Basics of Combinatorics

Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino

Definition

- 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
- Combinatorics is a topic of the course in Mathematical Methods for Engineering
- In problem-solving we need to enumerate the ways, not only to count them

Model

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

Grouping criteria

We can group k objects taken from a group S of n elements 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 $S_0 \ldots S_{n-1}$
  - $S = S_0 \cup S_1 \cup \ldots S_{n-1}$ && $\forall\, i \neq j\ S_i \cap S_j = \emptyset$

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

Basic principle: addition

- Alternative definition
  - If an object can be selected in p_0 ways from a group of size S_0 , …, and in p_{n-1} ways from a group of size S_{n-1}
  - Then selecting an object from any of the n groups may be performed in $\sum_{i=0}^{n-1} |p_i|$ ways

Example

- There are 4 Computer Science courses and 5 Mathematics courses
- A student can select just one
- In how many ways can a student choose?

- Solution
  - Disjoint sets $\Rightarrow$
  - Model: principle of addition
  - Number of choices = 4 + 5 = 9

Basic principle: multiplication

- Given n sets S_i $(0 \le i < n)$ each of cardinality $|S_i|$, the number of ordered t-uples $(s_0 \ldots s_{n-1})$ with $s_0 \in S_0 \ldots s_{n-1} \in S_{n-1}$ is
  - #tuples = $\prod_{i=0}^{n-1} |S_i|$
- Alternative definition
  - If an object x_0 can be selected in p_0 ways from a group, an object x_1 can be selected in p_1 ways, …, and an object x_{n-1} can be selected in p_{n-1} ways, the choice of a t-uple of objects $(x_0 \ldots x_{n-1})$ can be done in
  - #tuples = $p_0 \cdot p_1 \ldots . \cdot p_{n-1}$ ways

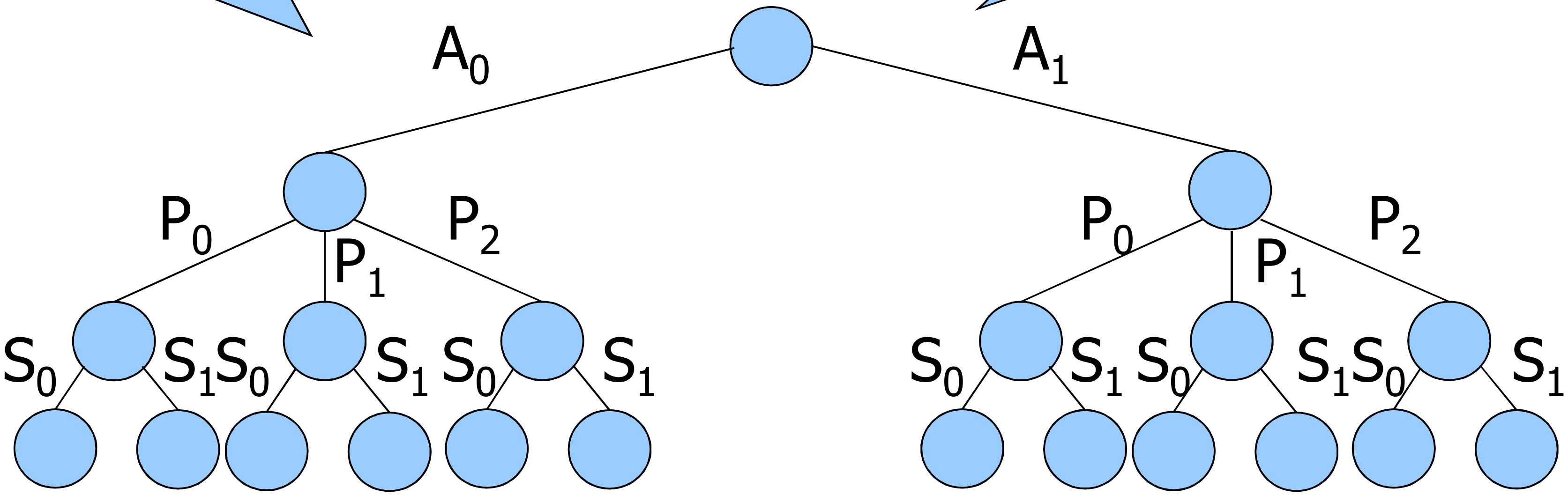
Example

- In a restaurant a menu is served made of appetizer, first course, second course and dessert
- The customer can choose among 2 appetizers, 3 first courses, 2 second courses
- How many different menus can the restaurant offer?
  - Model: principle of multiplication
  - Number of choices = 2 x 3 x 2 = 12

Example

2 appetizers (A0, A1)
3 main courses (P0, P1, P2)
2 second courses (S0,S1) (n=k=3)

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



Solution:
(A_0,P_0,S_0), (A_0,P_0,S_1), (A_0,P_1,S_0), (A_0,P_1,S_1), (A_0,P_2,S_0), (A_0,P_2,S_1),
(A_1,P_0,S_0), (A_1,P_0,S_1), (A_1,P_1,S_0), (A_1,P_1,S_1), (A_1,P_2,S_0), (A_1,P_2,S_1)

Basic search principles

- Choices are made in sequence
  - They are represented by a tree
  - The number of choices is fixed for a level, but varies from level to level, then nodes have a number of children that varies according to the level
  - Each 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
- Solutions are the labels of the edges along each path from root to node

Basic search principles

- The goal is to enumerate all solutions, searching their space
- All solutions are valid
- Recursive calls are associated to the solution, whose size grows by 1 at each call
- Termination
  - Size of current solution equals final desired size

Basic search principles

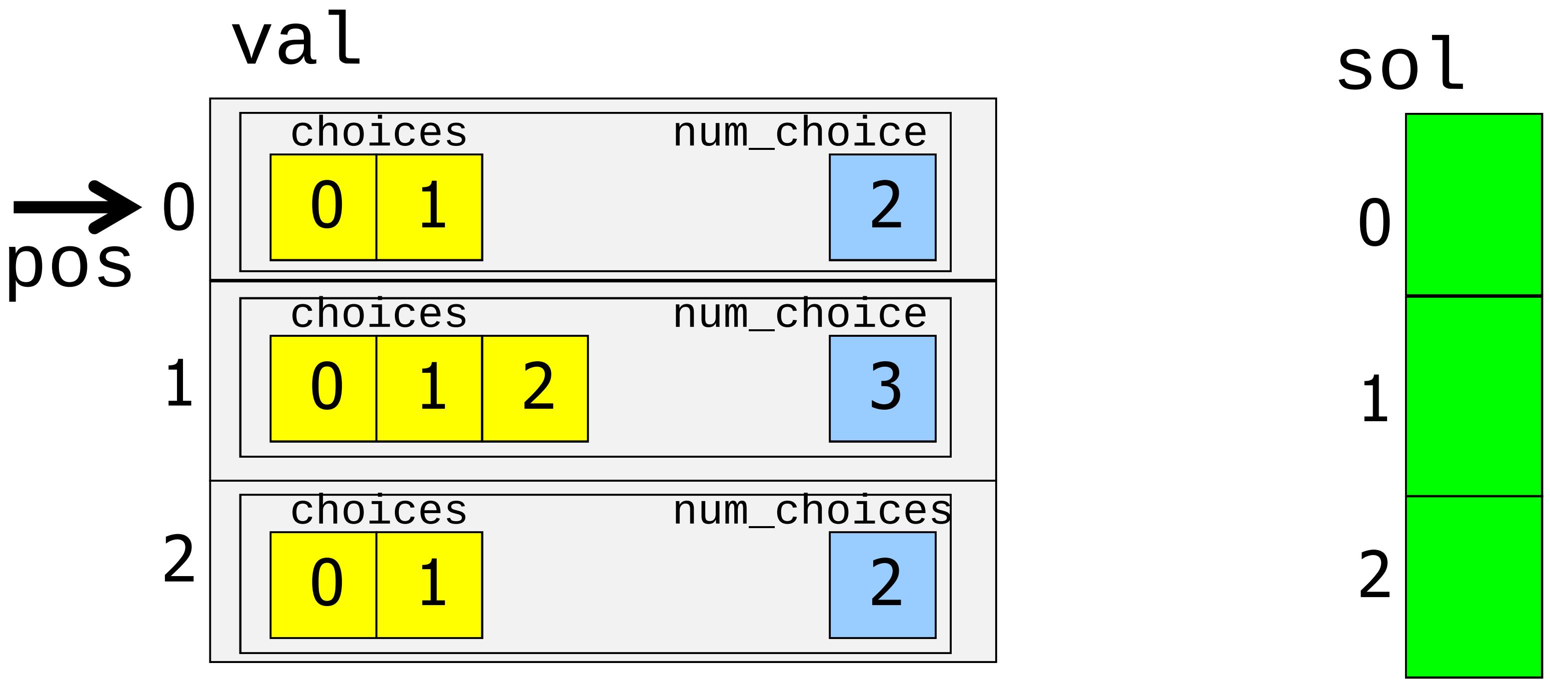
- There is a 1:1 matching between choices and a (possibly non contiguous) subset of integers
- Possible choices are stored in array val of size n containing structures of type Level
  - Each structure contains an integer field num_choice for the number of choices at that level and an array *choices of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

Basic search principles

- 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, i.e., 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

Solution

Referring to the example



Solution

Check for NULL Pointers

```
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));
```

Solution

```
int princ_mult(int pos, Level *val, int *sol,
               int n, int count) {
  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 = princ_mult(pos+1, val, sol, n, count);
  }
  return count;
}
```

Simple arrangements

No repetitions

Set

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

Order matters

There are

$$D_{n,k} = \frac{n!}{(n-k)!} = n \cdot (n-1) \cdot \ldots\ldots \cdot (n-k+1)$$

Simple arrangements of n objects k by k

Simple arrangements

- Note that simple arrangements are
  - distinct $\Rightarrow$ the group is a set
  - ordered $\Rightarrow$ order matters
  - simple $\Rightarrow$ in each grouping 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!

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

no repeated digits

k = 2

n = 4

val = {4, 9, 1, 0 }

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

Solution

- In order not to generate repeated elements
  - An array mark records already taken elements (mark[i]=0 $\Rightarrow$ 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
  - Count records the number of solutions

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

```c
int arr(int pos,int *val,int *sol,int *mark,
         int n, int k,int count){
  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(pos+1, val, sol, mark, n, k,count);
      mark[i] = 0;
    }
  }
  return count;
}
```

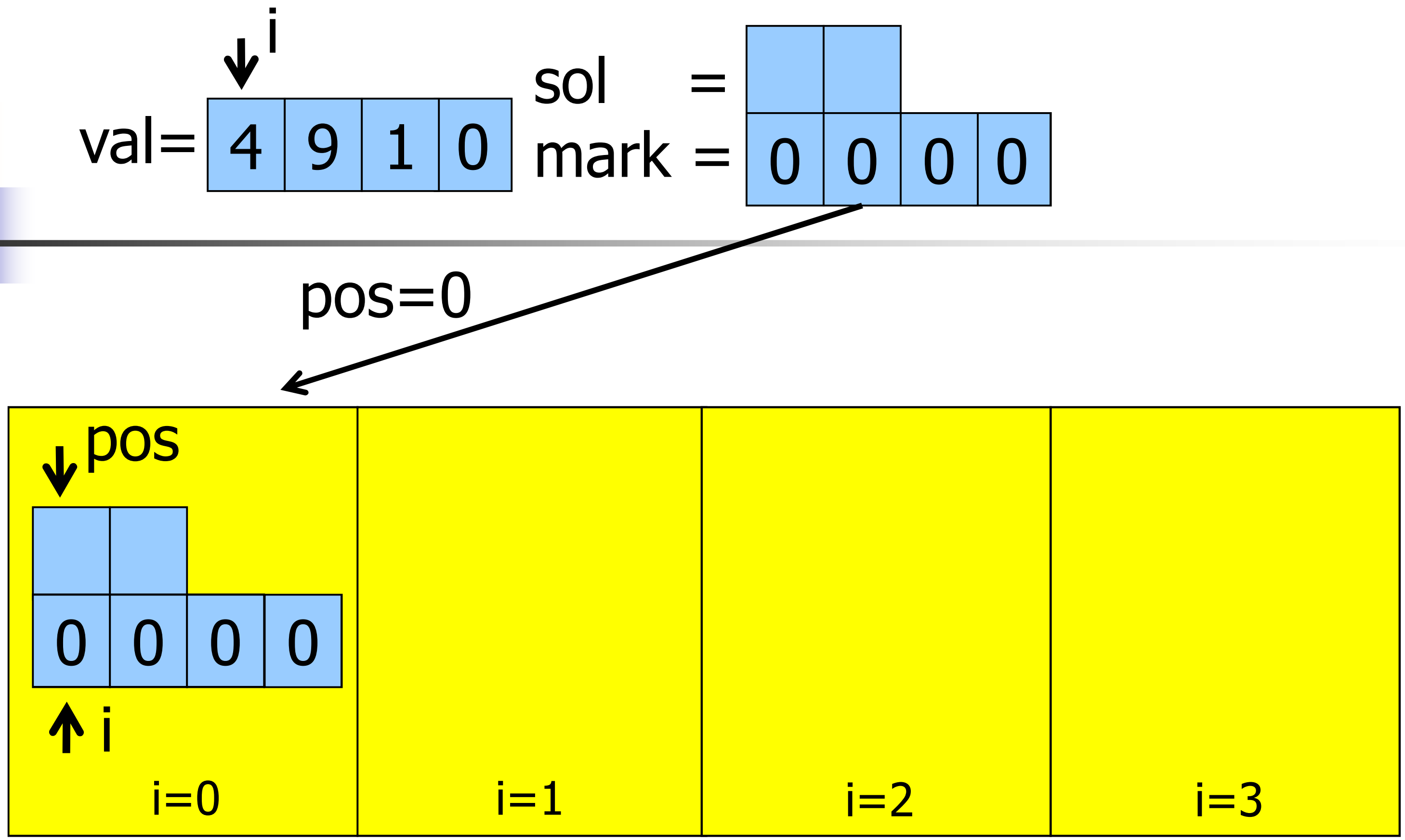
Termination

Iteration on n choices

Mark and choose

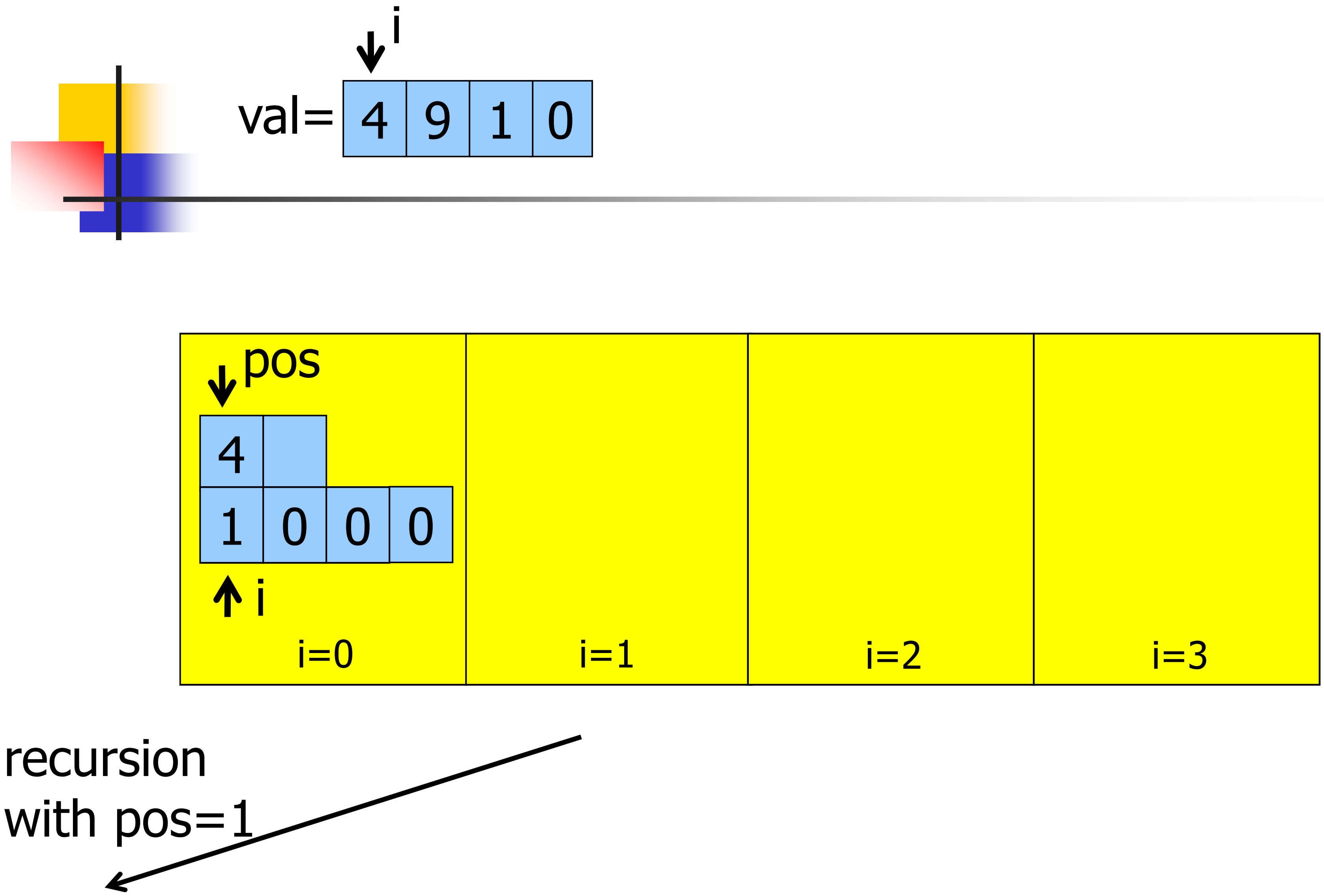
Recursion

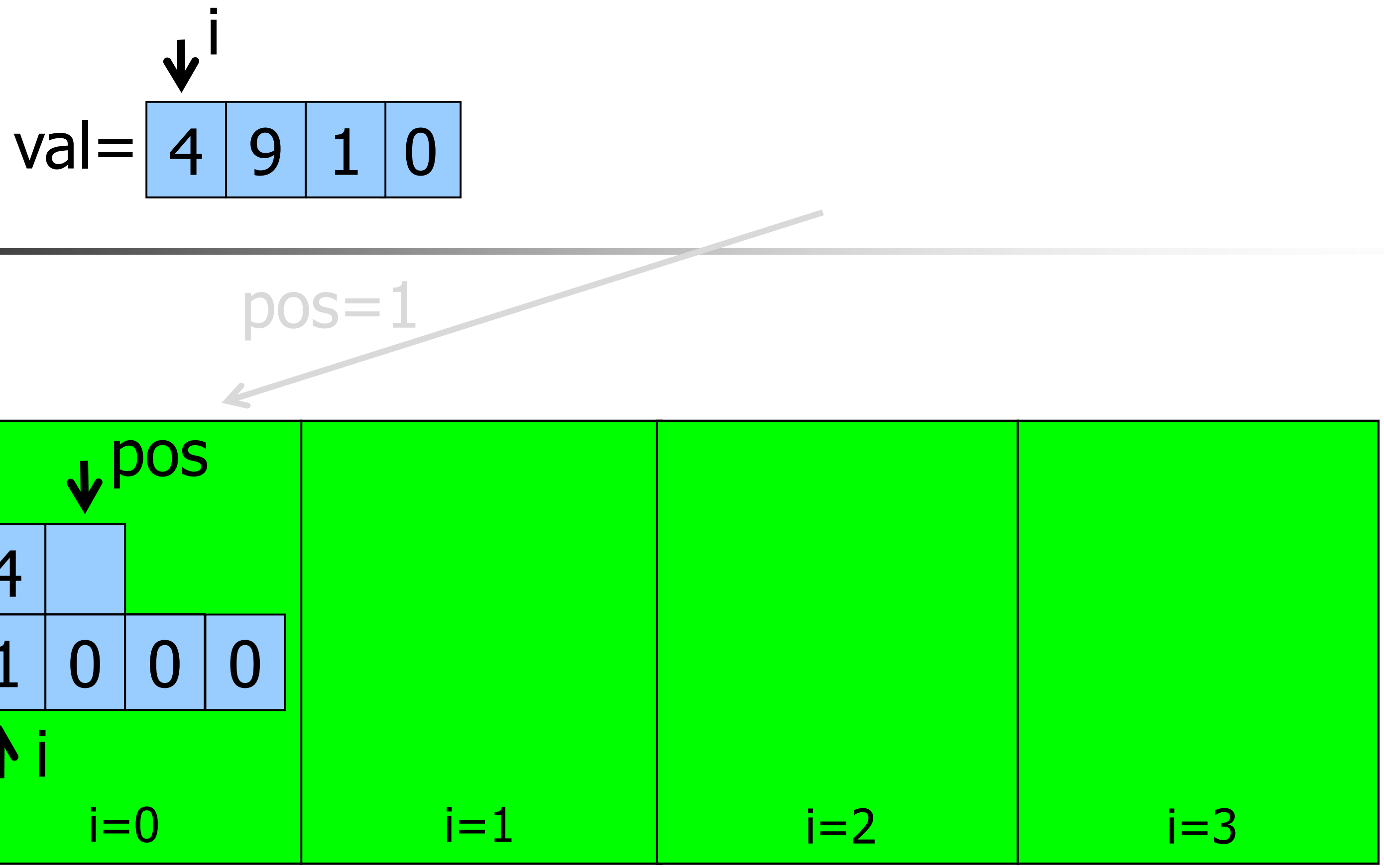
Unmark



mark[i] = 0

sol[pos] = val[i]
mark[i] = 1

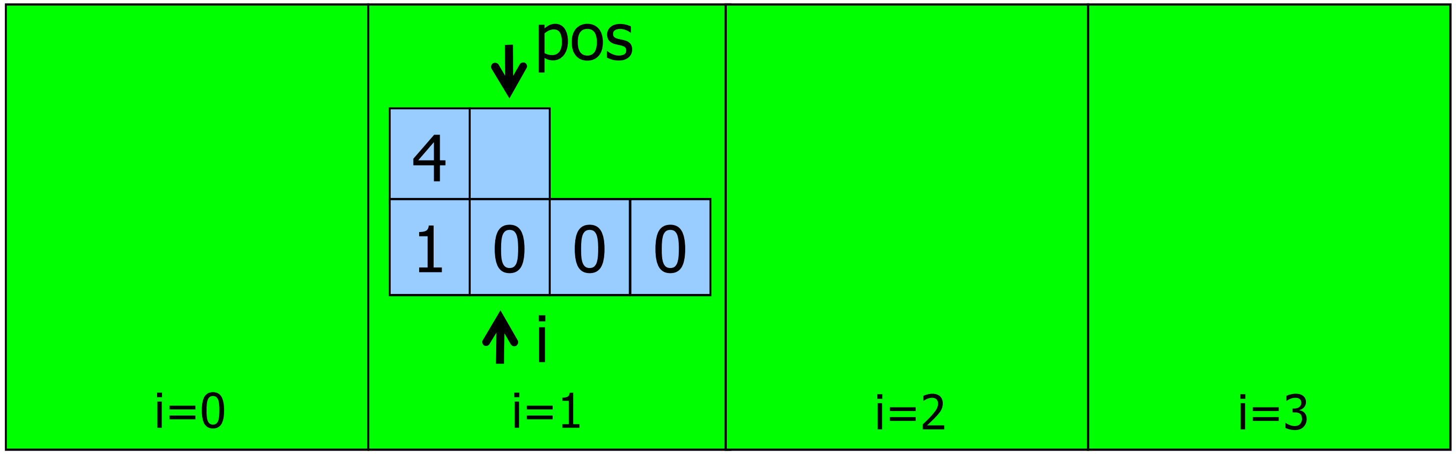

i
val=
4
9
1
0
pos
4
1
0
0
0
i
i=0
i=1
i=2
i=3
recursion
with pos=1



mark[i] = 1

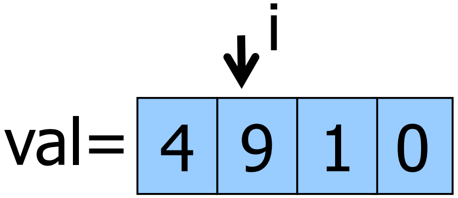
i

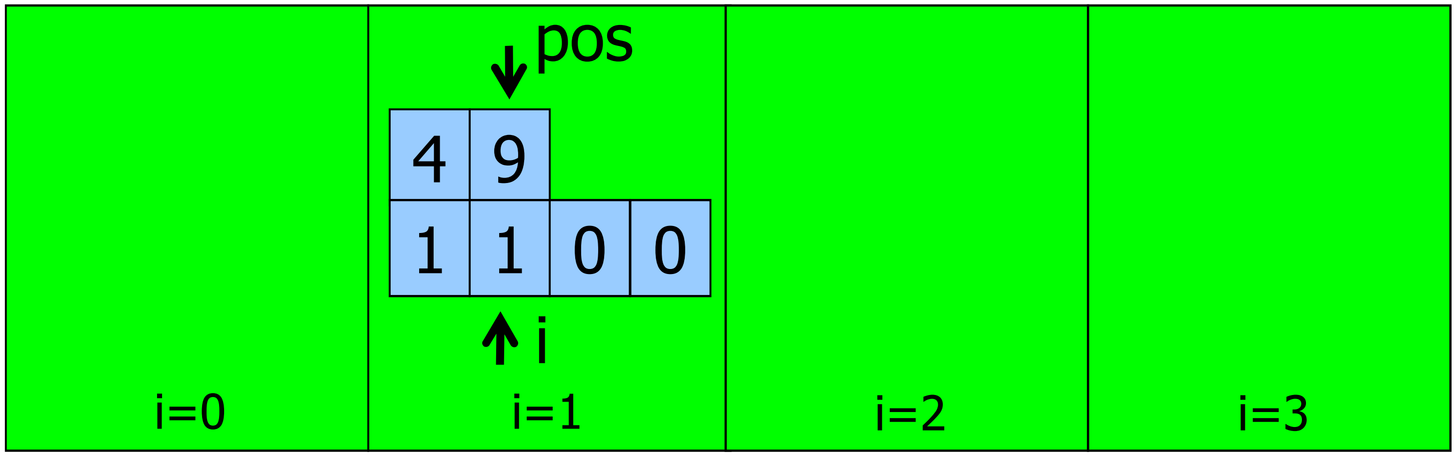
val= | 4 | 9 | 1 | 0 |



mark[i] = 0

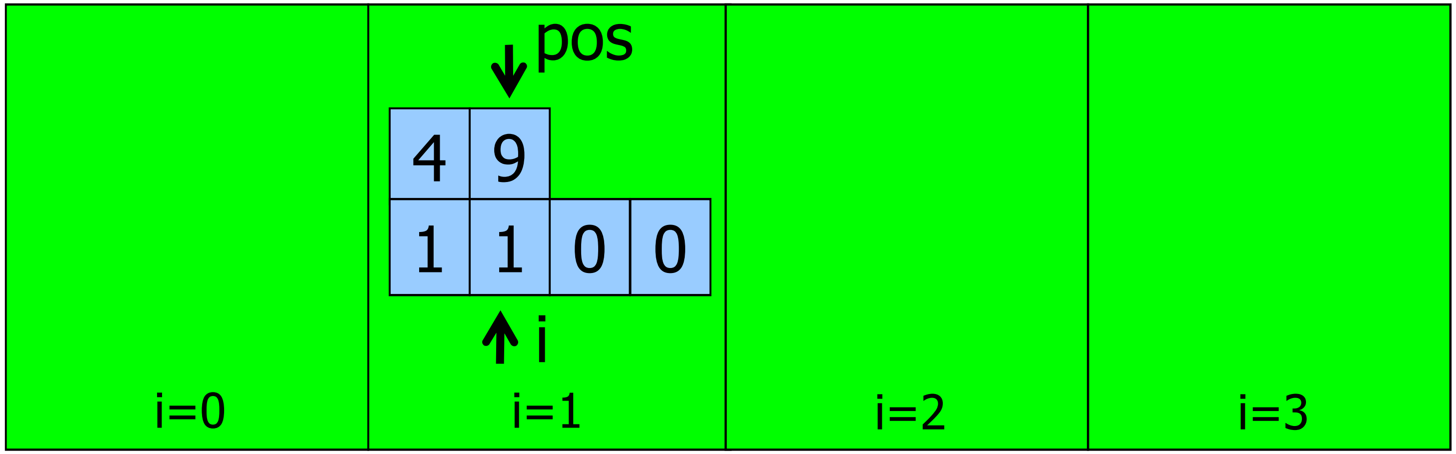
sol[pos] = val[i]
mark[i] = 1





recursion
with pos=2

val= 4 9 1 0

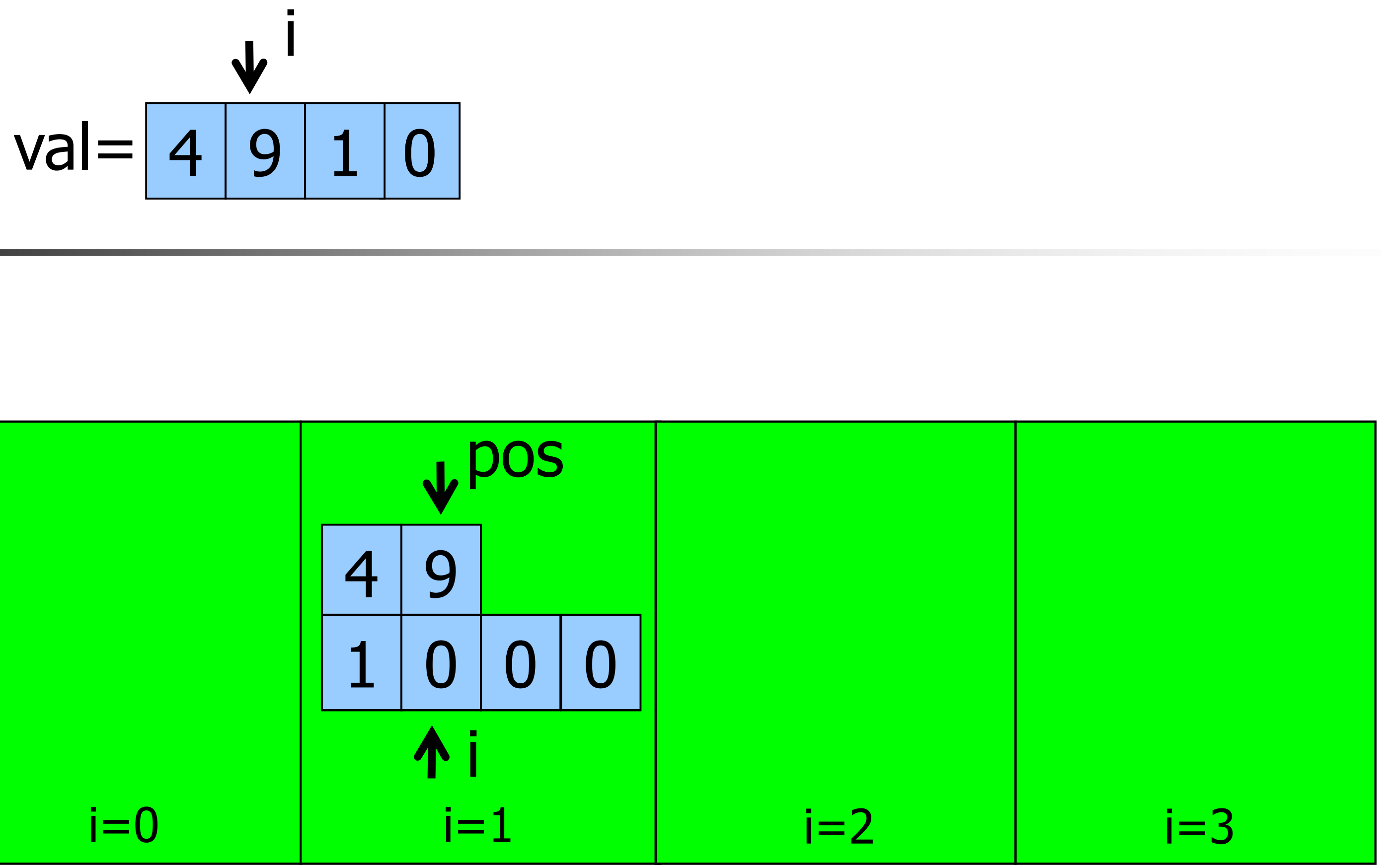


termination: display, update count

return

sol= 4 9

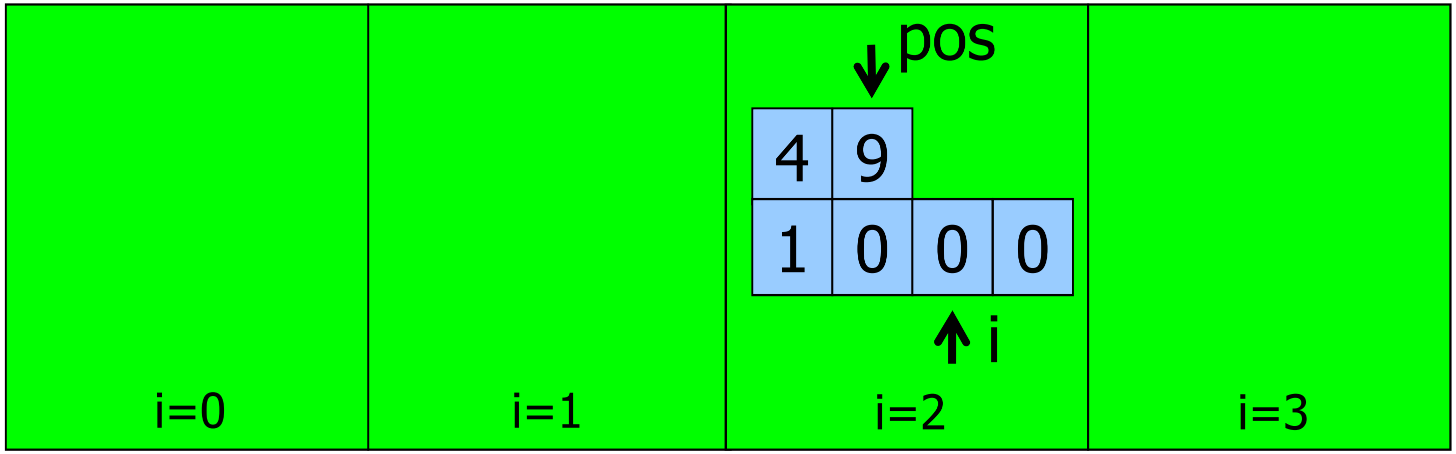
First couple



unmark mark[i]

i

val=	4	9	1	0

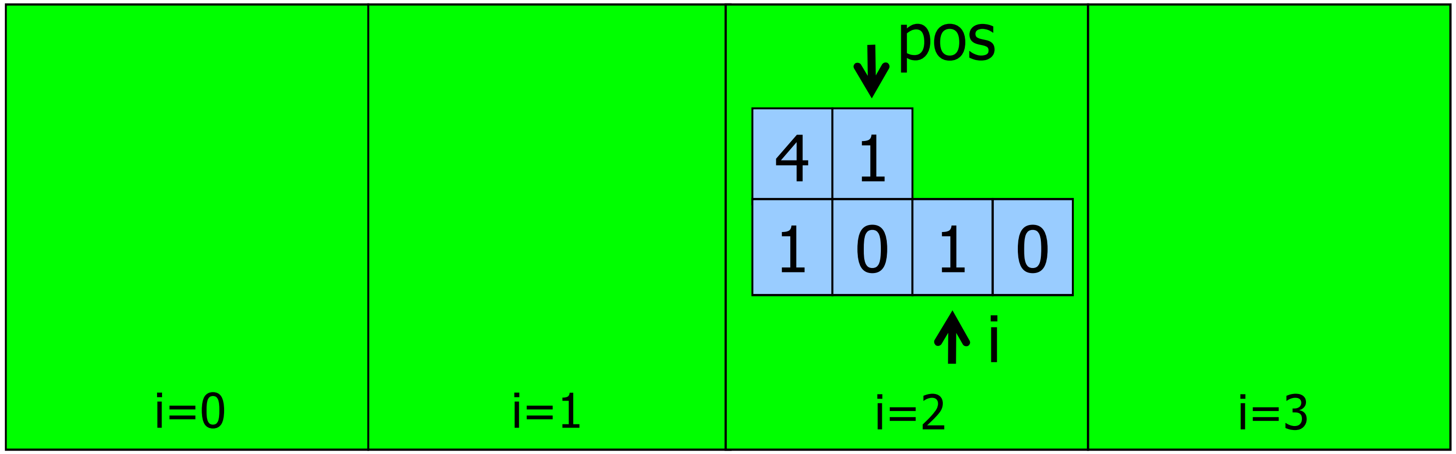


mark[i] = 0

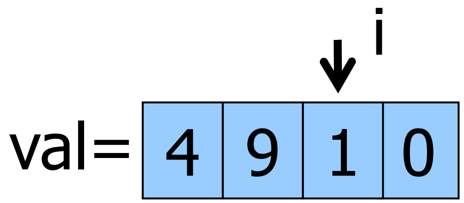
sol[pos] = val[i]
mark[i] = 1

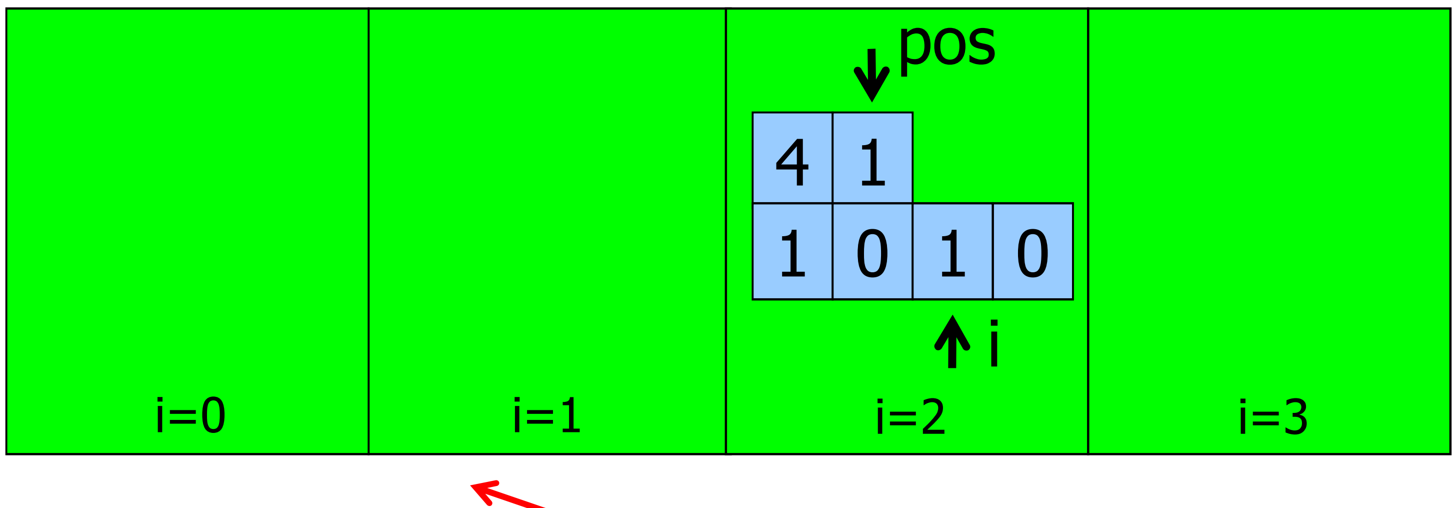
i

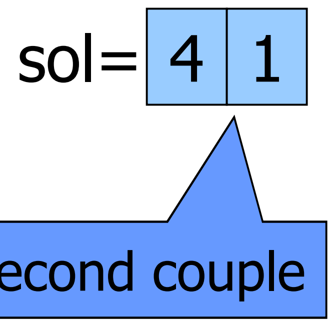
val= | 4 | 9 | 1 | 0 |



recursion
with pos=2

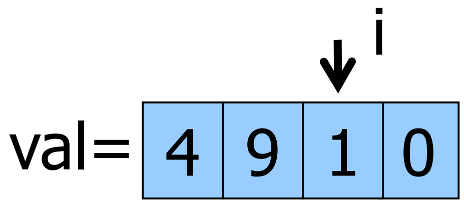


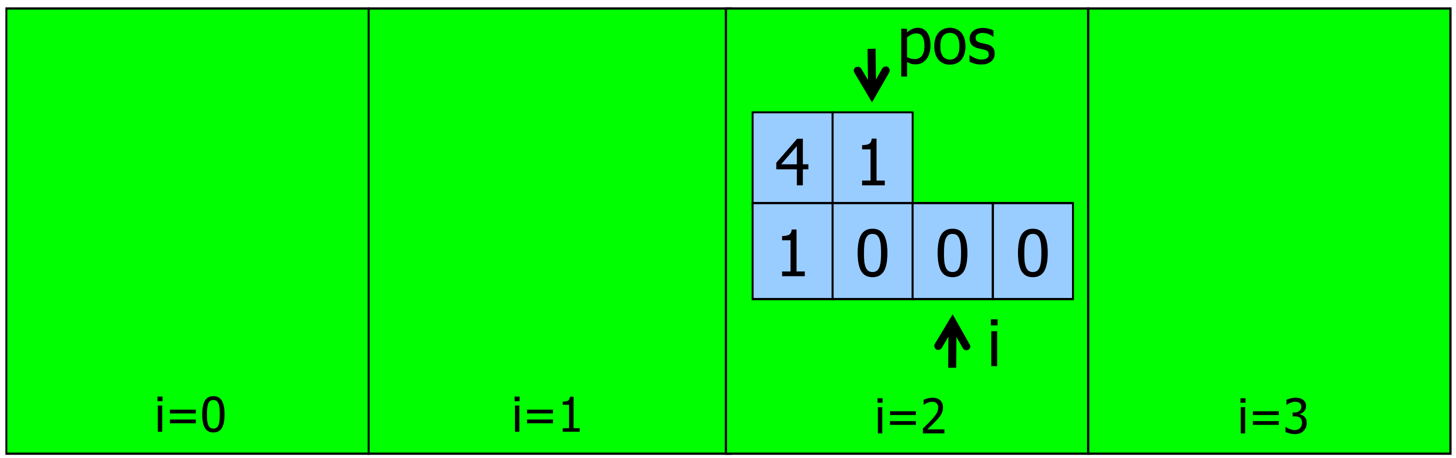




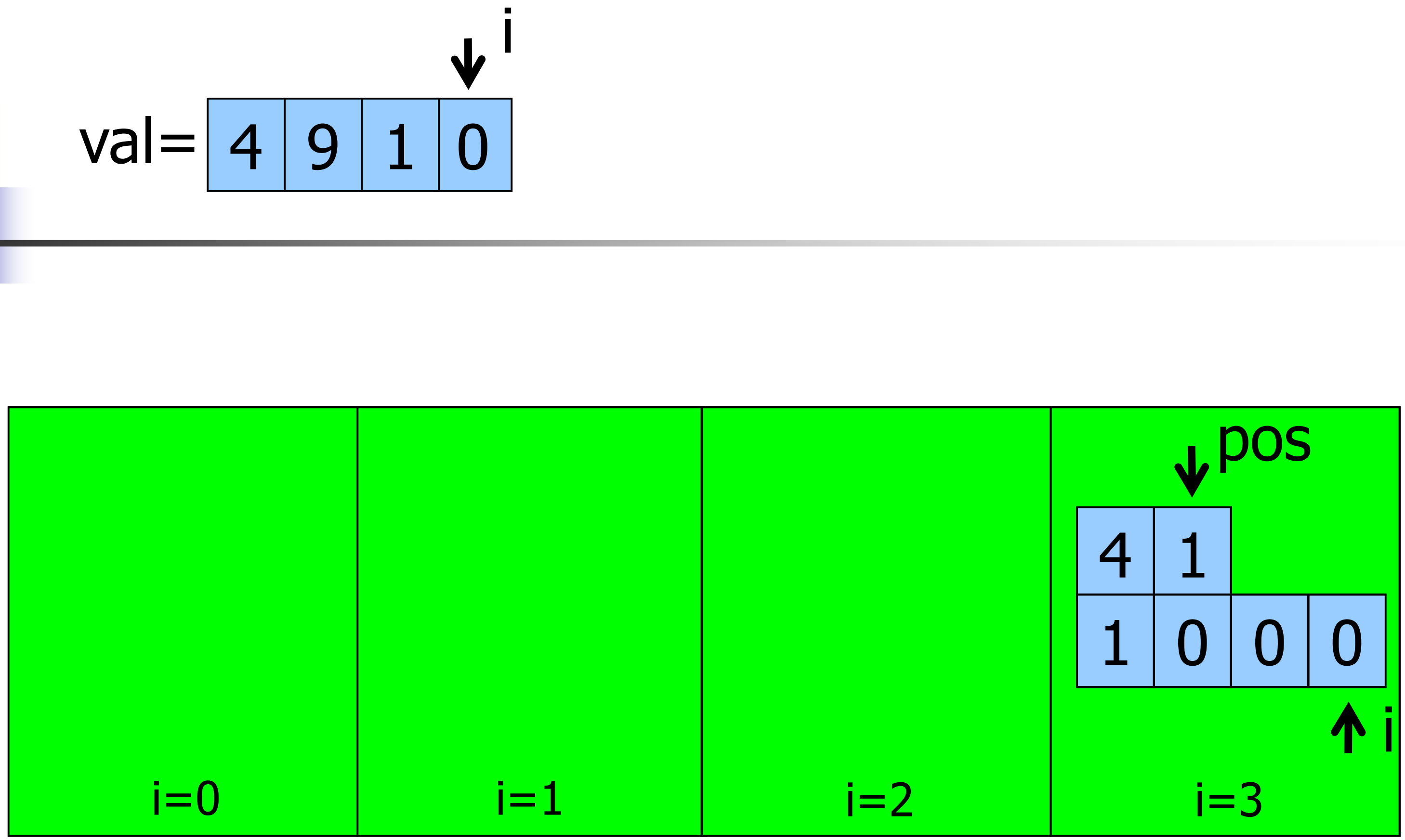
terminatione: display, update count

return



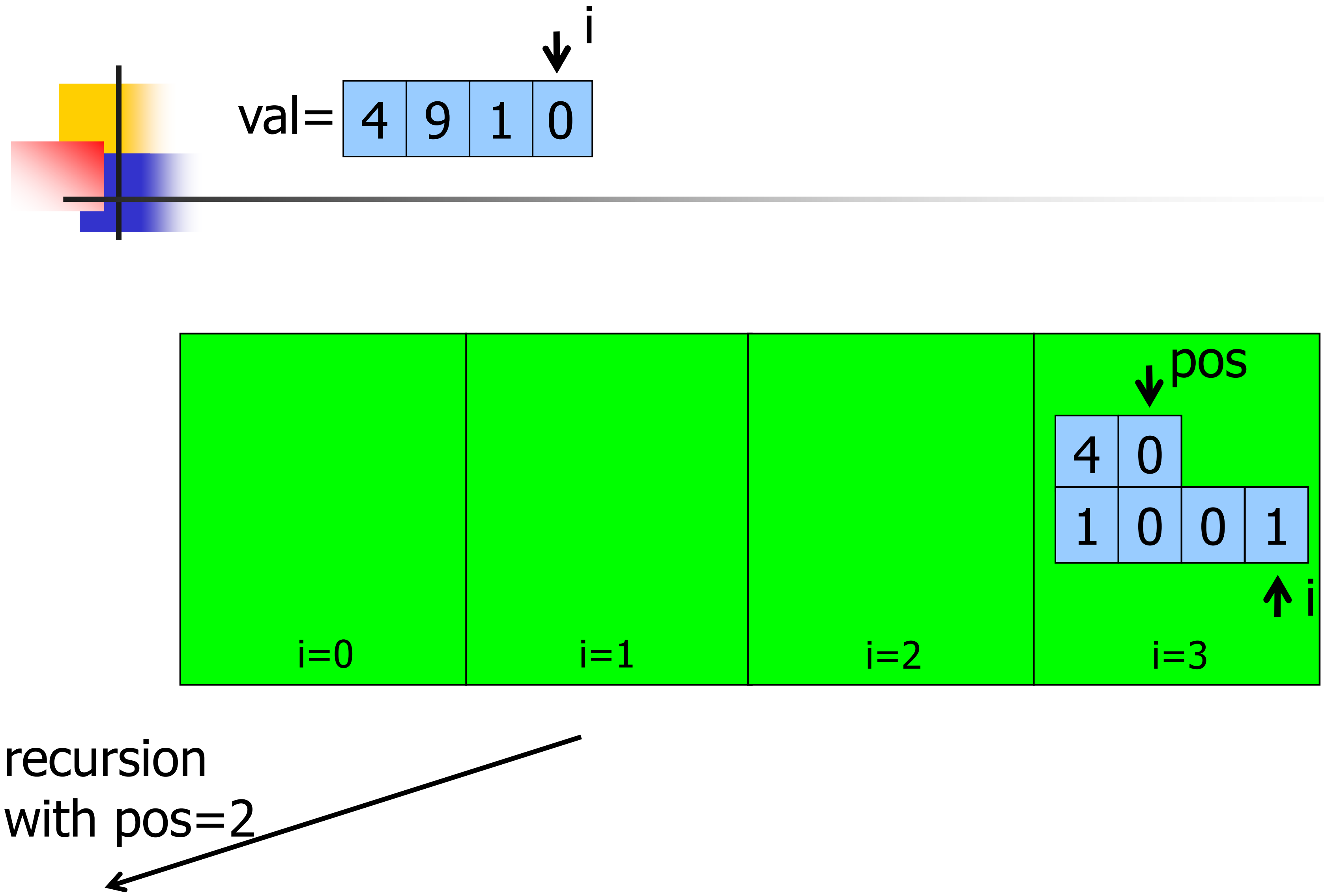


unmark mark[i]



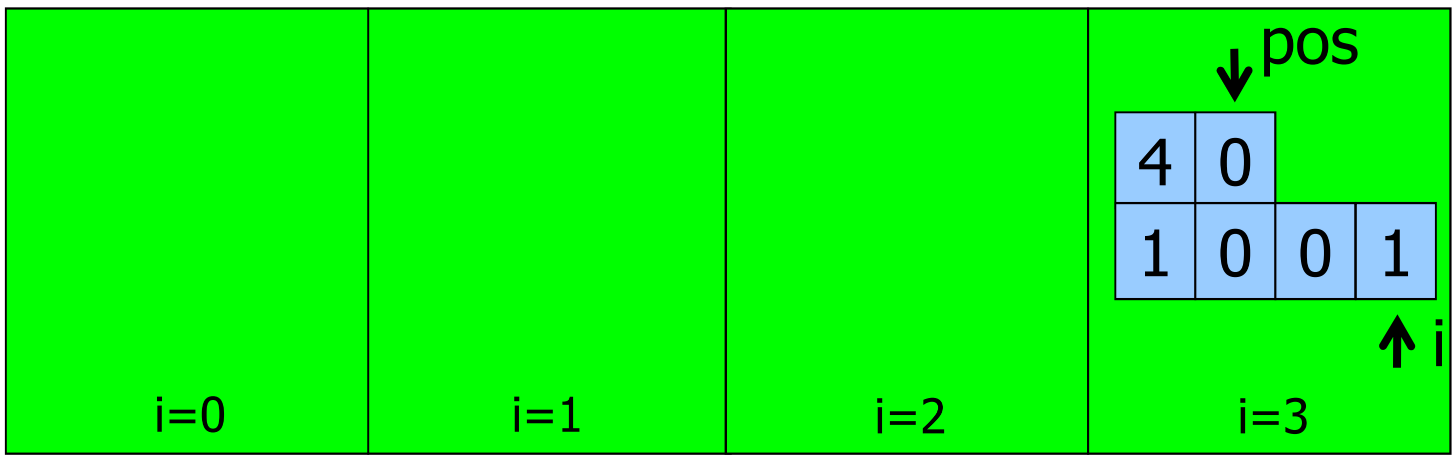
mark[i] = 0

sol[pos] = val[i]
mark[i] = 1



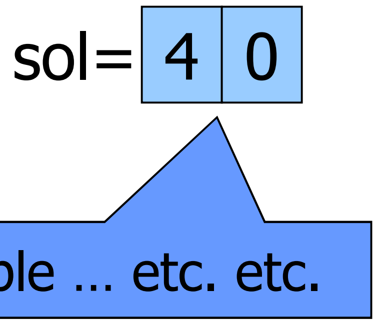
i
val=
4
9
1
0
pos
4
0
1
0
0
1
i
i=0
i=1
i=2
i=3
recursion
with pos=2

val= | 4 | 9 | 1 | 0 |



termination: display, update count

return



Arrangements with repetitions

Repetitions

Set

An <u>arrangement with repetitions</u> $D'_{n,\,k}$ of *n* <u>distinct</u> objects of class *k* (k by k) is an <u>ordered subset</u> composed of *k* out of *n* objects ($0 \leq k$) each of whom may be taken up to k times.

No upper bound!

Order matters

There are

$$D'_{n,\,k} = n^k$$

Arrangements with repetitions of n objects taken k by k

Arrangements with repetitions

Note that:

- distinct $\Rightarrow$ the group is a set
- ordered $\Rightarrow$ order matters
- "simple" not mentioned $\Rightarrow$ in every grouping the same object can occur repeatedly at most *k* times
- k may be > n

Arrangements with repetitions

- 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 ot times

Example

positional representation: order matters!

How many and which are pure binary numbers on 4 bits?

Each bit can take either value 0 or 1

k = 4

n = 2

Model: arrangements with repetitions

$D'_{2,\,4} = 2^4 = 16$

Solution

{0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111
1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 }

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
- `count` stores the number of solutions.

Solution

```
int rep_arr (
  int pos,int *val,int *sol,int n,int k,int count
) {
  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 = rep_arr(pos+1, val, sol, n, k, count);
  }
  return count;
}
```

termination

Iteration on n choices

Choice

Recursion

Simple Permutations

No repetitions

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

Order matters

Set

There are

$$P_n = D_{n,\,n} = n!$$

simple permutations of n objects

Simple Permutations

Note that:

- distinct $\Rightarrow$ the group is a set
- ordered $\Rightarrow$ order matters
- simple $\Rightarrow$ in each grouping there are exactly n non repeated objects.

Two groupings differ because the elements are the same, but appear in a different order.

Example

Positional representation: order matters!

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

n = 3

No repetitions

Model: simple permutations

$P_3 = 3! = 6$

Solution

{ ORA, OAR, ROA, RAO, AOR, ARO }

Example

Given a set val of n integers, generate all their possible permutations

The number of permutations is n!

Example

val = {1, 2, 3}   n = 3

n! = 6

The 6 permutations are

{1,2,3} {1,3,2} {2,1,3} {2,3,1} {3,1,2} {3,2,1}

Solution

In order not to generate repeated elements:

- an array `mark` records already taken elements (`mark[i]=0` ⇒ `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

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

```c
int perm(int pos,int *val,int *sol,int *mark,
         int n, int count){
  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(pos+1, val, sol, mark, n, count);
      mark[i] = 0;
  }
  return count;
}
```

Termination

Iteration on n choices

Mark and choose

Recursion

Unmark

Permutations with repetitions

Repeated elements

Given a multiset of n objects among which α are identical, β are identical, etc., the number of distinct permutations with repeated objects is:

order matters

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

Permutations with repetitions

Note that:

- "distinct" not mentioned $\Rightarrow$ the group is a multiset
- permutations $\Rightarrow$ order matters

Two groupings 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

Model: permutations with repetitions

$P^{(2)}_3 = 3!/2! = 3$

Solution

{ OOR, ORO, ROO }

Solution

Same as for simple permutations, with these changes

- `n` is the cardinality of the multiset
- store in array `dist_val` of `n_dist` cells the distinct elements of the multiset
  - sort array `val` with an O(nlogn) algorithm
  - "compact " `val` eliminating duplicate elements and store it in array dist_`val`

Solution

- the array `mark` of `n_dist` elements records at the beginning the number of occurrences of the distinct elements of the multiset
- element `dist_val[i]` is taken if `mark[i]> 0`, `mark[i]` is decremented
- upon return from recursion `mark[i]` is incremented
- `count` stores the number of solutions.

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

```
int rep_perm (int pos, int *dist_val, int *sol,
    int *mark, int n, int n_dist, int count) {
  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] = dist_val[i];
      count=perm_r (
       pos+1,dist_val,sol,mark,n, n_dist,count);
      mark[i]++;
    }
  }
  return count;
}
```

Termination

Iteration on n_dist choices

Occurrence control

Mark and choose

Recursion

Unmark

Simple Combinations

No repetitions

Set

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 doesn't matter

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

Simple Combinations

Note that

- Distinct $\Rightarrow$ the group is a set
- Non ordered $\Rightarrow$ order doesn't matter
- Simple $\Rightarrow$ in each grouping there are exactly k non repeated objects

Two groupings differ because there is at least a different element

Simple Combinations

Binomial coefficient

There are

$$C_{n,k} = \binom{n}{k} = \frac{D_{n,k}}{P_k} = \frac{n!}{k!(n-k)!}$$

simple combinations of n objects k by k (n choose k)

- Recursive definition of the binomial coefficient
  - $\binom{n}{0} = \binom{n}{n} = 1$
  - $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$

Example

- How many simple combinations there are with val = {7, 2, 0, 4, 1}, n=5, k=4?
  - $C_{n,k} = \frac{n!}{k!(n-k)!} = \frac{5!}{4!1!} = 5$
  - {7,2,0,4} {7,2,0,1} {7,2,4,1} {7,0,4,1} {2,0,4,1}
- How many simple combinations there are with val = {1, 9, 5, 4}, n=4, k=3?
  - $C_{n,k} = \frac{n!}{k!(n-k)!} = \frac{4!}{3!1!} = 4$
  - {1,9,5} {1,9,4} {1,5,4} {9,5,4}

Solution

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`
- array `sol` is filled in 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`
- array `mark` is not needed
- `count` stores the number of solutions

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

```
int comb(int pos, int *val, int *sol, int n, int k,
         int start, int count) {
  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(pos+1, val, sol, n, k, i+1, count);
  }
  return count;
}
```

termination

iteration on choices

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

Recursion on next position and next choice

Combinations with repetitions

No upper bound!

Repetitions

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 $(0 \leq k)$. Each of them may be taken at most k times

Order doesn't matter

There are

$$C'_{n,k} = \frac{(n+k-1)!}{k!(n-1)!}$$

Combinations with repetitions of n objects k by k

Combinations with repetitions

Note that

- Distinct $\Rightarrow$ the group is a set
- Non ordered $\Rightarrow$ order doesn’t matter
- "Simple " not mentioned $\Rightarrow$ in each grouping the same object may occur repeatedly at most *k* times
- k may be > n

Combinations with repetitions

Two groupings 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 doesn't matter

When simultaneously casting two dice, how many compositions of values may appear on 2 faces?

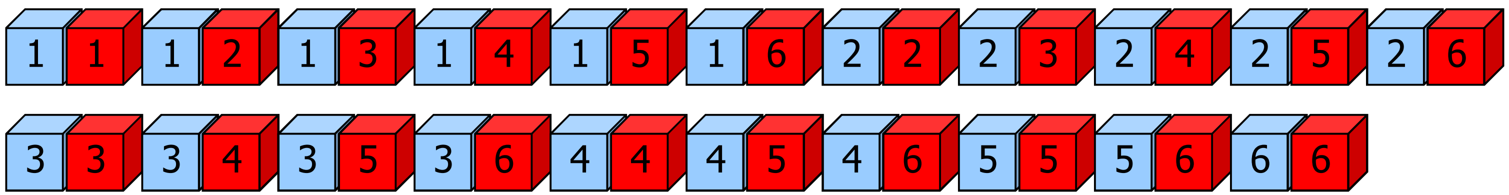
k = 2

n = 6

Model: combinations with repetitions

$C'_{6,\,2} = (6 + 2 - 1)!/2!(6-1)! = 21$

Solution



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

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

```
int rep_comb(int pos,int *val,int *sol,int n,int k,
             int start, int count) {
  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 = rep_comb(pos+1, val, sol, n, k, i, count);
  }
  return count;
}
```

Iteration on choices

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

Recursion on next position

The powerset

Given a set S of k elements (k=card(S)), its powerset $\wp$(S) is the set of the subsets of S, including S itself and the empty set

- Example
  - S = {1, 2, 3, 4 } and k = 4
  - $\wp$(S) = {{}, {4}, {3}, {3,4}, {2}, {2,4}, {2,3}, {2,3,4}, {1}, {1,4}, {1,3}, {1,3,4}, {1,2}, {1,2,4}, {1,2,3}, {1,2,3,4} }

Models

There are 3 models

1. divide and conquer
2. arrangements with repetitons
3. simple combinations

Divide and conquer

- Terminal case: empty set
- Recursive case: powerset for k-1 elements union either the empty set or the k-th element s_k
- Iteration on all the elements in S

$$\wp(S_k) = \begin{cases} \emptyset & \text{se } k = 0 \\ \{\ \wp(S_{k-1}) \cup s_k\ \} \cup \{\ \wp(S_{k-1})\ \} & \text{se } k > 0 \end{cases}$$

Divide and conquer

- 2 distinct recursive branches are used, depending on the current element being included or not in the solution
- in `sol` we directly store the element, not a flag to indicate its presence/absencd
- index `start` is used to exclude symmetrical solutions
- return value `count` represents the total number of sets.

Divide and conquer

```
int powerset(int pos, int *val, int *sol, int k,
             int start, int count) {
   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(pos+1, val, sol, k, i+1, count);
   }
   count = powerset(pos, val, sol, k, k, count);
   return count;
}
```

For all elements from start onwards

Include element and recur

Do not add and recur

Arrangements with repetitions

Each subset is represented by the `sol` array having k elements:

- The set of possible choices for each position in the array is {0, 1}, thus n = 2. The for loop is replaced by 2 explicit assignments
- `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

```
int powerset(int pos,int *val,int *sol,int k,int count) {
  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(pos+1, val, sol, k, count);
  sol[pos] = 1;
  count = powerset(pos+1, val, sol, k, count);
  return count;
}
```

Termination: print solution

Do not take element pos

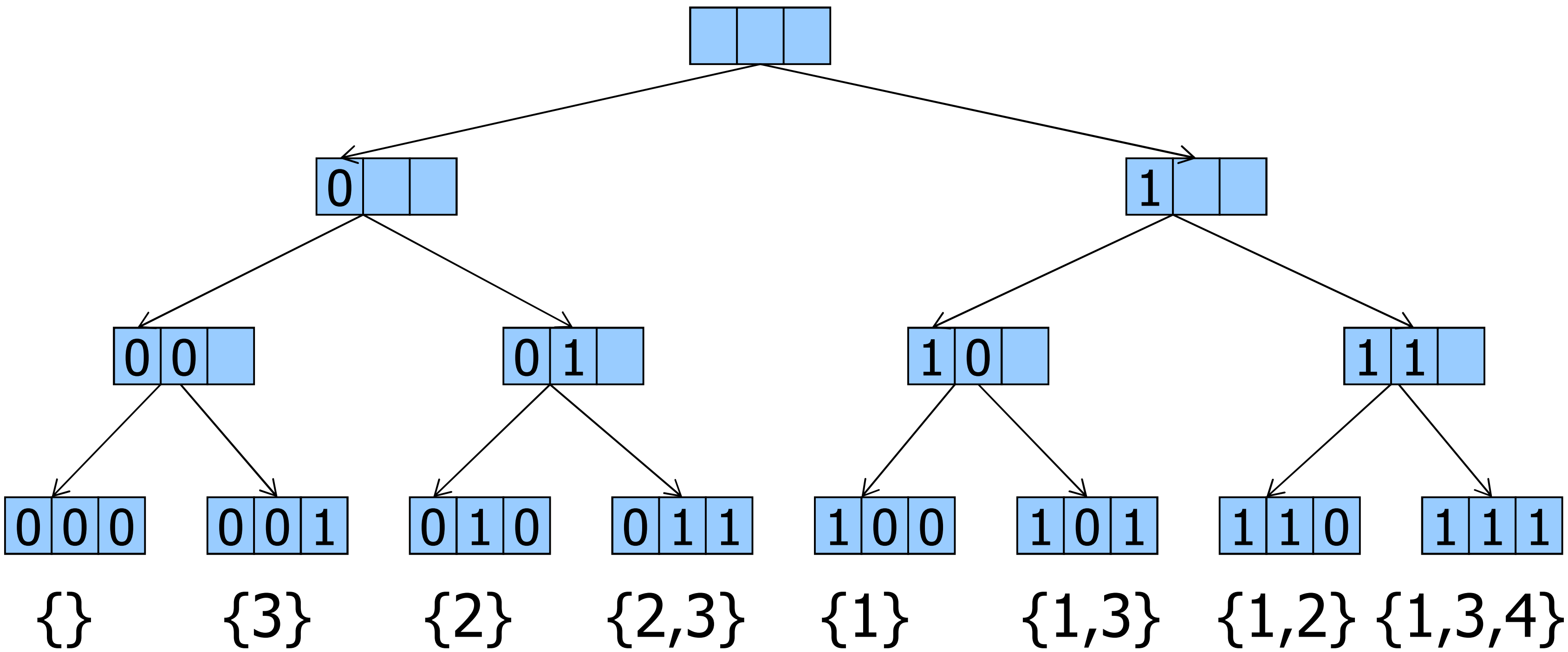
Recur on pos+1

Backtrack take element pos

Recur on pos+1

Solution

n = 2, k = 3, val = {1, 2, 3}



Simple combinations

- Union of the empty set and of the powerset of size 1, 2, 3, …., k
- Model: simple combinations of k elements taken by groups of n
  - $\wp(S) = \{ \varnothing \} \cup \bigcup_{n=1}^{k} \binom{k}{n}$
- the 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

Wrapper

```
int powerset(int* val, int k, int* sol){
    int count = 0, n;
    printf("{ }\n");
    count++;
    for(n = 1; n <= k; n++){
        count += powerset_r(val, k, sol, n, 0, 0);
    }
    return count;
}
```

Empty set

Iteration on recursive calls

Solution

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

Terminal case: predefined number of elements reached

For all elements from start onwards

Partitions of a set

Given a set I of n elements, a collection $S = \{S_i\}$ of non empty blocks forms a partition of I iff both the following conditions hold:

- blocks are pairwise disjoint:

$$\forall\ S_i, S_j \in S \text{ con } i \neq j \quad S_i \cap S_j = \varnothing$$

- their union is I:

$$I = \cup_i S_i$$

The number of blocks k ranges from 1 (block = set I) to n (each block contains only 1 element of I).

Example

I = {1, 2, 3, 4 } n = 4

K=1	k = 2	k = 3	k = 4
1 partition	7 partitions	6 partitions	1 partition
{1, 2, 3, 4}	{1, 2, 3}, {4}	{1, 2}, {3}, {4}	{1}, {2}, {3}, {4}
	{1, 2, 4}, {3}	{1, 3}, {2}, {4}	
	{1, 2}, {3, 4}	{1}, {2 ,3}, {4}	
	{1, 3, 4}, {2}	{1, 4}, {2}, {3}	
	{1, 3}, {2, 4}	{1}, {2, 4}, {3}	
	{1, 4}, {2, 3}	{1}, {2}, {3, 4}	
	{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 {1, 3}, {2}, {4} AND {2}, {3, 1}, {4} are identical

Number of partitions

The global number of partitions of a set I of n objects is given by Bell's numbers, defined by the following recurrence

$B_0 = 1$

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

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

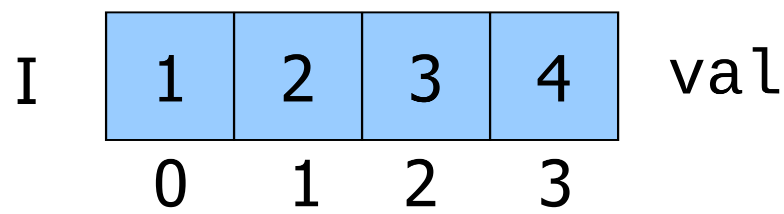
Partitions of a set S

Representing partitions

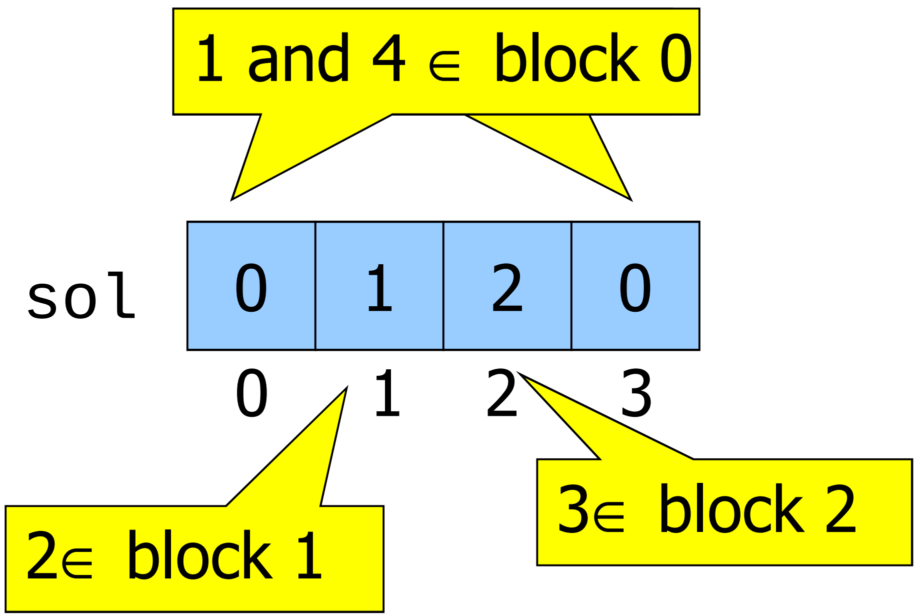
- Given the element, identify the unique block it belongs to
- Given the block, list the elements that belong to it

First approach preferrable, as it works on an array of integers and not on lists

Example



If I = {1, 2, 3, 4}, n= card(I)=4 and if a partitioning on k = 3 blocks (having index 0, 1 e 2) is requested, partition {1, 4}, {2}, {3} is represented as:



Problems

Given I and n=card(I), find:

- Any partition
- All partitions in k blocks where k ranges between 1 and n
- All partitions in k blocks

Arrangements with repetitions

Er's algorithm

Arrangements with repetitions

- 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.

Arrangements with repetitions

- The role of n and k is reverse with respect to previous examples (where n was the number of choices and k the size of the solution)
- It is a generalization of the powerset removing the constraint on the choice being restricted to 0 or 1
- Need to check in the terminal case that the block is not empty (by computing how many times each block occurs).

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

```
void arr_rep(int pos,int *val,int *sol,int n,int k) {
  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(pos+1, val, sol, n, k);
  }
}
```

Block occurrence array

Occurrence computation

Occurrence control

Discarded solution

Recursion

Er's algorithm (1987)

Compute all the partitions of `k` objects stored in array `val` in `m` blocks with `m` ranging from `1` to `k`:

- index `pos` to walk through the `k` objects. Recursion terminates when `pos >= k`
- index `m` to walk through the blocks that may be used at that step
- array `sol` of `k` elements for the solution

Er's algorithm (1987)

- 2 recursions
  - Assign the current object to one of the block with index in the range from 0 to `m` and recur on the next object
  - Assign the current object to block `m` and recur on the next object and on the number of blocks increased by 1

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

```
void SP_rec(int n, int m, int pos, int *sol, int *val) {
  int i, j;
  if (pos >= k) {
    printf("partition in %d blocks: ", m);
    for (i=0; i<m; i++)
      for (j=0; j<k; j++)
        if (sol[j]==i)
          printf("%d ", val[j]);
    printf("\n");
    return;
  }
  for (i=0; i<m; i++) {
    sol[pos] = i;
    SP_rec(n, m, pos+1, sol, val);
  }
  sol[pos] = m;
  SP_rec(n, m+1, pos+1, sol, val);
}
```

Termination condition

Recursion on objects

Recursion on objects and blocks

Er's algorithm version 02

Computing all the partitions of `k` objects stored in array `val` in exactly `n` blocks

- As before, passing parameter `n` used in the terminal case to "filter" valid solutions

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

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

Termination condition

Filter

Recursion on objects

Recursion on objects and blocks