrow$ the group is a set
    - Ordered $\Rightarrow$ order matters
    - Simple $\Rightarrow$ in each group there are exactly k non repeated objects
  - Two groupings differ
    - Either because there is at least a different element
    - Or because the ordering is different

Example

Positional representation: order matters!

k = 2

- How many and which are the numbers on 2 distinct digits composed with digits 4, 9, 1 and 0?

No repeated digits

val = { 4, 9, 1, 0 }

n = 4

- Model
  - Simple arrangements
  - $D_{4,\,2} = \frac{n!}{(n-k)!} = \frac{4!}{(4-2)!} = 4 \cdot 3 = 12$
- Solution
  - Numbers = { 49, 41, 40, 94, 91, 90, 14, 19, 10, 04, 09, 01 }

Example

Positional representation: order matters!

k = 2

- How many strings of 2 characters can be formed selecting chars within the group of 5 vowels {A, E, I, O, U}?

val = { A, E, I, O, U }

No repeated digits

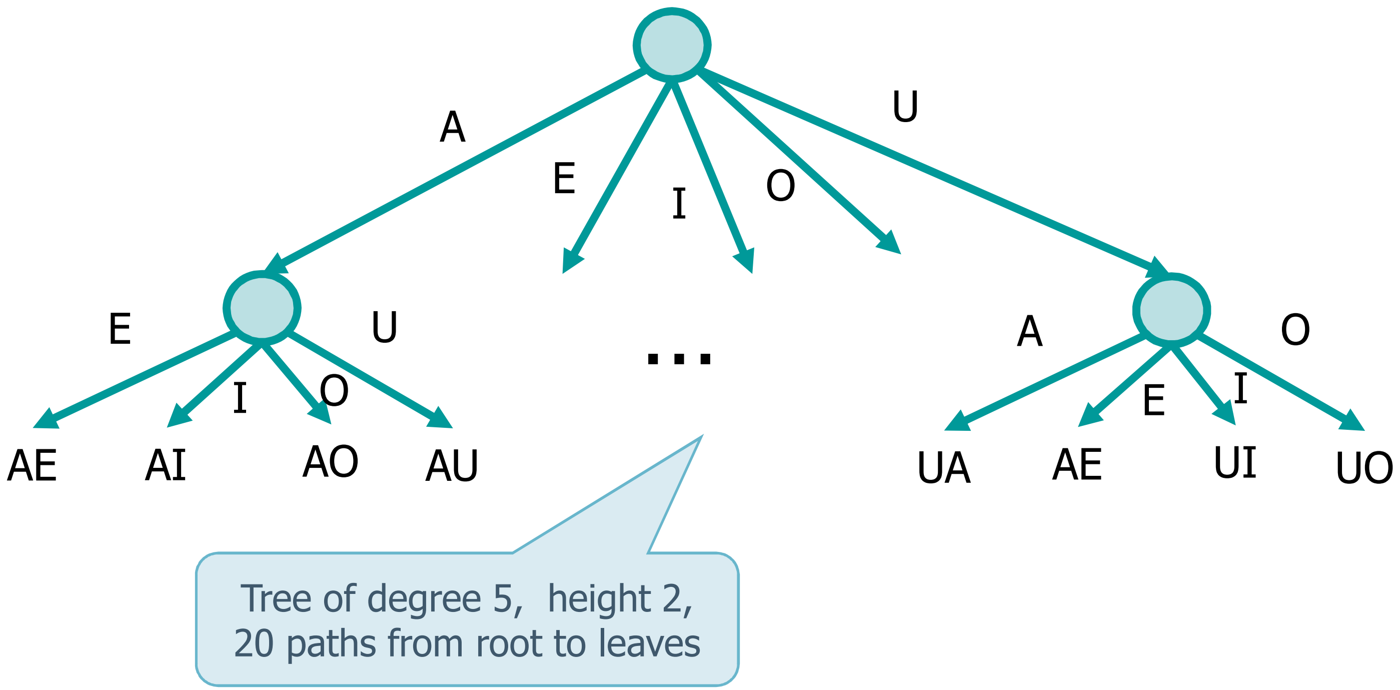
n = 5

- Model
  - Simple arrangements
  - $D_{5,\,2} = \frac{n!}{(n-k)!} = \frac{5!}{(5-2)!} = 5 \cdot 4 = 20$
- Solution
  - Strings = { AE, AI, AO, AU, EA, EI, EO, EU, IA, IE, IO, IU, OA, OE, OI, OU, UA, UE, UI, UO }

Example

Solution

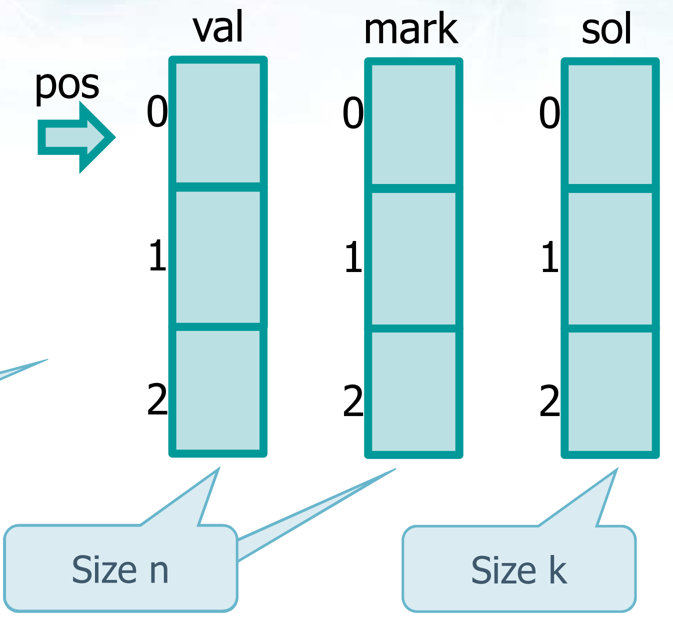
- Strings = { AE, AI, AO, AU, EA, EI, EO, EU, IA, IE, IO, IU, OA, OE, OI, OU, UA, UE, UI, UO }



Implementation

As for the multiplication principle with the same set to which one element is extracted, recursion level after recursion level

As the set is the same, the array **val** become an array of flags **mark**




```
val = malloc (n * sizeof(int));
mark = malloc (n * sizeof(int));

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

Implementation

- In order not to generate repeated elements
  - An array **mark** records already taken elements
    - **mark[i]=0** implies that i-th element not yet taken, else 1
    - The cardinality of mark equals the number of elements in val (all distinct, being a set)
  - While choosing
    - The i-th element is taken only if **mark[i]==0**, **mark[i]** is assigned with 1
  - While backtracking
    - **mark[i]** is assigned with 0
  - Array **count** records the number of solutions

Implementation

```
int arr (int *val, int *sol, int *mark,
         int n, int k, int count, int pos){
  int i;

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

Termination condition

Iteration on n choices

Mark and choose

Recur

Unmark

Arrangements with repetitions

Repetitions

Set

- An arrangement with repetitions $D'_{n,k}$ of n distinct objects of class k (k by k) is an ordered_subset composed of k out of n objects ($0 \le k$) each of whom may be taken up to k times

Order matters

- The number of arrangements with repetitions of n objects taken k by k is
  - $D'_{n,k} = n \cdot n \cdot \ldots\ldots \cdot n = n^k$

I select an object out of n, then I select an object out of n, etc.

Arrangements with repetitions

- Note that
  - Arrangements with repetitions are
    - Distinct ⇒ the group is a set
    - Ordered ⇒ order matters
    - As "simple" is not mentioned ⇒ in every grouping the same object can occur repeatedly at most k times
      - k may be > n
  - Two groupings differ if one of them
    - Contains at least an object that doesn't occur in the other group or
    - Objects occur in different orders or
    - Objects that occur in one grouping occur also in the other one but are repeated a different number of times

Example

Positional representation: order matters!

n = 4

- How many binary numbers can be created with 4 bits?

k=2, val ={ 0, 1}, repeated digits

- Model
  - Each bit can take either value 0 or 1
  - Arrangements with repetitions
    - $D'_{2,\,4} = 2^4 = 16$
- Solution
  - Numbers = { 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 }

Example

Positional representation: order matters!

k = 2

- How many strings of 2 characters can be formed selecting chars with repetitions within the group of 5 vowels {A, E, I, O, U}?

n = 5, val = {A, E, I, O , U }

Repeated digits

- Model
  - Arrangements with repetitions
  - $D'_{5,2} = n^k = 5^2 = 25$
- Solution
  - Strings = { **AA**, AE, AI, AO, AU, EA, **EE**, EI, EO, EU, IA, IE, **II**, IO, IU, OA, OE, OI, **OO**, OU, UA, UE, UI, UO, **UU** }

Solution

- Each element can be repeated up to **k** times
  - There in no bound on **k** imposed by **n**
  - For each position we enumerate all possible choices
  - Array **count** stores the number of solutions

As the multiplication principle but extracting from the same set over and over again

As simple arrangements with **NO mark** array, as all elements can be selected at any level

Implementation

As the multiplication principle but with the same set

```
int arr_rep (int *val, int *sol,
             int n, int k, int count, int pos) {
  int i;

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

Termination condition

Iteration on n choices

Choose

Recur

Simple Permutations

As simple arrangements with **k==n** (k does not exist)

- A simple arrangement $D_{n,\ n}$ of n distinct objects of class n (n by n) is a simple permutation P_n
  - A simple permutation is an ordered subset made of n objects

No repetitions

Order matters

Set

- The number of simple permutations of n objects is
  - $P_n = D_{n,\ n} = n \cdot (n-1) \cdot \ldots\ldots \cdot (n-n+1) = n!$

Simple Permutations

- Note that
  - Simple permutation
    - Distinct ⇒ the group is a set
    - Ordered ⇒ order matters
    - Simple ⇒ in each grouping there are exactly n non repeated objects
  - Two groups differ because
    - The elements are the same, but they appear in a different order

Example

Positional representation: order matters!

val = { 1, 2, 3 }

- Given a set **val** of 3 integers, generate all possible numbers containing these 3 digits once

No repetition

n = 3

- Solution
  - Model
    - Simple permutations
  - The number of permutations is
    - $P_3 = n! = 3! = 6$
  - Permutations = { 123, 132, 213, 312, 231, 321 }

Example

Positional representation: order matters!

val = { O, R, A }

- How many and which are the anagrams of the string ORA (string of 3 distinct letters)?

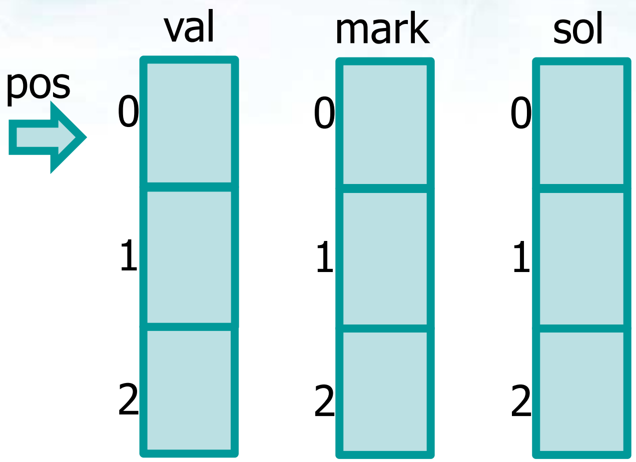
No repetition

n = 3

- Solution
  - Model
    - Simple permutations
  - The number of permutations is
    - $P_3 = n! = 3! = 6$
  - Anagrams = { ORA, OAR, ROA, AOR, RAO, ARO }

Implementation

As simple arrangements with **k==n** (we select n elements out of n)



Don’t forget to check for NULL

Size n

```
val = malloc (n * sizeof(int));
sol = malloc (n * sizeof(int));
mark = malloc (n * sizeof(int));
```

Solution

- In order not to generate repeated elements
  - An array **mark** records already taken elements
    - **mark[i]=0** implies that the i-th element not yet taken, else 1
    - The cardinality of **mark** equals the number of elements in **val** (all distinct, being a set)
  - While choosing
    - The i-th element is taken only if **mark[i]==0**, **mark[i]** is assigned with 1
  - During backtrack
    - **mark[i]** is assigned with 0
  - **Count** stores the number of solutions

Implementation

As simple arrangements with **k==n**

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

Termination condition

Iteration on n choices

Mark and choose

Unmark

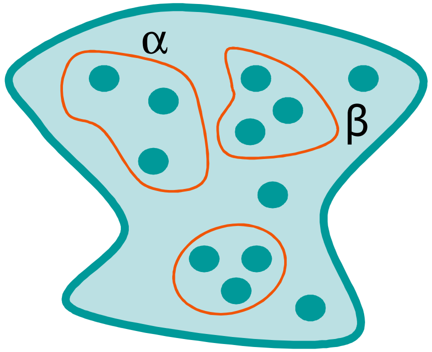
Recur

Permutations with repetitions

Repeated elements

Order matters

- Given a set (multiset) of n objects among which
  - α objects are identical
  - β objects are identical
  - etc.



the number of distinct permutations with repeated objects is

- $P_n^{(\alpha,\,\beta,\,..)} = \frac{n!}{(\alpha!\cdot\beta!\,\dots)}$

From permutation
$P_n = n!$
divided by the permutations of the repeated objects

Permutations with repetitions

- Note that
  - Permutation with repetetitions
    - "distinct" not mentioned ⇒ the group is a multiset
    - Permutations ⇒ order matters
  - Two groups differ
    - Either because the elements are the same but are repeated a different number of times or because the order differs

Example

Positional representation: order matters!

- How many and which are the distinct anagrams of string ORO (string of 3 characters, 2 being identical)?

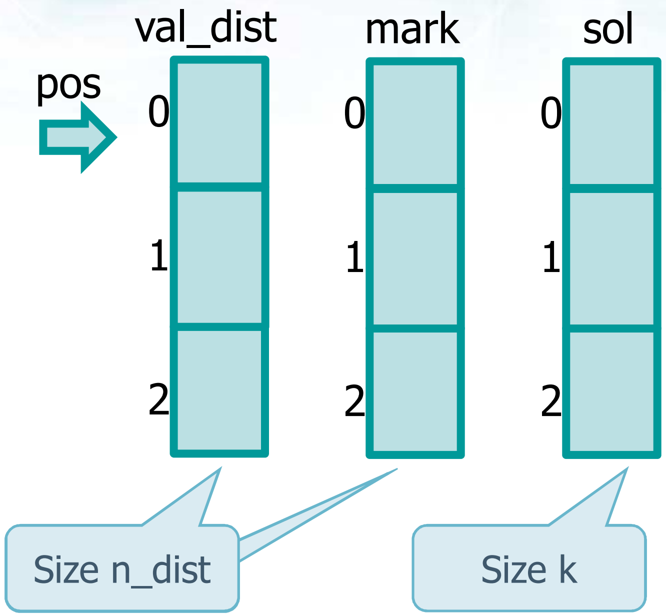
n = 3

$\alpha = 2$

- Solution
  - Model: permutations with repetitions
    - $P_3^{(2)} = \frac{n!}{(\alpha! \cdot \beta! \ ...)} = \frac{3!}{2!} = 3$
  - Anagrams = { OOR, ORO, ROO }

Implementation

As simple arrangements but **mark** is an array of counters not of flags and there are **val_dist** distinct values



Don't forget to check for NULL

```
dist_val = malloc (n_dist*sizeof(int));
mark = malloc (n_dist*sizeof(int));

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

Implementation

- As far as the data-base is concerned
  - It is the same as for simple permutations, with these changes
    - **n** is the cardinality of the multiset
    - **n_dist** is the number of distinct elements of the multiset
    - **val** is the set of (n) elements in the multuise4t
    - **val_dist** is the set of (n_dist) distinct elements of the multiset
    - **count** stores the number of solutions
  - Element **val_dist[i]** is taken if **mark[i]> 0**, **mark[i]** is decremented

Implementation

As simple arrangements but **mark** is an array of counters

```
int perm_rep (int *val_dist, int *sol, int *mark,
  int n, int n_dist, int count, int pos) {
  int i;
  if (pos >= n) {
    for (i=0; i<n; i++)
      printf("%d ", sol[i]);
    printf("\n");
    return count+1;
  }
  for (i=0; i<n_dist; i++) {
    if (mark[i] > 0) {
      mark[i]--;
      sol[pos] = val_dist[i];
      count = perm_rep (
        val_dist,sol,mark,n,n_dist,count,pos+1);
      mark[i]++;
    }
  }
  return count;
}
```

Termination condition

Iteration on n_dist choices

Occurrence control

Mark and choose

Recur

Unmark

Simple combinations

No repetitions

- A simple combination $C_{n,\ k}$ of n distinct objects of class k (k by k) is a non ordered subset composed by k of n objects ($0 \leq k \leq n$)

Order does not matter

Set

For the first time order does not matter !

Simple combinations

- The number of combinations of n elements k by k equals the number of arrangements of n elements k by k divided by the number of permutations of k elements
  - $C_{n,k} = \frac{D_{n,k}}{P_k} = \binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$

Binomial coefficient (n choose k, k≤n)

Simple combinations

- Note that
  - Simple combinations
    - Distinct ⇒ the group is a set
    - Non ordered ⇒ order doesn't matter
    - Simple ⇒ in each grouping there are exactly k non repeated objects
  - Two groups differ
    - Because there is at least a different element

Example

Order does not matter

k = 3

- How many sets of 3 characters can be formed with the 4 characters {A, B, C, D}?

val = {A, B, C, D}, n = 4

- Model
  - Simple combinations
- Solution
  - $C_{n,k} = \binom{n}{k} = \binom{4}{3} = \frac{n!}{k! \cdot (n-k)!} = \frac{4!}{3! \cdot 1!} = 4$
  - Simple combinations = { ABC, ABD, ACD, BCD }

Example

Order does not matter

k = 4

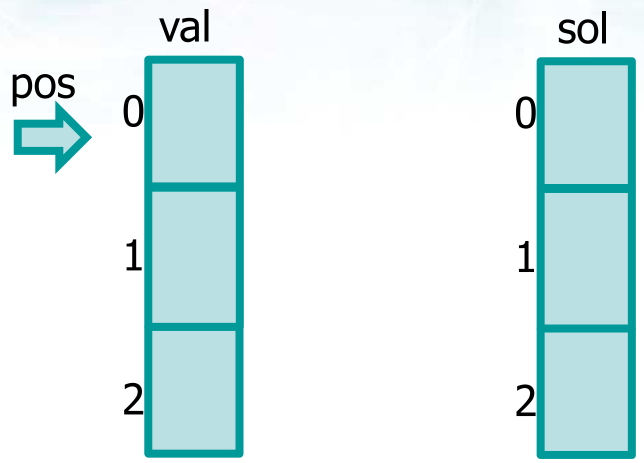
- How many sets of 4 digits can be formed with the 5 digits {7, 2, 0, 4, 1}?

val = {7, 2, 0, 4, 1}, n = 5

- Model
  - Simple combinations
- Solution
  - $C_{n,k} = \binom{n}{k} = \binom{5}{4} = \frac{n!}{k! \cdot (n-k)!} = \frac{5!}{4! \cdot 1!} = 5$
  - Simple combinations = { 7204, 7201, 7241, 7041, 2041 }

Implementation

As simple arrangements but **mark** does not exist and we begin from **start** at each selection iteration



Size n

Size k

Don’t forget to check for NULL

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

Implementation

- With respect to simple arrangements it is necessary to "force" one of the possible orderings
  - Index **start** determines from which value of **val** we start to fill-in **sol**
  - Array
    - **val** is visited thanks to index **i** starting from **start**
    - **sol** is assigned starting from index **pos** with possible values of **val** from start onwards
    - Once value **val[i]** is assigned to **sol**, recur with **i+1** and **pos+1**
    - **mark** is not needed
  - Variable count stores the number of solutions

Implementation

As simple arrangements but **start** forces a specific order

```
int comb (int *val, int *sol, int n, int k,
          int start, int count, int pos) {
  int i, j;

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

  for (i=start; i<n; i++) {
    sol[pos] = val[i];
    count = comb(val,sol,n,k,i+1,count,pos+1);
  }
  return count;
}
```

Termination condition

Iteration on n choices

sol[pos] filled with possible values of val from start onwards

Recur (next position and next choice)

Combinations with repetitions

No upper bound

Repetition

Set

- A combination with repetitions $C'_{n,k}$ of n distinct objects of class k (k by k) is a non ordered subset made of k of the n objects (k ≥ 0)

Order does not matter

  - Each object may be taken at most k times
- The number of combinations with repetitions of n objects k by k is
  - $C'_{n,k} = \binom{n+k-1}{k} = \binom{n+k-1}{n-1} = \frac{(n+k-1)!}{k! \cdot (n-1)!}$

Combinations with repetitions

- Note that
  - Combinations with repetitions
    - Distinct ⇒ the group is a set
    - Non ordered ⇒ order doesn't matter
    - "Simple " not mentioned ⇒ in each grouping the same object may occur repeatedly at most k times
    - k may be > n
  - Two groups differ if
    - One of them contains at least an object that doen't occur in the other one
    - The objects that appear in one group appear also in the other one but are repeated a different number of times

Example

Order does not matters!

n = 6

k = 2

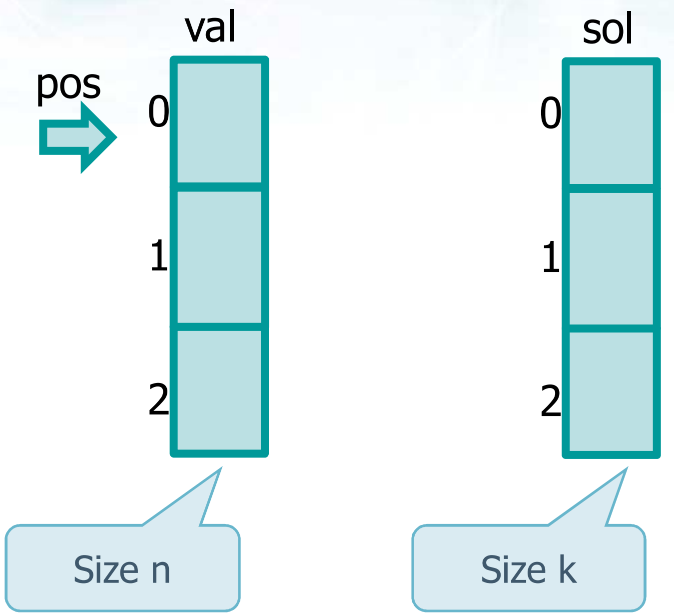
- When simultaneously casting two dice, how many compositions of values may appear on 2 faces?
- Model
  - Combinations with repetitions
- Solution
  - $C'_{6,2} = \binom{n + k - 1}{k} = \frac{(n+k-1)!}{k! \cdot (n-1)!} = \frac{(6+2-1)!}{2! \cdot (6-1)!} = 21$
  - Compositions = { 11, 12, 13, 14, 15, 16, 22, 23, 24, 25, 26, 33, 34, 35, 36, 44, 45, 46, 55, 56, 66 }

Solution

- Same as simple combinations, but
  - Recursion occurs only for **pos+1** and not for **i+1**
  - Index **start** is incremented each time the for loop on choices
  - **count** records the number of solutions

Implementation

As simple combinations but **i** is not incremented when recurring to re-consider the same object over and over again



Don't forget to check for NULL

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

Implementation

As simple combinations but we must re-consider the same object

```
int comb_rep (int *val, int *sol, int n, int k,
              int start, int count, int pos) {
  int i, j;

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

  for (i=start; i<n; i++) {
    sol[pos] = val[i];
    count = comb_rep(val,sol,n,k,i,count,pos+1);
  }
  return count;
}
```

Termination condition

Iteration on n choices

sol[pos] filled with possible values of val from start onwards

Recur (next position)

The powerset

- Given a set S, its powerset is the set of the subsets of S, including S itself and the empty set

- Example
  - S = { 1, 2, 3, 4 }
  - k = 4
  - $Powerset_S$ = { ∅, 4, 3, 34, 2, 24, 23, 234, 1, 14, 13, 134, 12, 124, 123, 1234 }

K = |S|

Models

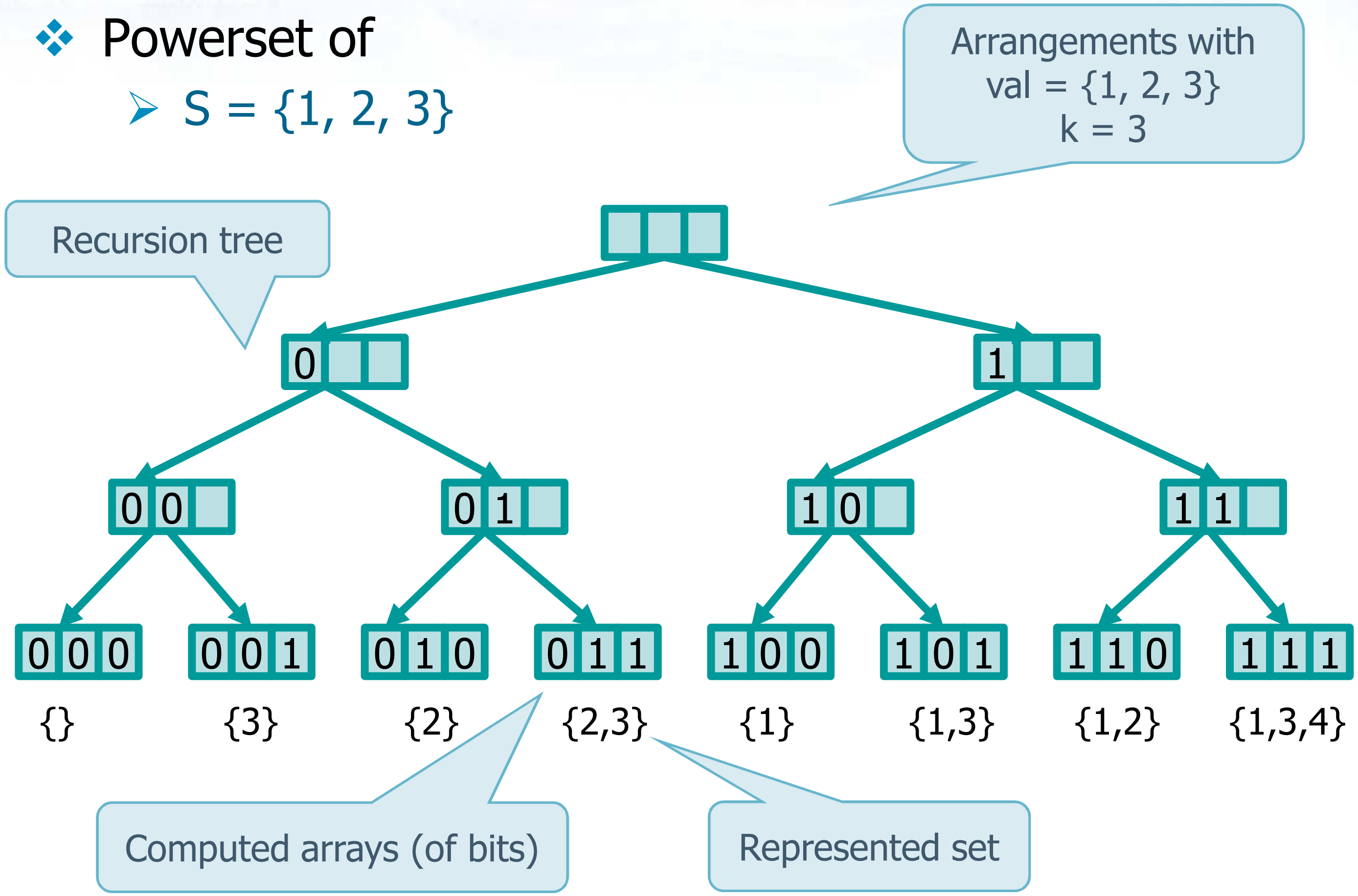
- The powerset can be computed using 3 different models
  - Arrangements with repetitions
  - Simple combinations
    - Re-activating the procedure k times
  - Simple combinations
    - Adopting a divide and conquer strategy

The powerset: 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 n positions
    - The computed array will tell which elements have to be selected within the powerset

Implementation

- Powerset of
  - S = {1, 2, 3}



Implementation

- Each subset is represented by the **sol** array having **k** elements
  - Each element represent the set of possible choices, thus 0 and 1 (thus and 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

Implementation

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

The powerset: 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, …, |S|)
  - In this case the recursive function generates the desired set (not an array of bits previously generated)

Implementation

- We must
  - Union of the empty set and
  - The powerset of size 1, 2, 3, …., k
- To compute the powerset, we use simple combinations of k elements taken by groups of n
  - $\text{Powerset}_S = \{ \emptyset \} \cup \bigcup_{n=1}^{k} \binom{k}{n}$
- 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

Implementation

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)

Implementation

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)

The powerset: 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
    - If k=0
      - $Powerset_{S_k} = \{\varnothing\}$
    - If k>0
      - $Powerset_{S_k} = \{Powerset_{S_{k-1}} \cup S_k\} \cup \{Powerset_{S_{k-1}}\}$

Terminal case: empty set

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

Implementation

- 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

Implementation

```
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

Partitions of a set

- Given a set S of |S| elements, a collection S = {Si} of non empty blocks forms a partition only iff both the following conditions hold
  - Blocks are pairwise disjoint
    - $\forall S_i, S_j \in$ S with i ≠ j then $S_i \cap S_j = \emptyset$
  - The union of those blocks is S
    - $S = \cup_i S_i$
- The number of blocks k ranges
  - From 1, in that case the block coincides with the set S
  - To n, in that case each block contains only 1 element of S

Example

- Given the following set S generate all possibile partitions
  - S = {1, 2, 3, 4 }
- Solution
  - K=1
    - 1 partition: {1234}
  - K=2
    - 7 partitions: {123, 4}, {124, 3} , {12, 34}, {134, 2}, {13, 24}, {14, 23}, {1, 234}
  - K=3
    - 6 partitions: {12, 3, 4}, {13, 2, 4} , {1, 23, 4}, {14, 2, 3}, {1, 24, 3}, {1, 2, 34}
  - K=4
    - 4 partitions: {1}, {2}, {3}, {4}

The order of the blocks and of the elements within each block doesn't matter. As a consequence the 2 partitions {123, 4} AND {4, 312} are identical

Problem

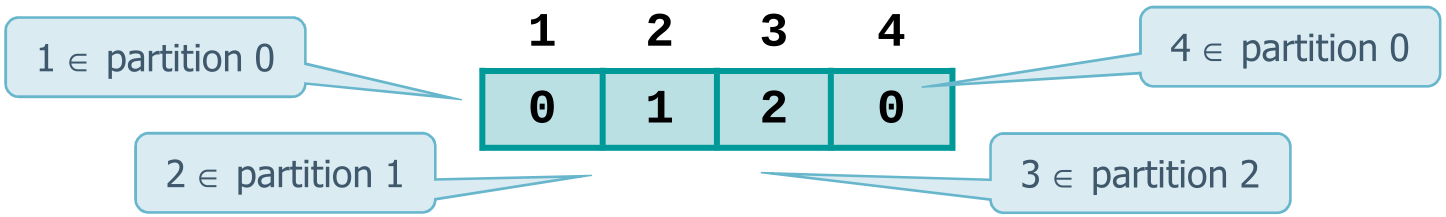
- Given the set S of cardinality n=|S|, it is possibile to find
  - All partitions in exactly k blocks, where k is a constant value
    - This problem can be solved with arrangements with repetitions
  - All partitions in k blocks, where k is a variable value and it ranges between 1 and n
    - This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

Number of partitions

- 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
  - $B_0 = 1$
  - $B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} \cdot Bk$
- The first Bell numbers are
  - $B_0 = 1$, $B_1 = 1$, $B_2 = 2$, $B_3 = 5$, $B_4 = 15$, $B_5 = 52$, …
- Their search space is not modelled in terms of combinatorics

Partition of a set S

- To represent a partitions at least two approaches are possibile
  - Given the element, identify the unique block it belongs to
  - Given the block, list the elements that belong to it
- First approach preferrable, as it works on an array of integers and not on lists
  - Example
    - S={1,2,3,4}, partition={14, 2, 3}
    - Partitions are numbered from 0 to 3



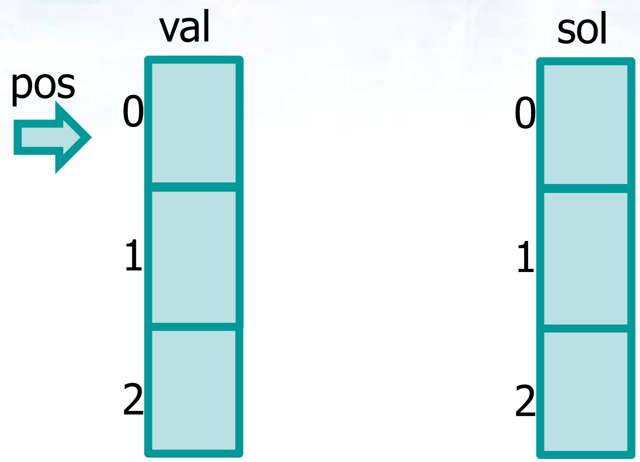
Problems

- To solve the first problem arrangements with repetitions are sufficient
  - 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
- As we do not want to have empty partitions (we would generate less than k partitions)
  - We must check whether all partitions are not empty once a solution has been generated

Implementation

- 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



Size k

Size k

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;
  occ = calloc(n, sizeof(int))
  if (pos >= n) {
    for (j=0; j<n; j++) occ[sol[j]]++;
    i=0;
    while ((i < k) && ok) {
      if (occ[i]==0) ok = 0;
        i++;
    }
    if (ok == 0) return;
    else { /*PRINT SOLUTION ... */ }
  }
  for (i=0; i<k; i++) {
    sol[pos] = i;
    arr_rep(val,sol,n,k,pos+1);
  }
}
```

Block occurrence array

Occurrence computation

Occurrence check

Discard solution

Print solution

Recur:
Simple arrangements