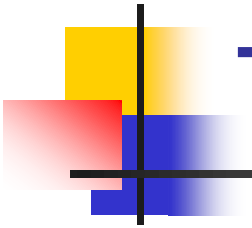
Search and optimization problems

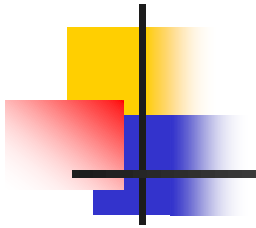
Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino



Types of problems

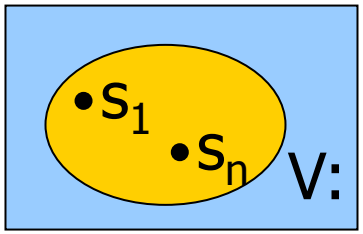
Computational problems:

- Solution based on a mathematical procedure that lead without choices and in a finite number of steps to the solution
- Examples:
  - factorial
  - determinant
  - Fibonacci, Catalan, Bell numbers etc.



Search problems:

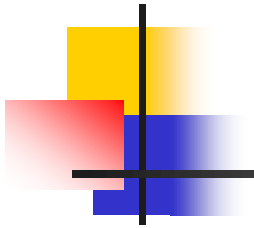
- given:
  - S: space (set) of all possible solutions
  - V: space of all valid solutions
  - in general $V \subset S$
- check whether $V=\varnothing$
- list the elements of V
  - at least 1
  - all in case of enumeration.



S = solution space

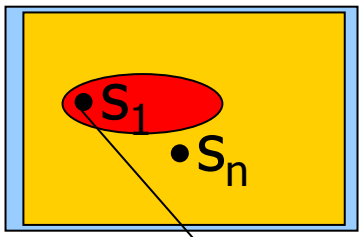
valid solutions

- Examples:
  - powerset that satisfies a condition
  - 8 queens,
  - Sudoku,
  - all simple paths from a source in a graph



Optimization problems:

- $S \equiv V$
- given a goal function f (cost or advantage), select one or more solutions for which f is minimum or maximum
- enumeration required



S = solution space

≡

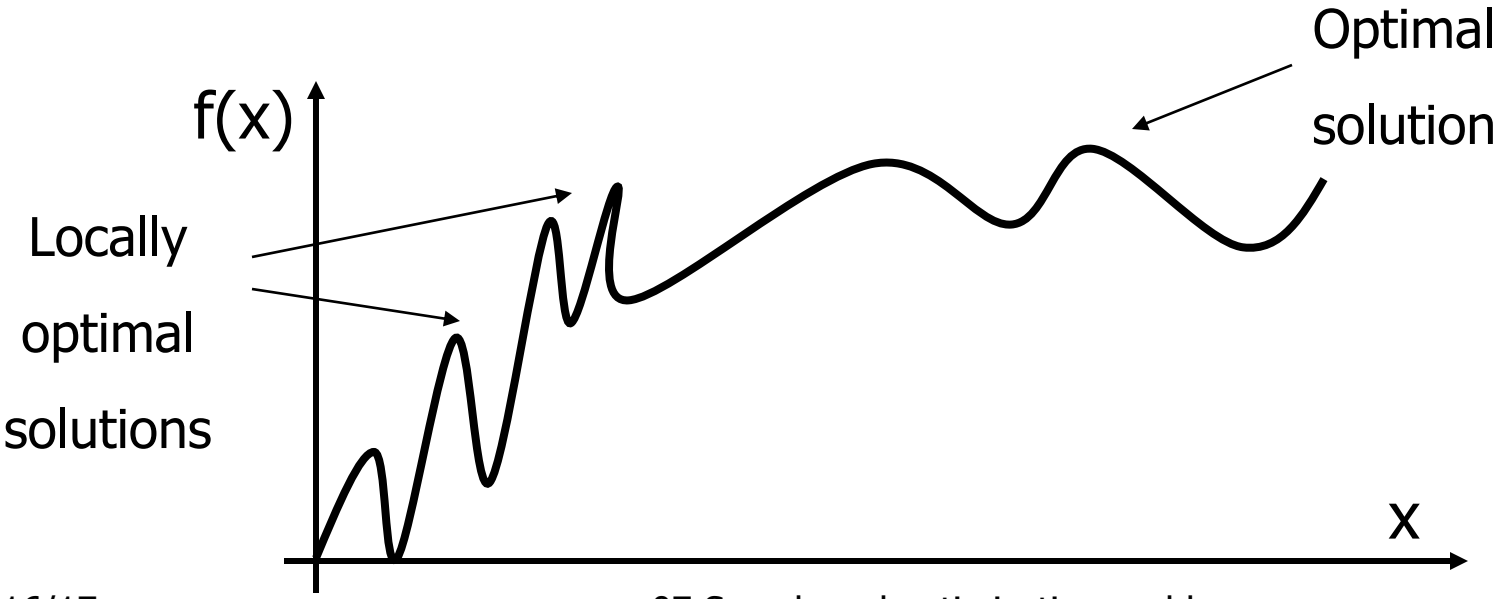
V: valid solutions

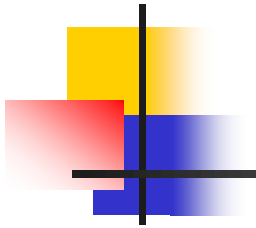
$f(s_1)$ minimum (maximum)

- Examples:
  - maximize the value of objects compatible with the maximum capacity of a box
  - in a graph find all simple paths from a source having maximum length

Minimum (maximum): absolute or in a contiguous domain:

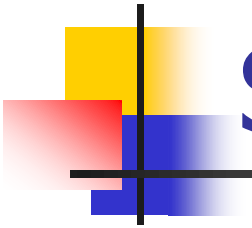
- *Optimal* solution: absolute min/max
- *Locally optimal solution*: local min/max





Possible answers:

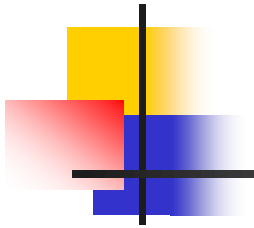
- count the number of valid solutions
- find at least one valid solution
- find all valid solutions
- among valid solutions, find the one (ones) that is (are) optimal according to an optimization criterion.



Solution space search

Incremental approach:

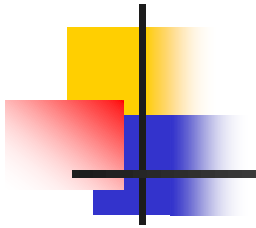
- initially empty solution
- solution extension applying choices
- termination when solution found



Generic algorithm using a data structure DS:

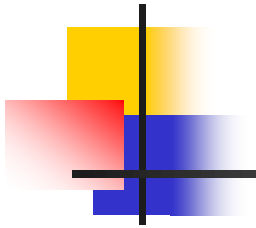
Search():

- put initial solution in DS
- while DS doesn't become empty:
  - extract a partial solution from DS;
  - if it is valid, Return Solution
  - apply possible choices and put resulting partial solutions in DS
- return failure.



When DS is:

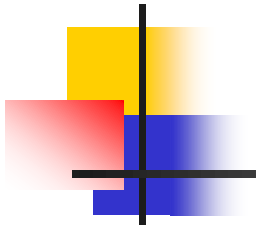
- a queue (FIFO), search is breadth-first
- a stack (LIFO), search is depth-first
- a priority queue, search is best-first.



If the algorithm:

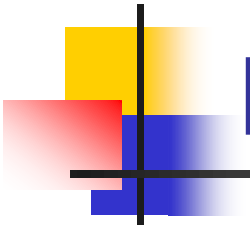
- doesn't know anything about the problem, it is non informed
- has specific knowledge (heuristics), it is informed

If the algorithm can explore the whole space, it is complete.



Approach: search algorithm

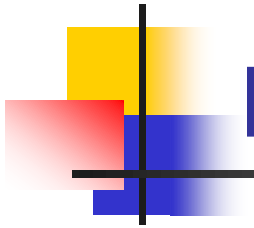
- depth-first
- non informed
- complete
- recursive.



Reresentation

Solution space represented as a search tree:

- of height n, where n is the solution size
- of degree k, where k is the maximum number of possible choices
- the root is the initially empty solution
- intermediate nodes are labelled with partial solutions
- leaves are solutions. A function checks whether they are valid

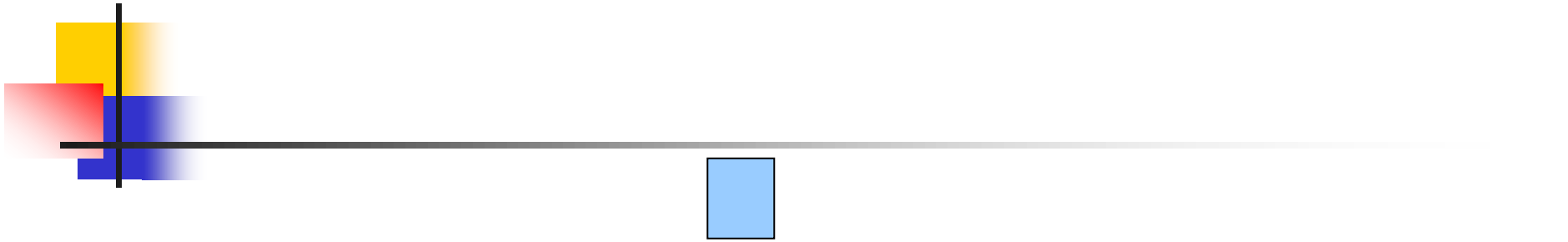


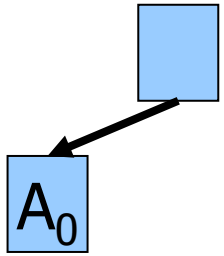
Example

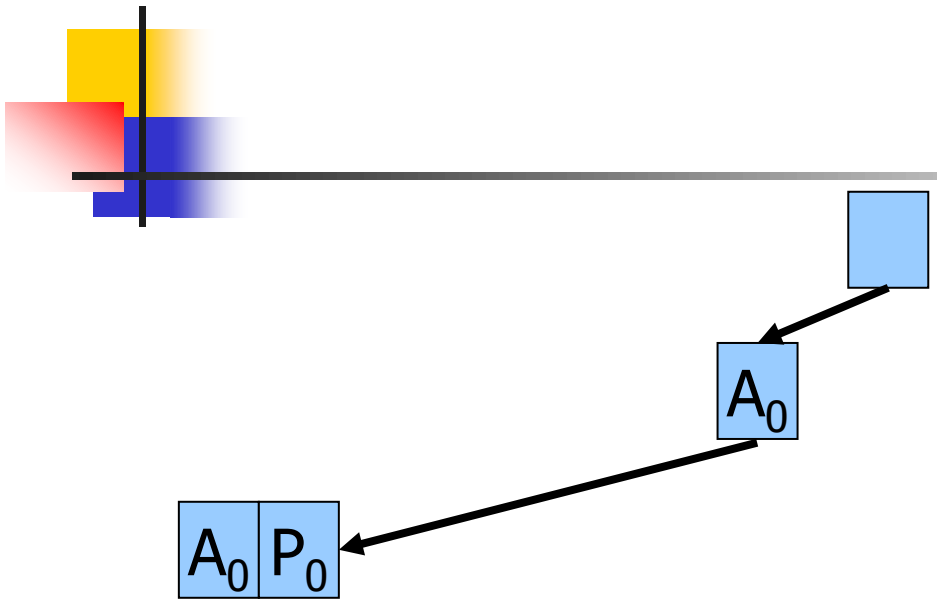
In a restaurant there is a 3-course menu with a choice of an appetizer, a first course and a second course.

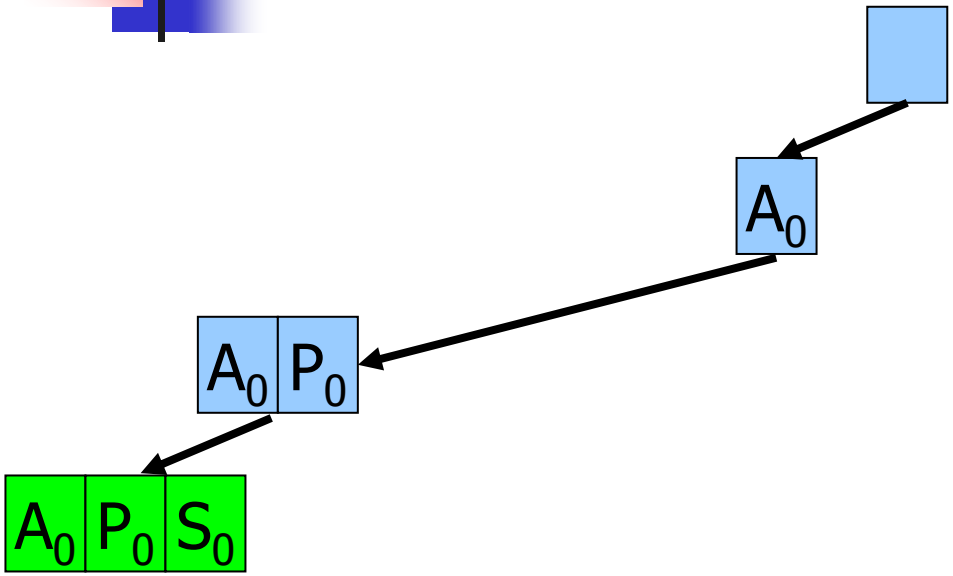
There are 2 appetizers A_0, A_1, 3 first-course dishes P_0, P_1 and P_2 and 2 second-course dishes S_0, S_1.

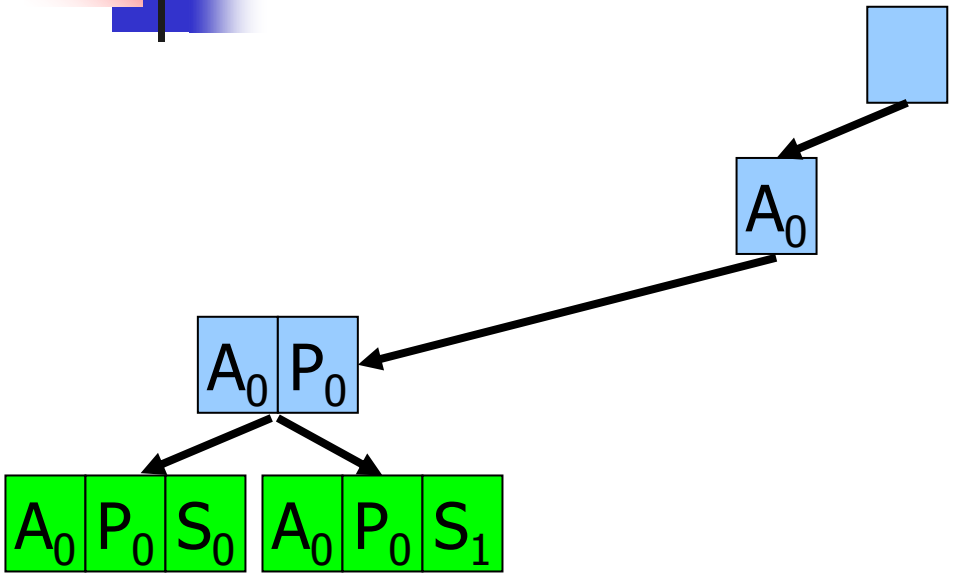
How many and which menus can be offered to customers?

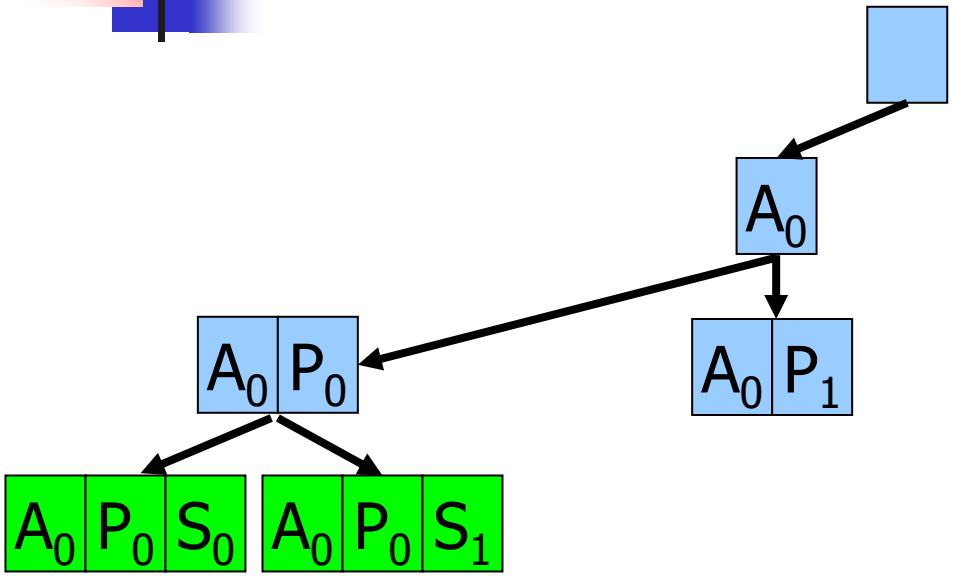


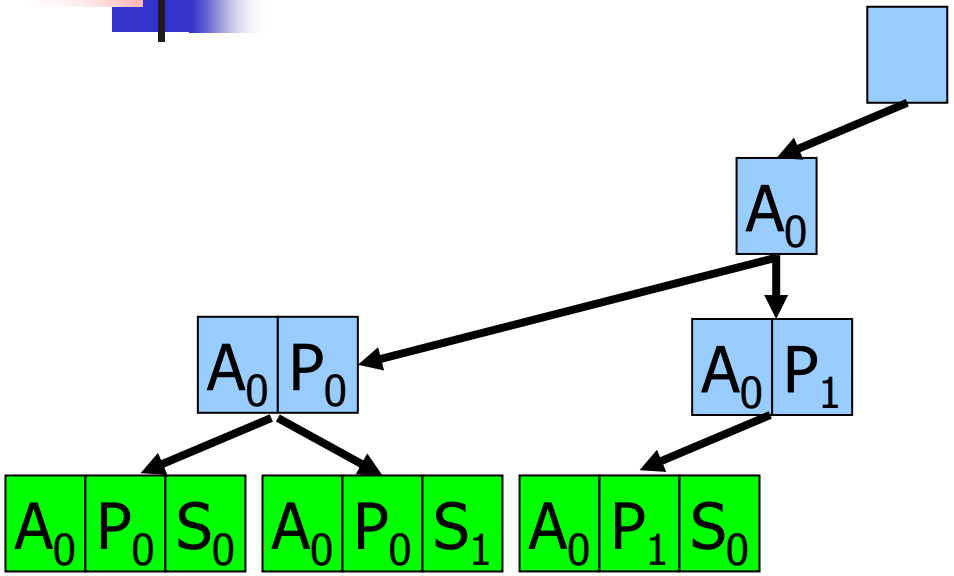

A_0

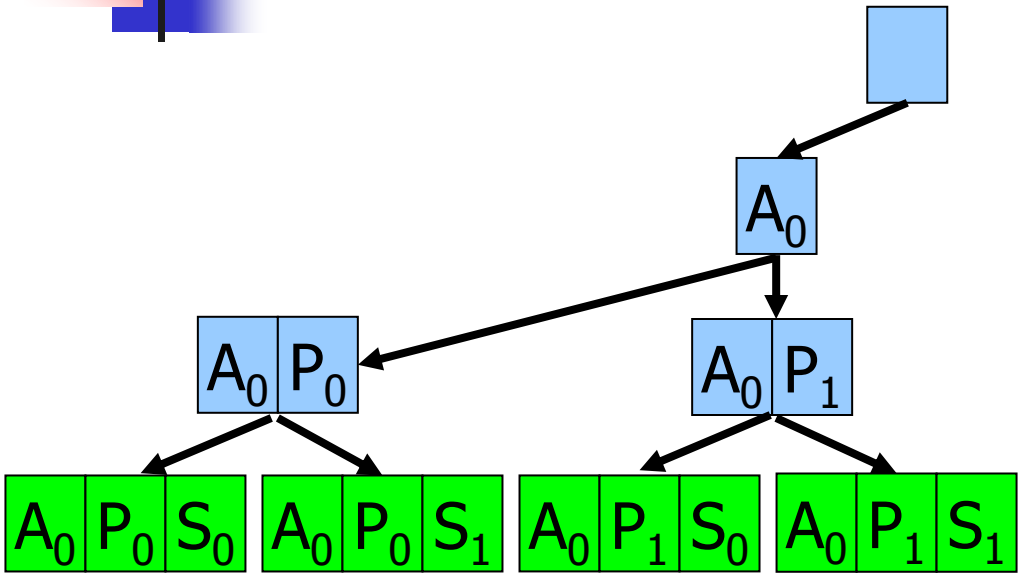

A_0
A_0 P_0

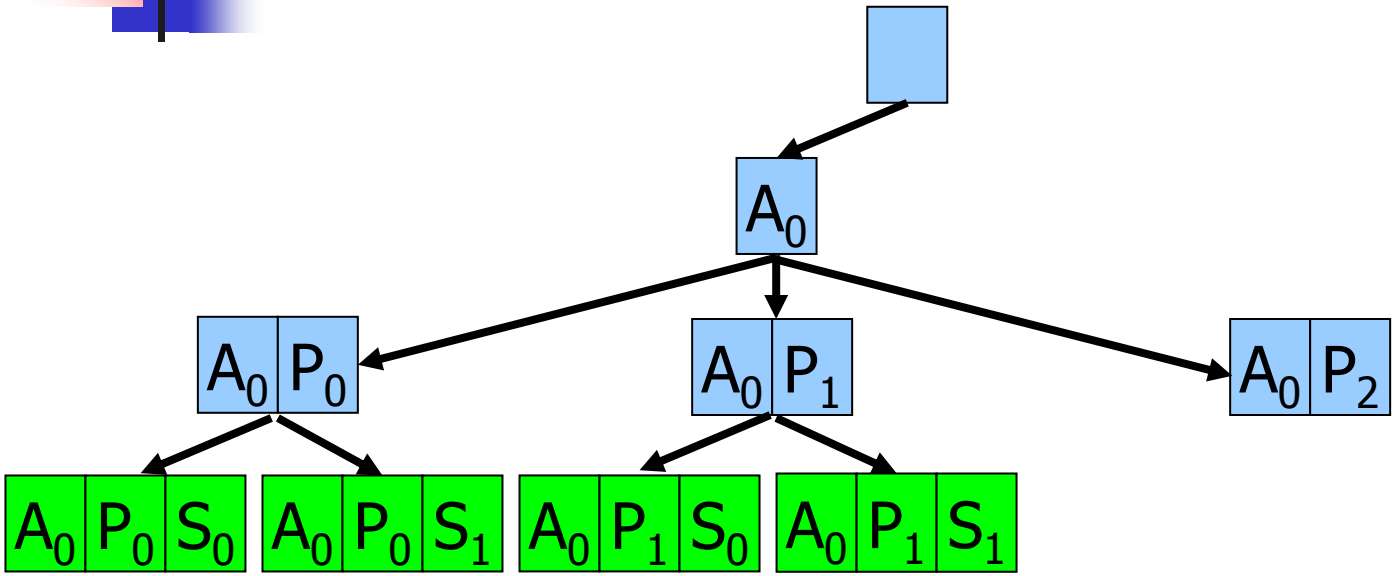

A_0
A_0 P_0
A_0 P_0 S_0

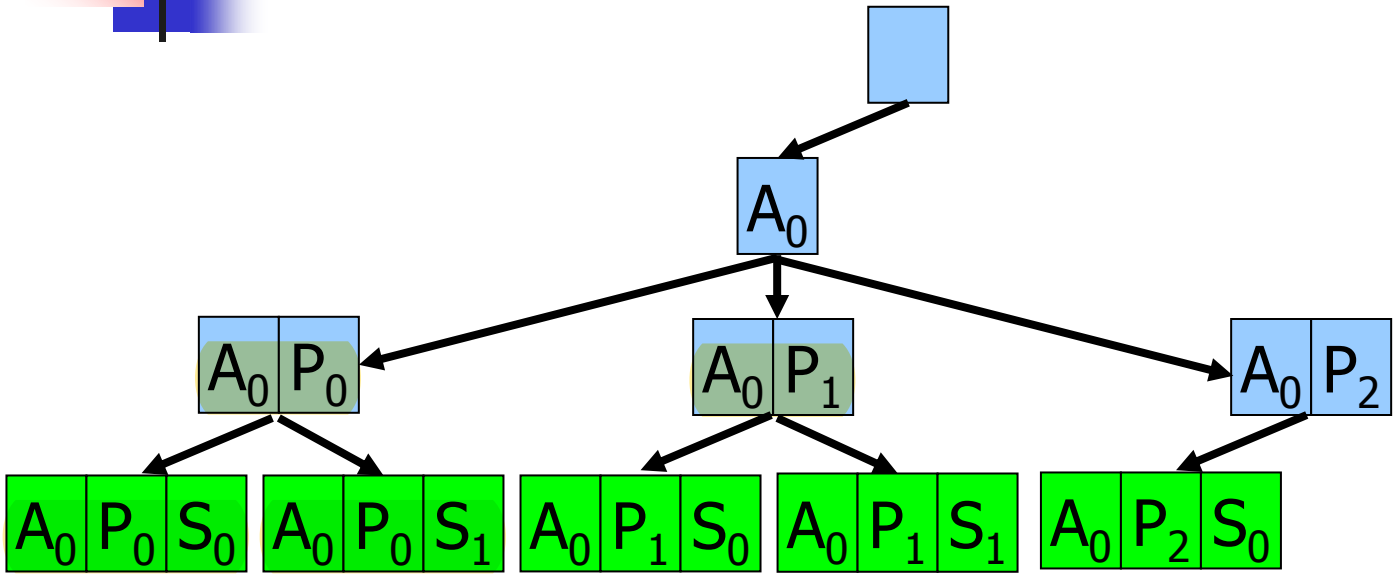

A_0
A_0 P_0
A_0 P_0 S_0
A_0 P_0 S_1

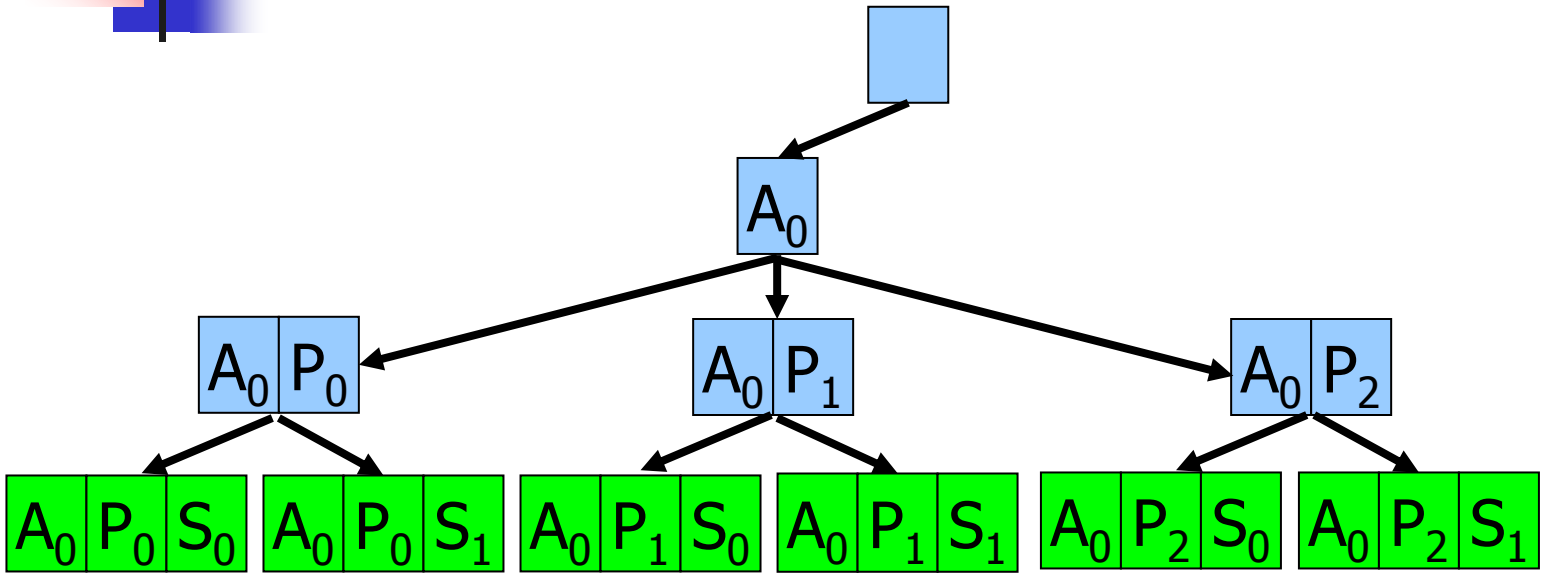

A_0
A_0 P_0
A_0 P_1
A_0 P_0 S_0
A_0 P_0 S_1

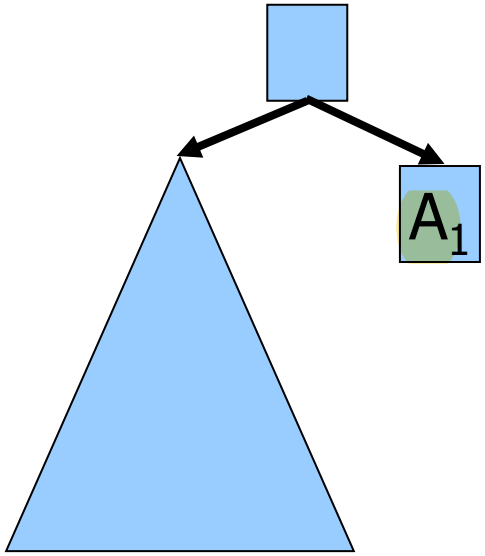

A_0
A_0 P_0
A_0 P_1
A_0 P_0 S_0
A_0 P_0 S_1
A_0 P_1 S_0


A0
A0 P0
A0 P1
A0 P0 S0
A0 P0 S1
A0 P1 S0
A0 P1 S1


A_0
A_0 P_0
A_0 P_1
A_0 P_2
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
A_0 P_0
A_0 P_1
A_0 P_2
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
A_0 P_0
A_0 P_1
A_0 P_2
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



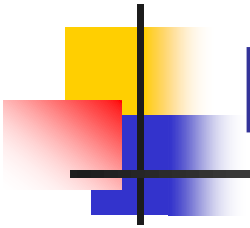
And so on

Combinatorics and Search Space

- Choices as elements of a group
- Solution space determined by item grouping (arrangenent) rules
- Combinatorics: association rules
  - distinct items
  - sorted items
  - repeated items

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.

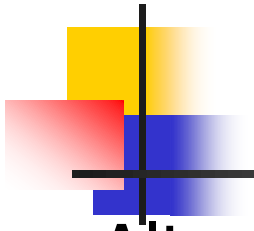
Basic principle: addition

If a set S of objects is partitioned in pair-wise disjoint subsets S_0 .. S_{n-1}

$S = S_0 \cup S_1 \cup .... S_{n-1}$ && $\forall$ i $\neq$ j $S_i \cap S_j = \varnothing$

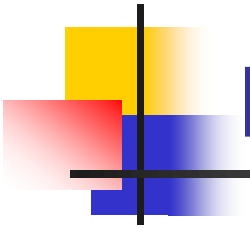
The number of objects in S may be determined adding the number of objects of each of the sets S_0 .. S_{n-1}

$$|S| = \sum_{i=0}^{n-1} |Si|$$



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 separate 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?

Disjoint sets $\Rightarrow$

Model: principle of addition

Number of choices = 4 + 5 = 9

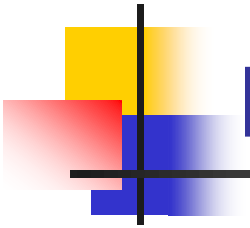
Basic principle: multiplication

Given n sets S_i $(0 \leq i < n)$ 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:

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

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, 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 $p_0 \cdot p_1 \ldots \cdot p_{n-1}$ 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 and 4 desserts.
How many different menus can be the restaurant offer?

Model: principle of multiplication

Number of choices = 2 x 3 x 2 x 4 = 48

Grouping criteria

We can group k objects taken from a group S of n elements keeping into account:

- **unicity** of the elements: 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?

Simple arrangements

no repetitions

set

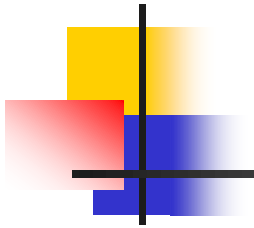
A simple arrangement $D_{n,k}$ of n distinct objects of class k (k by k) is an ordered subset composed by k out of n objects ($0 \leq k \leq n$).

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.



Note that:

- 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.

Exam

positional representation: order matters!

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

k = 2

n = 4

no repeated digits

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 }

Arrangements with repetitions

no upper bound!

repetitions

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 ($0 \leq k$) each of whom may be taken up to k times.

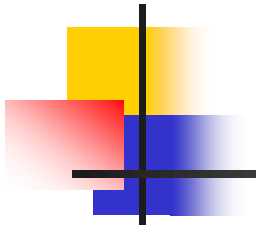
set

order matters

There ar

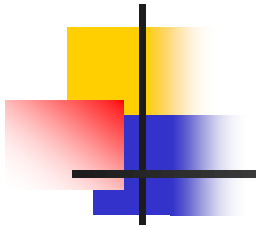
$$D'_{n,k} = n^k$$

Arrangements with repetitions of n objects taken k by k.



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



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 }

Simple Permutations

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.

There are

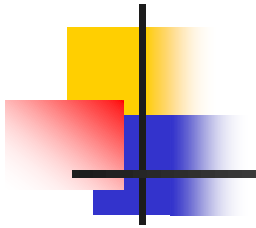
$$P_n = D_{n,n} = n!$$

simple permutations of n objects.

no repetitions

order matters

set



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 }

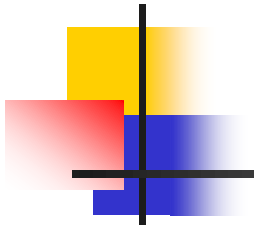
Permutations with repetitions

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

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



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

$\alpha = 2$

Model: permutations with repetitions

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

Solution
{ OOR, ORO, ROO }

Simple Combinations

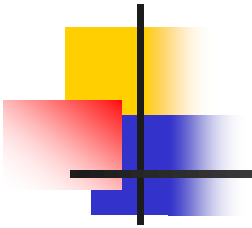
no repetitions

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

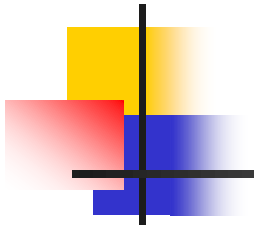
The number of combinations of n elements k by k equals the number of arrangements of n elements k by k divided by the number of permutations of k elements.



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.

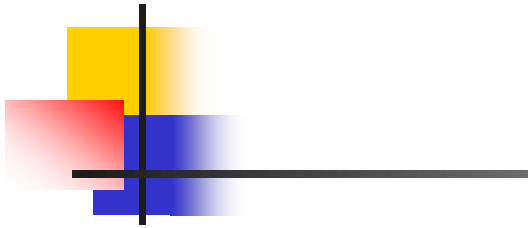


There are:

binomial coefficient

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

simple combination 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

order doesn't matter

How many 3-tuples can be composed with the lottery's numbers (from 1 to 90)?

k = 3

n = 90

Model: simple combinations

$$C_{90,3} = \frac{90!}{3!(90-3)!} = \frac{90 \cdot 89 \cdot 88 \cdot 87!}{3 \cdot 2 \cdot 87!} = 117480$$

Combinations with repetitions

no upper bound!

repetitions

set

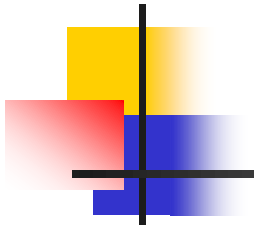
A combination with repetitions $C'_{n,k}$ of *n* distinct objects of class *k* (k by k) is a non ordered subset made of *k* of the *n* objects ($0 \leq k$). Each of them may be taken at most k times.

order doesn't matter

There are

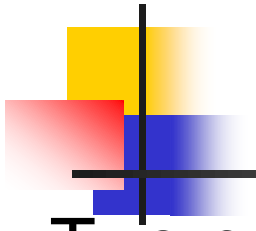
$$C'_{n,k} = \frac{(n+k-1)!}{k!(n-1)!}$$

Combinations with repetitions of n objects k by k.



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.



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?

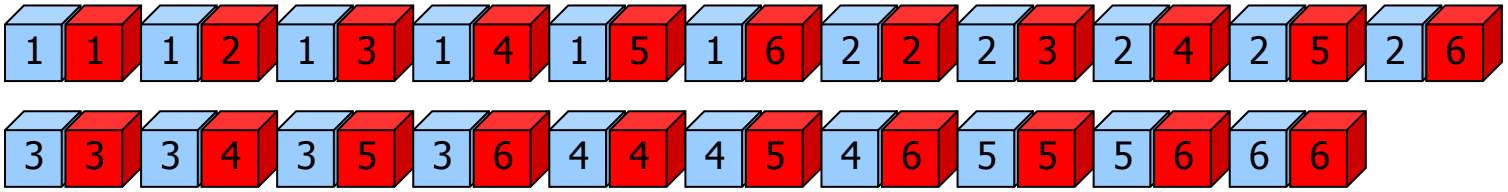
k = 2

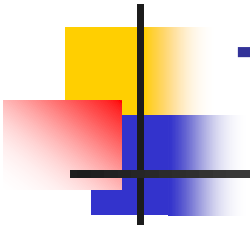
n = 6

Model: combinations with repetitions

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

Solution





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} }

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

1 partition:

{1, 2, 3, 4}

k = 2

7 partitions:

{1, 2, 3}, {4}

{1, 2, 4}, {3}

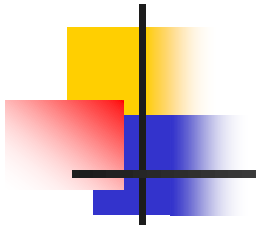
{1, 2}, {3, 4}

{1, 3, 4}, {2}

{1, 3}, {2, 4}

{1, 4}, {2, 3}

{1}, {2, 3, 4}



k = 3

6 partitions:

{1, 2}, {3}, {4}

{1, 3}, {2}, {4}

{1}, {2 ,3}, {4}

{1, 4}, {2}, {3}

{1}, {2, 4}, {3}

{1}, {2}, {3, 4}

k = 4

1 partition:

{1}, {2}, {3}, {4}

Number of partitions

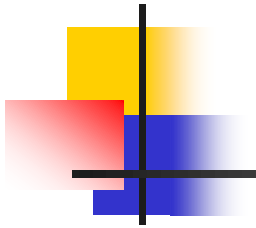
The global number of partitions of a set I of n objects is given by Bell's numbers, defined by the following recurrence:

$B_0 = 1$

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

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.



Note: the order of the blocks and of the elements within each block doesn't matter.

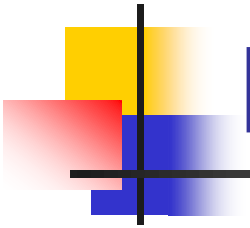
As a consequence the 2 partitions:

{1, 3}, {2}, {4} e {2}, {3, 1}, {4}

are identical.

Exhaustive search of the solutions space

Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino

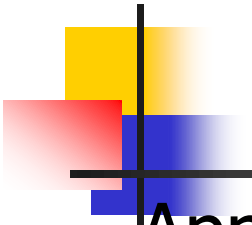


Decomposition in subproblems

It is the most important step while designing a recursive solution:

- it is necessary to identify the problem solved by each single recursion
- i.e., to share the task among all recursive calls.

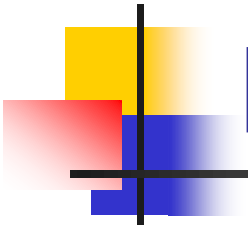
The approach is distributed, without a unique vision of the solution.



Approaches:

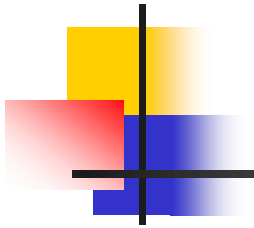
- each recursion selects an element of the solution. Termination: the solution has reached the required size or no other choices are available
- recursion considers one of the elements of the set to decide whether to take it or not and where to place it in the solution.

We follow the first approach as it is more intuitive.



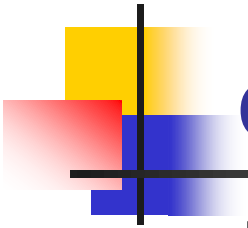
Data structures

- Data structures
  - may be global, i.e., common to all instances of the recursive function
  - local, i.e. proper to each instance
- Global data structures:
  - data of the problem (matrix, map, graph), constraints, available choices, solution
- Local data structures:
  - indices of the level of recursive call, local copies of data structures, indices or pointers to parts of global data structures

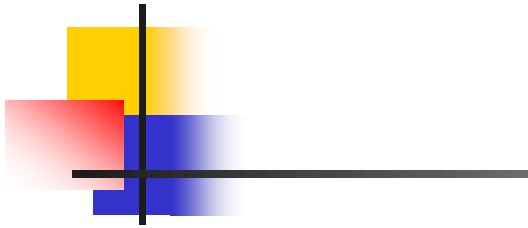


- Global doesn’t necessarily imply the use of global C variables
- Use of global C variables for global data structures:
  - discouraged but not forbidden when recursive functions work on few and well known data
  - pro: few parameters passed to the recursive function
- Solution: all data (global and local) passed as parameters. It’s possible to group them in a C struct to improve readability.

Types of data structures for integer objects



- Non integer objects: use symbol tables to reduce to integers
- starting set (sets):
  - just one: array `val`
  - many: subarrays of type `Level`
  - alternative: lists
- solution: storage not requested, it is enough to list them:
  - array `sol`
  - scalar variable (e.g. `count`) passed as parameter by value and returned



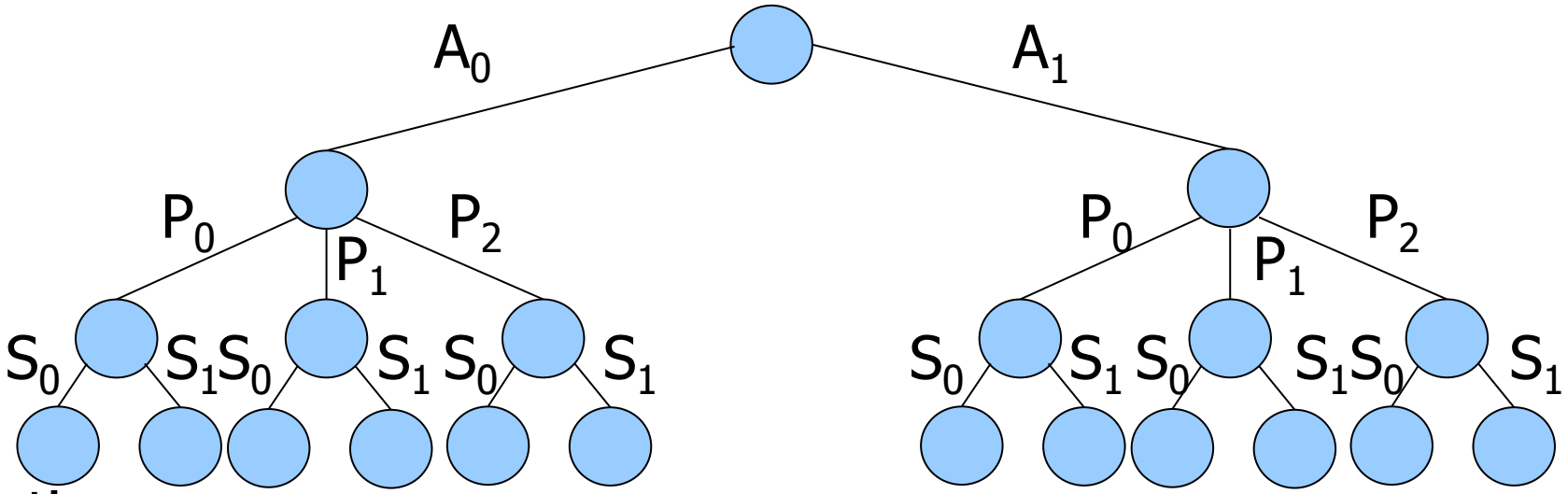
- indices:
  - `pos` identifies the level of recursion and is used to decide which selection slots should be used or which solution slots should be filled
  - `n` and `k`: indicate the size of the problem and of the solution
- constraints: not all choices are allowed. The allowed ones comply with:
  - static
  - dynamic constraints (e.g. array `mark`).

Principle of multiplication

- n choices are made in sequence, represented by a tree
- 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.

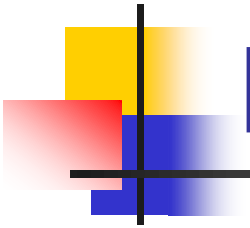
Example

Menu with selection among 2 appetizers (A_0, A_1), 3 main courses (P_0, P_1, P_2) and 2 second courses (S_0,S_1) (n=k=3). Tree of degree 3 and height 3, 12 paths from root to leaves.



Solution:

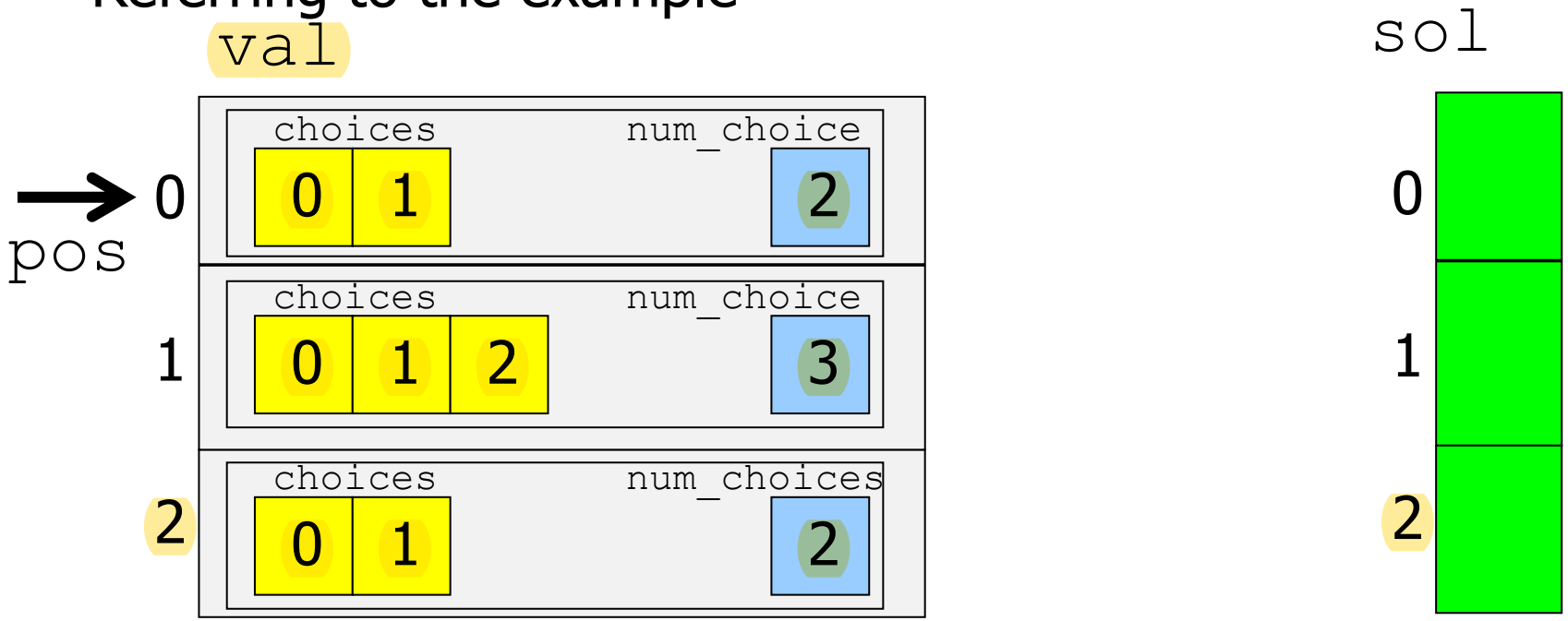
(A_0,P_0,S_0), (A_0,P_0,S_1), (A_0,P_1,S_0), (A_0,P_1,S_1), (A_0,P_2,S_0), (A_0,P_2,S_1), (A_1,P_0,S_0), (A_1,P_0,S_1), (A_1,P_1,S_0), (A_1,P_1,S_1), (A_1,P_2,S_0), (A_1,P_2,S_1)

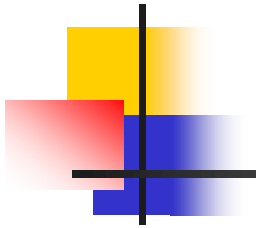


Basic search principles

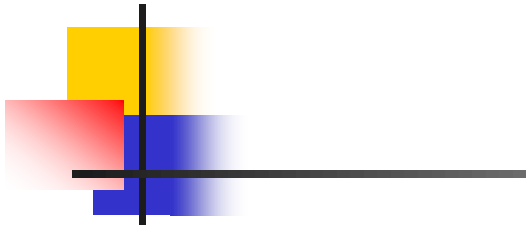
- n *decisions in sequence* are taken, each among different *choices*, whose number is fixed for a level, but varies from level to level
- there is a 1:1 matching between choices and a (possibly non contiguous) subset of integers
- possible choices are stored in array `val` of size `n` containing structures 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.

Referring to the example

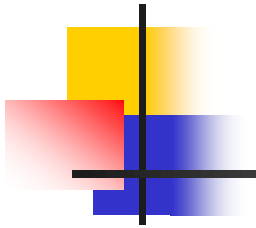




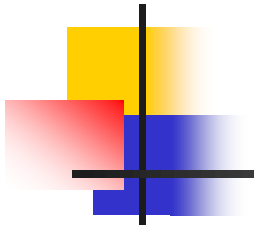
- the goal is to enumerate all solutions, searching their space
- all solutions are valid
- recursive calls are associated to the solution, whose size grows by 1 at each call. Termination: size of current solution equals final desired size



- a solution is represented as an array `sol` of n elements that stores the choices at each step
- at each step index `pos` indicates the size of the partial solution
- if `pos>=n` a solution has been found

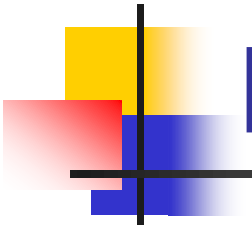


- the recursive step iterates on possible choices for the current value of `pos`, i.e., the contents of `sol[pos]` is taken from `al[pos].choices[i]` extending each time the solution's size by 1
- and recurs on the `pos+1-th` choice
- `count` is the integer return value for the recursive function and counts the number of solutions.



```
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));
```

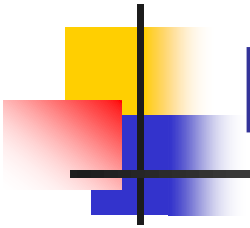
```
int princ_mult(int pos, Level *val, int *sol,
               int n, int count) {
  int i;
  if (pos >= n) {
    for (i = 0; i < n; i++)
      printf("%d ", sol[i]);
    printf("\n");
    return count+1;
  }
  for (i = 0; i < val[pos].num_choice; i++) {
    sol[pos] = val[pos].choices[i];
    count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



Model

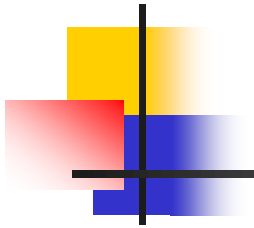
The search space may modelled as the space of:

- simple arrangements
- arrangements with repetitions (+ powerset)
- simple permutations
- permutations with repetition
- simple combinations
- combinations with repetitions
- (partitions).



Basic search principles

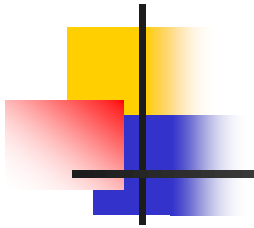
- Imagine to take *decisions in sequence,* each among different *choices* without any information to drive the decision
- possible choices are stored in integer array `val` of size `n`
- we want to enumerate all solutions, searching the whole space and deciding once a solution is reached whether it is valid or optimal.



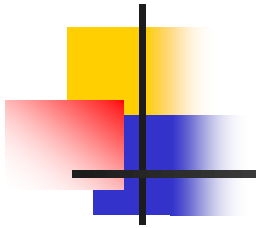
Alternatives:

- recursive calls are associated to the solution, whose size grows by 1 at each of them. Termination: current solution size is desired size
- recursive calls are associated to choices. At each call a choice is made. Termination: all choices are exhausted.

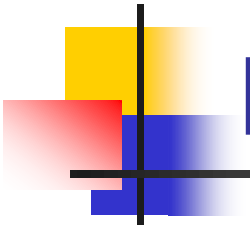
We follow the first approach.



- a solution is represented by an integer array `sol` of `k` elements that stores choices made at each step
- at each step `pos` indicates the size of the partial solution
  - if `pos>=k`, a solution has been found whose validity/optimality is checked



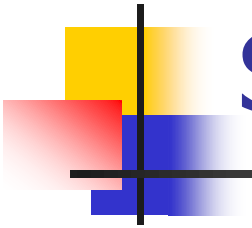
- else, if a `pos+1-th` choice is possible, use it to extend `sol` and proceed recursively
  - else, cancel the last choice `pos` (backtrack) and restart from the `pos-1-th` choice
- iteration: the contents of `sol[pos]` is taken from `val`.



Backtracking

Backtracking is not a paradigm, unlike divide and conquer, the greedy approach or dynamic programming, as there is no general scheme.

It is rather an algorithmic technique to orderly explore all possible states of a search space identifying solutions and valid solutions.



Simple arrangements

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.

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

```
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

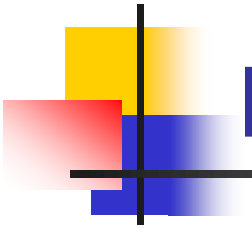
iteration on n choices

repetion control

mark and choose

unmark

recursion



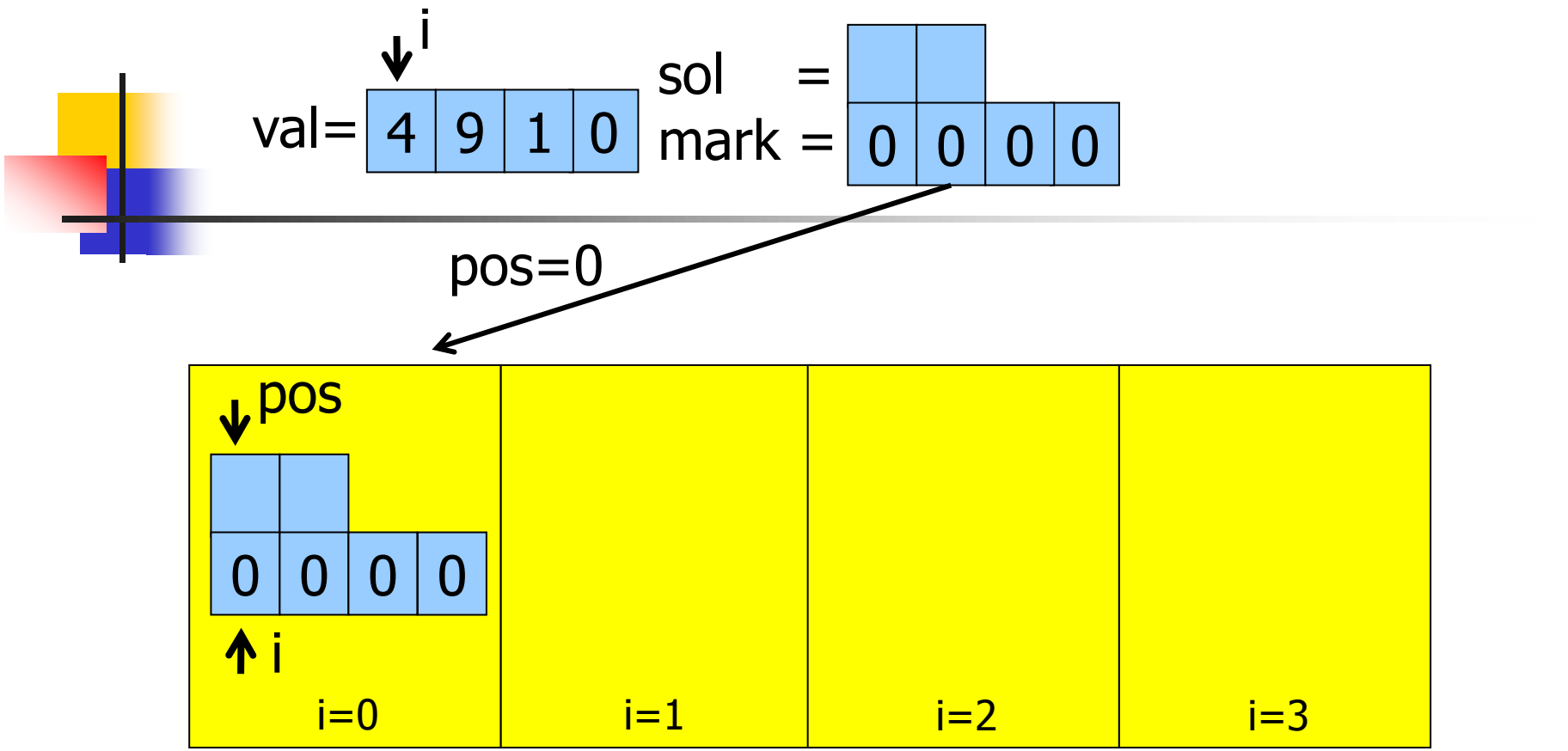
Example

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

n = 4, k = 2, val = {4, 9, 1, 0 }

$D_{4,2} = 4!/(4-2)! = 4 \cdot 3 = 12$

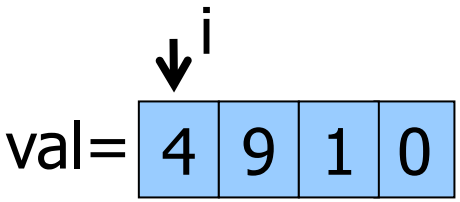
Solution:
{49, 41, 40, 94, 91, 90, 14, 19, 10, 04, 09, 01 }

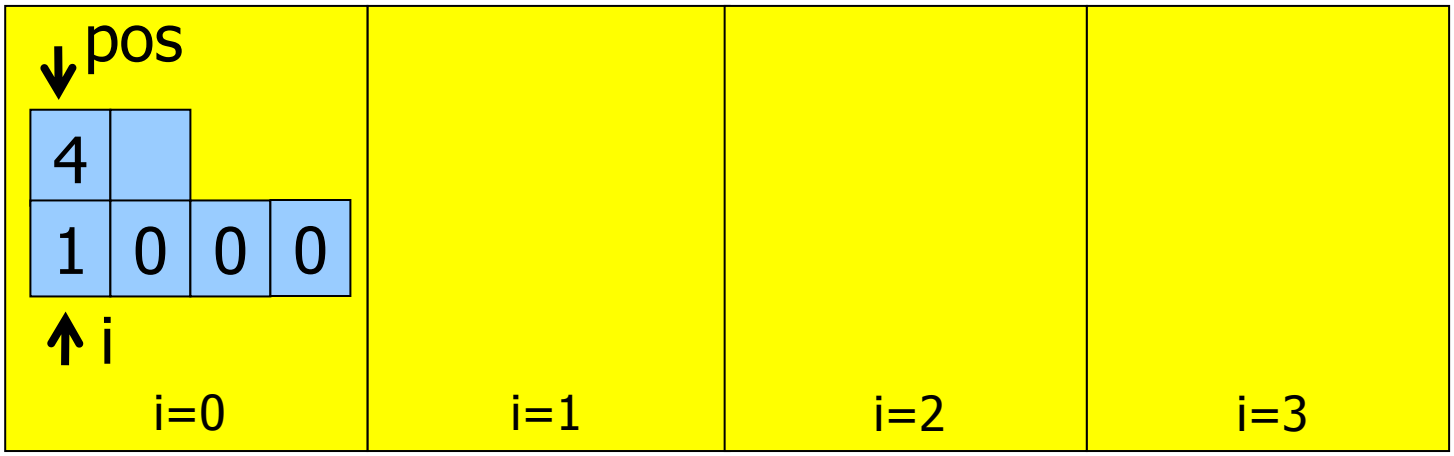


mark[i] = 0

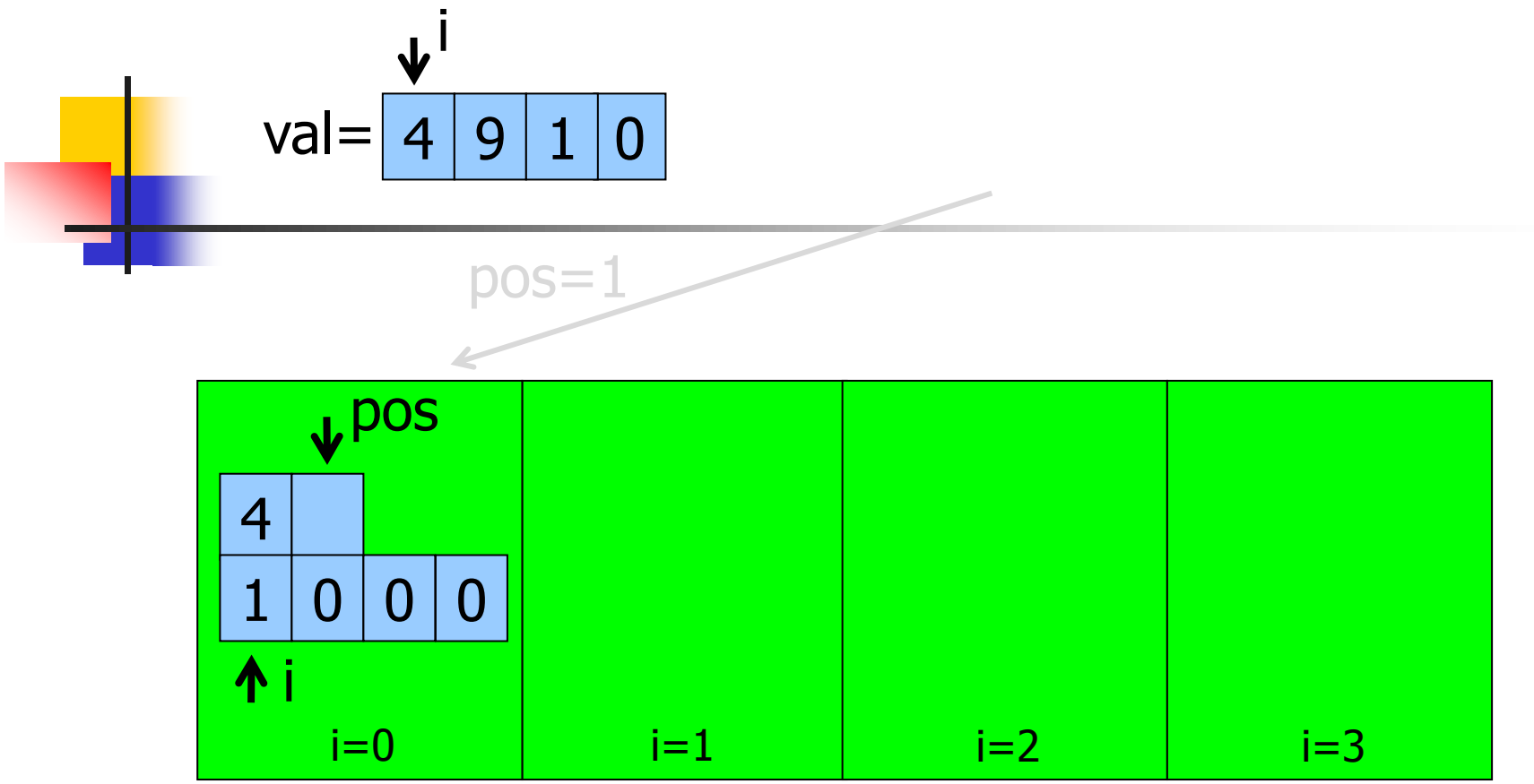
sol[pos] = val[i]

mark[i] = 1

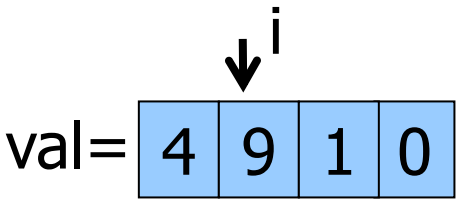


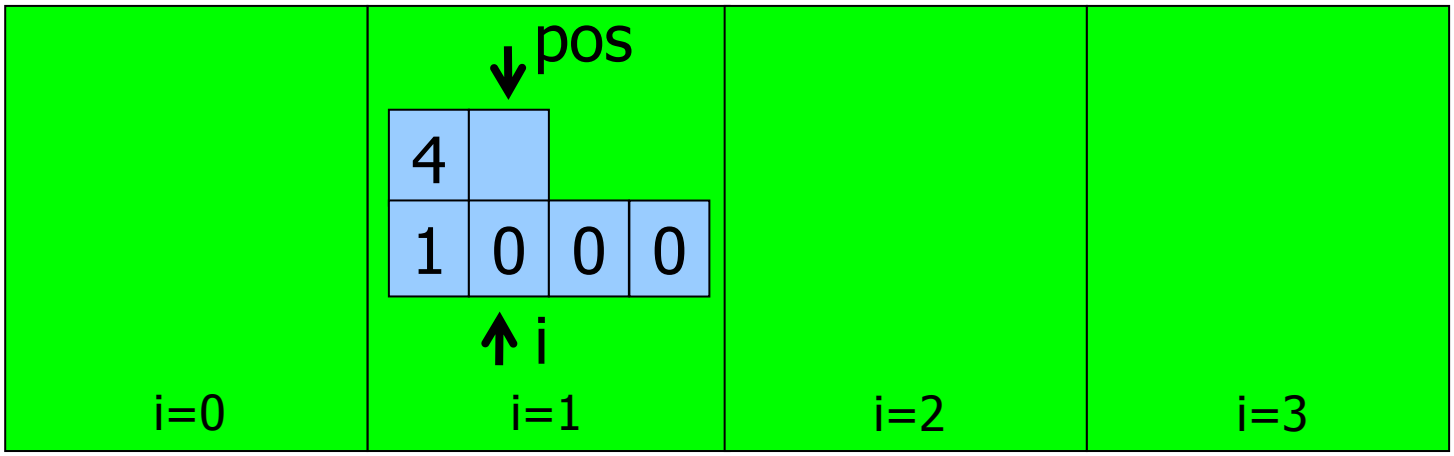


recursion
with pos=1



mark[i] = 1

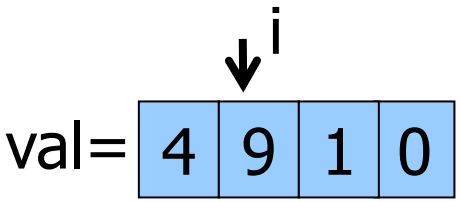


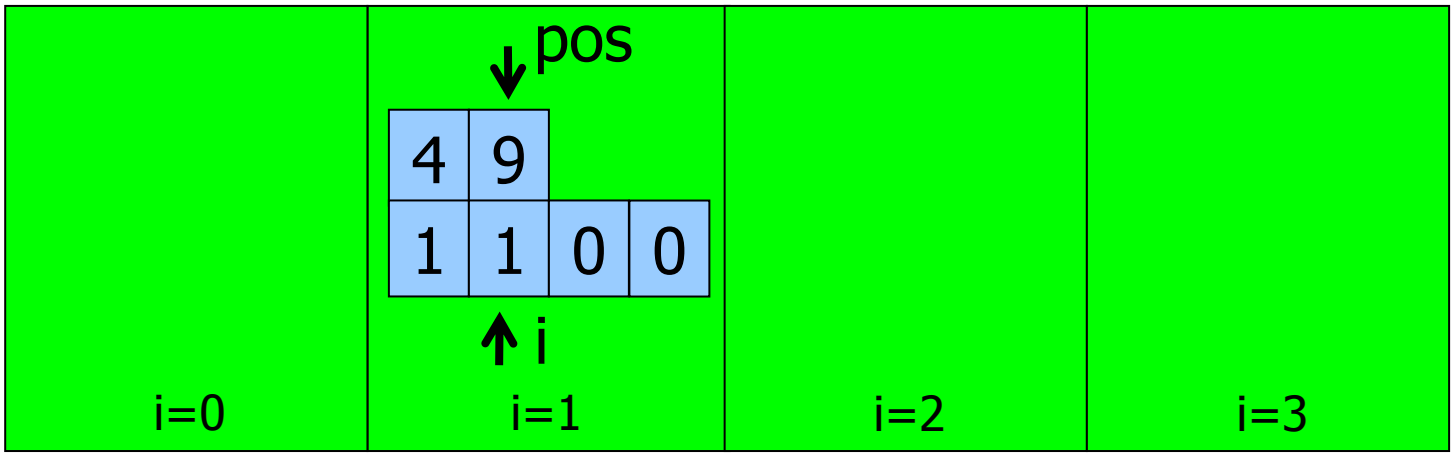


mark[i] = 0

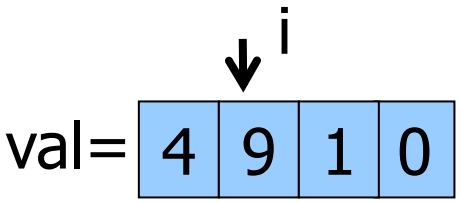
sol[pos] = val[i]

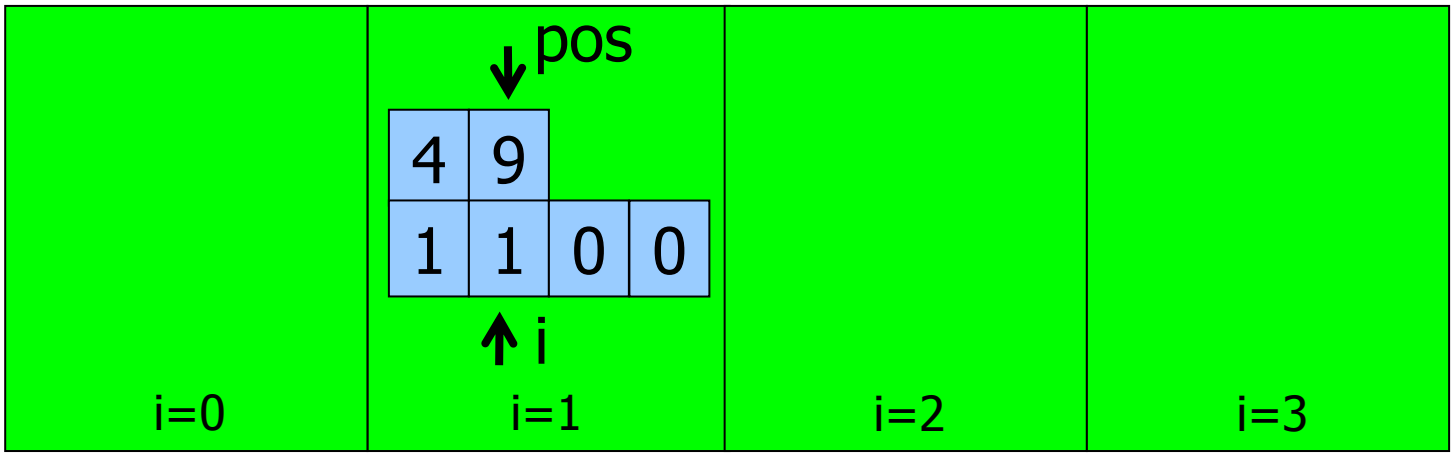
mark[i] = 1





recursion
with pos=2

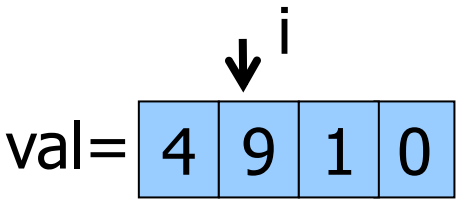


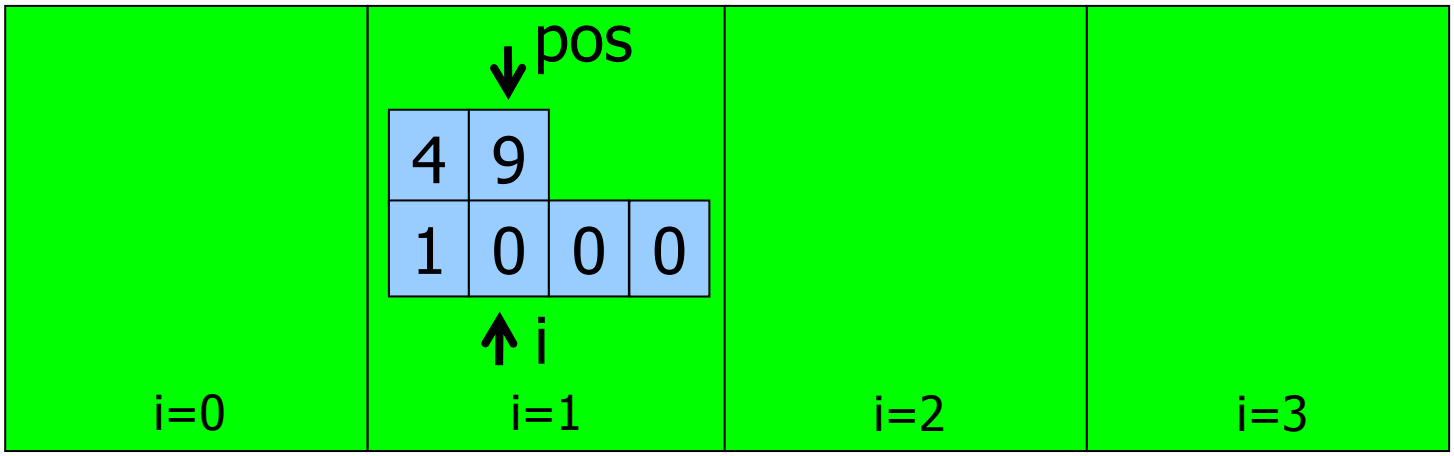


termination: display, update count

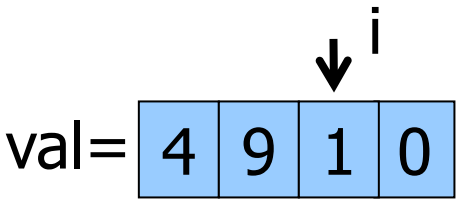
return

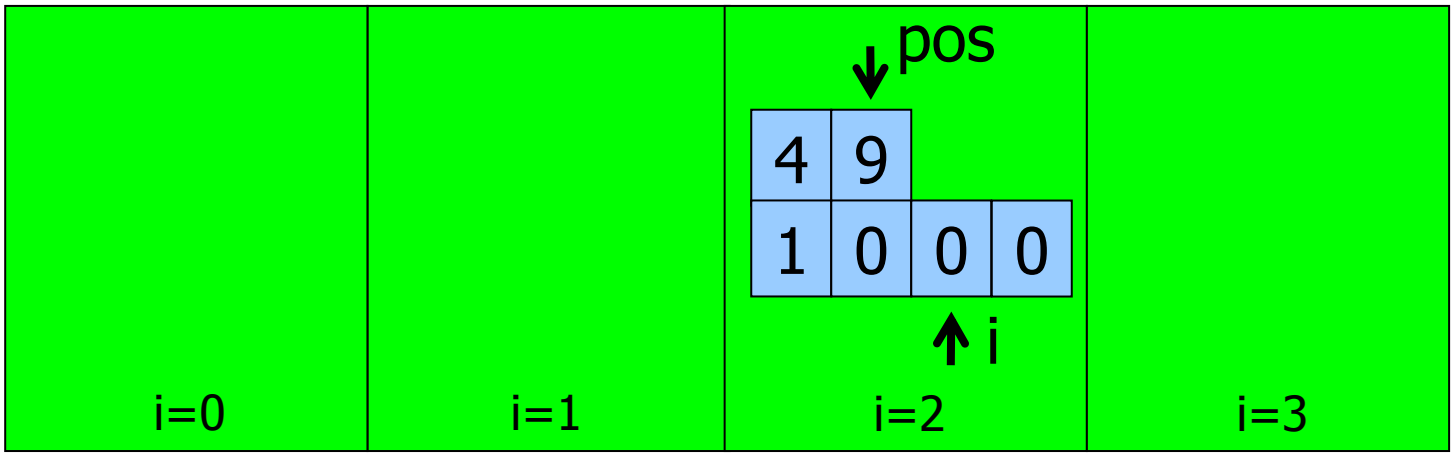
sol= 4 9





unmark mark[i]

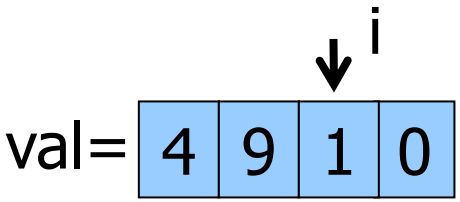


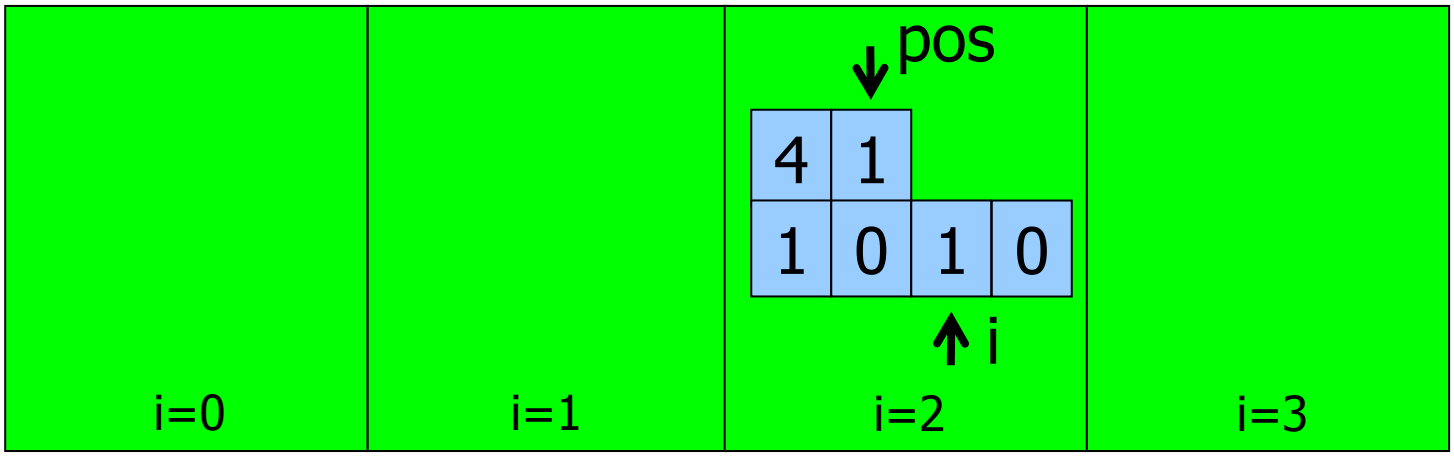


mark[i] = 0

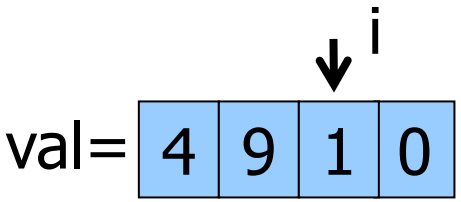
sol[pos] = val[i]

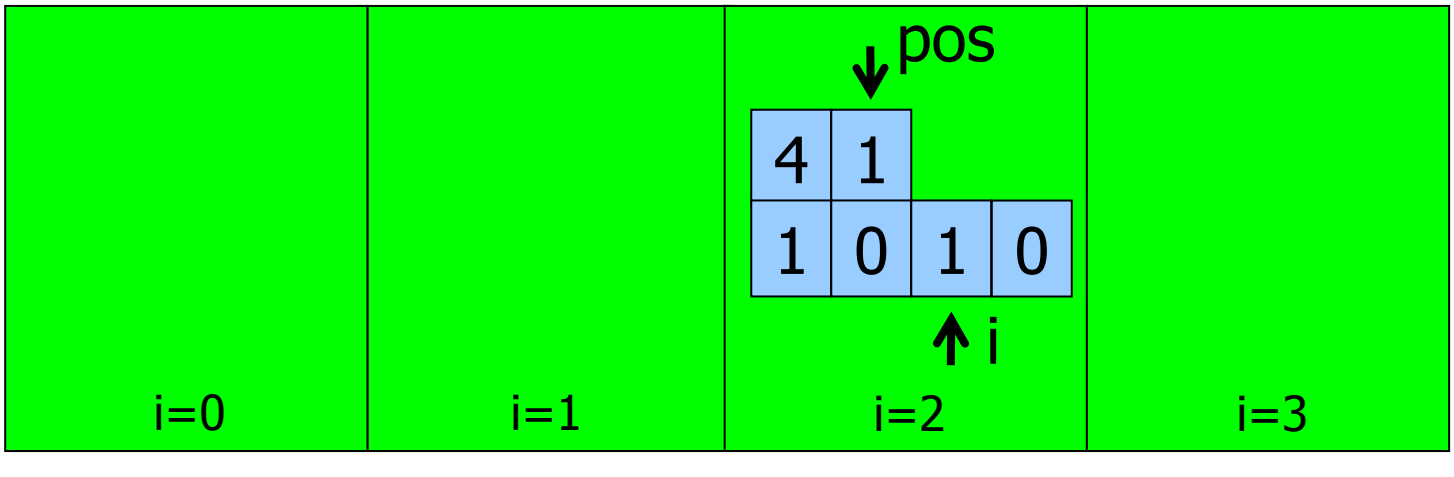
mark[i] = 1





recursion
with pos=2

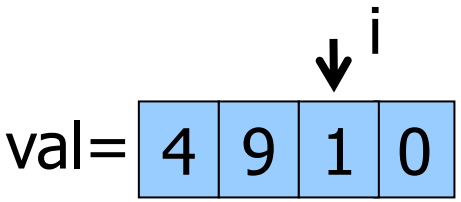


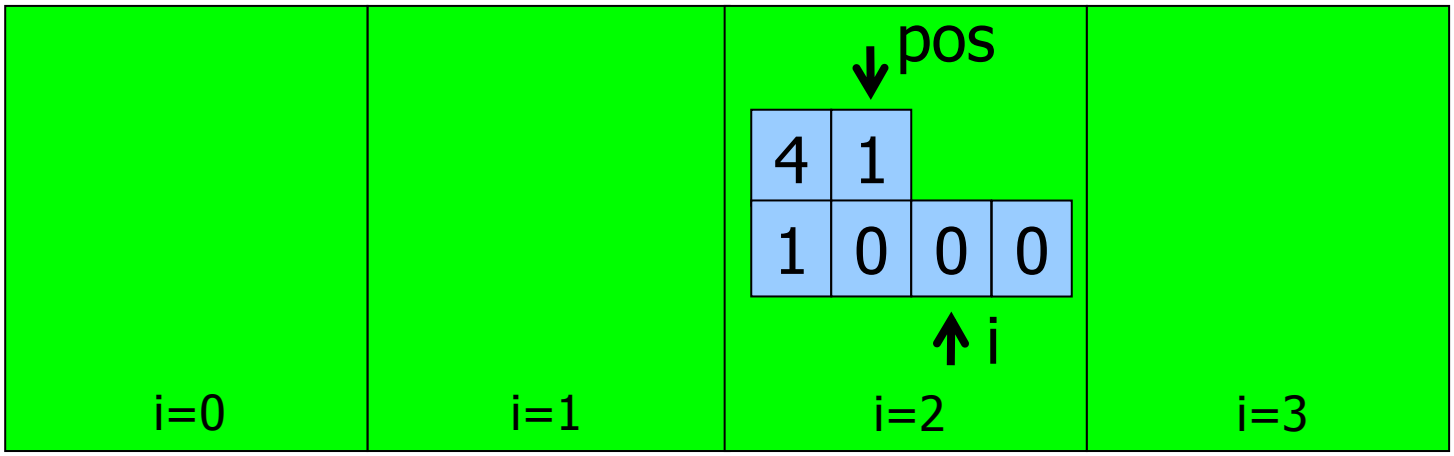


terminatione: display, update count

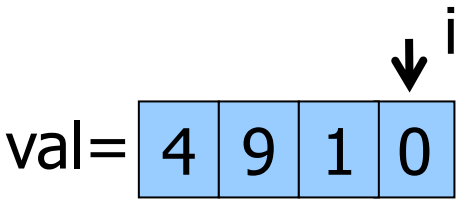
return

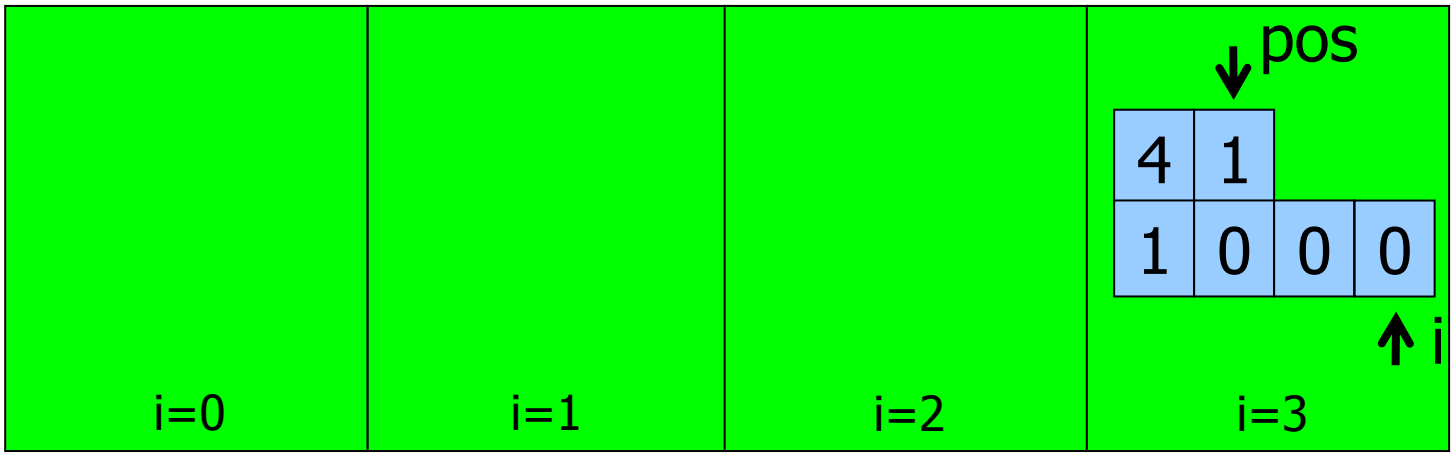
sol= 4 1





unmark mark[i]

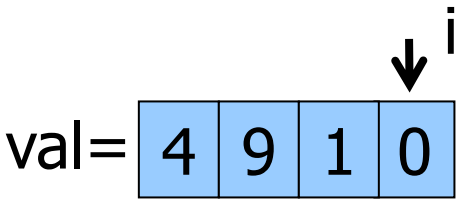


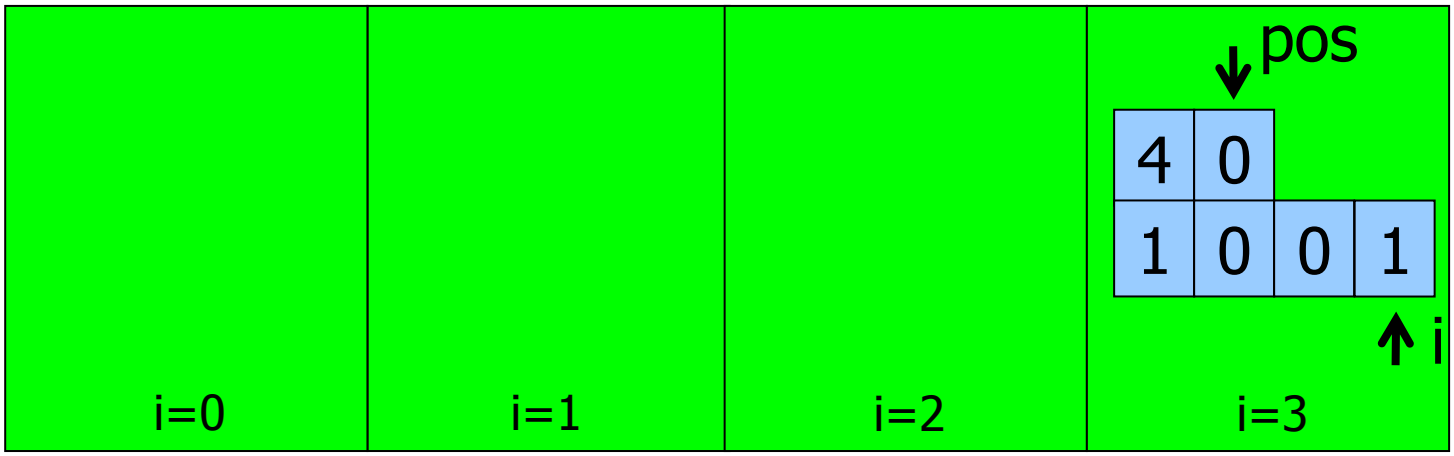


mark[i] = 0

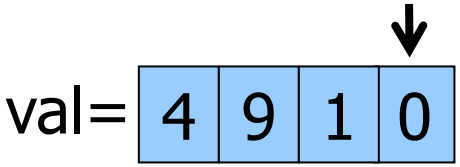
sol[pos] = val[i]

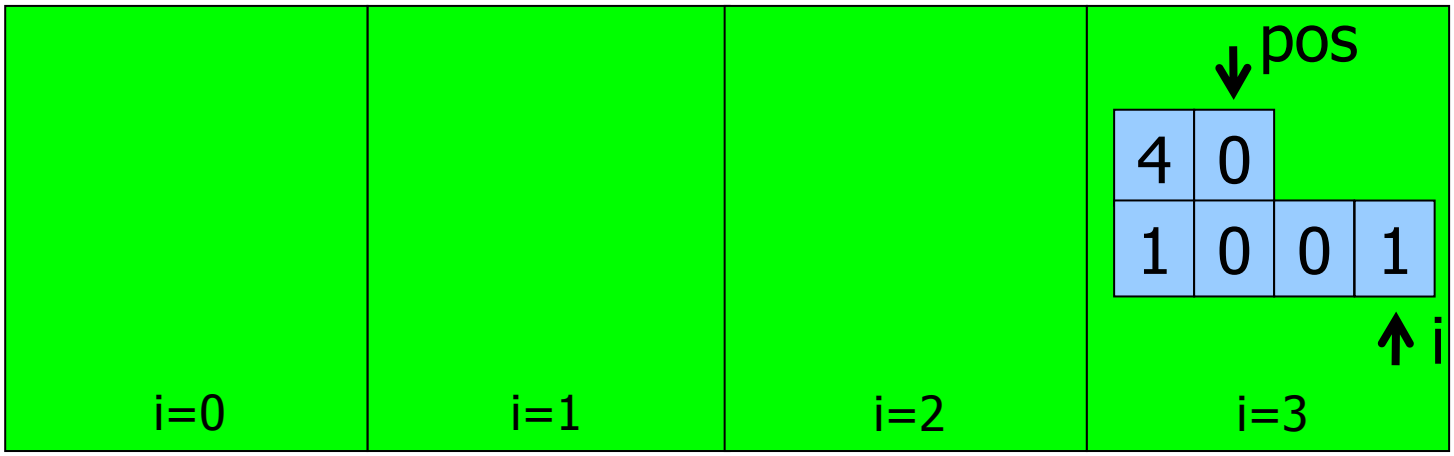
mark[i] = 1





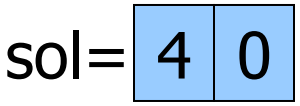
recursion
with pos=2

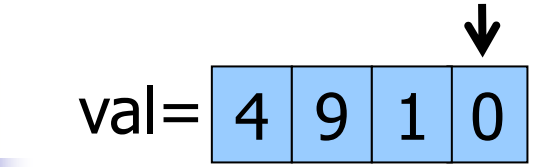


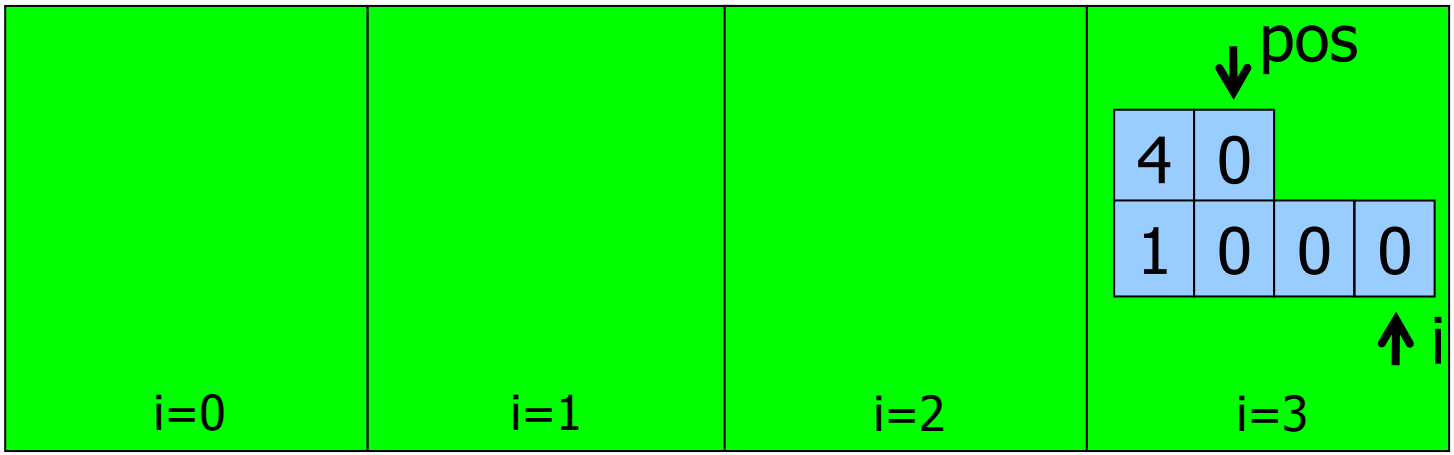


termination: display, update count

return

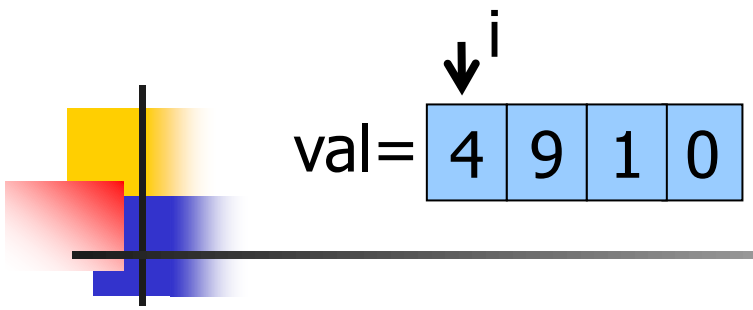


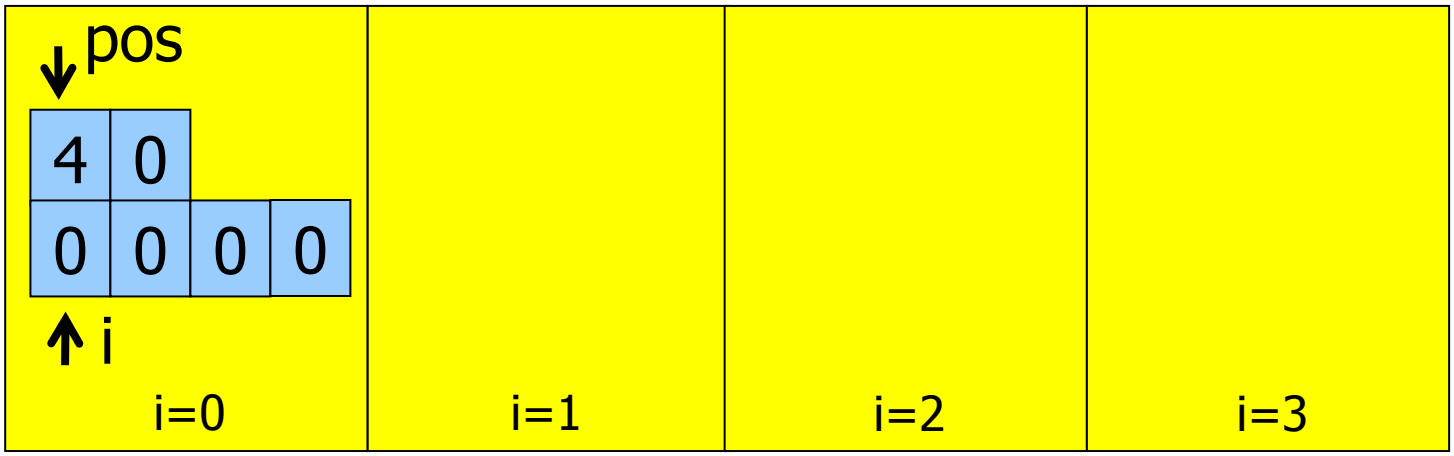




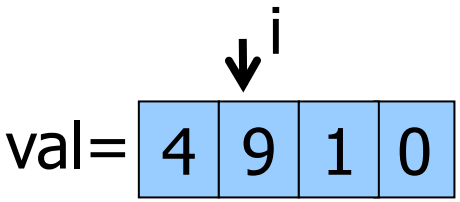
unmark mark[i]

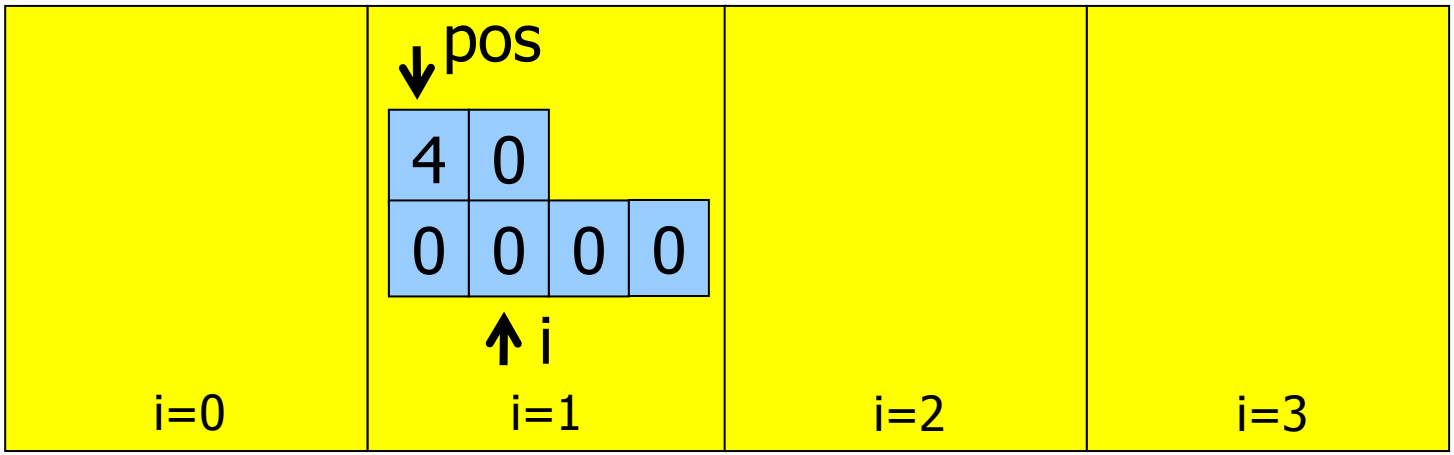
for loop finished, return





unmark mark[i]

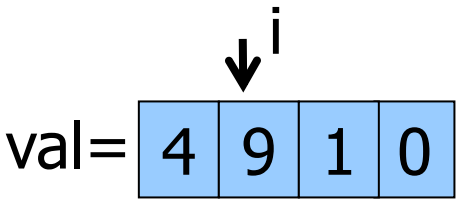


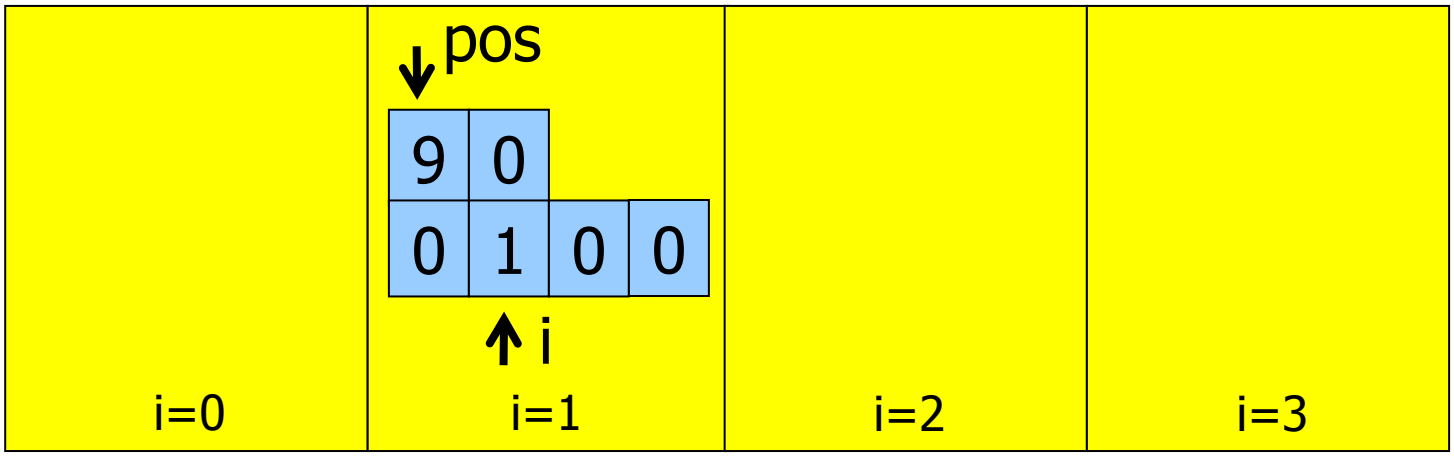


mark[i] = 0

sol[pos] = val[i]

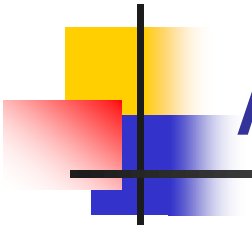
mark[i] = 1





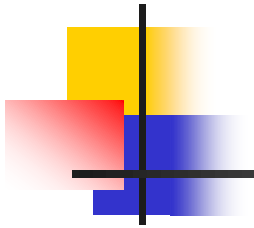
recursion
with pos=1

etc. etc.



Arrangements with repetitions

- 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.



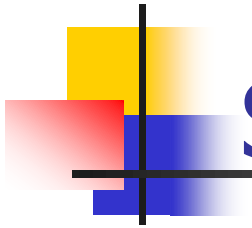
```
int rep_arr(int pos,int *val,in                n,int k,int count){
  int i;
  if (pos >= k) {
    for (i=0; i<k; i++)
      printf("%d ", sol[i]);
    printf("\n");
    return count+1;
  }
  for (i = 0; i < n; i++) {
    sol[pos] = val[i];
    count = rep_arr(pos+1, val, sol, n, k, count);
  }
 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

termination

iteration on n choices

choice

recursion



Simple permutations

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`
- during backtrack, `mark[i]` is assigned with 0
- `count` stores the number of solutions.

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

```
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

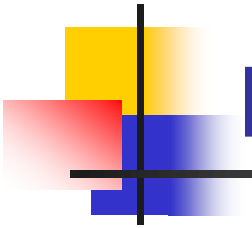
iteration on n choices

repetion control

mark and choose

recursion

unmark



Example

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

The number of permutations is n!.

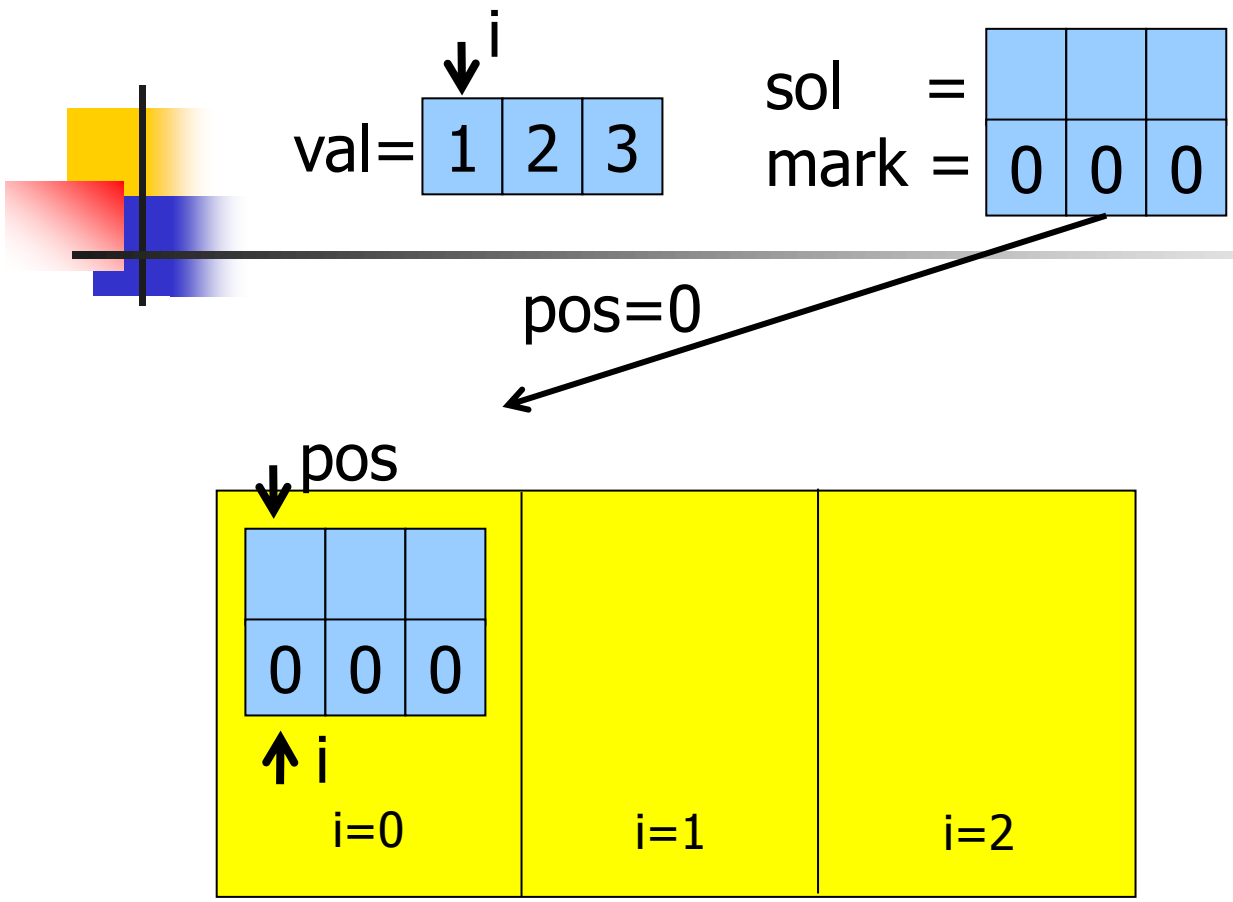
Example

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

n! = 6.

The 6 permutations are:

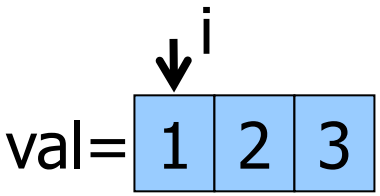
{1,2,3} {1,3,2} {2,1,3} {2,3,1} {3,1,2} {3,2,1}

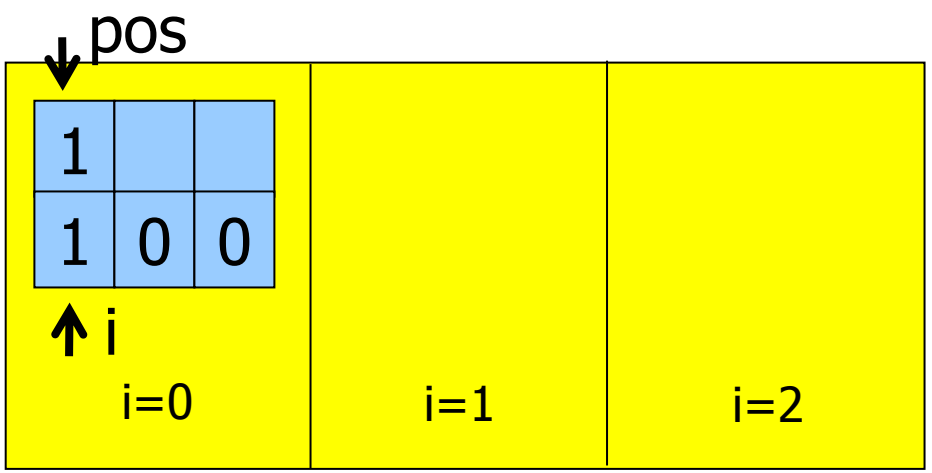


mark[i] = 0

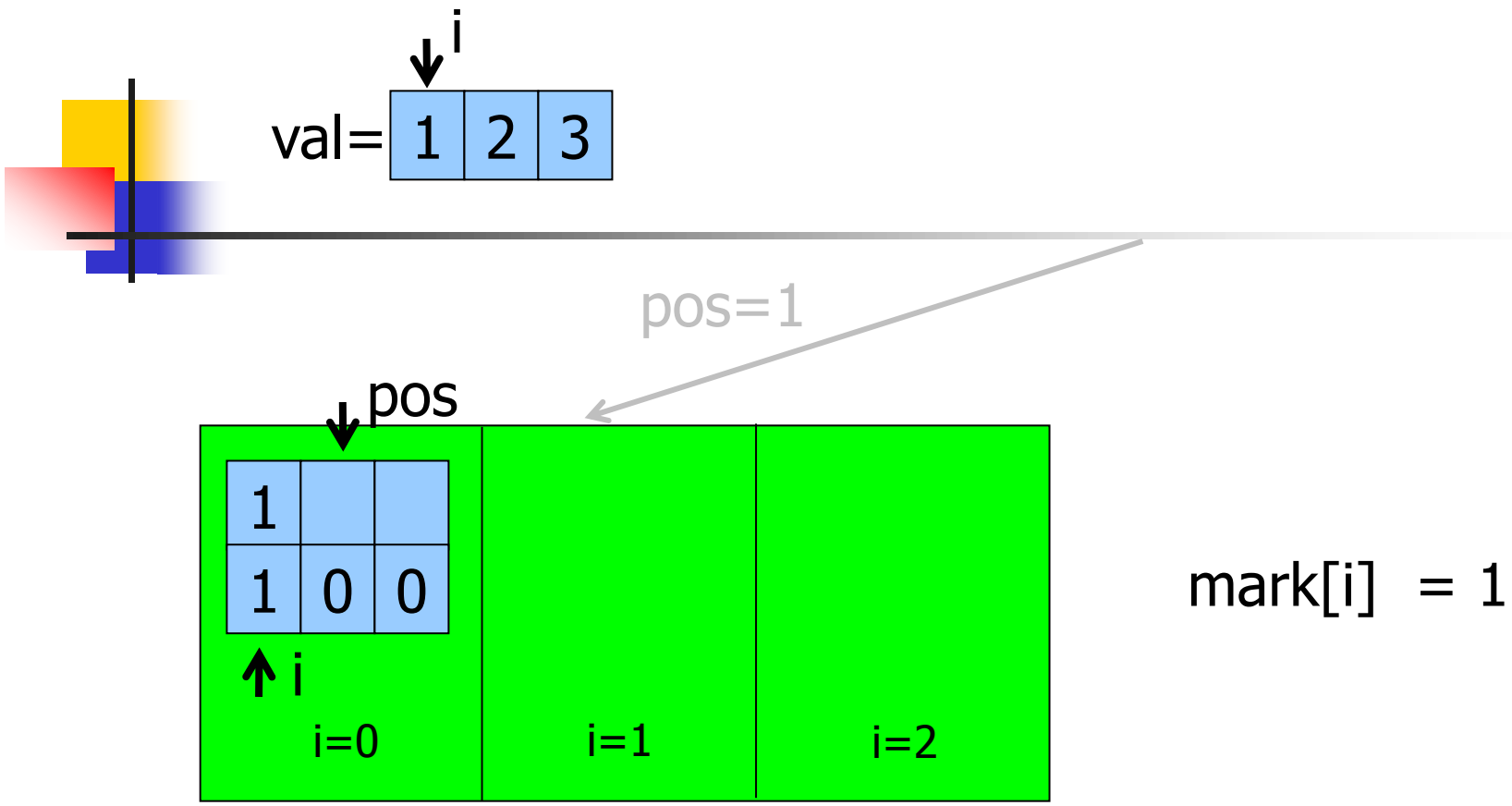
sol[pos] = val[i]

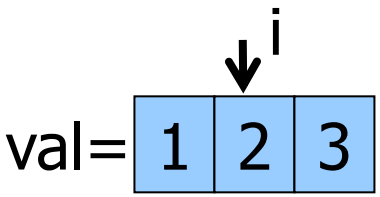
mark[i] = 1

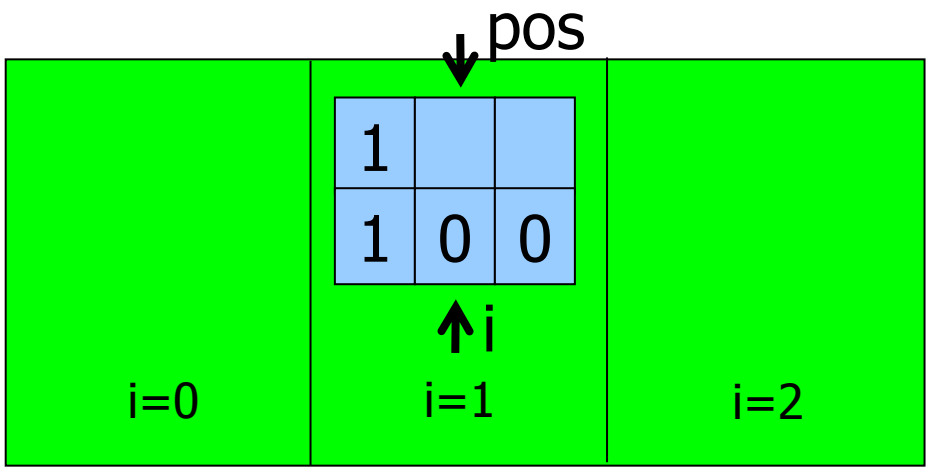




recursion
with pos=1


i
val=
1
2
3
pos=1
pos
1
1
0
0
i
i=0
i=1
i=2
mark[i] = 1

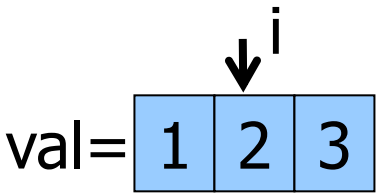


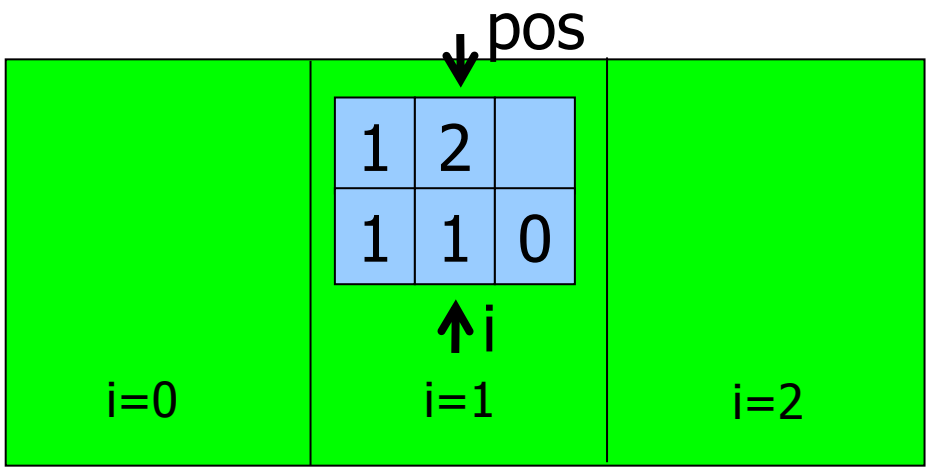


mark[i] = 0

sol[pos] = val[i]

mark[i] = 1

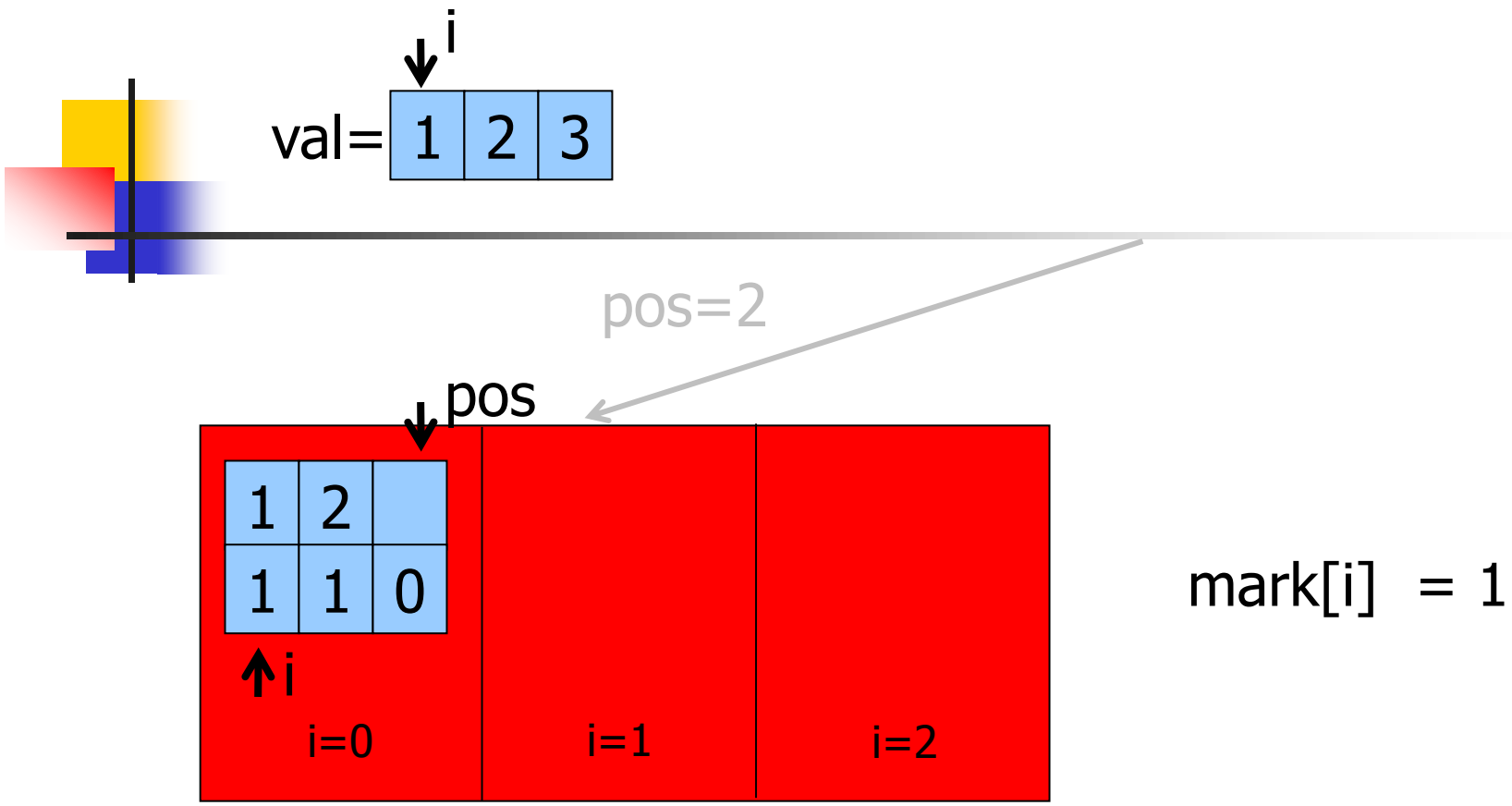


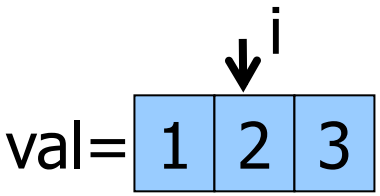


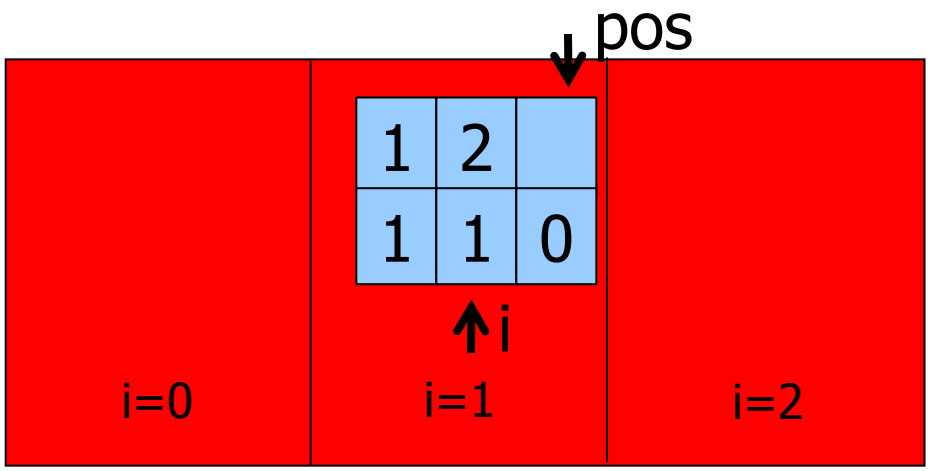
recursion
with pos=2

mark[i] = 0

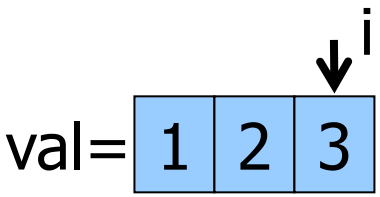
sol[pos] = val[i]
mark[i] = 1

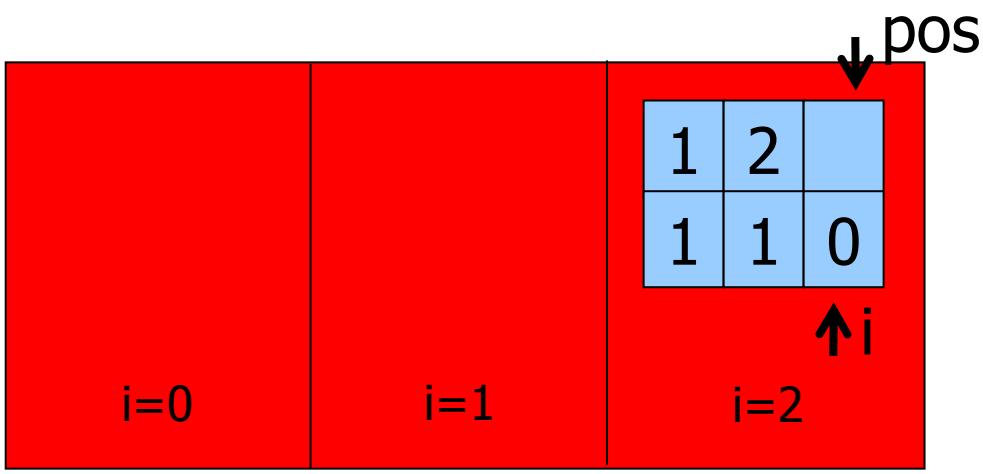

i
val=
1
2
3
pos=2
pos
1
2
1
1
0
i
i=0
i=1
i=2
mark[i] = 1





mark[i] = 1

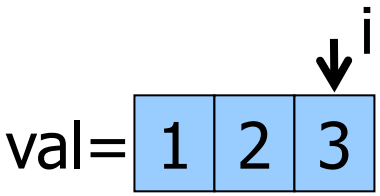


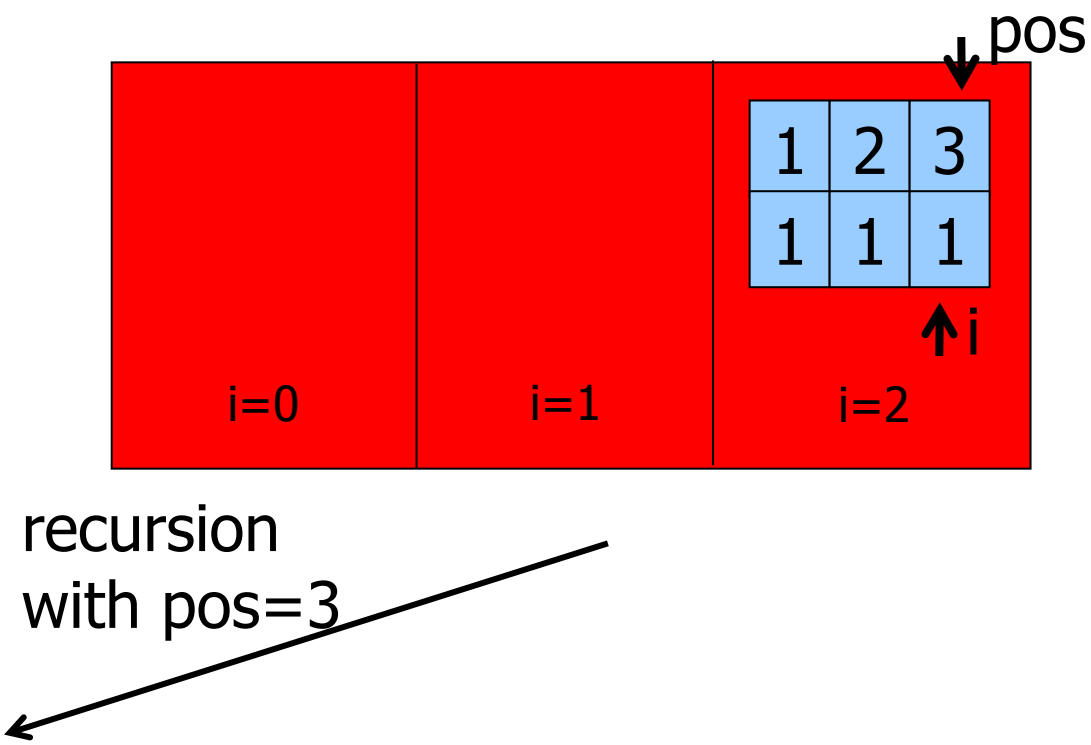


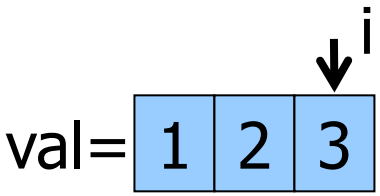
mark[i] = 0

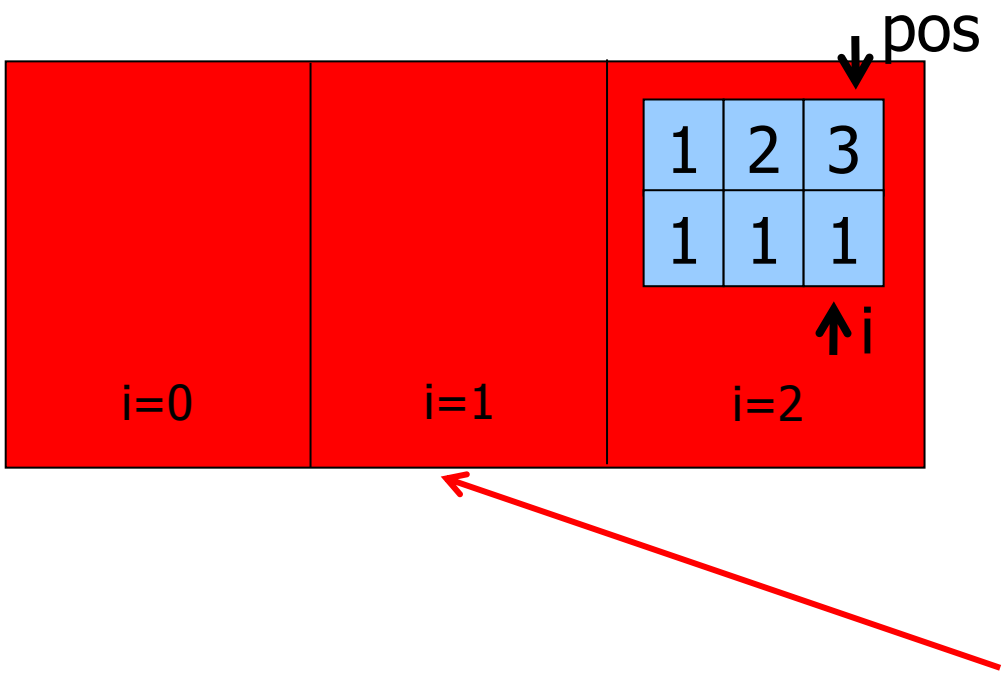
sol[pos] = val[i]

mark[i] = 1

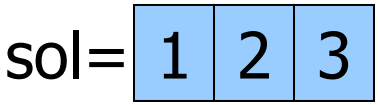

i
val=
1
2
3

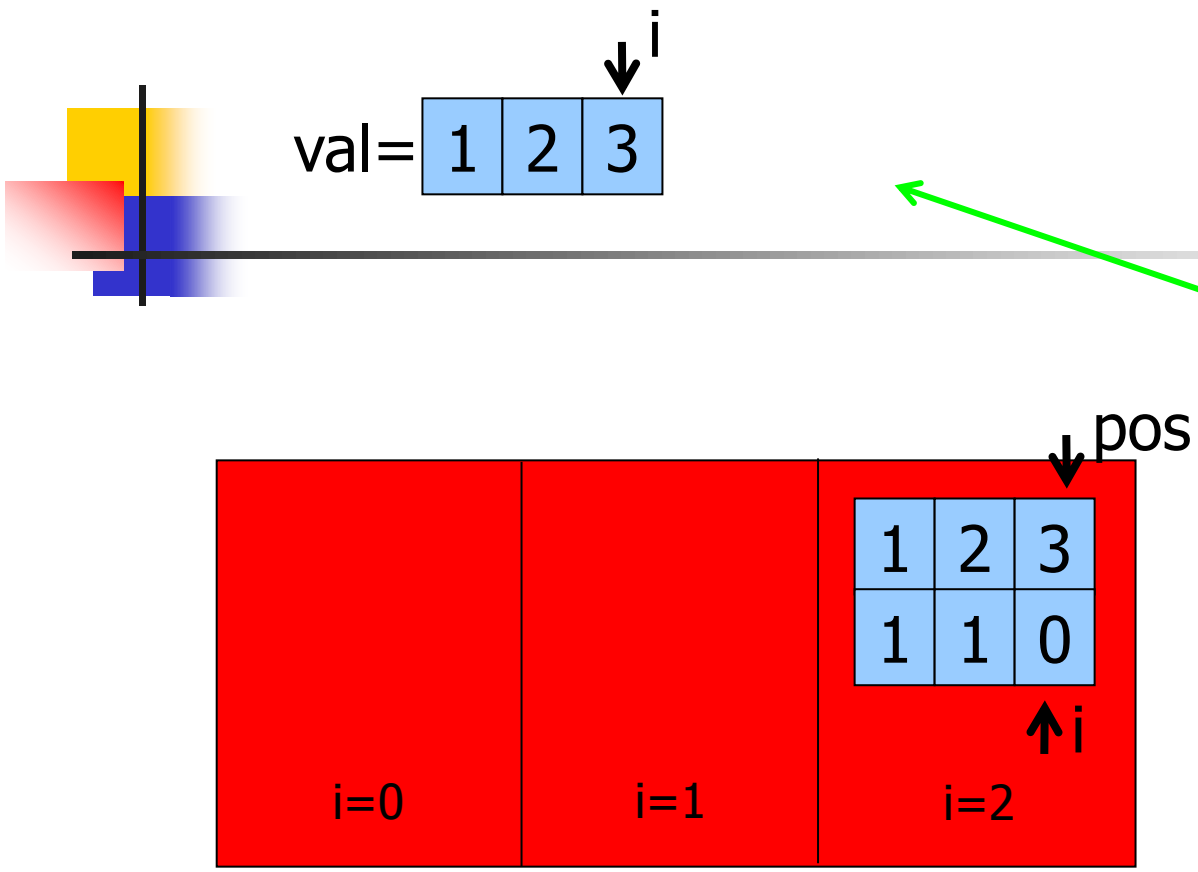

pos
1
2
3
1
1
1
i
i=0
i=1
i=2
recursion
with pos=3





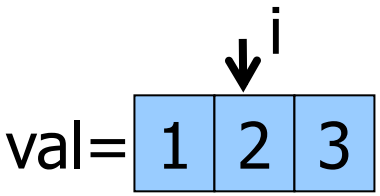
termination: display, update count return

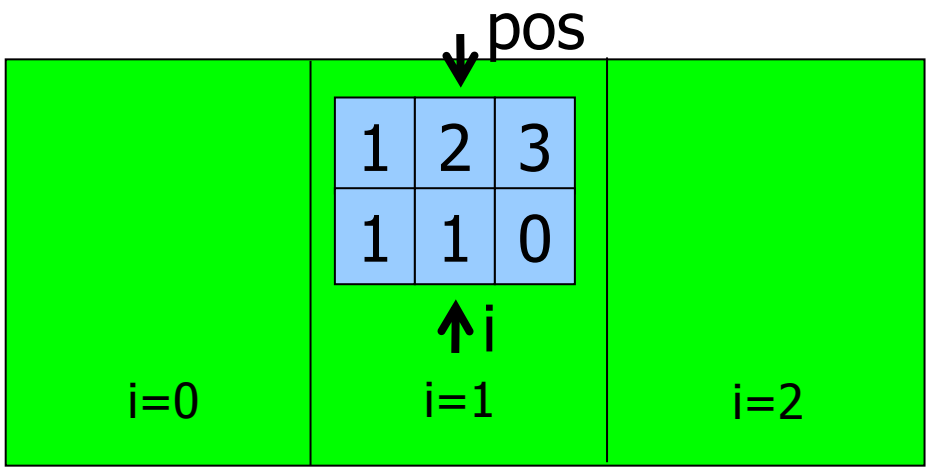


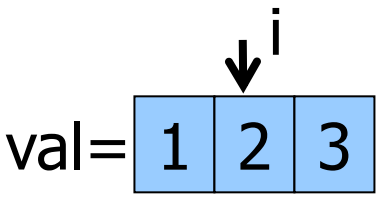


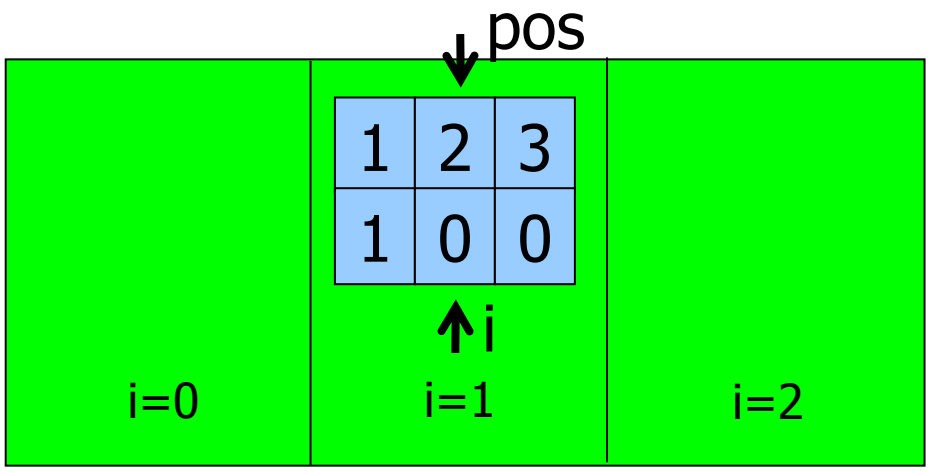
unmark mark[i]

for loop finished, return

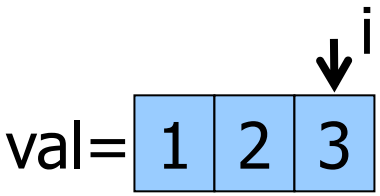

i
val=
1
2
3

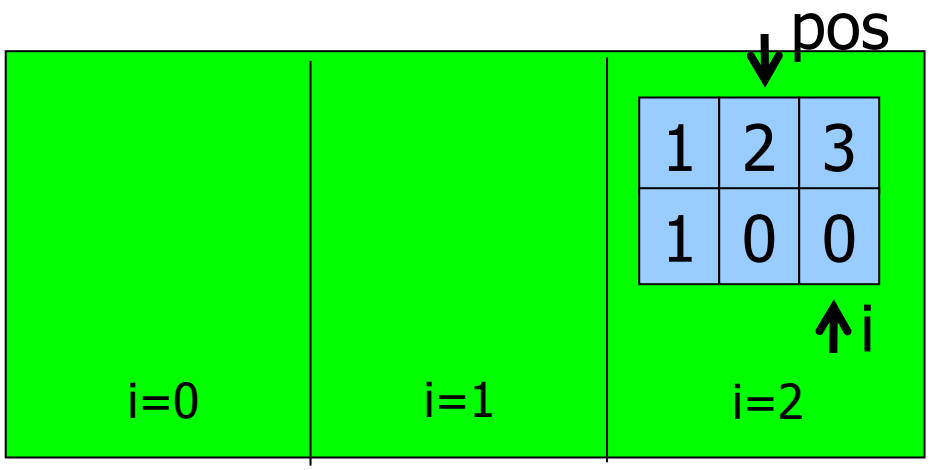

pos
1
2
3
1
1
0
i
i=0
i=1
i=2





unmark mark[i]

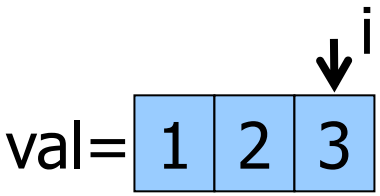


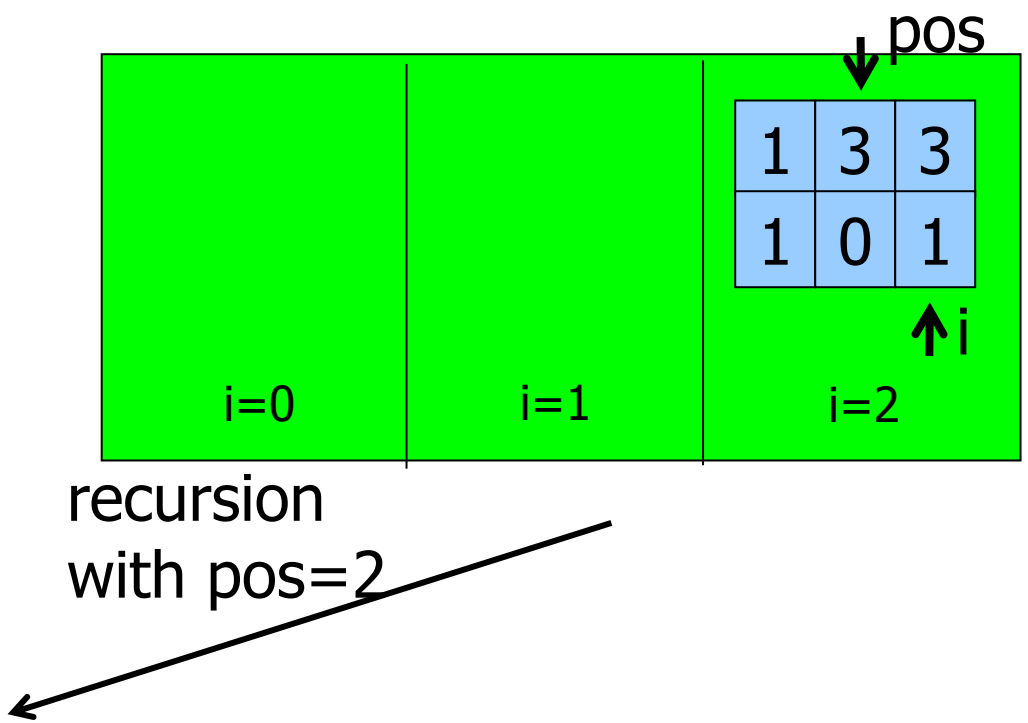


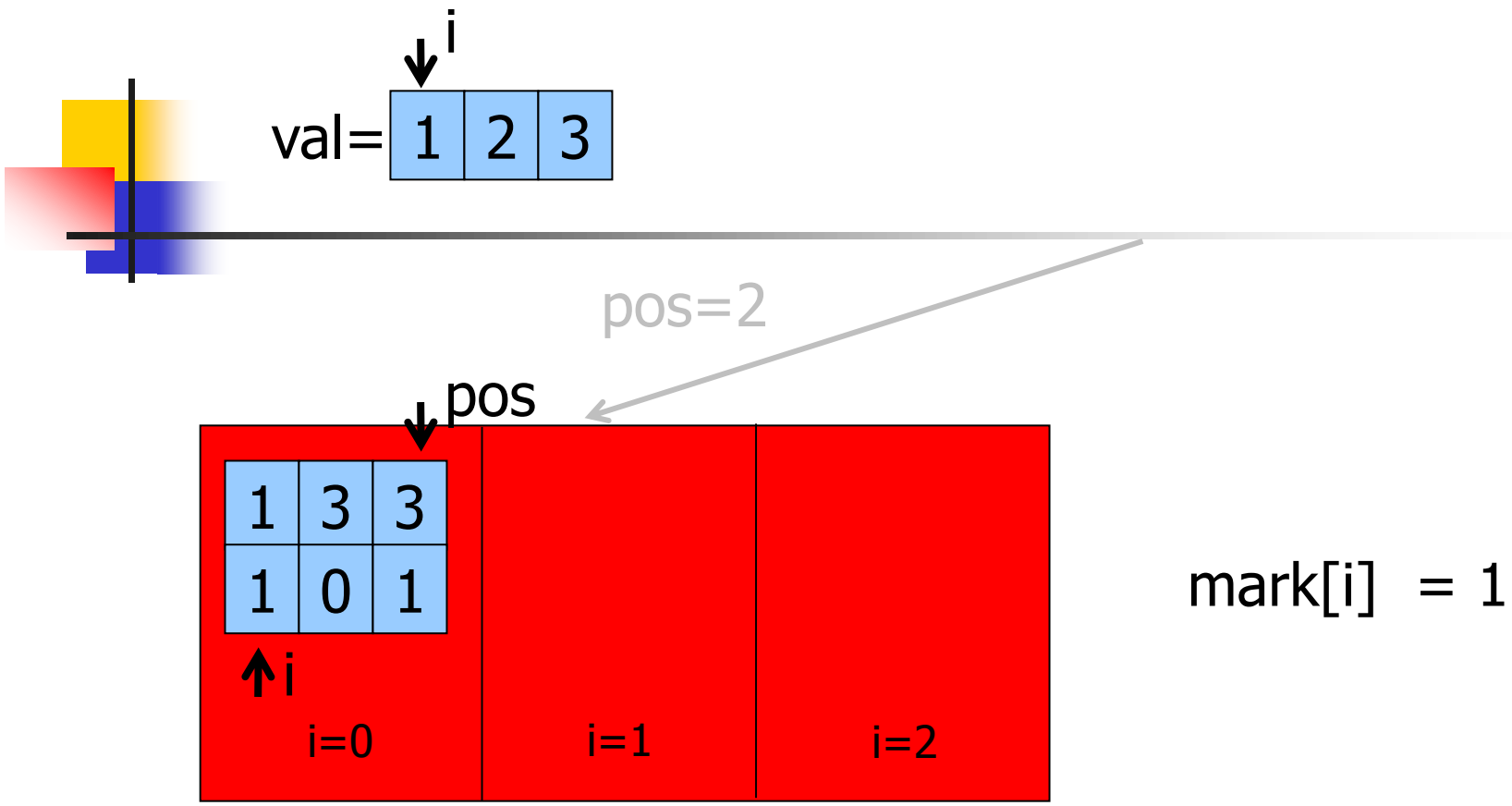
mark[i] = 0

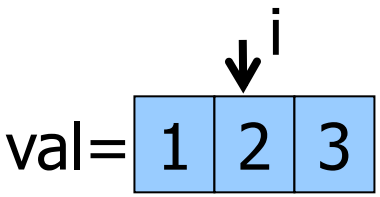
sol[pos] = val[i]

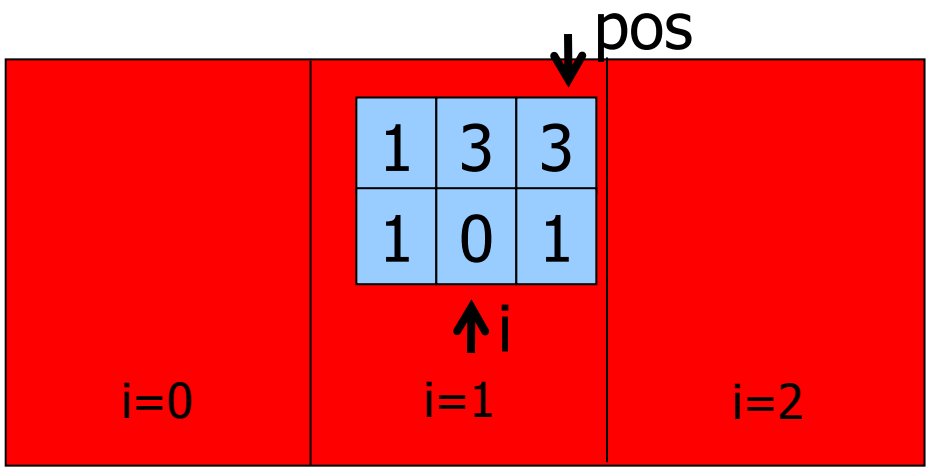
mark[i] = 1


i
val=
1
2
3


pos
1
3
3
1
0
1
i
i=0
i=1
i=2
recursion
with pos=2


i
val=
1
2
3
pos=2
pos
1
3
3
1
0
1
i
i=0
i=1
i=2
mark[i] = 1

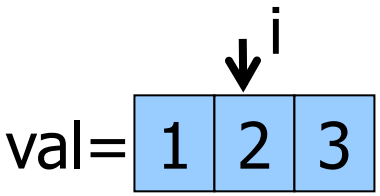


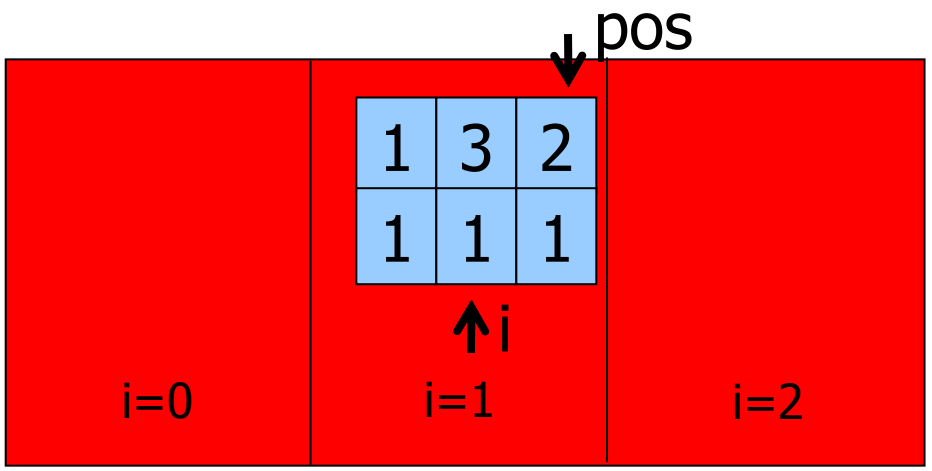


mark[i] = 0

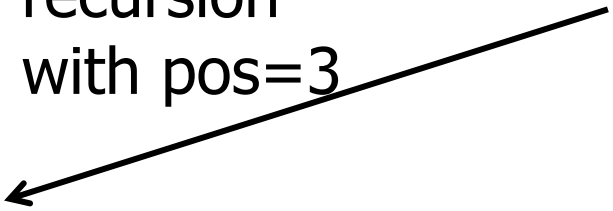
sol[pos] = val[i]

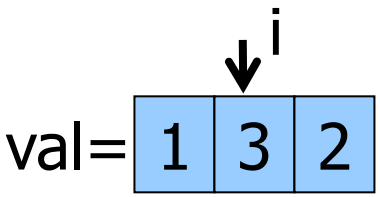
mark[i] = 1

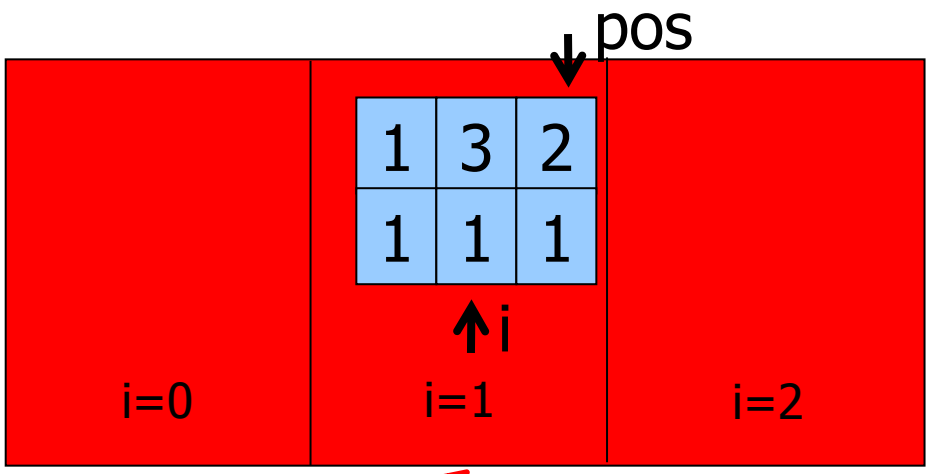




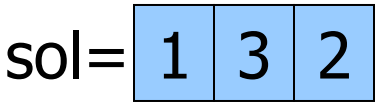
recursion
with pos=3

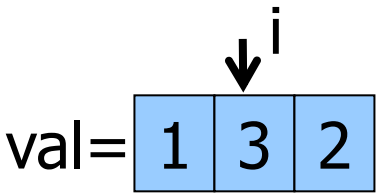


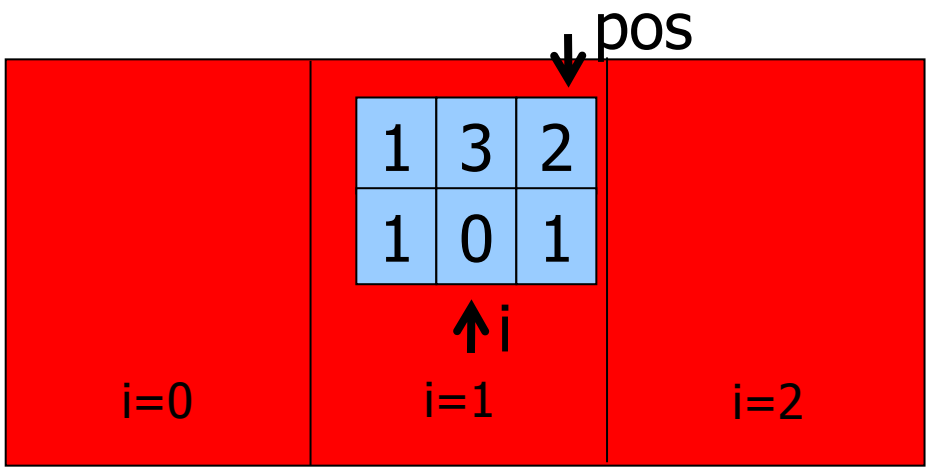




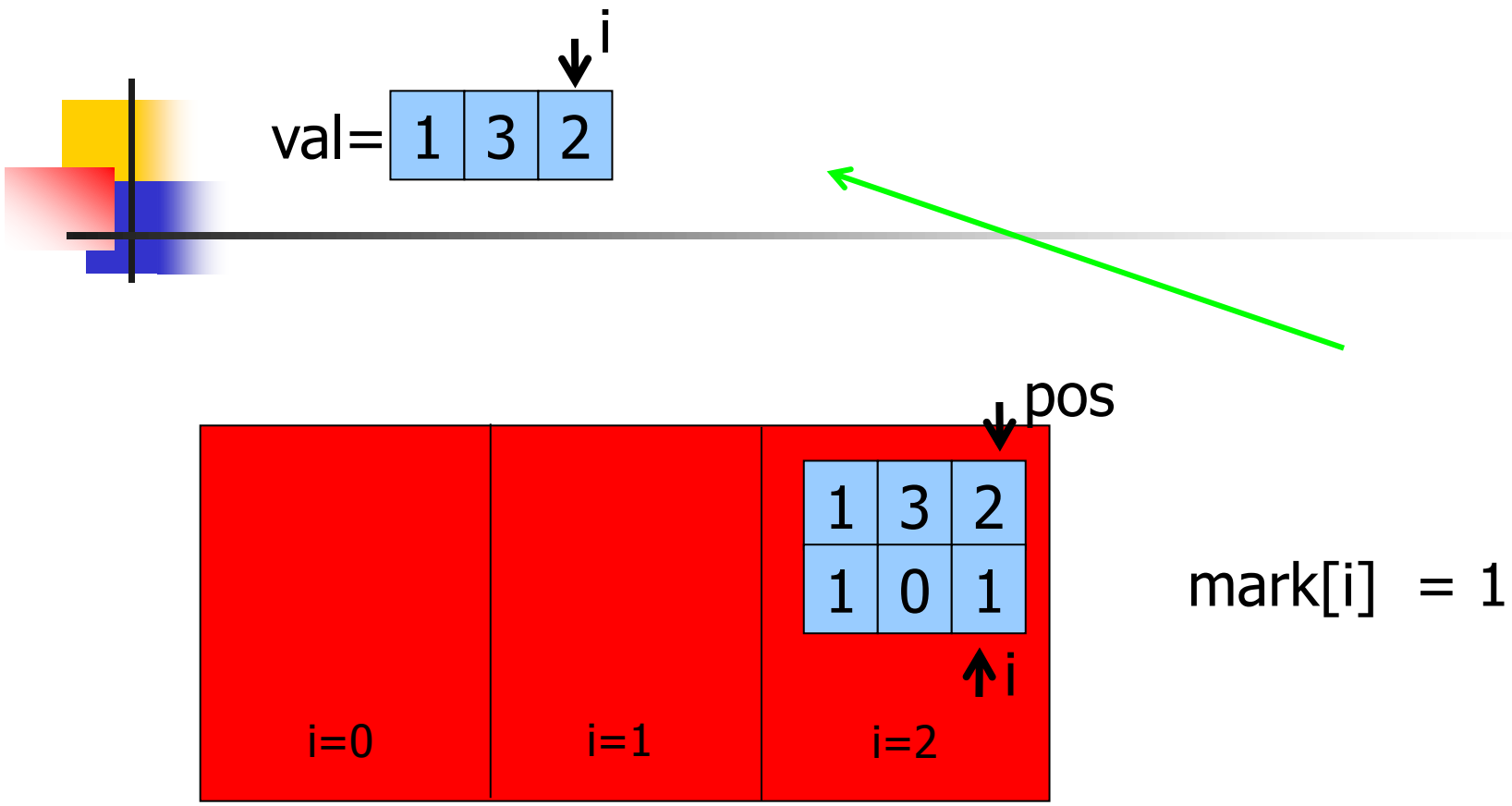
termination: display, update count return



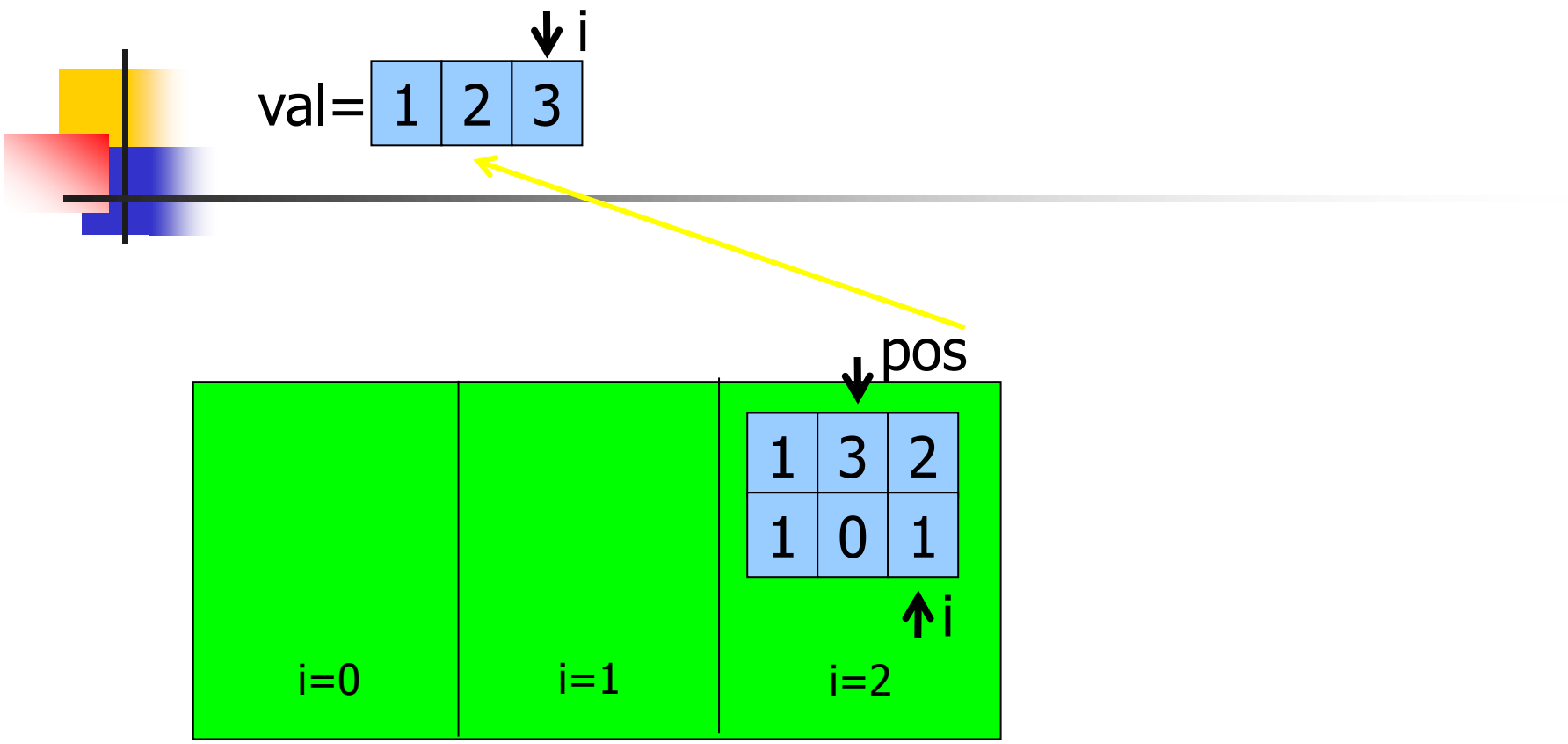




unmark mark[i]

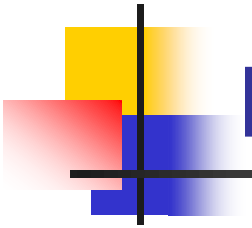


for loop finished, return



for loop finished, return

etc. etc.



Example: anagram with repetitions

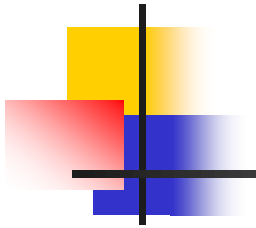
Read a string and generate all its anagrams.

If all the letters in the string are distinct, anagrams are distinct.

If there are repeated letters, anagrams are repeated.

Assuming that repeated letters are somehow distinguishable, the underlying model are simple permutations.

If the string has length n, the number of anagrams is n!



Example

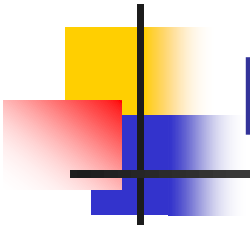
string = ORO, n = 3

n! = 6.

The 6 anagrams with repetitions are:

{ ORO, OOR, ROO, ROO, OOR, ORO }

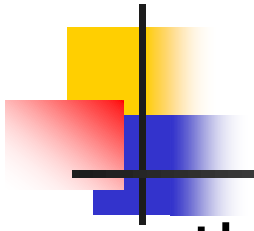
```
int anagrams(int pos, char *sol, char *val
             int *mark, int n,  int count) {
  int i;
  if (pos >= n) {
     sol[pos] = '\0';
     printf("%s\n", sol);
     return count+1;
  }
  for (i=0; i<n; i++)
    if (mark[i] == 0) {
      mark[i] = 1;
      sol[pos] = val[i];
      count = anagrams(pos+1,sol,val,mark,n,count);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



Permutations with repetition

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`



- 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.

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

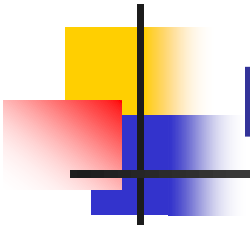
iteration on n_dist choices

occurrence control

mark and choose

recursion

unmark



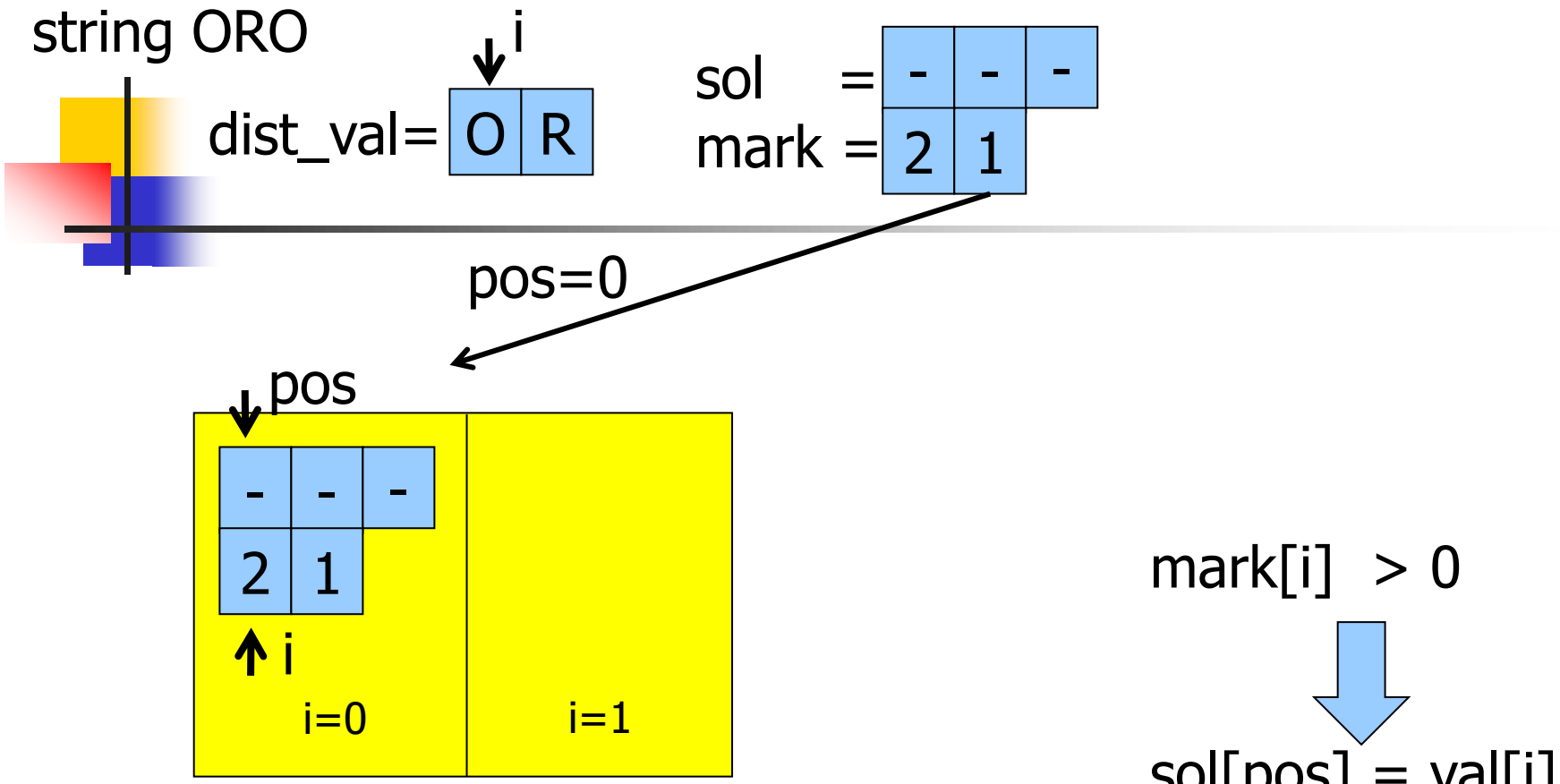
Example: distinct anagrams

Given a string with possibly repeated letters, generate all its distinct anagrams.

Example:
string ORO n = 3
dist_val = { O, R }, n_dist = 2

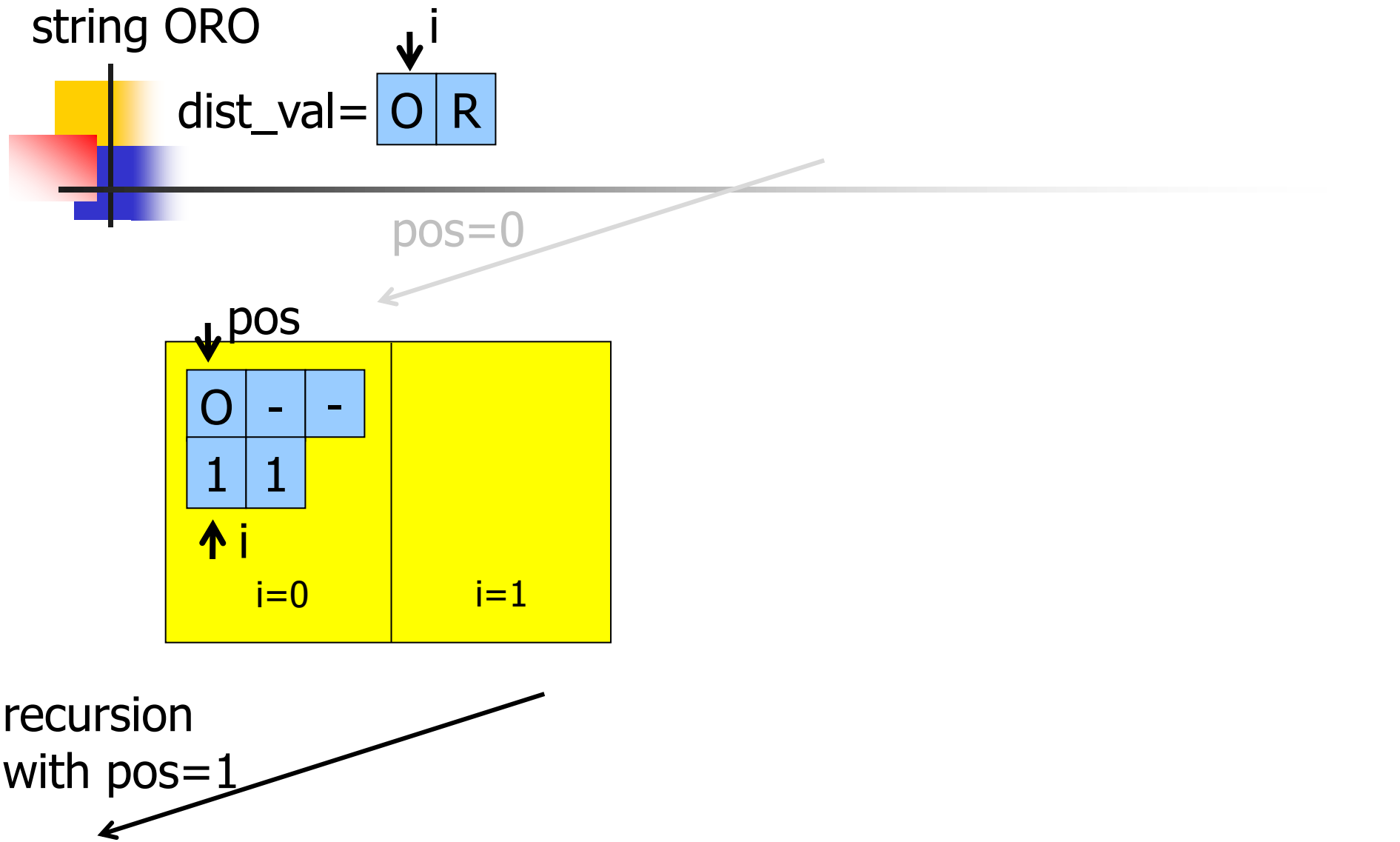
$P^{(2)}_3 = 3!/2! = 3$

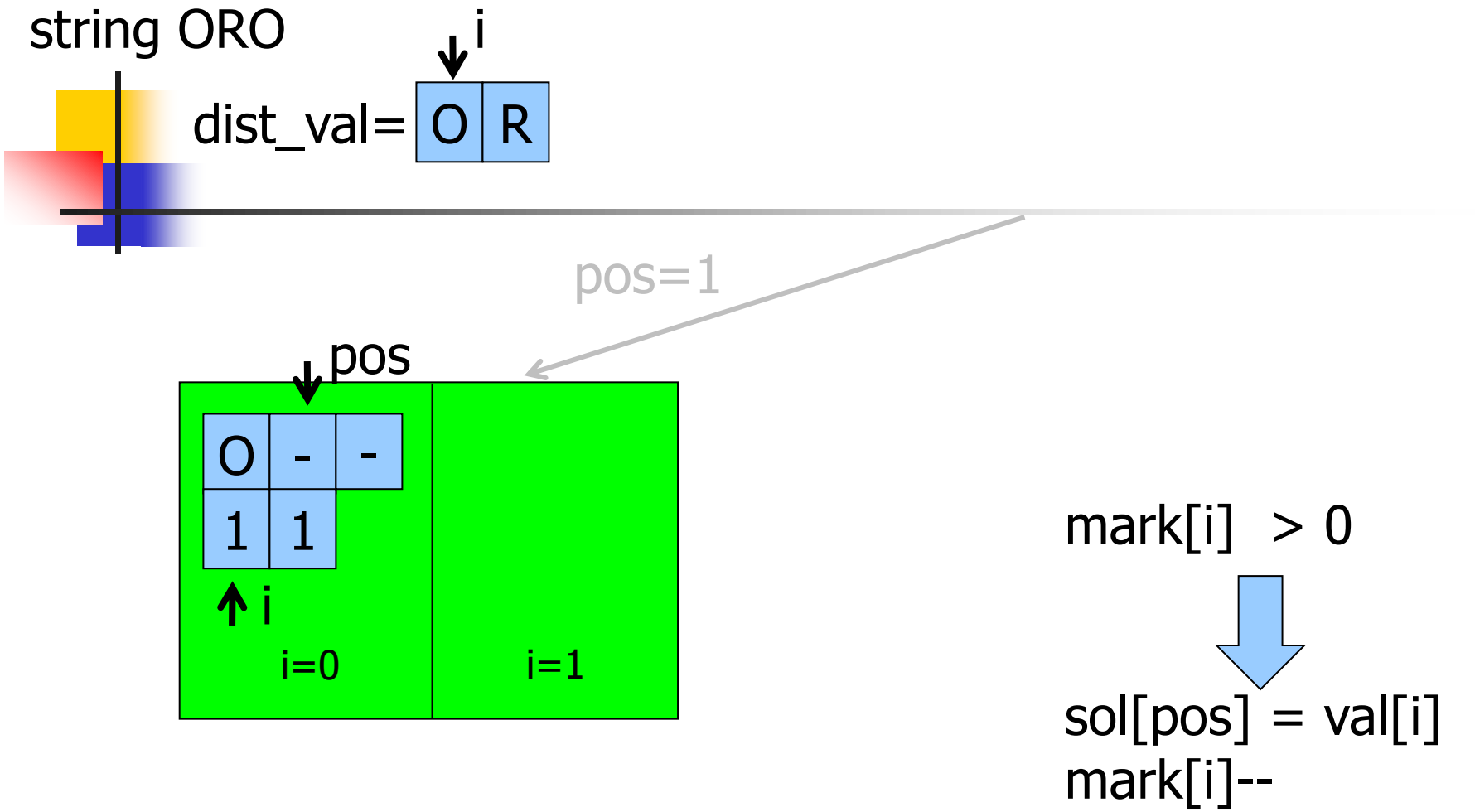
Solution
{ OOR, ORO, ROO }

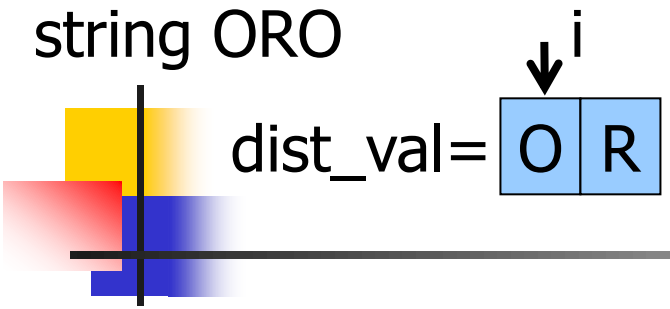


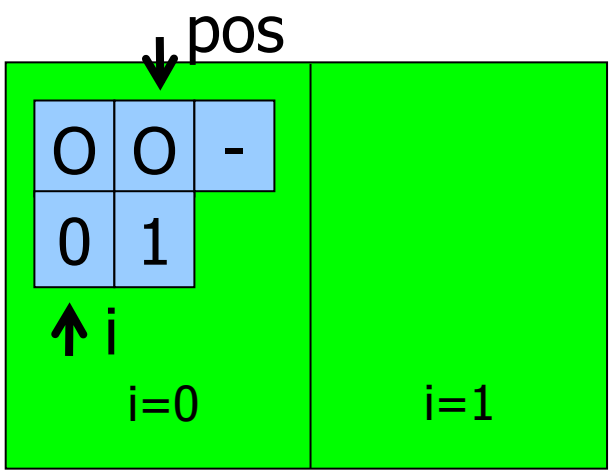
mark[i] > 0

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

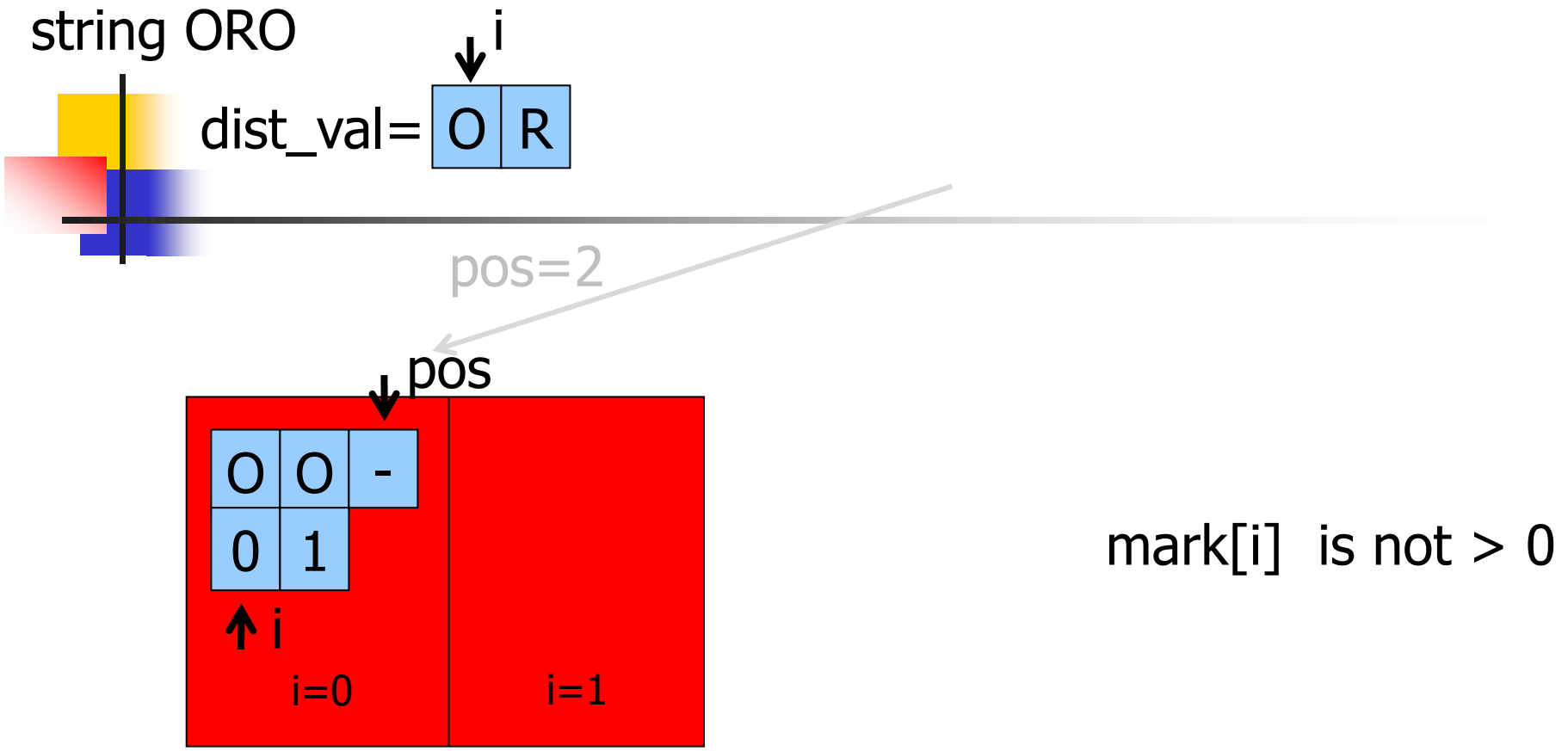

string ORO
i
dist_val=
O
R
pos=0
pos
O
-
-
1
1
i
i=0
i=1
recursion
with pos=1


string ORO
i
dist_val=
O
R
pos=1
pos
O
-
-
1
1
i
i=0
i=1
mark[i] > 0
sol[pos] = val[i]
mark[i]--

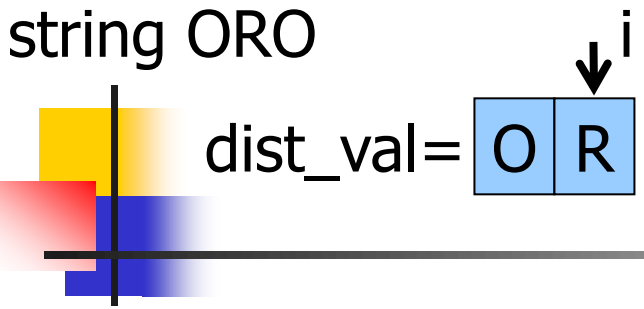


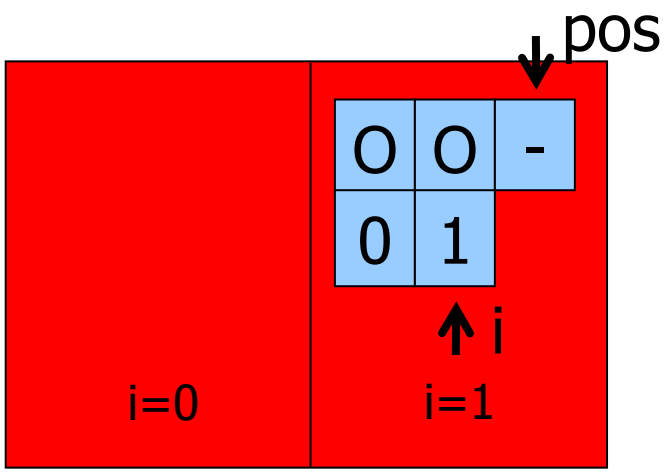


recursion
with pos=2



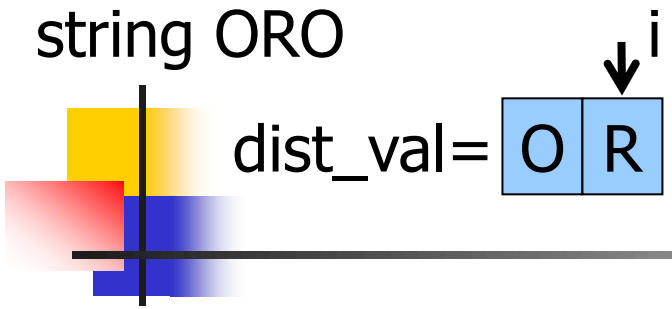
mark[i] is not > 0

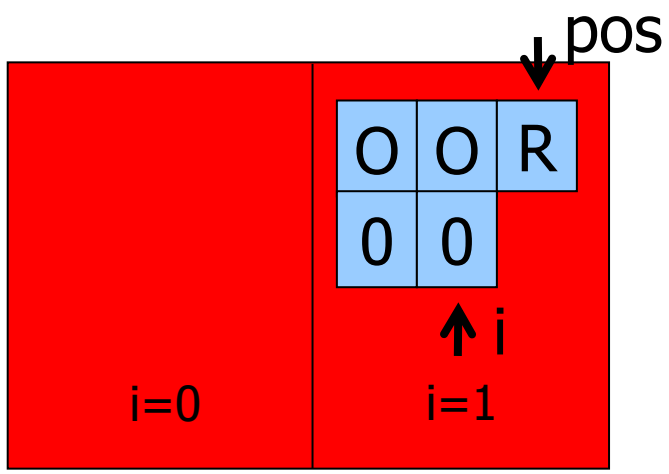




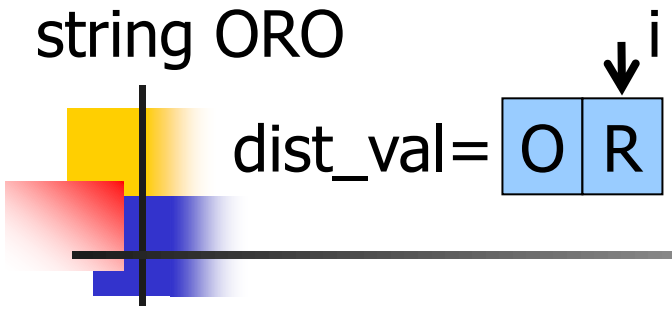
mark[i] > 0

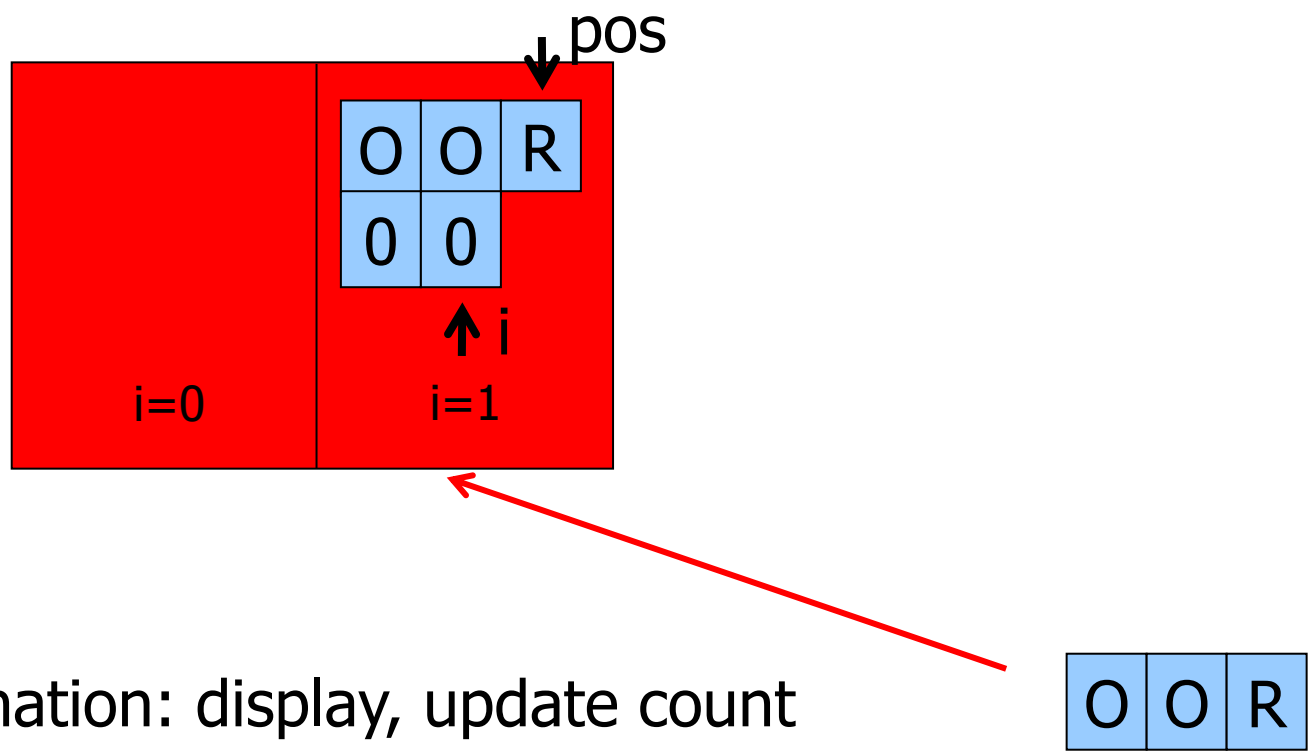
sol[pos] = val[i]
mark[i]--



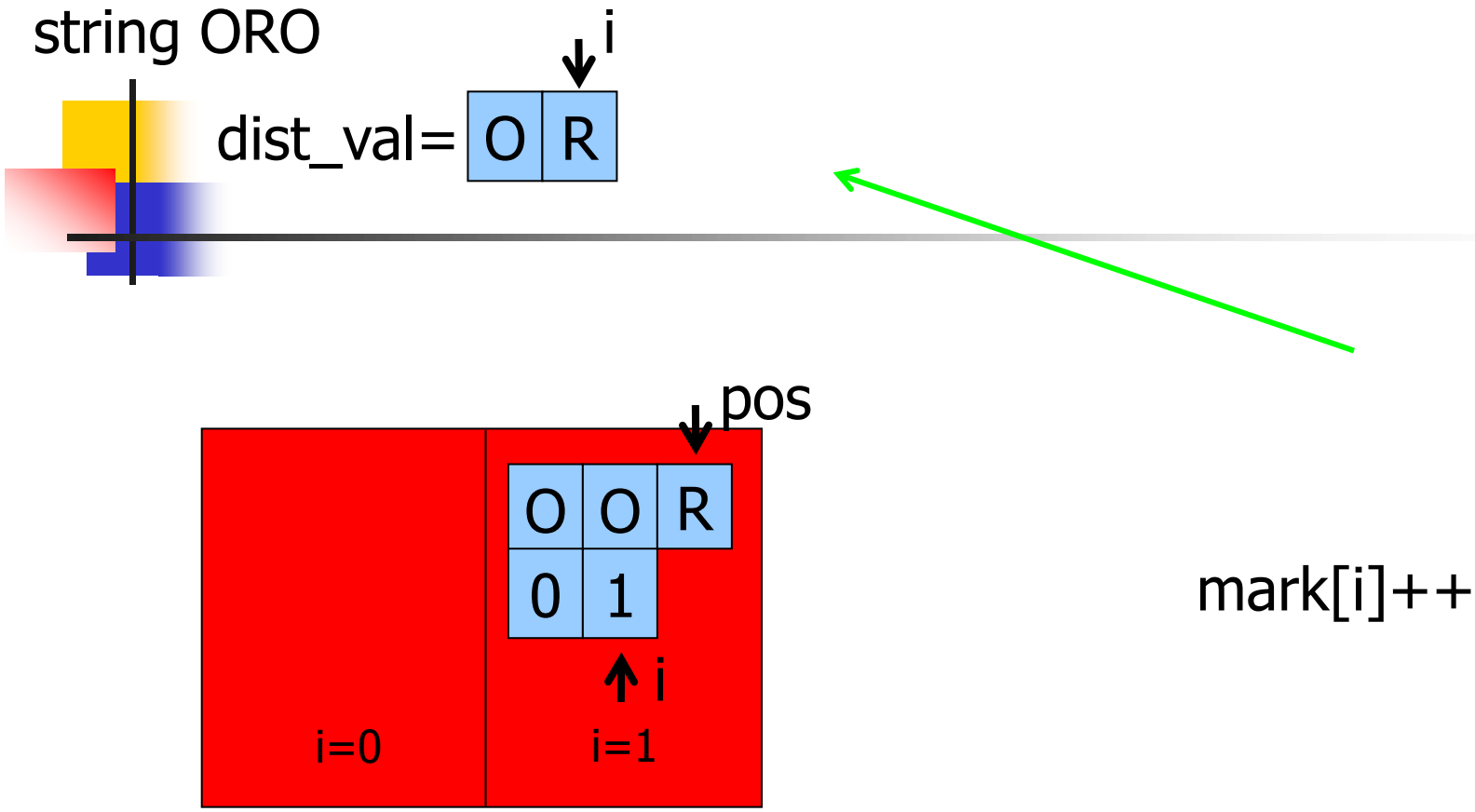


recursion
with pos=3



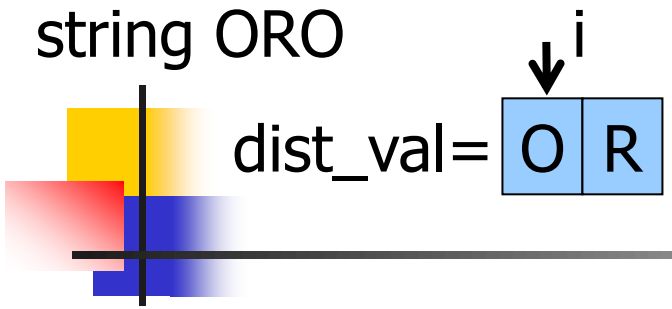


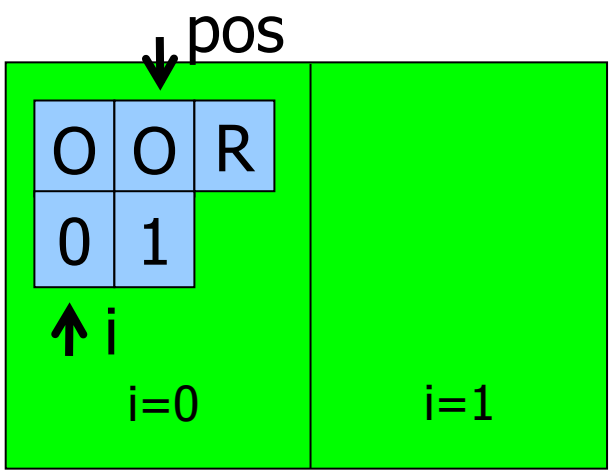
termination: display, update count
return



backtrack

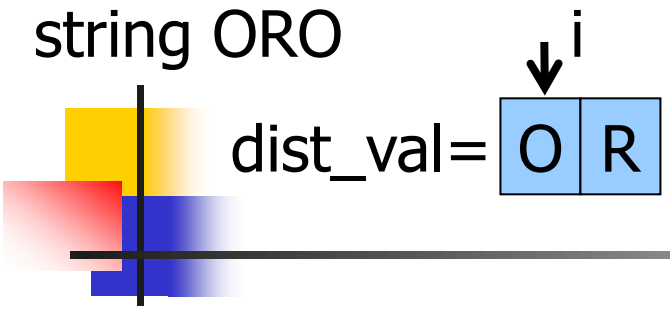
for loop finished, return

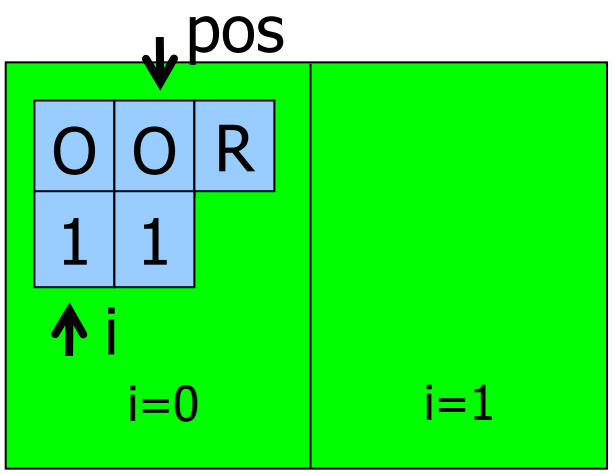


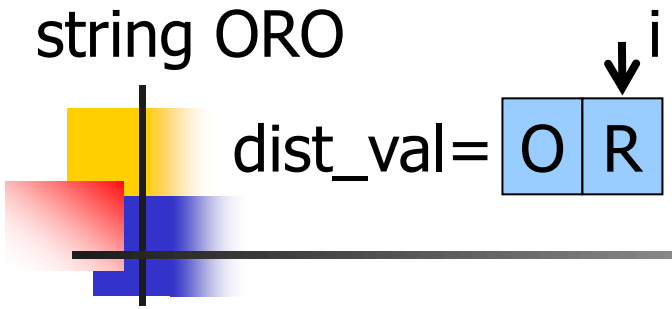


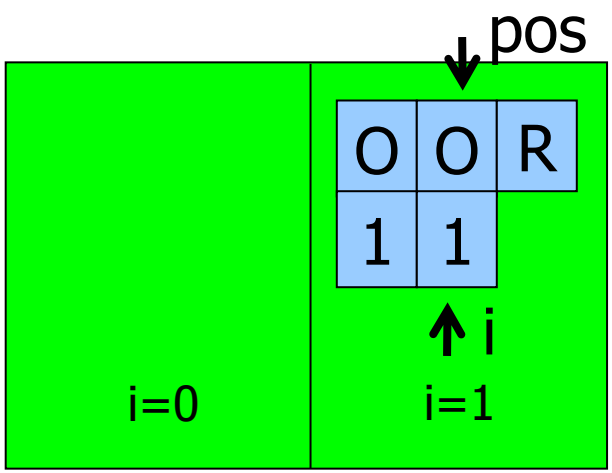
mark[i]++

backtrack

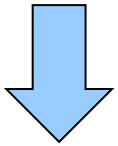

string ORO
i
dist_val=
O
R


pos
O
O
R
1
1
i
i=0
i=1

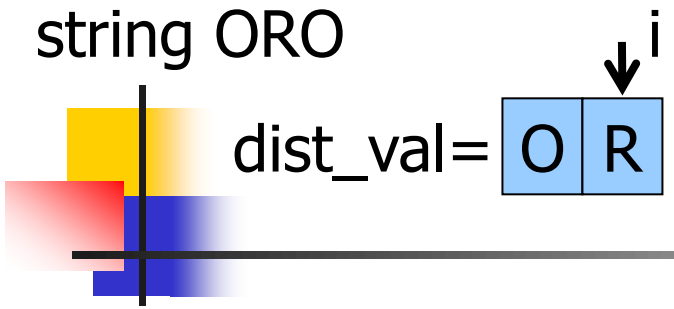


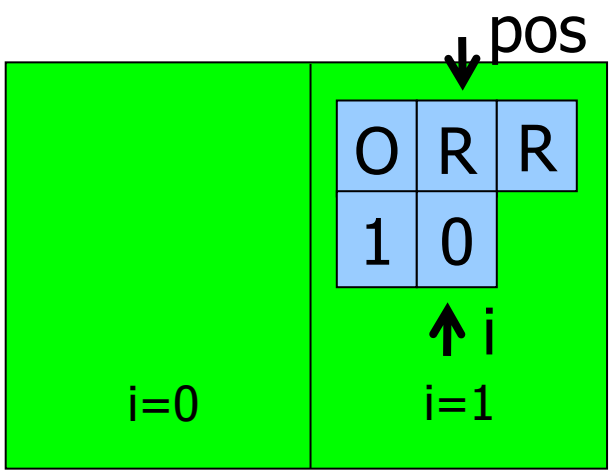


mark[i] > 0

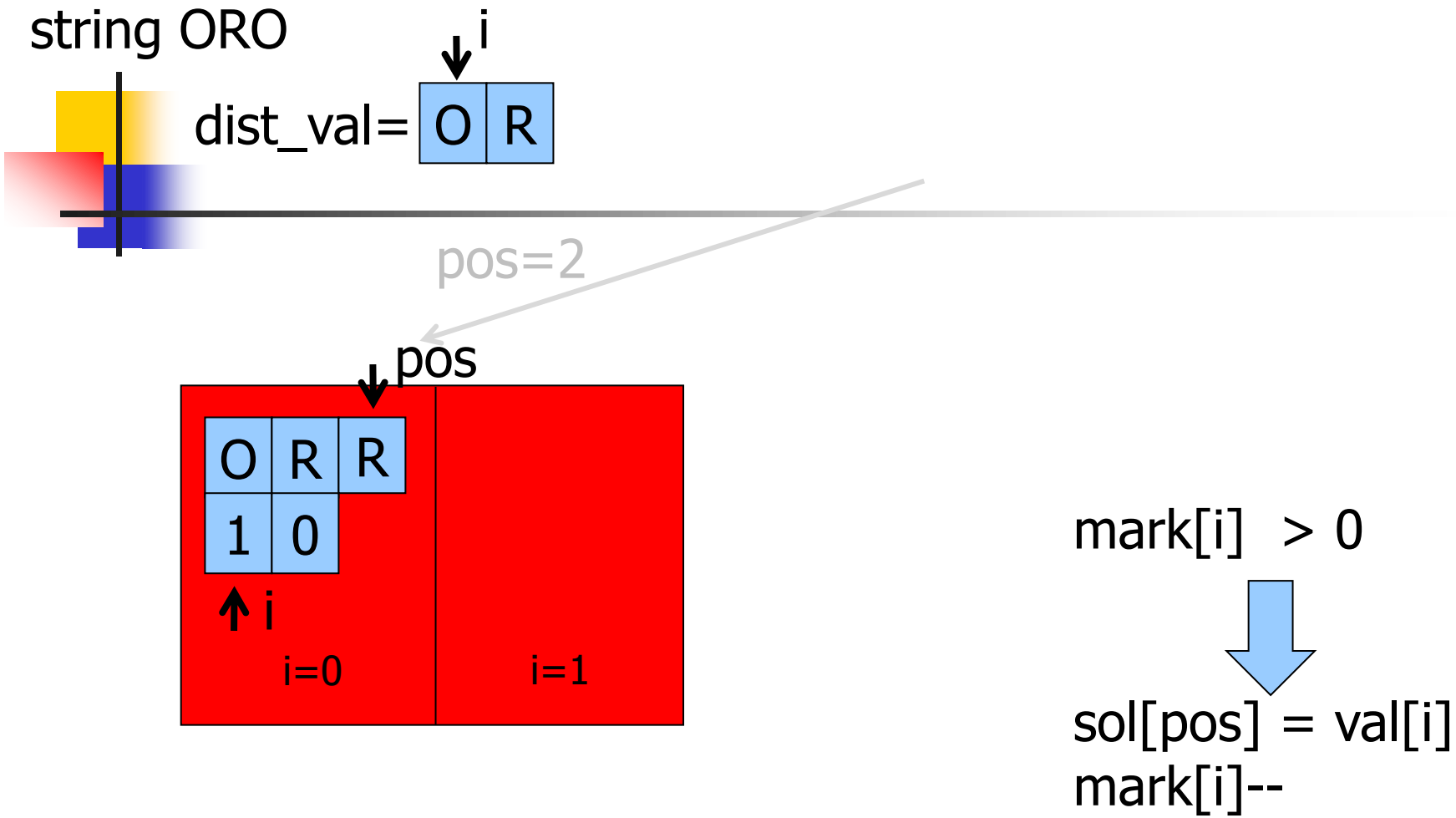


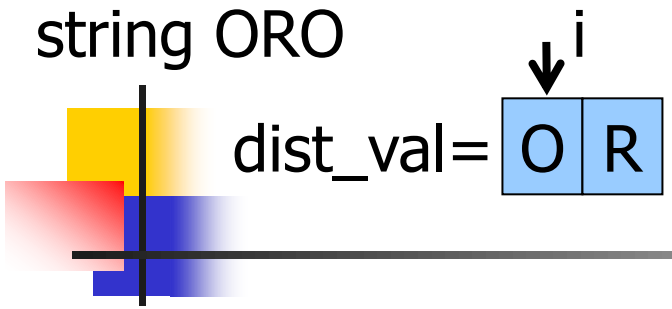
sol[pos] = val[i]
mark[i]--

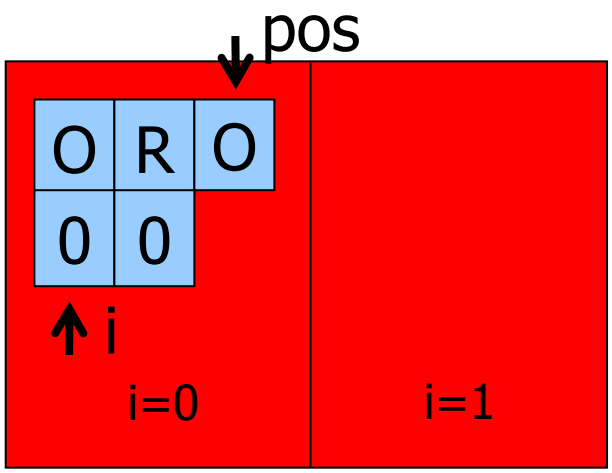




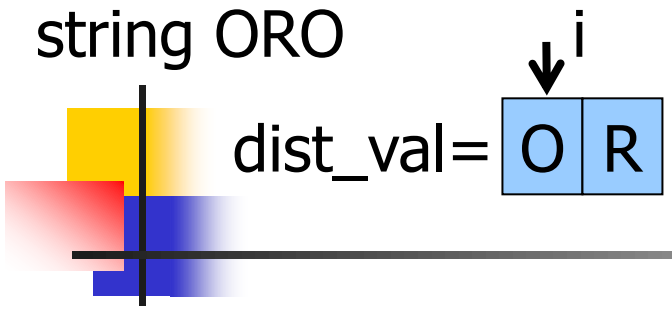
recursion
with pos=2

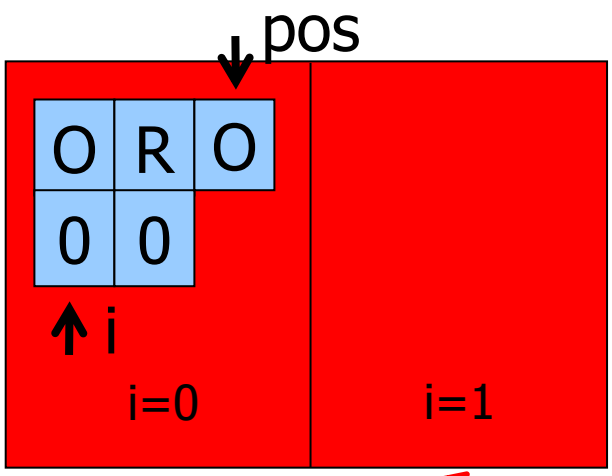

string ORO
i
dist_val=
O
R
pos=2
pos
O
R
R
1
0
i
i=0
i=1
mark[i] > 0
sol[pos] = val[i]
mark[i]--



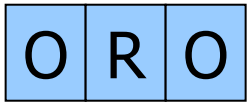


recursion
with pos=3

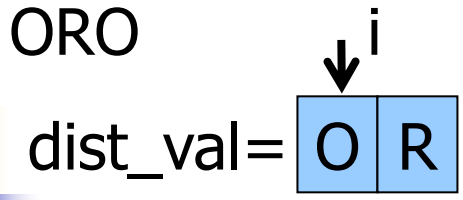


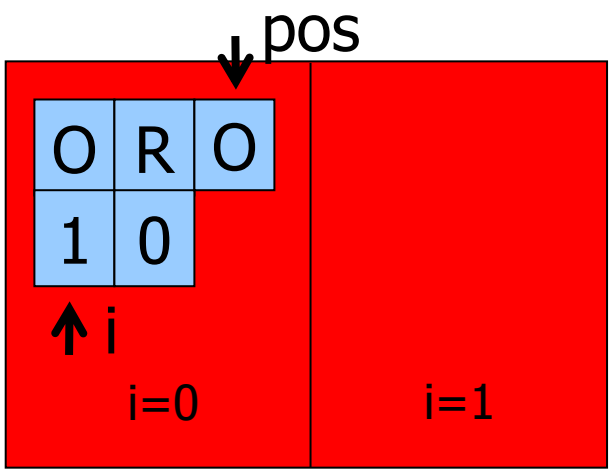


termination: display, update count
return



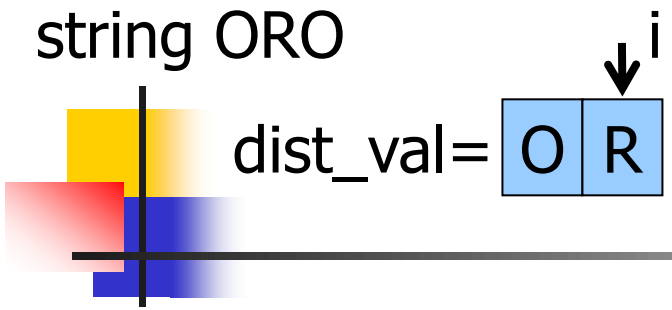
string ORO

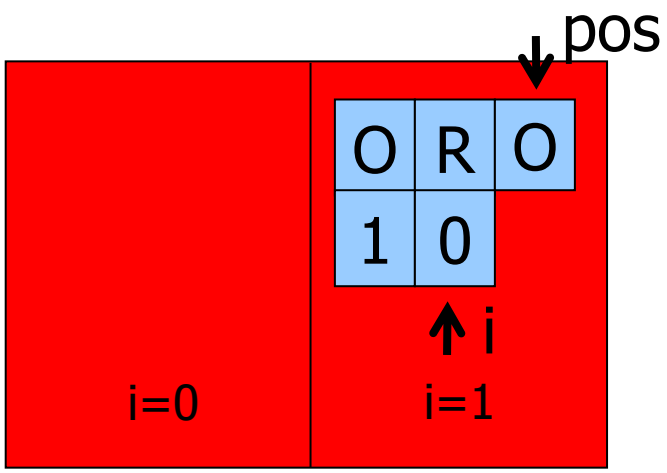




mark[i]++

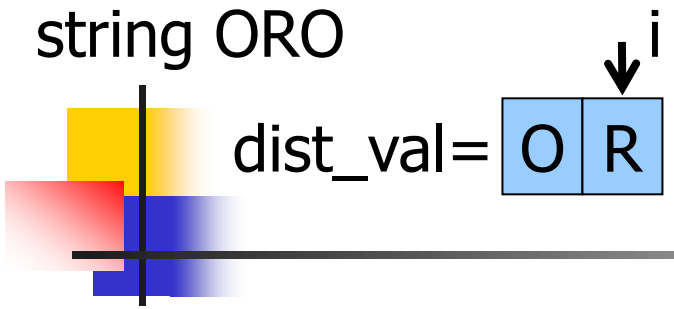
backtrack

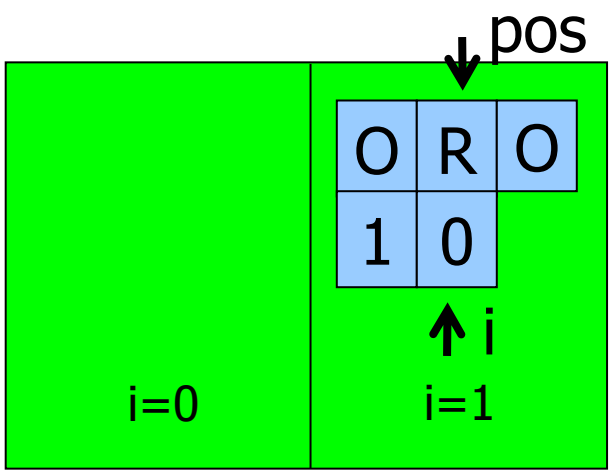




mark[i] is not > 0

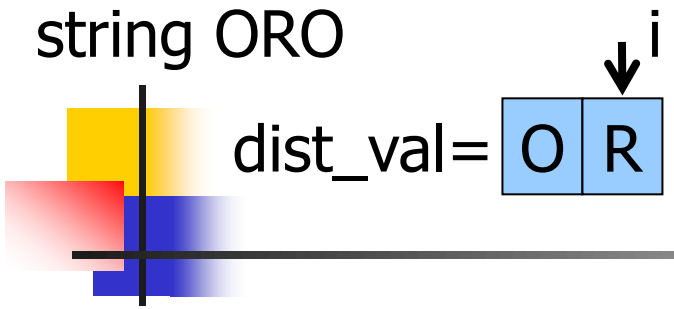
for loop finished, returna

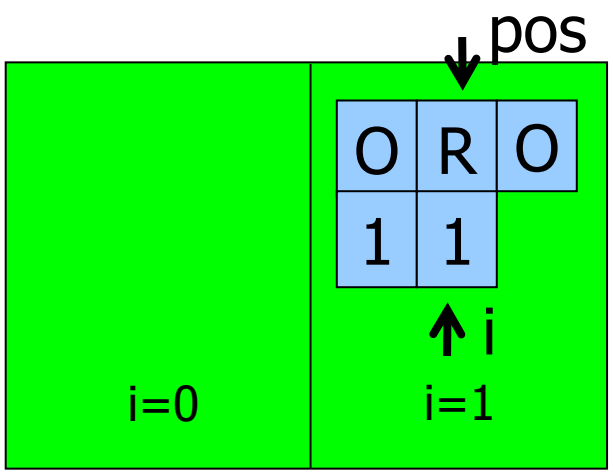




mark[i]++

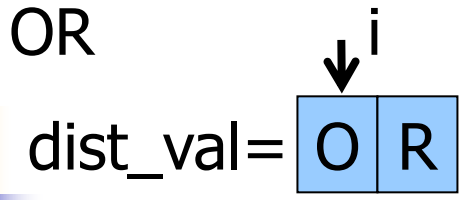
backtrack

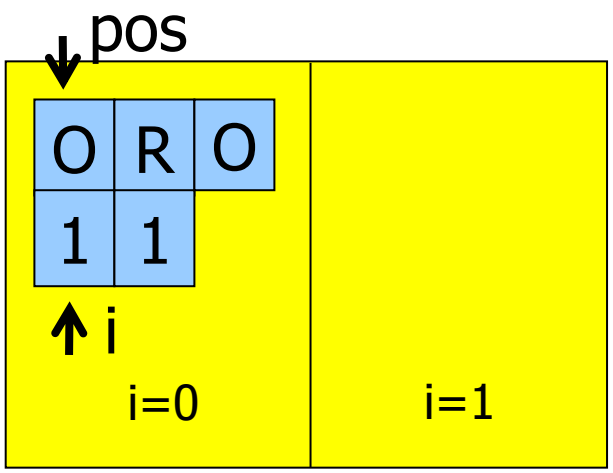




for loop finished, return

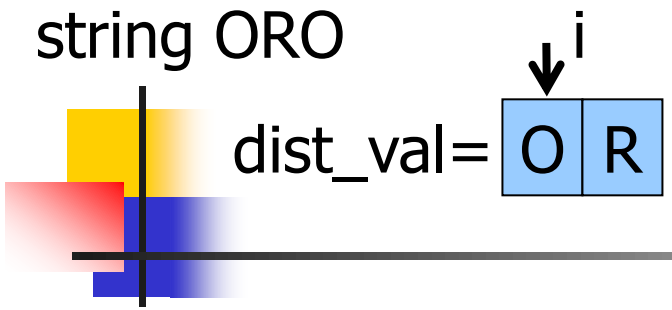
string OR

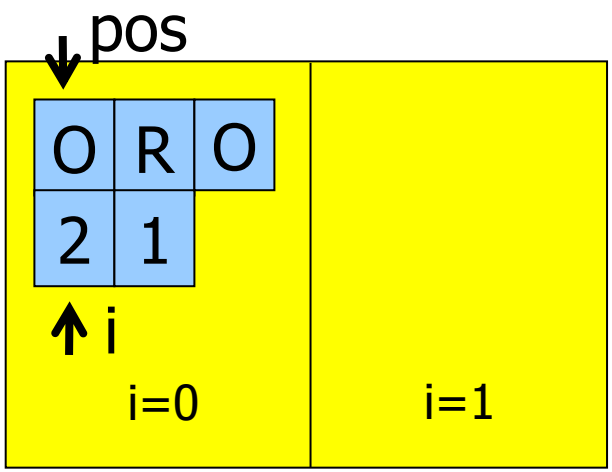




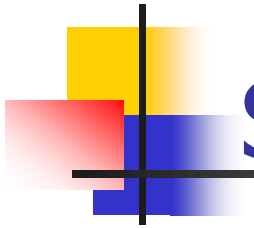
mark[i]++

backtrack





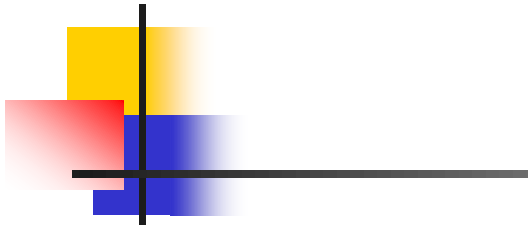
etc. etc.



Simple combinations

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

Example: n choose k

Given a set of n integers, generate all simple combinations of k among these values.

The number of combinations is n!/((n-k)!*k!) .

Example

for val = {7, 2, 0, 4, 1}:

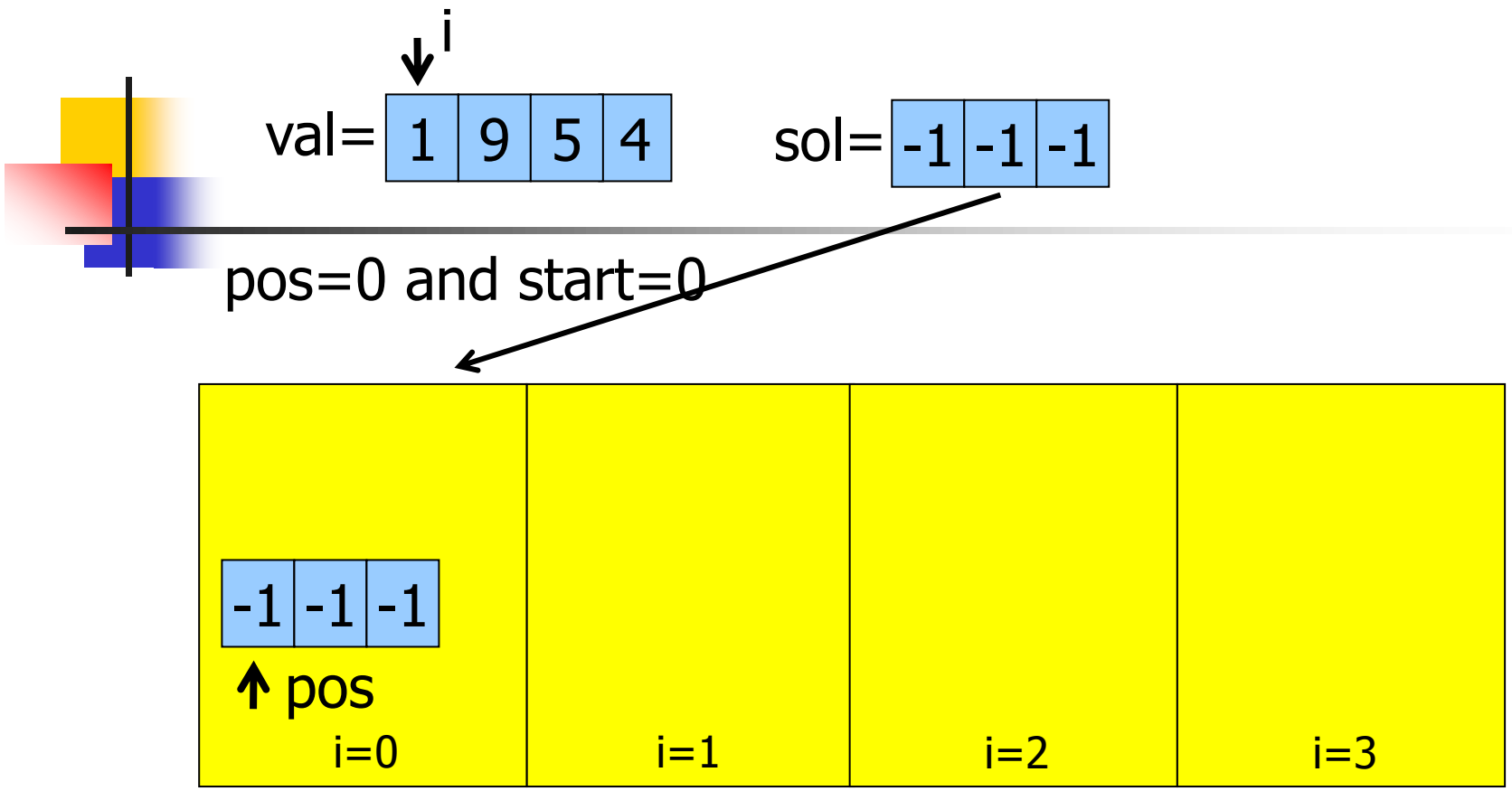
- n = 5 e k = 4: 5 combinations

{7,2,0,4} {7,2,0,1} {7,2,4,1} {7,0,4,1} {2,0,4,1}

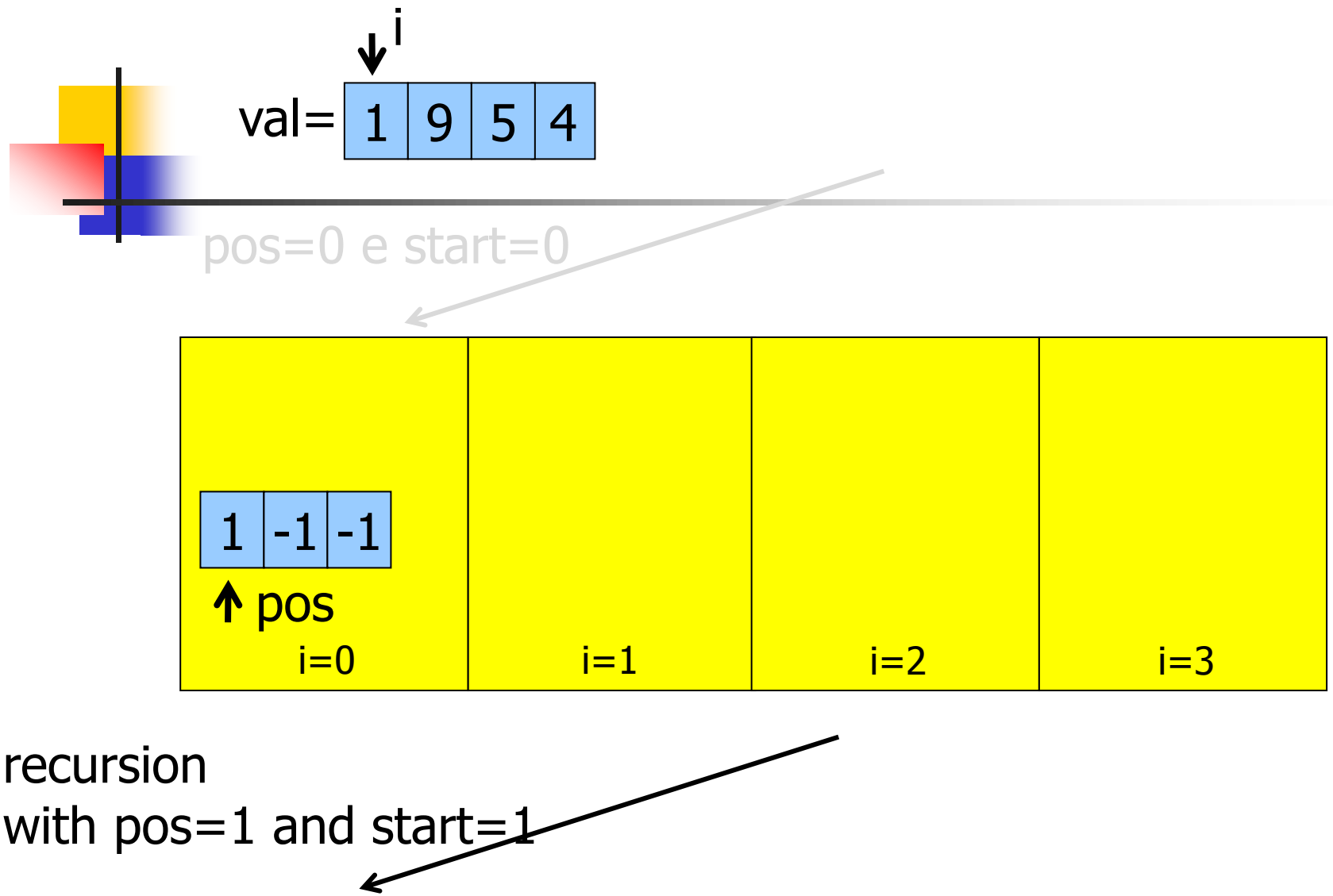
for val = {1, 9, 5, 4} :

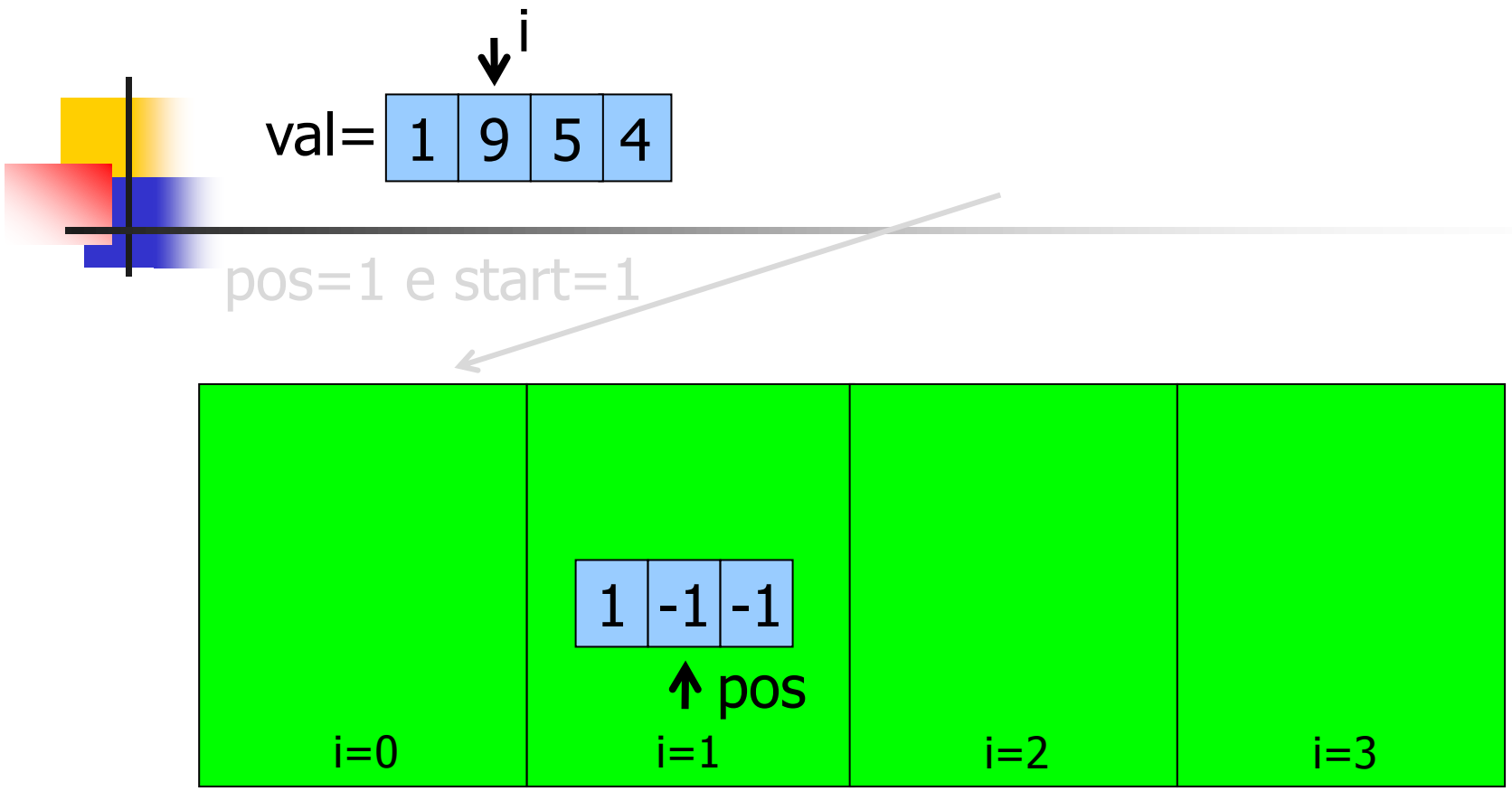
- n = 4 e k = 3: 4 combinations

{1,9,5} {1,9,4} {1,5,4} {9,5,4}

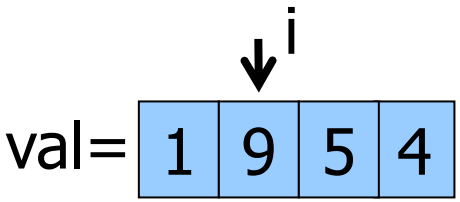


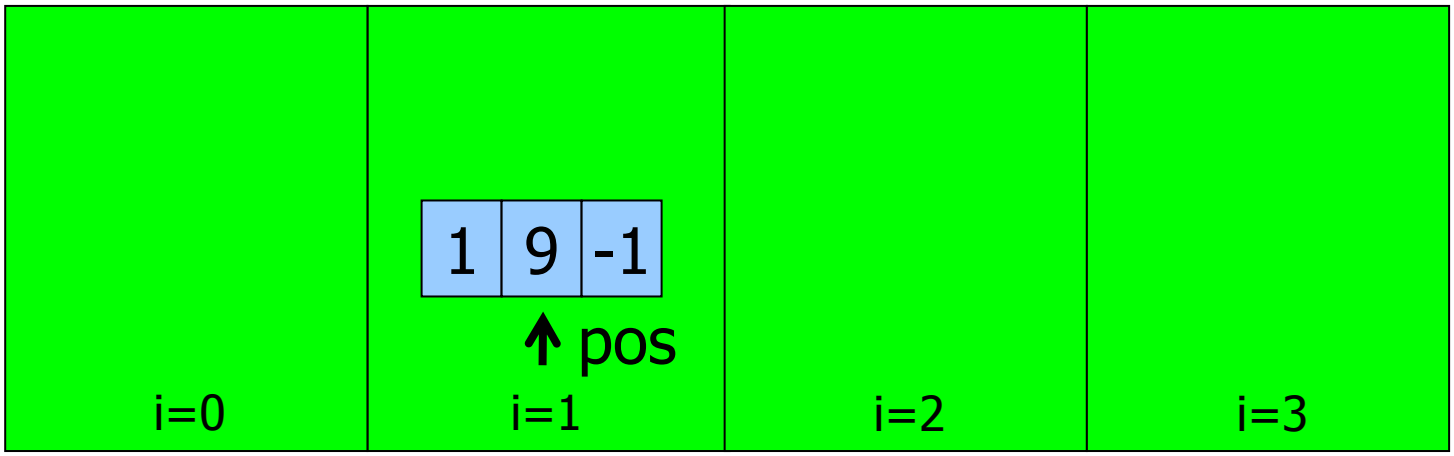
sol[pos] = val[i]


i
val=
1
9
5
4
pos=0 e start=0
1
-1
-1
pos
i=0
i=1
i=2
i=3
recursion
with pos=1 and start=1

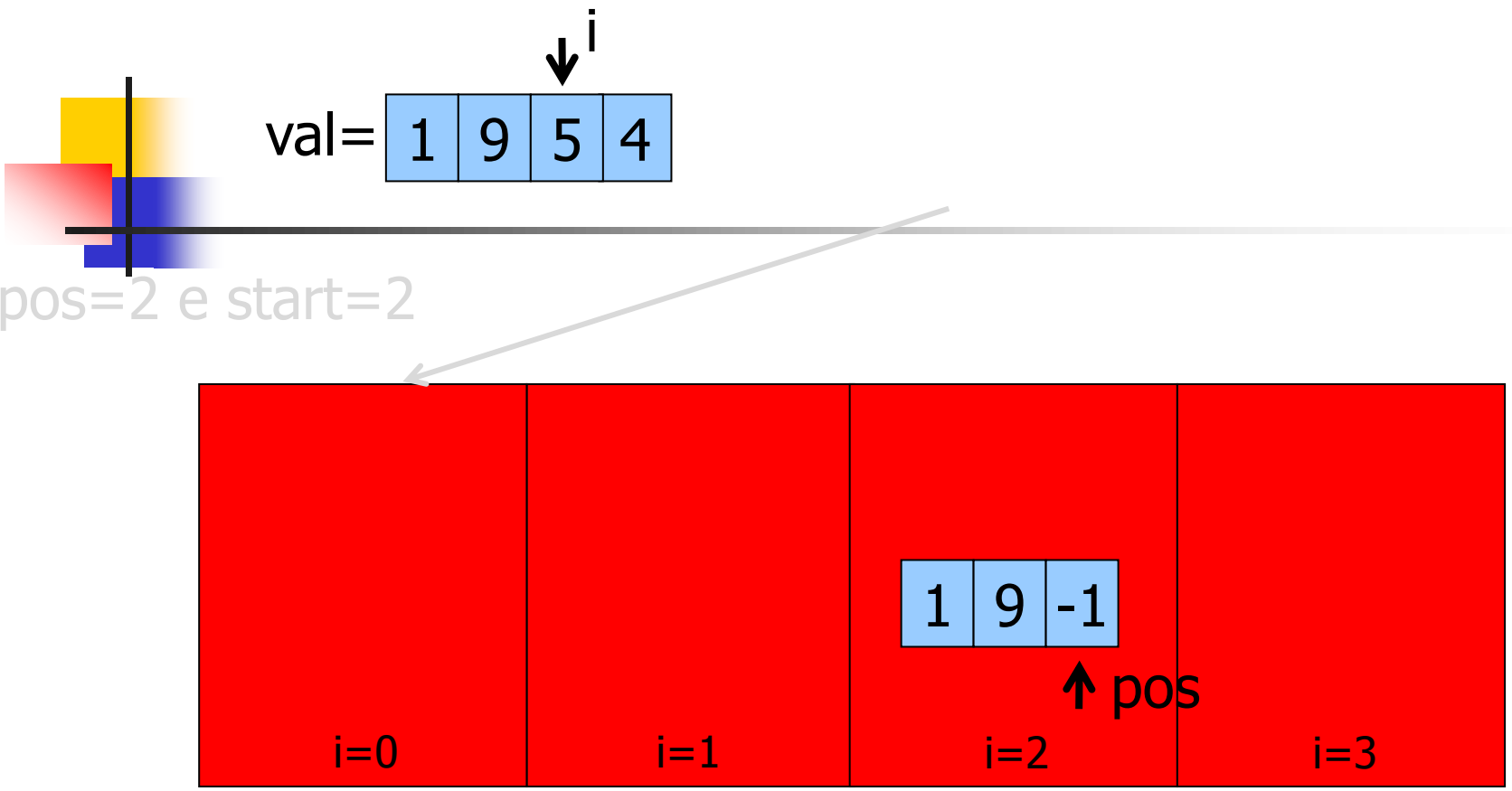


sol[pos] = val[i]

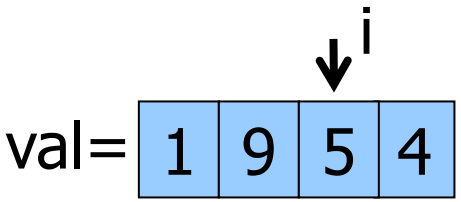


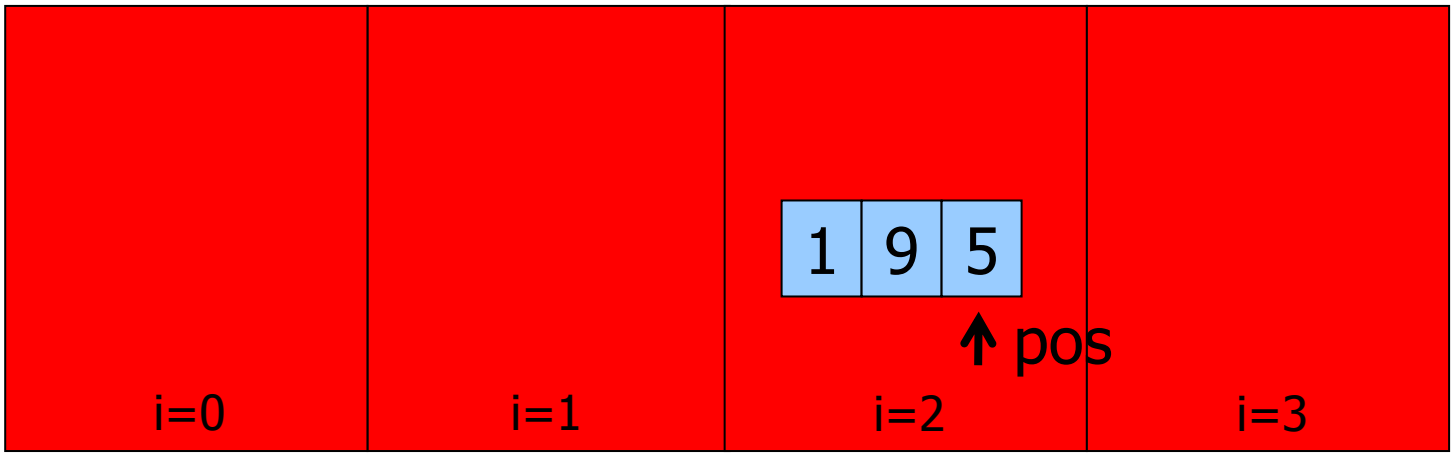


recursion
with pos=2 and start=2

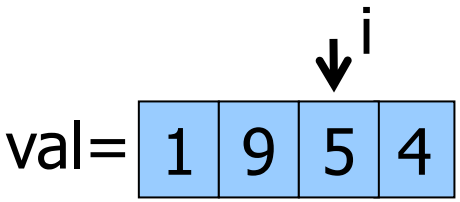


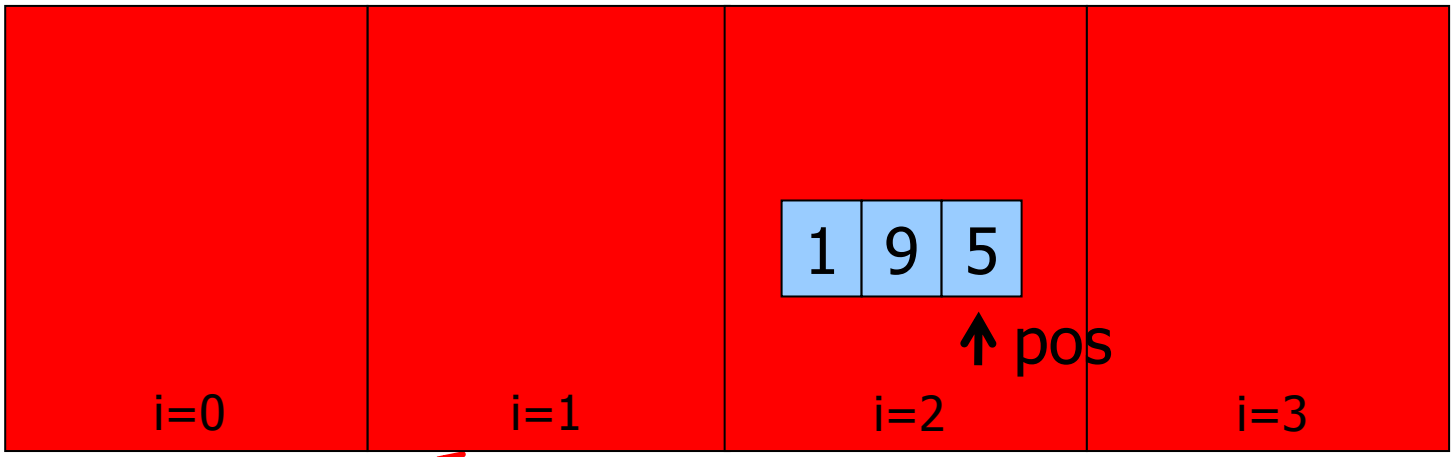
sol[pos] = val[i]





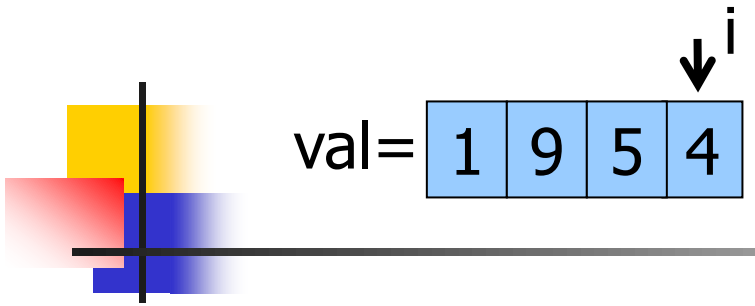
recursion
with pos=3 and start=3

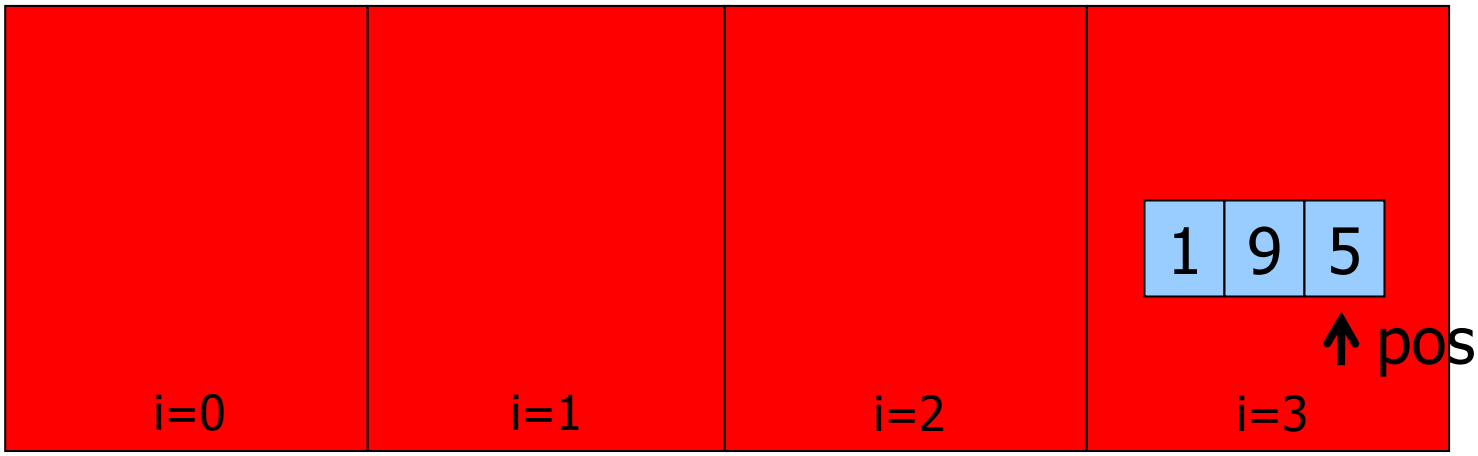




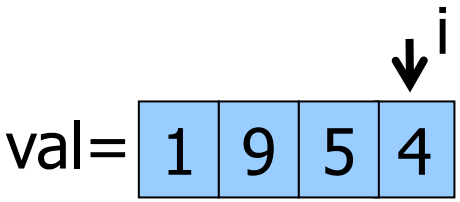
termination: display, update count return

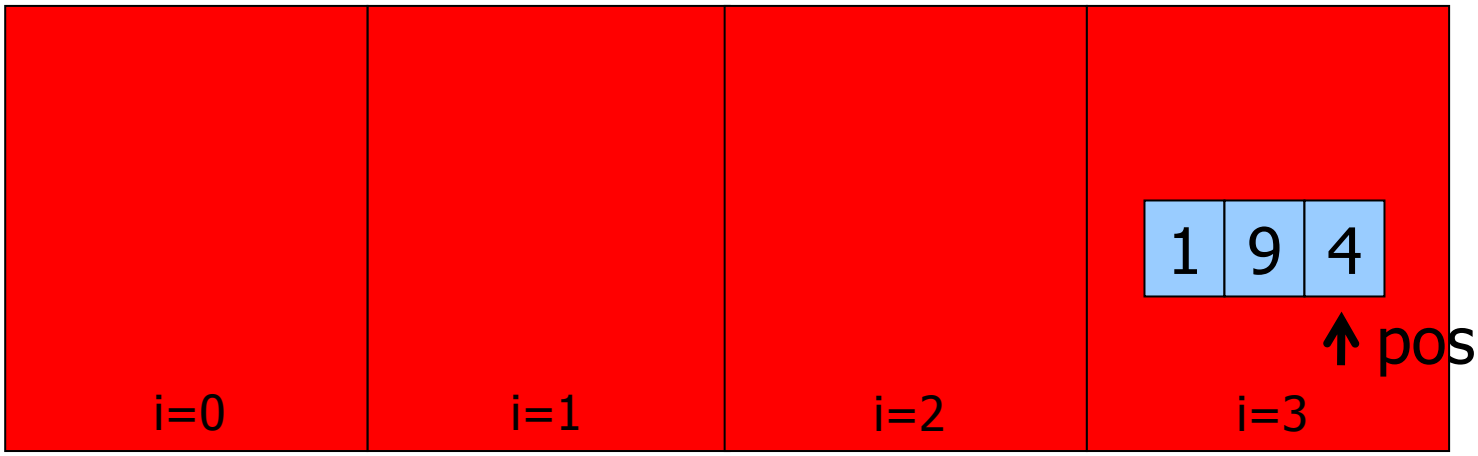
sol= 1 9 5



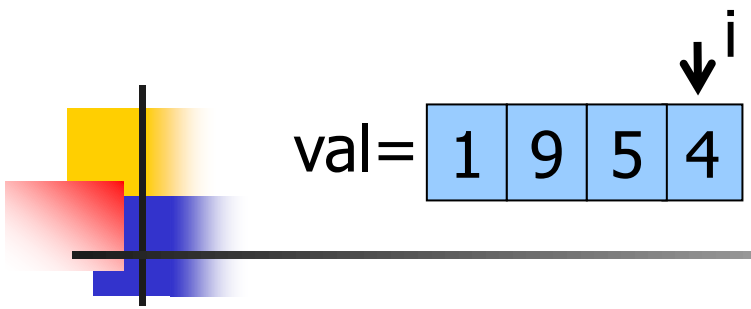


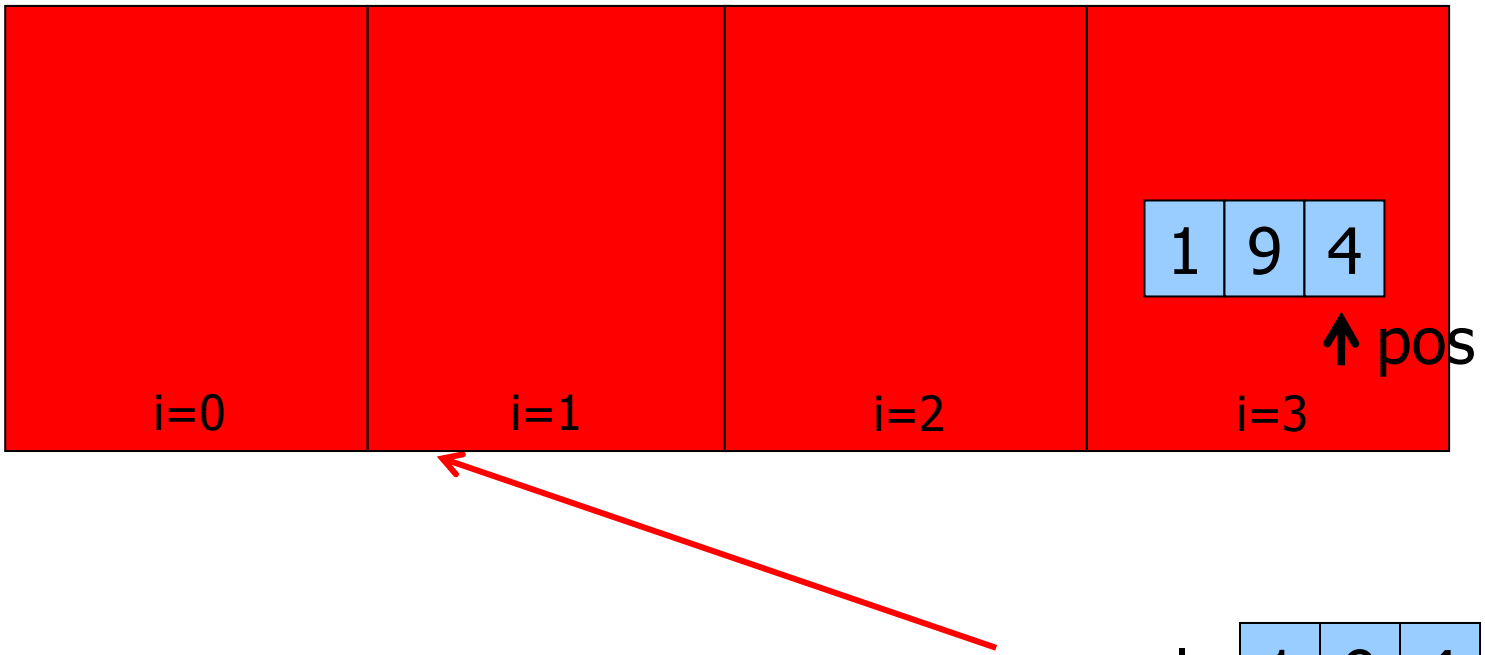
sol[pos] = val[i]





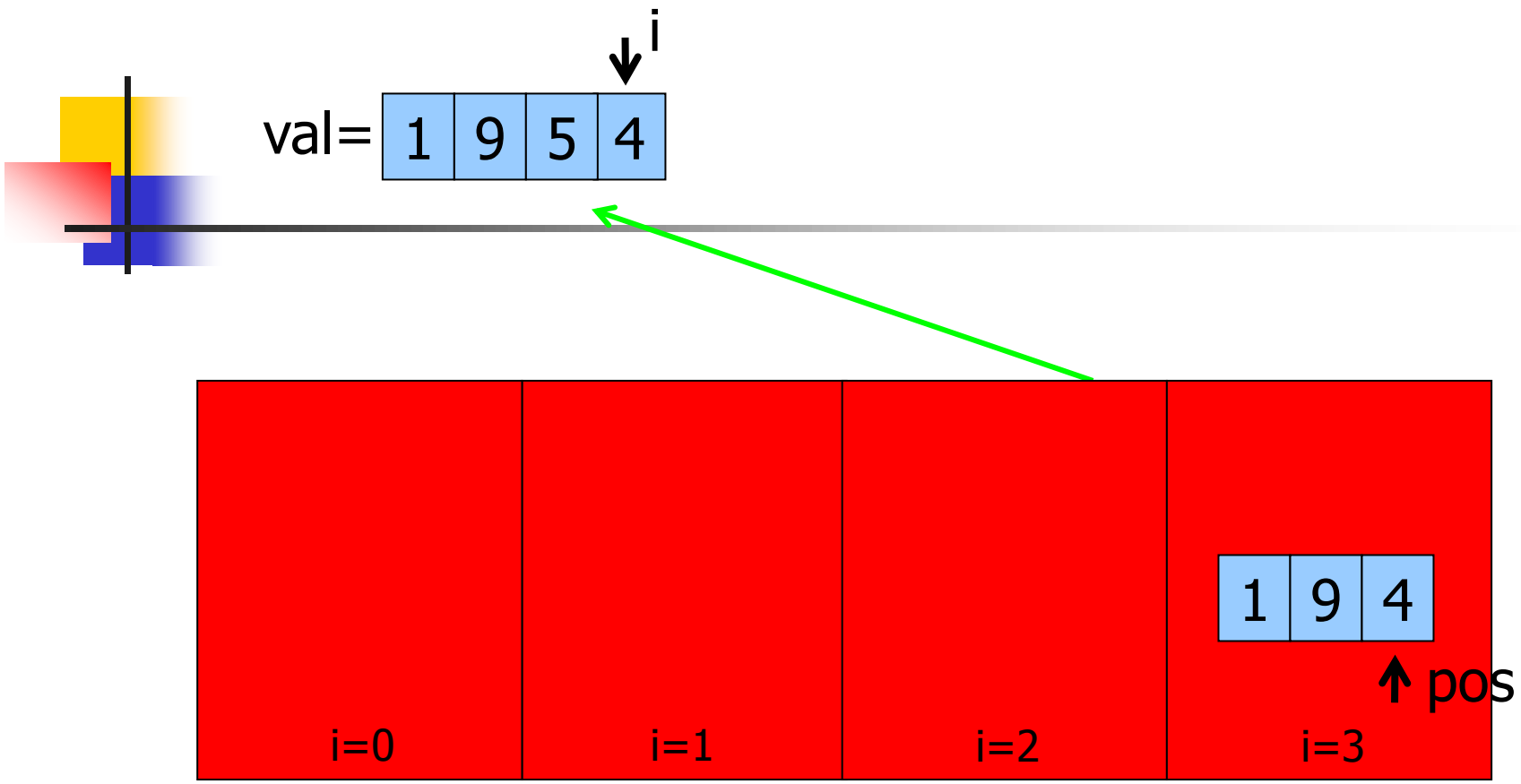
recursion
with pos=3 and start=4



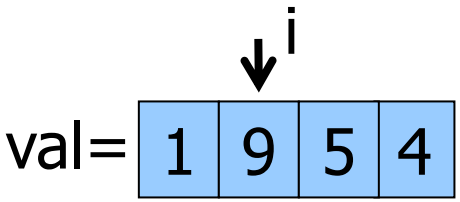


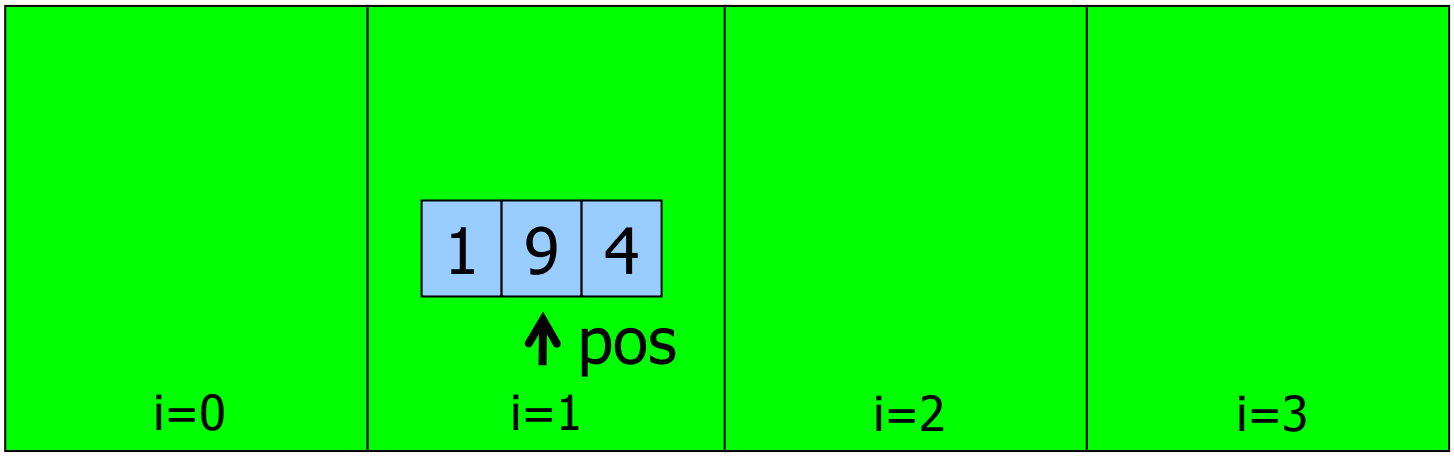
termination: display, update count return

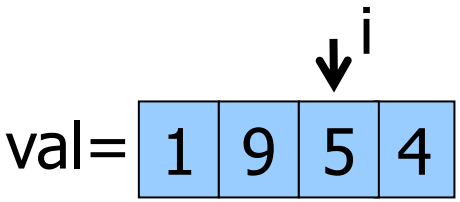
sol= 1 9 4

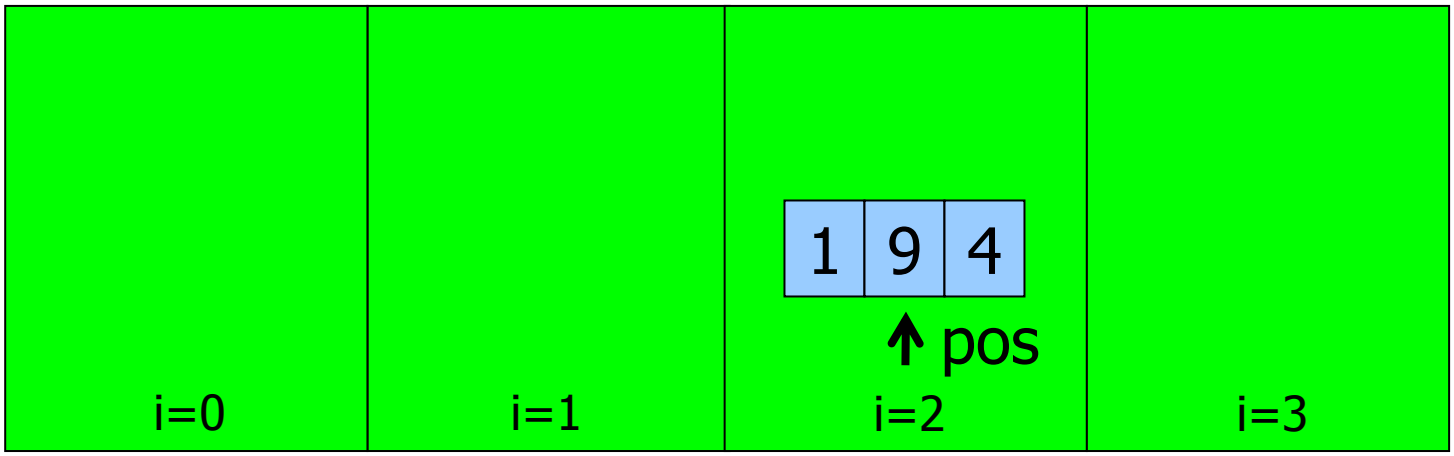


for loop finished, return

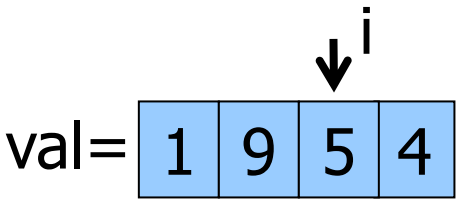

i
val=
1
9
5
4

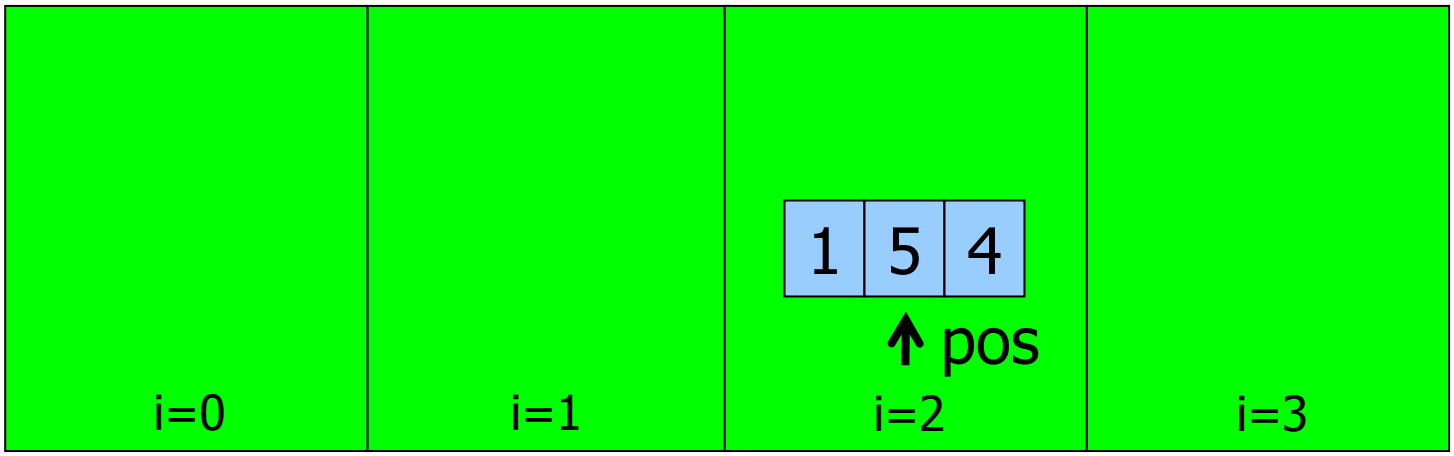

1
9
4
pos
i=0
i=1
i=2
i=3



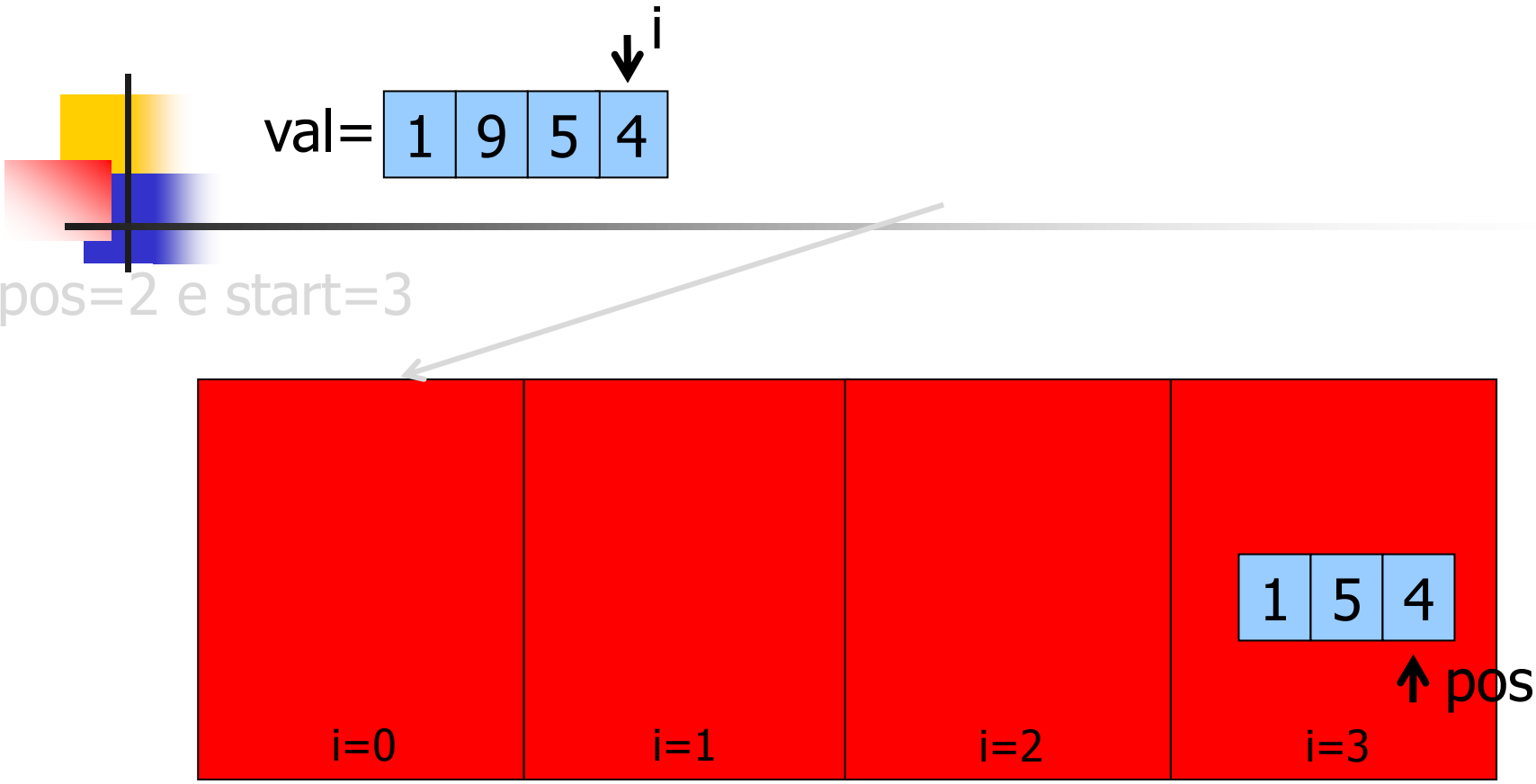


sol[pos] = val[i]

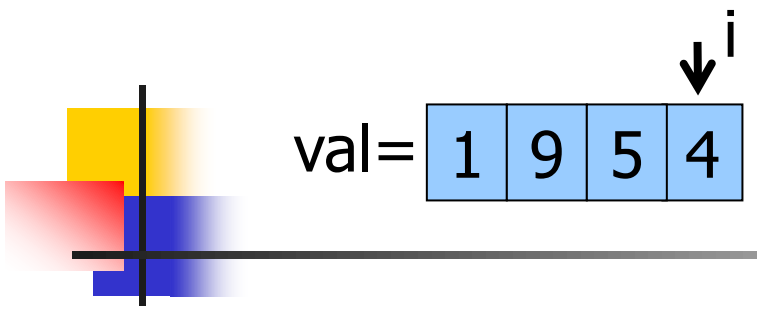


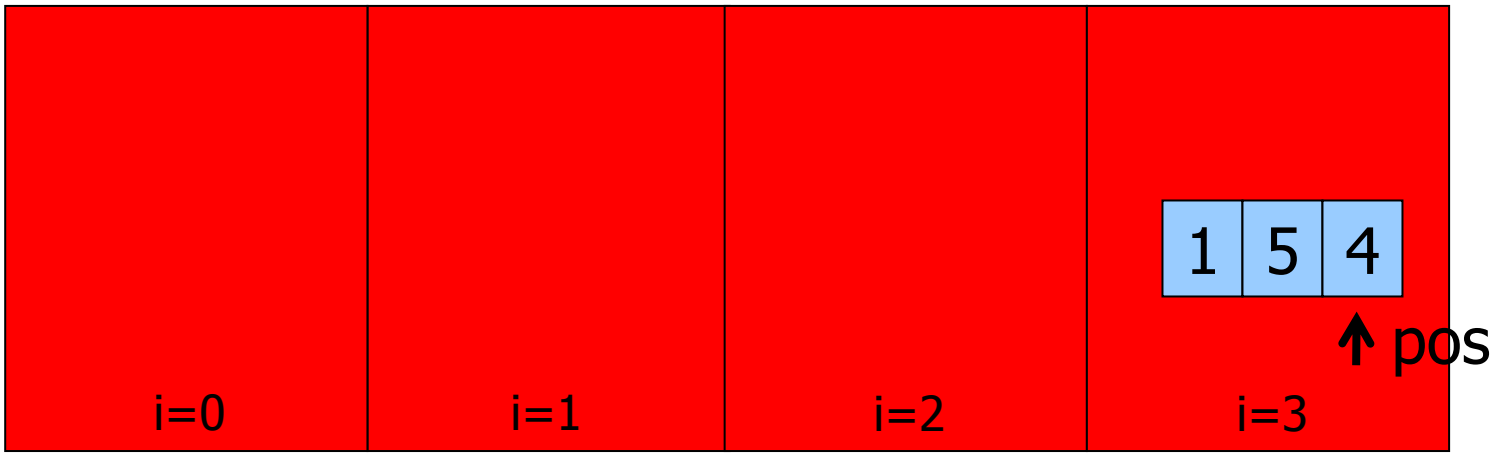


recursion
with pos=2 and start=3

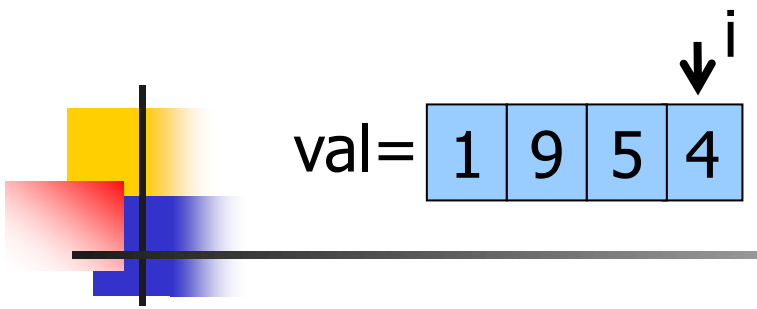


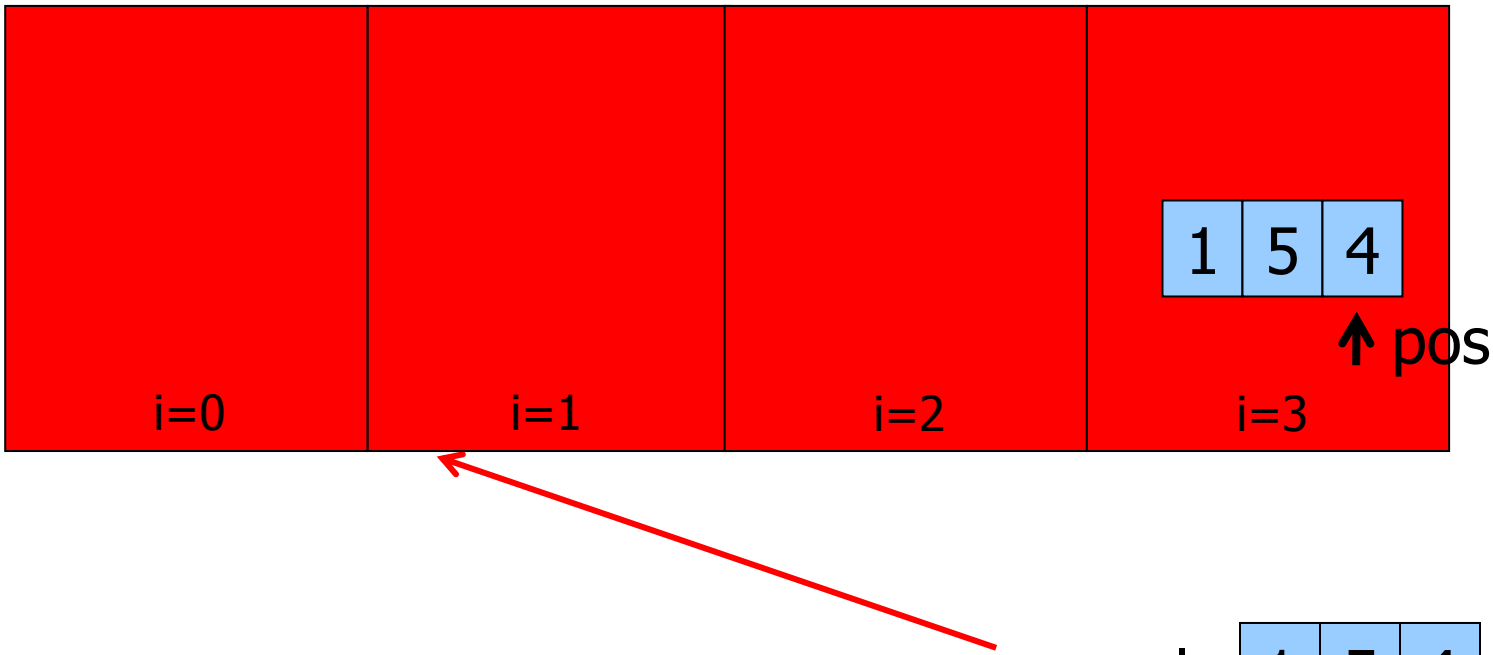
sol[pos] = val[i]





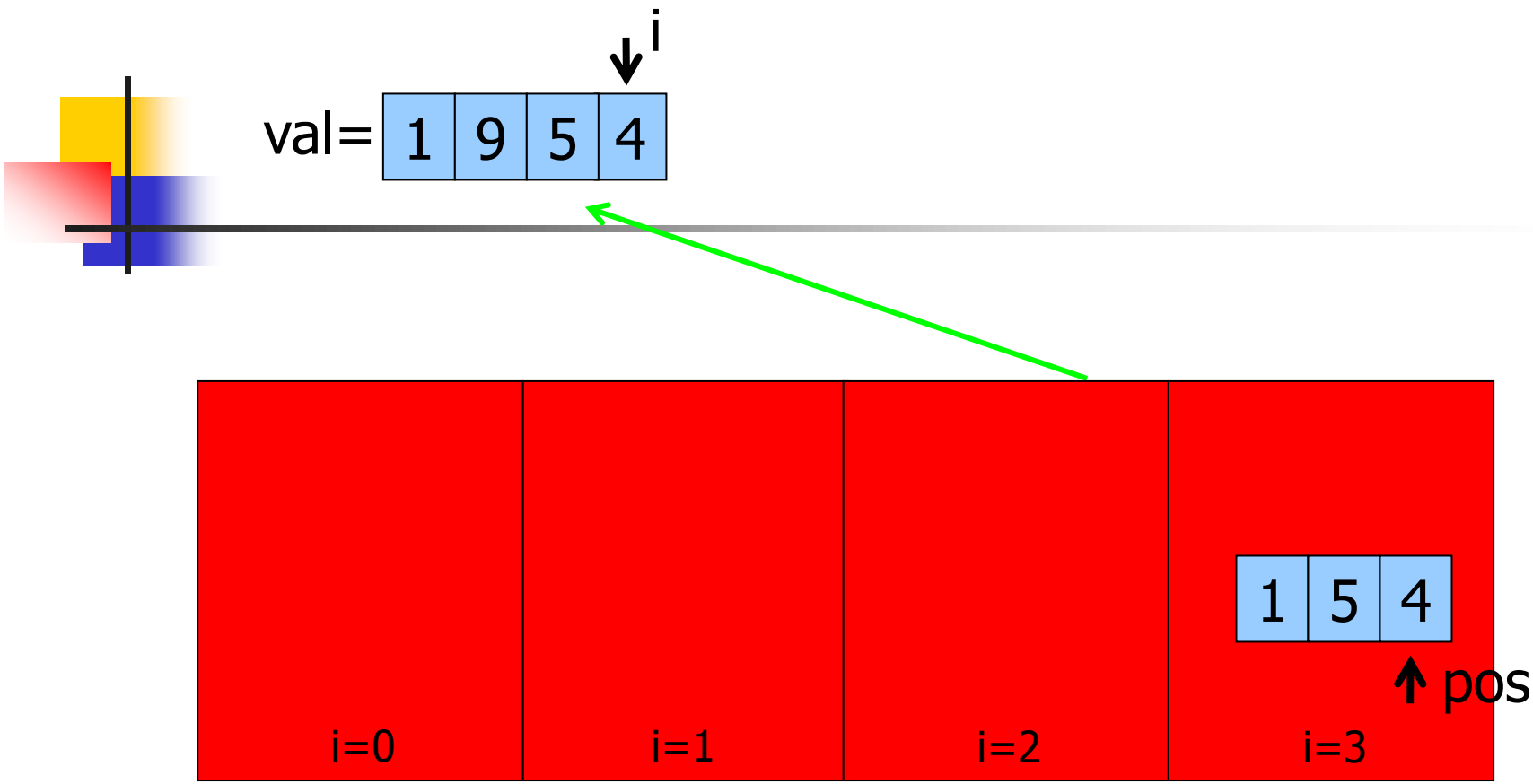
recursion
with pos=3 and start=4



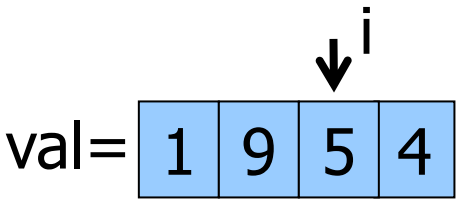


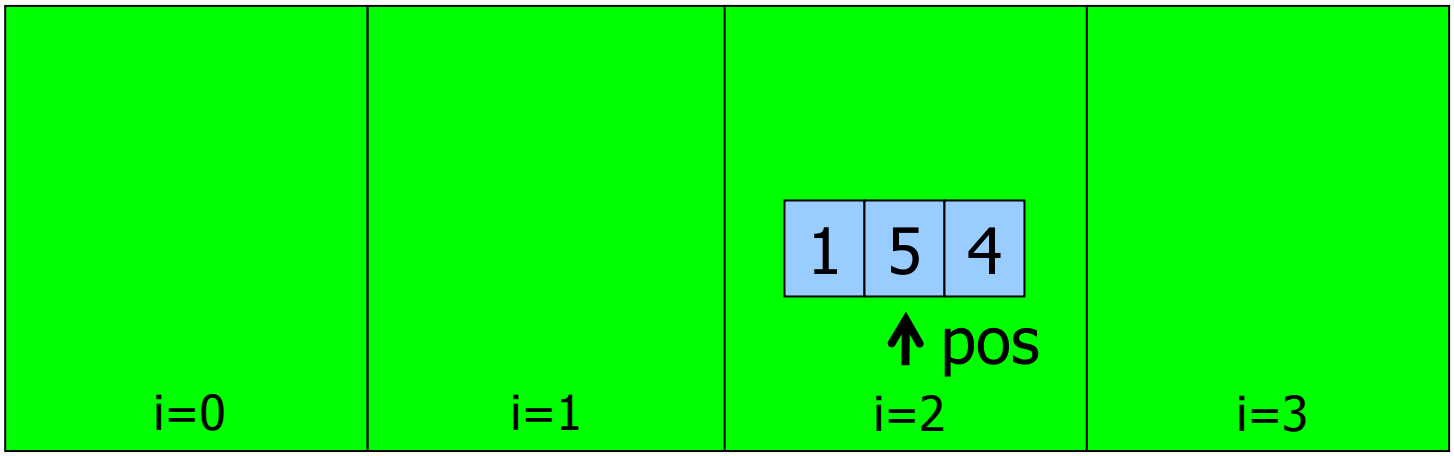
termination: display, update count return

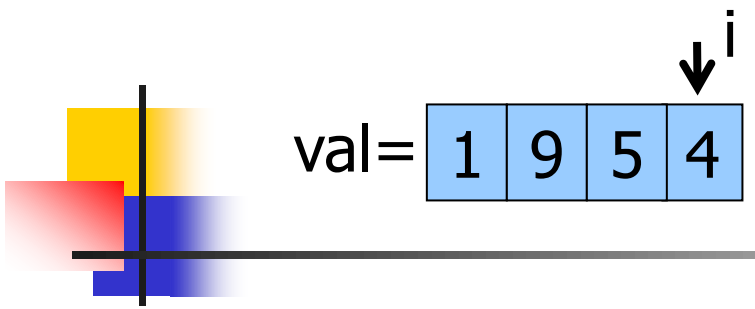
sol= 1 5 4

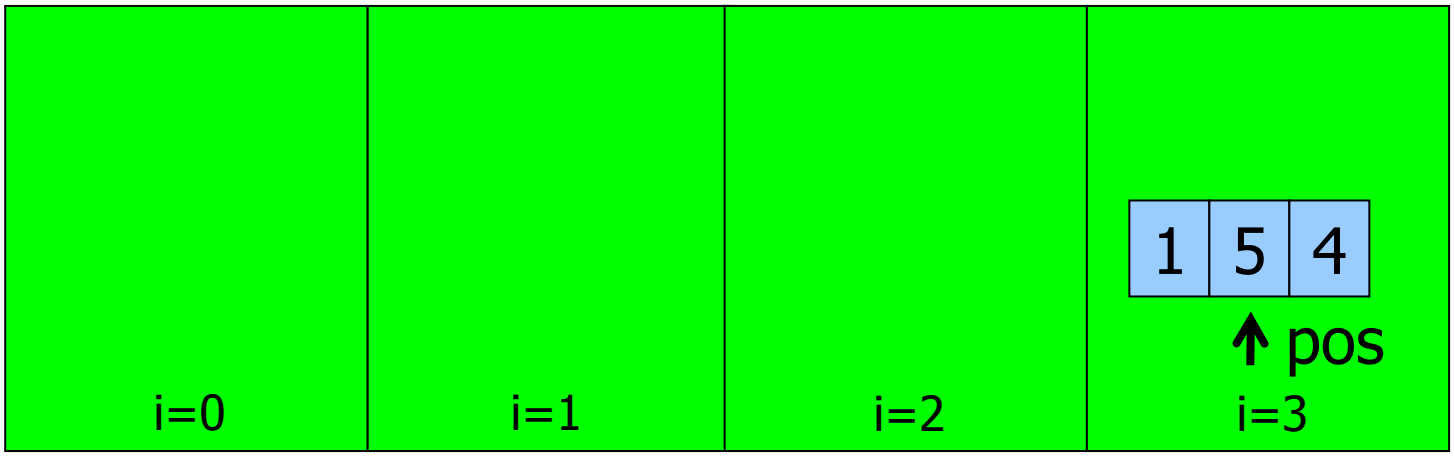


for loop finished, return

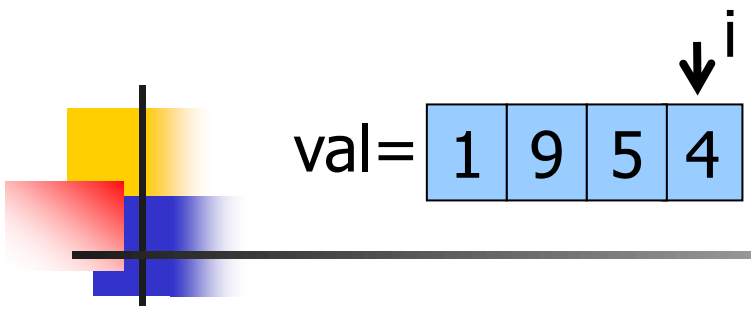

i
val=
1
9
5
4

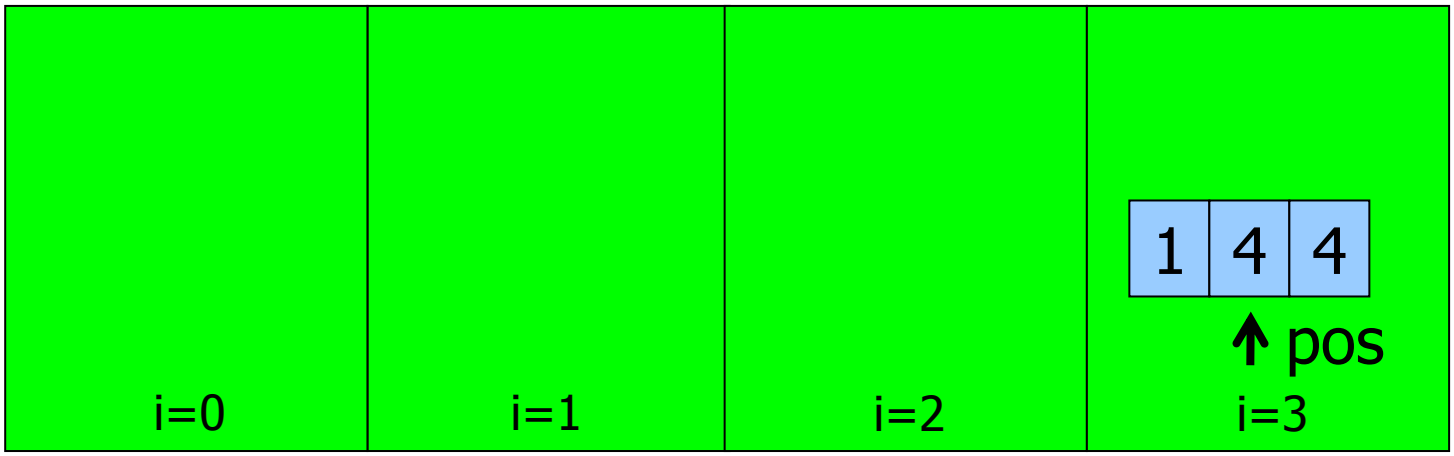

1
5
4
pos
i=0
i=1
i=2
i=3



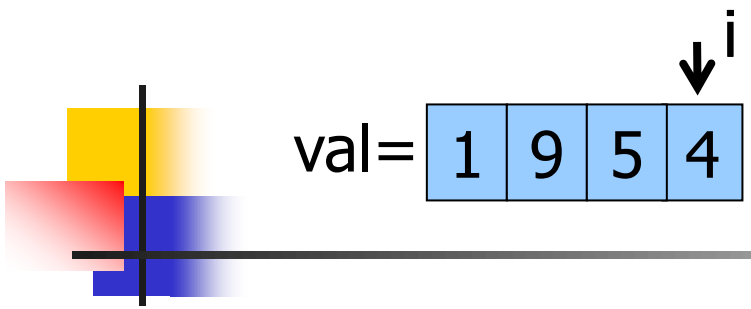


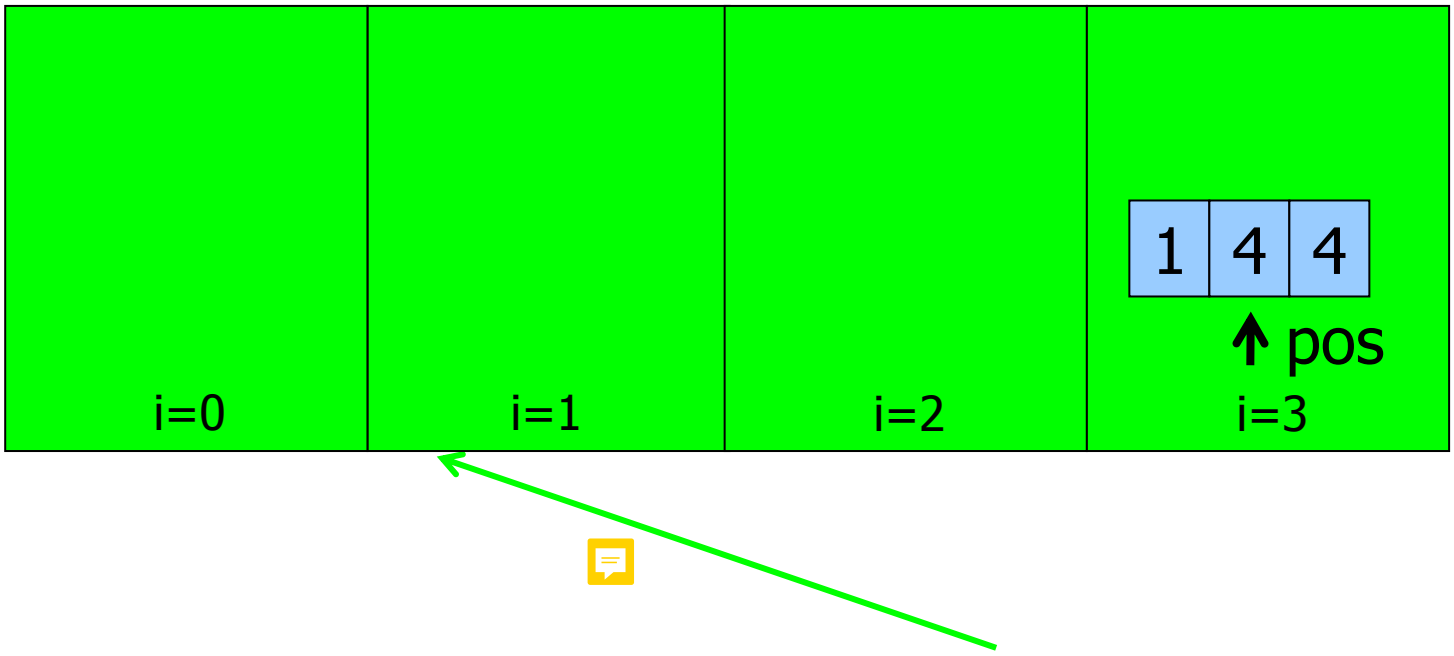
sol[pos] = val[i]



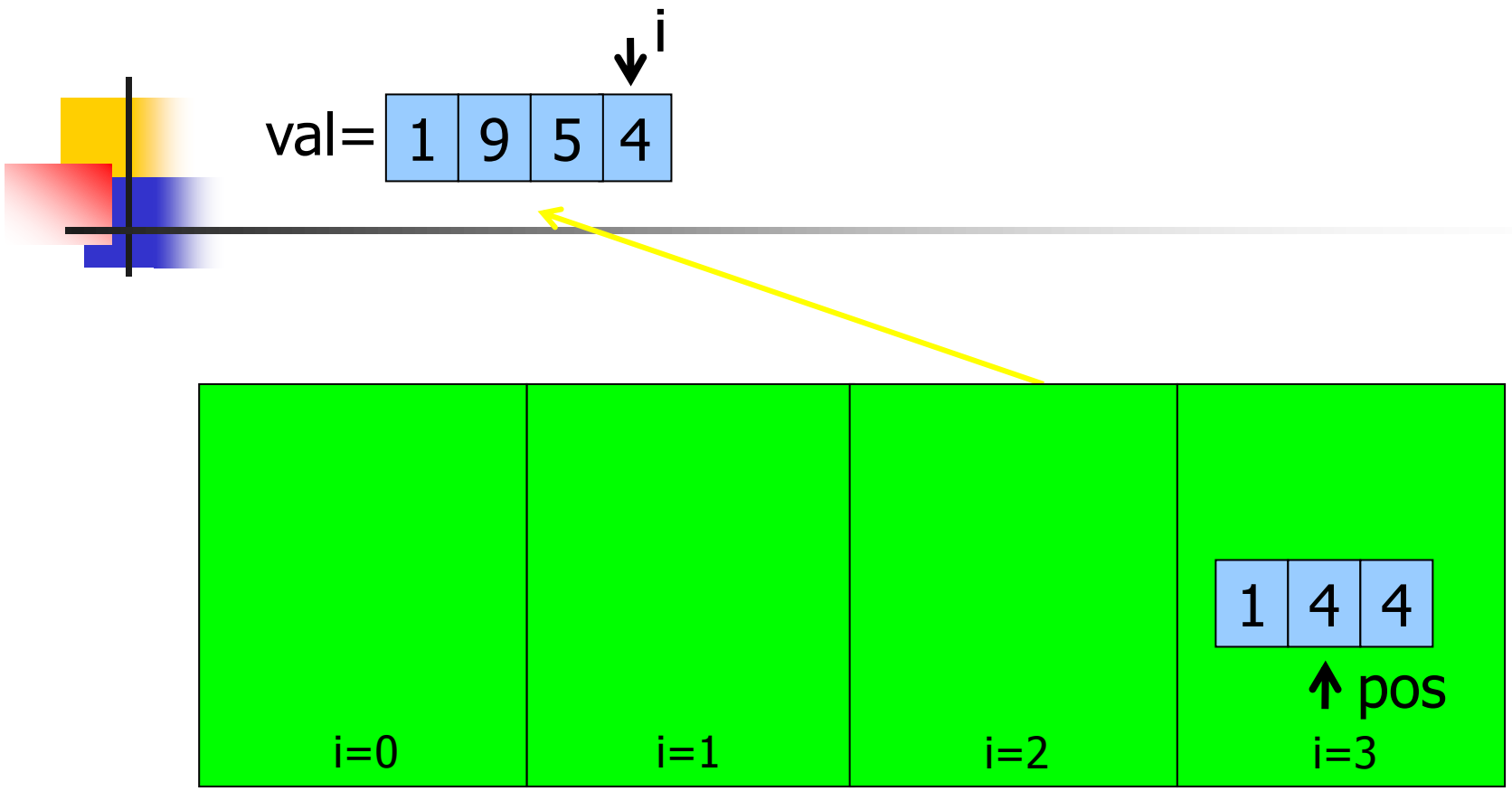


recursion
with pos=2 and start=4

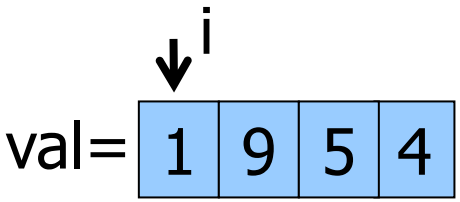


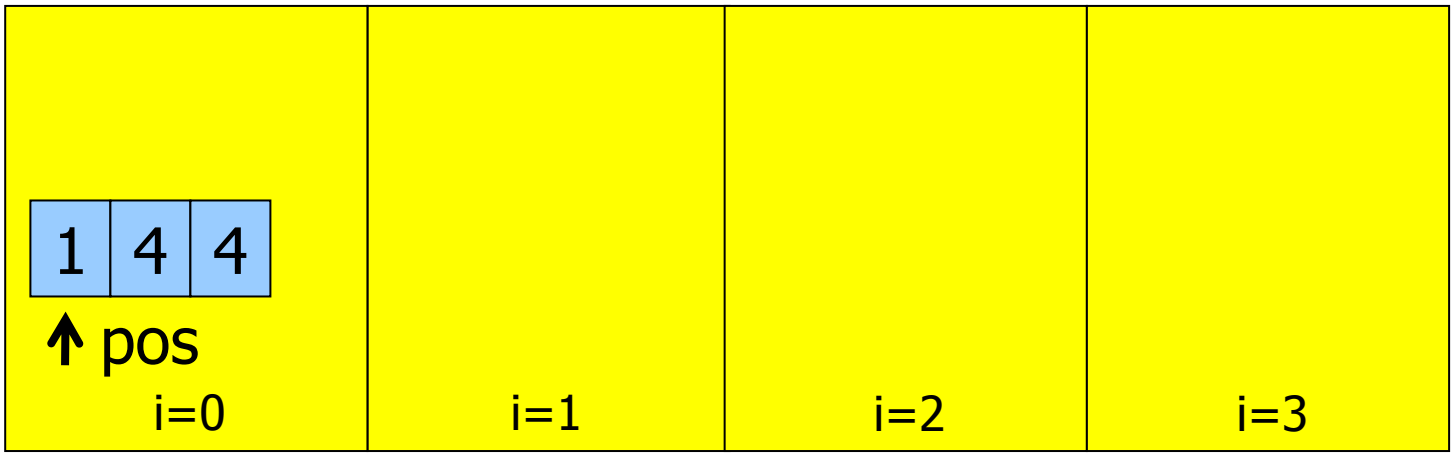


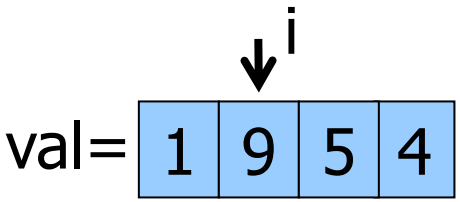
the for loop doesn't event start
return

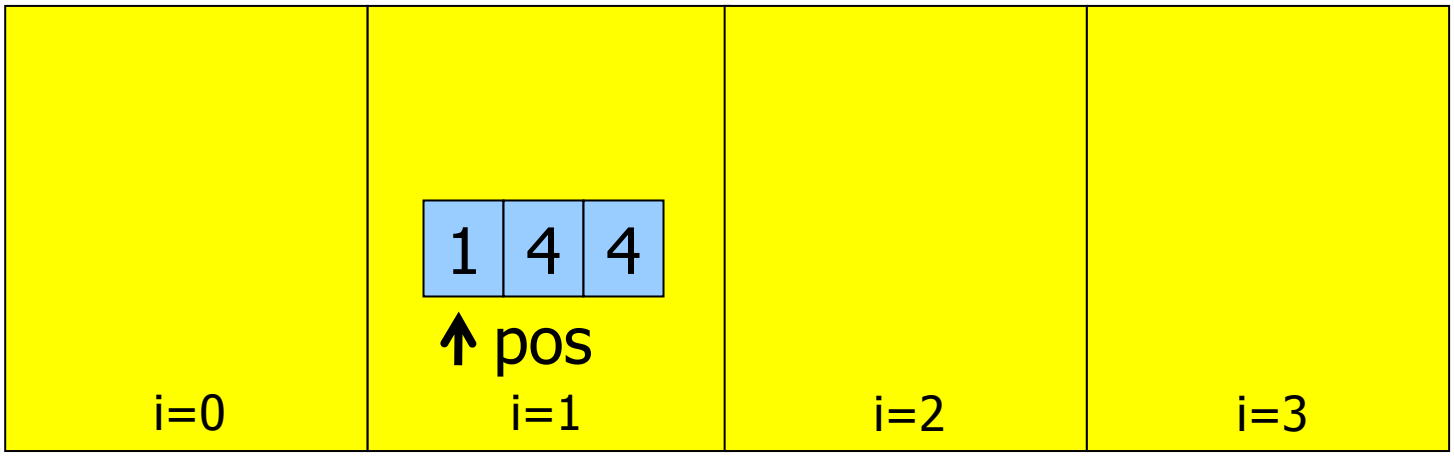


for loop finished, return

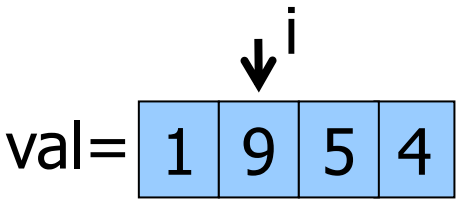

i
val=
1
9
5
4

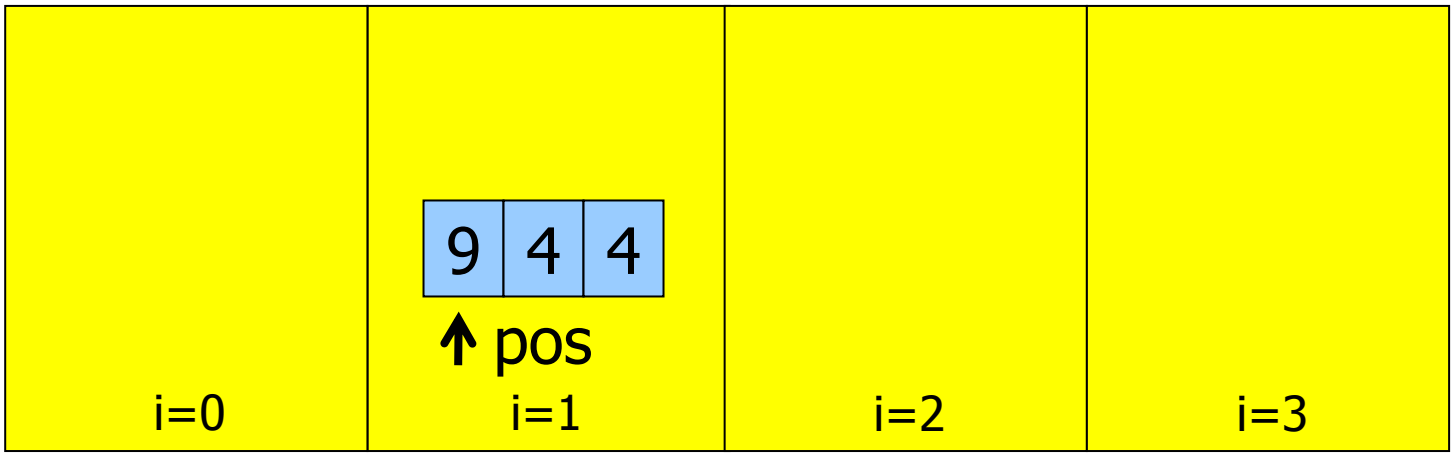

1
4
4
pos
i=0
i=1
i=2
i=3





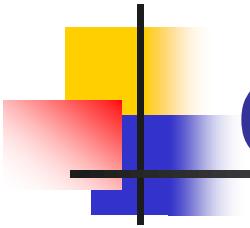
sol[pos] = val[i]





recursion
with pos=1 and start=2

etc. etc.



Combinations with repetitions

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 terminates.

`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 co
  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, start, count);
    start++;
  }
  return co
}
```

termination

iteration on choices

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

recursion on next position

update of start

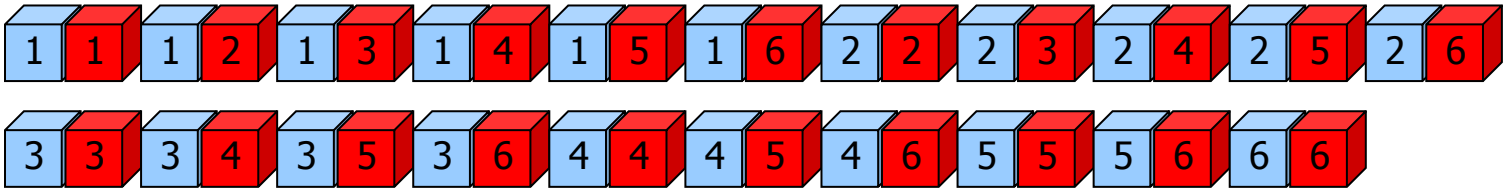
Esempio

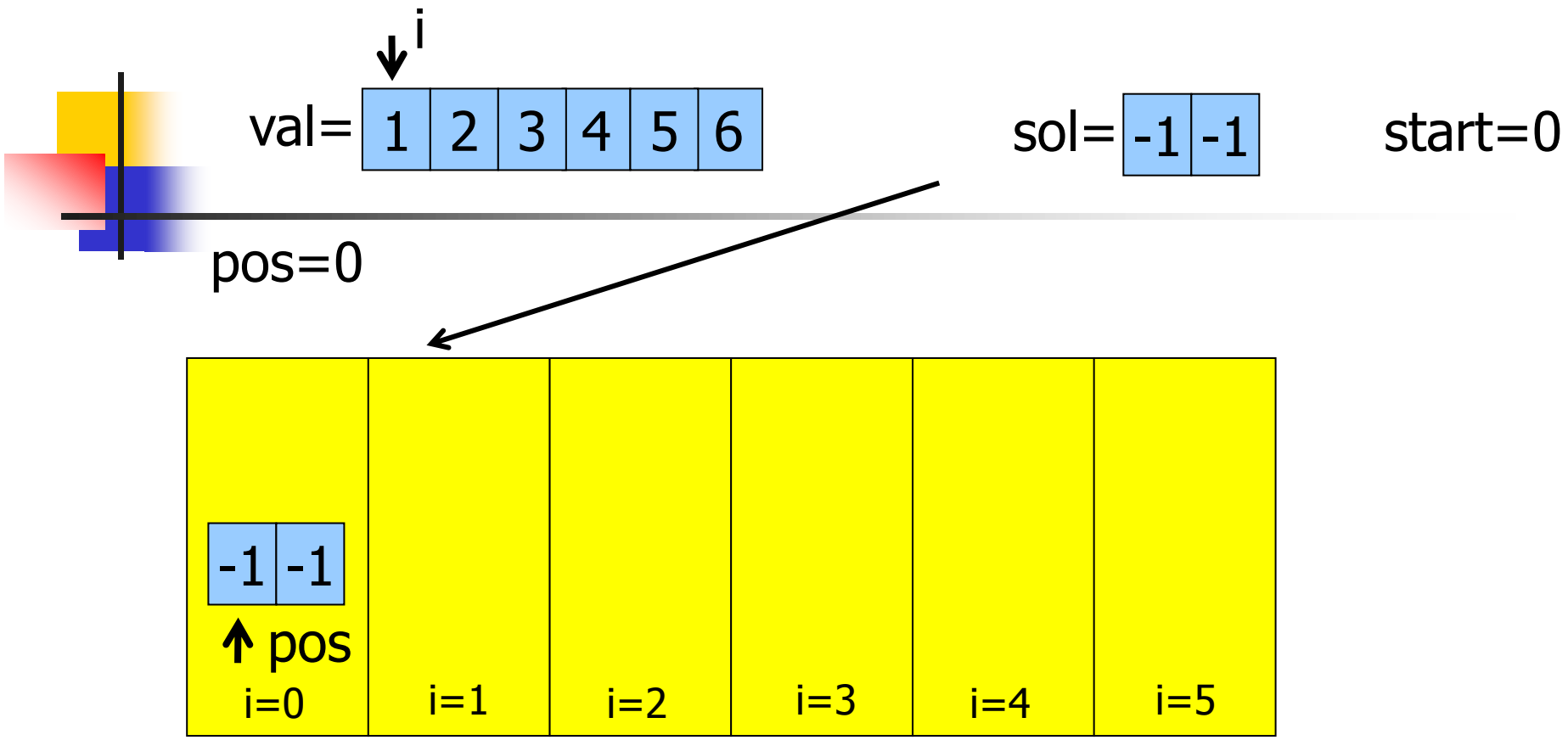
When simultaneously casting two dice, how many compositions of values may appear on 2 faces?

Modello: combinazioni con ripetizione

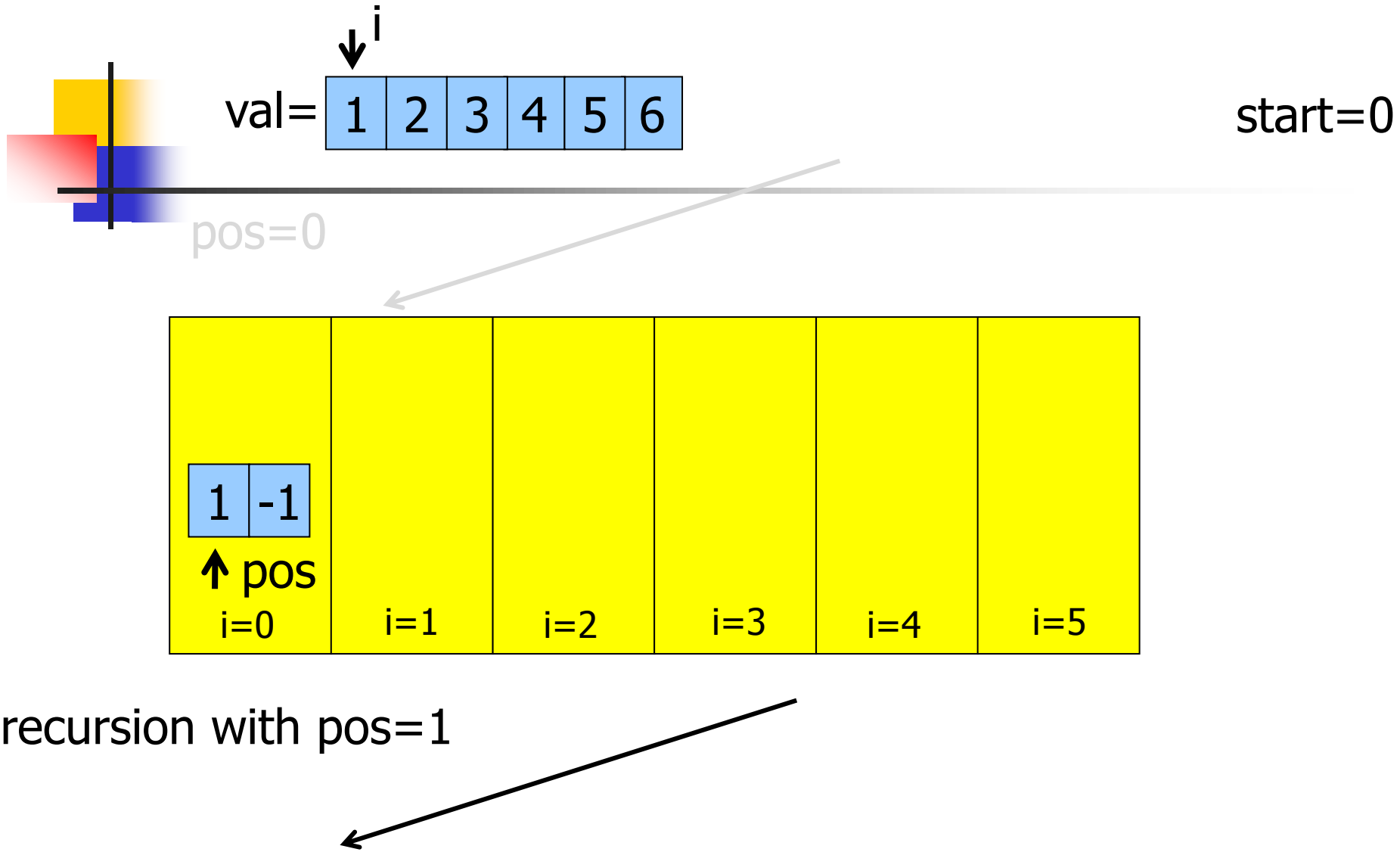
$C'_{6,3} = (6 + 2 - 1)!/2!(6-1)! = 21$

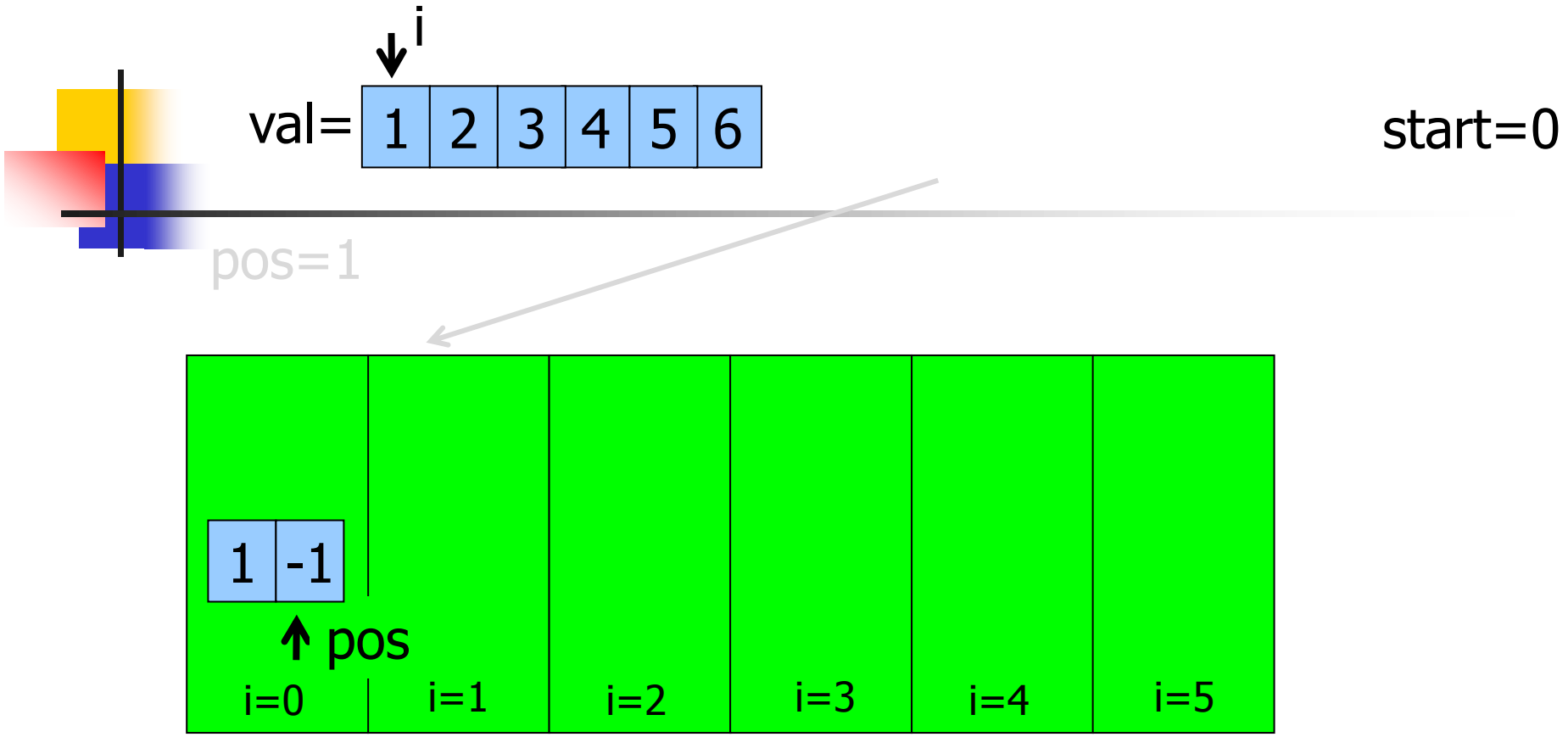
Soluzione



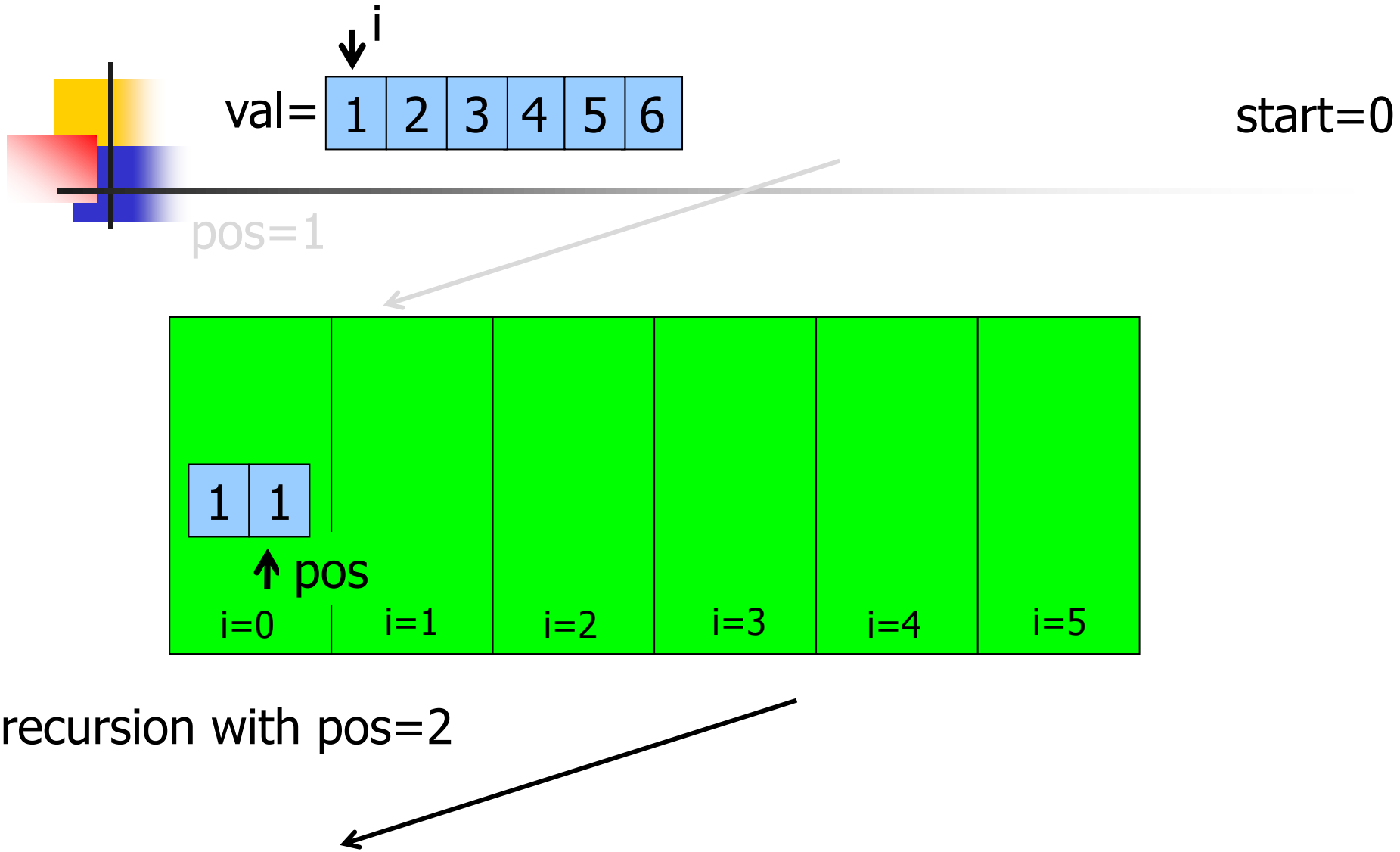


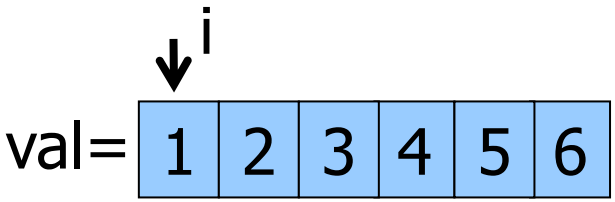
sol[pos] = val[i]


i
val=
1
2
3
4
5
6
start=0
pos=0
1
-1
pos
i=0
i=1
i=2
i=3
i=4
i=5
recursion with pos=1

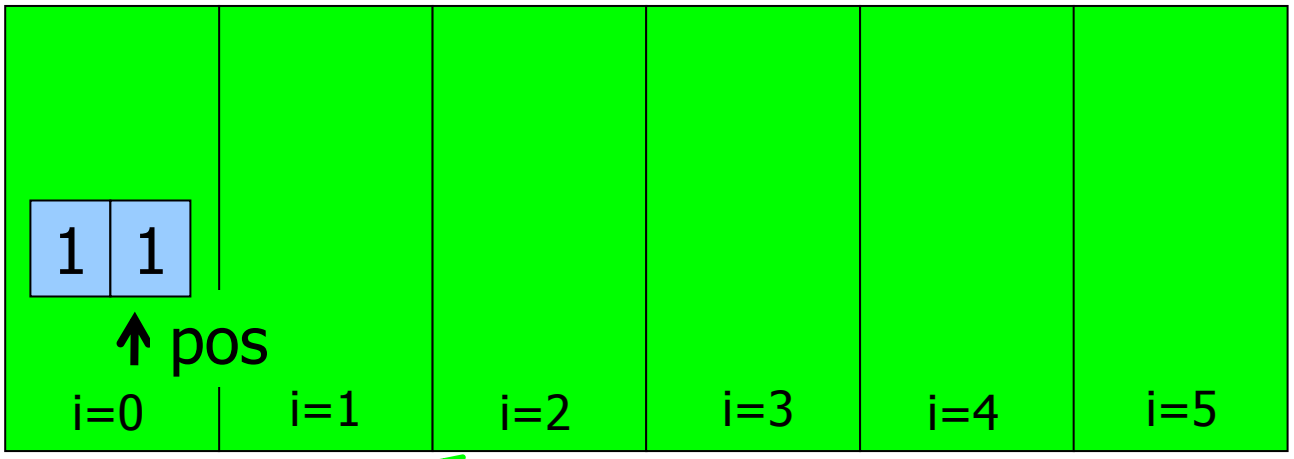


sol[pos] = val[i]


i
val=
1
2
3
4
5
6
start=0
pos=1
1
1
pos
i=0
i=1
i=2
i=3
i=4
i=5
recursion with pos=2

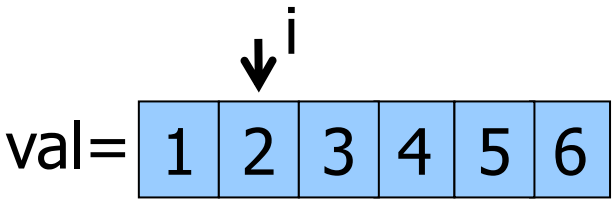


start=1

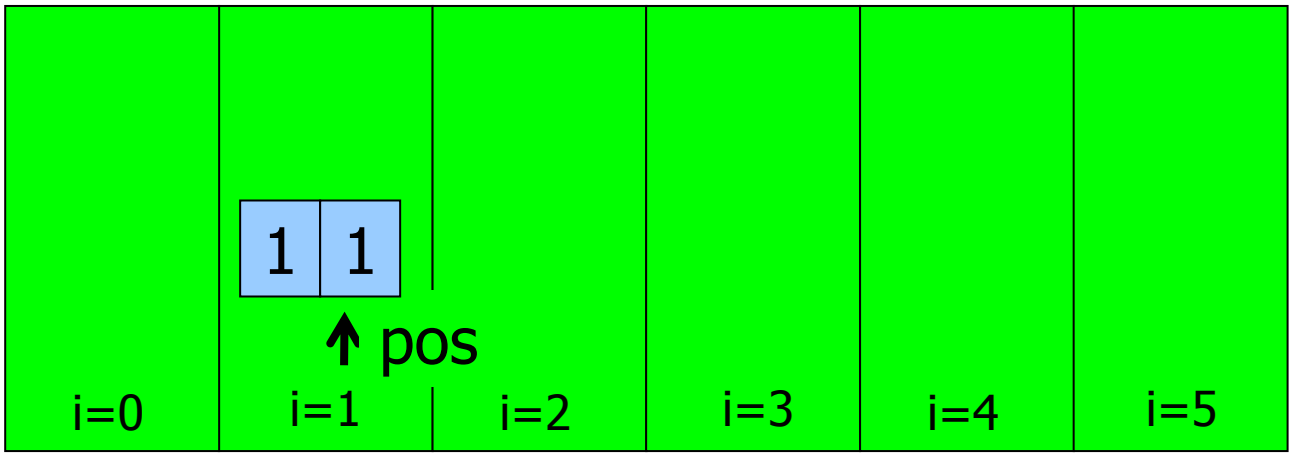


termination: display, update count
return and update start

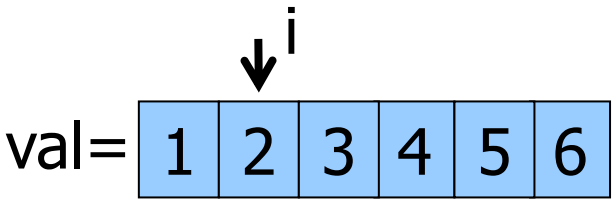
sol= 1 1



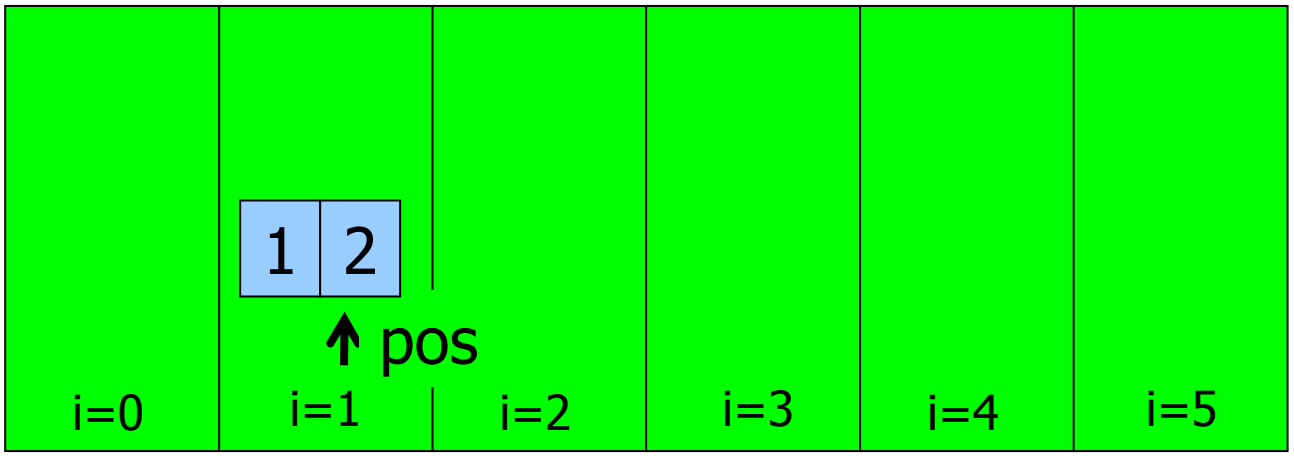
start=1



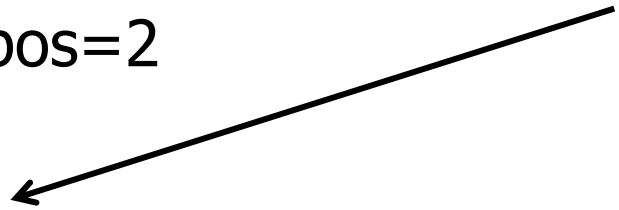
sol[pos] = val[i]

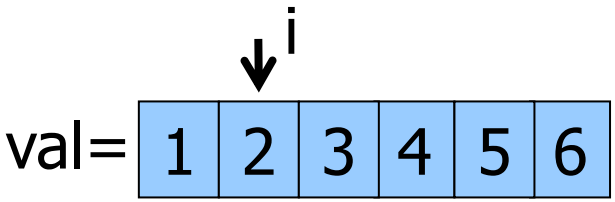


start=1



recursion with pos=2

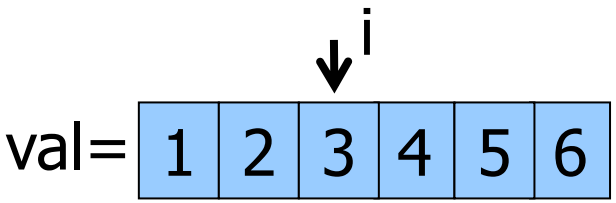




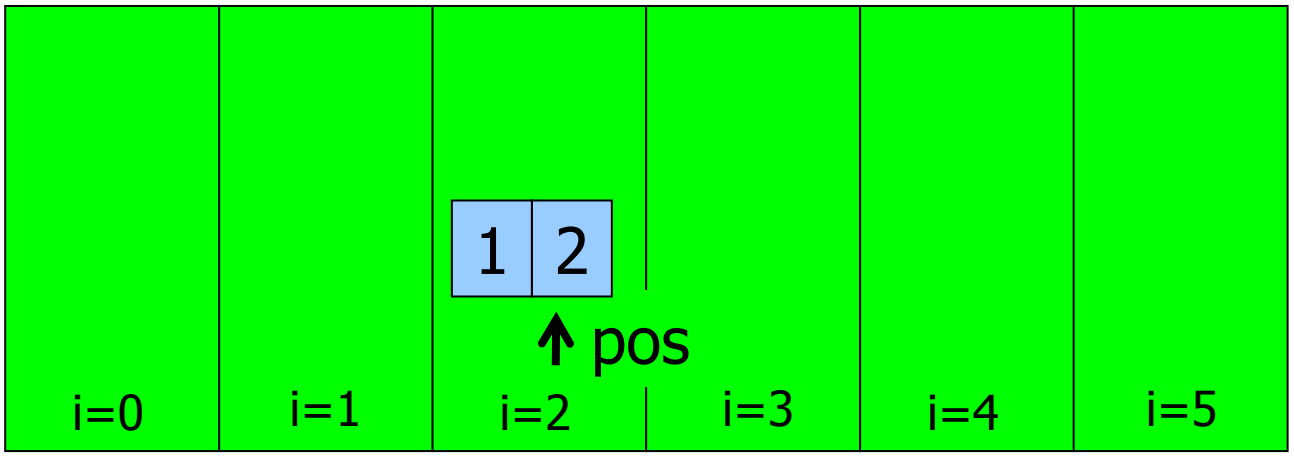
start=2

termination: display, update count
return and update start

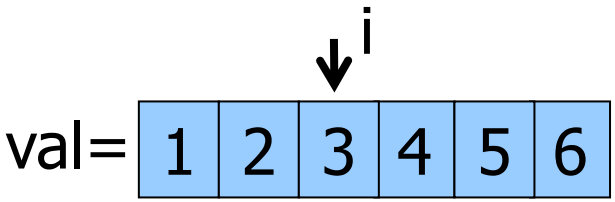
sol= 1 2



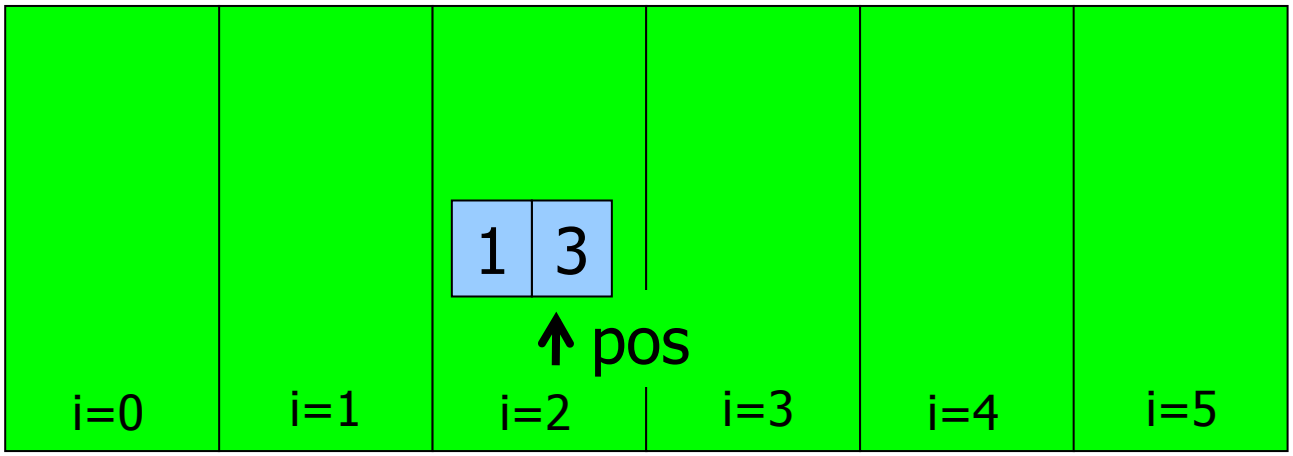
start=2



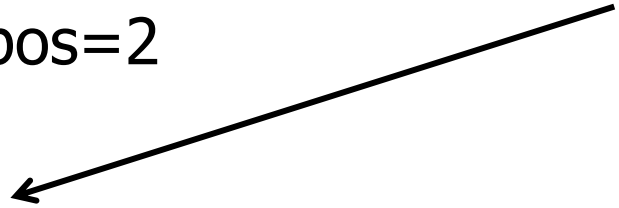
sol[pos] = val[i]

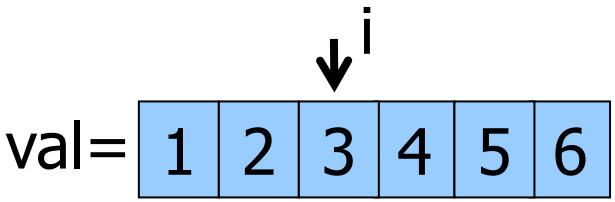


start=2

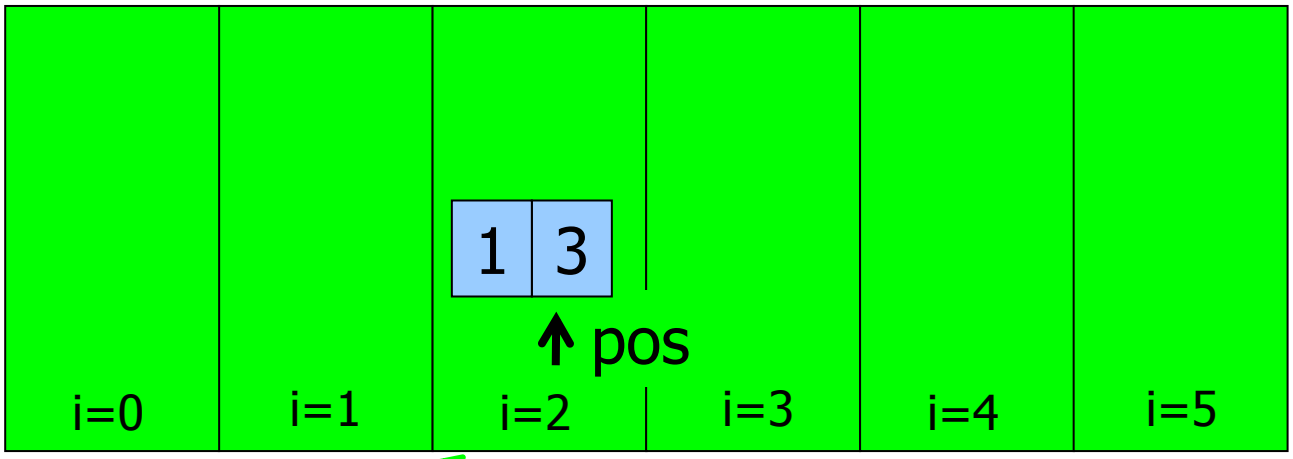


recursion with pos=2



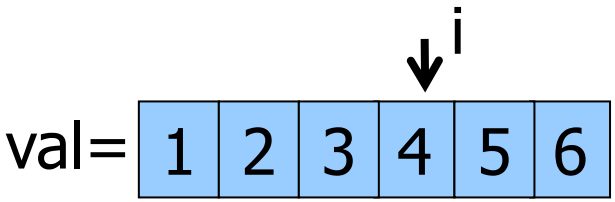


start=3

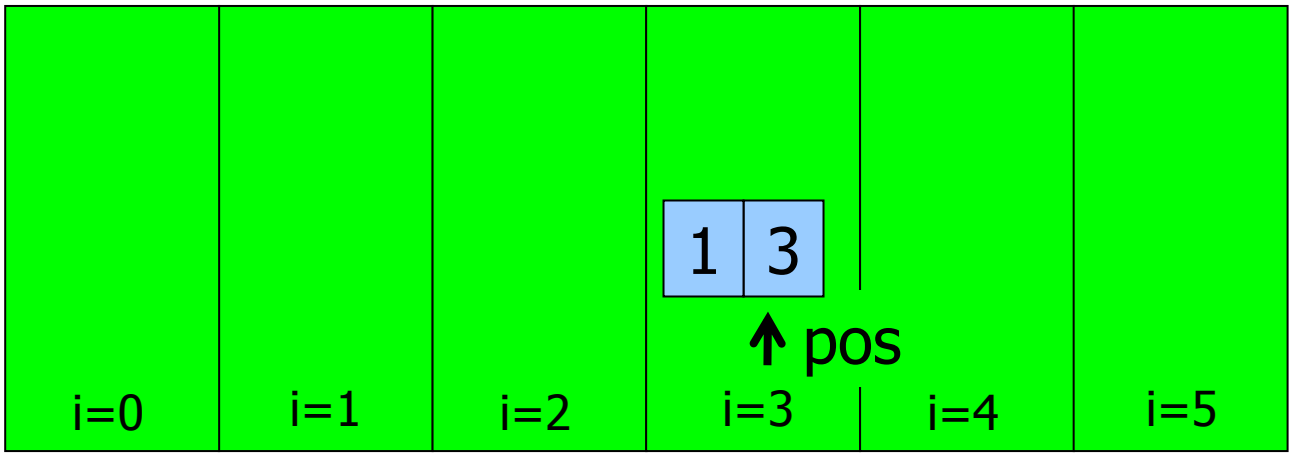


termination: display, update count
return and update start

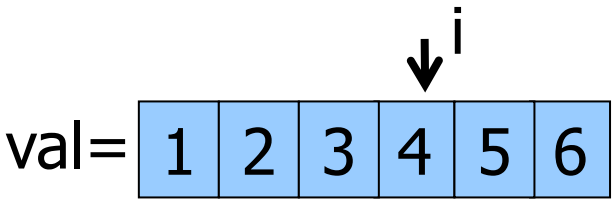
sol= 1 3



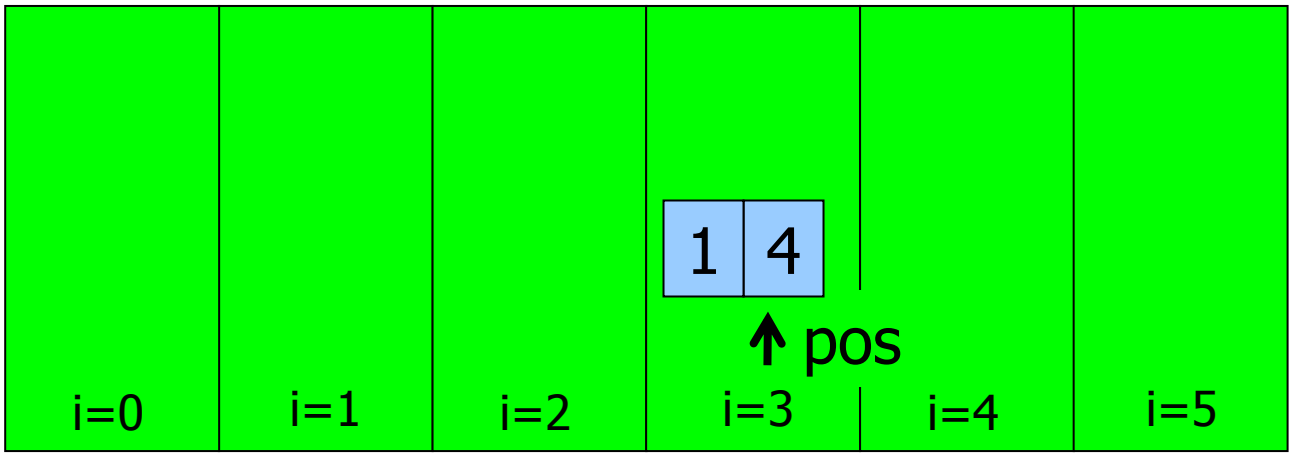
start=3



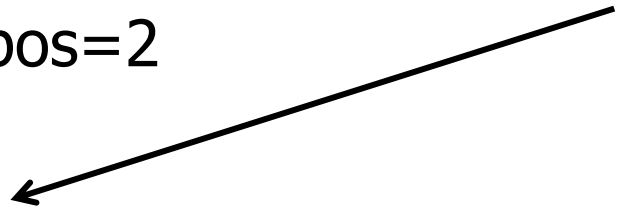
sol[pos] = val[i]

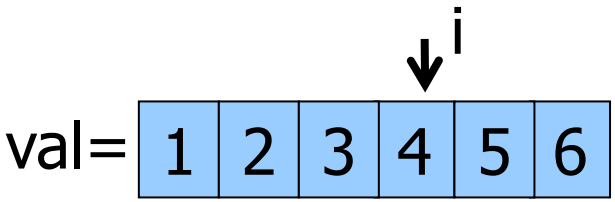


start=3

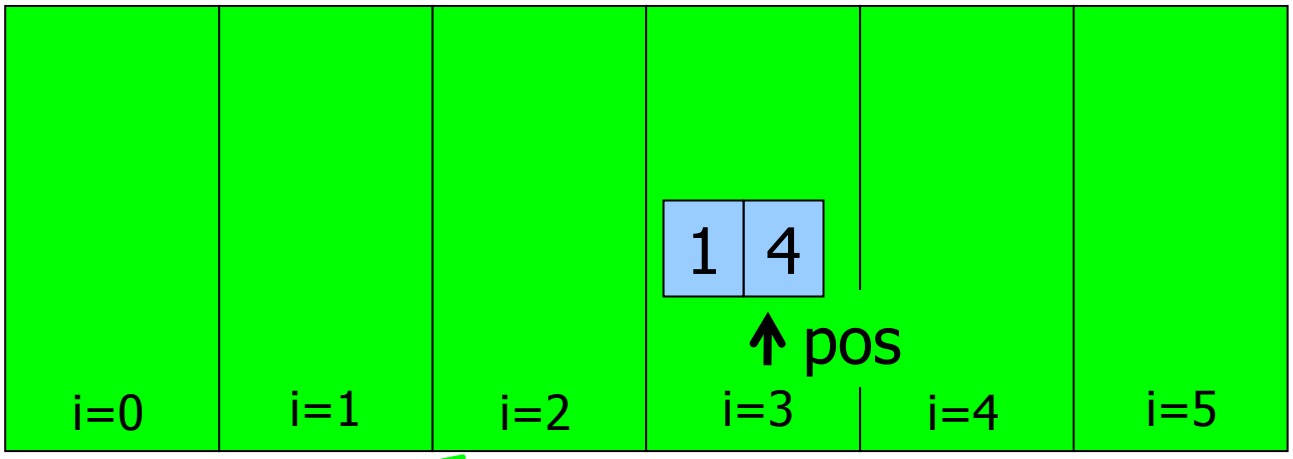


recursion with pos=2



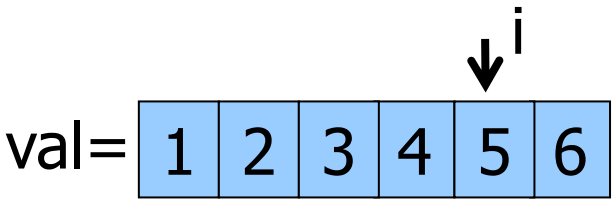


start=4

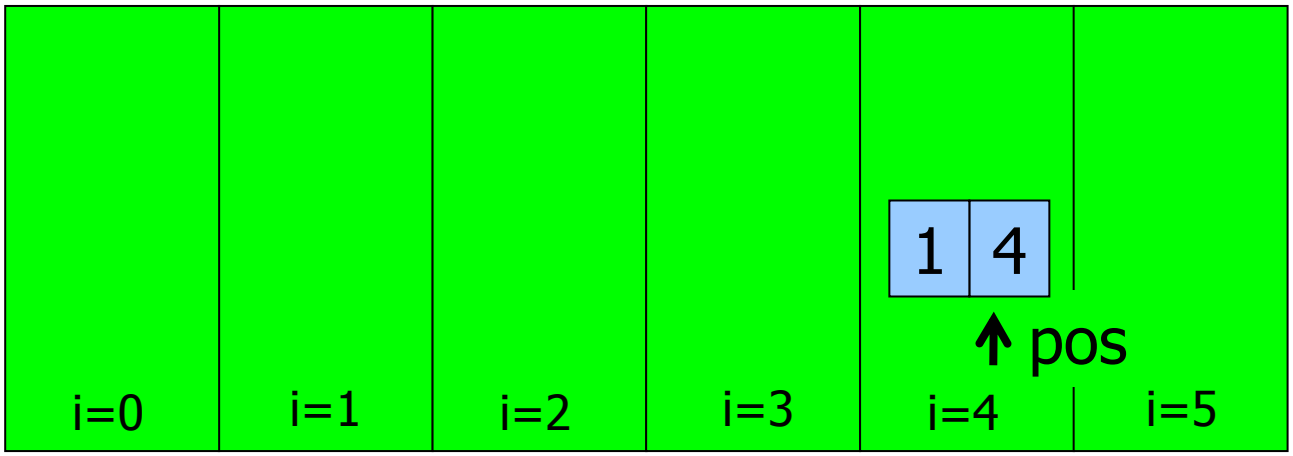


termination: display, update count
return and update start

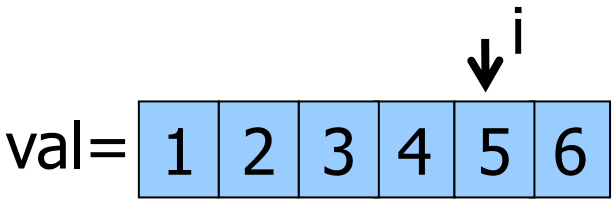
sol= 1 4



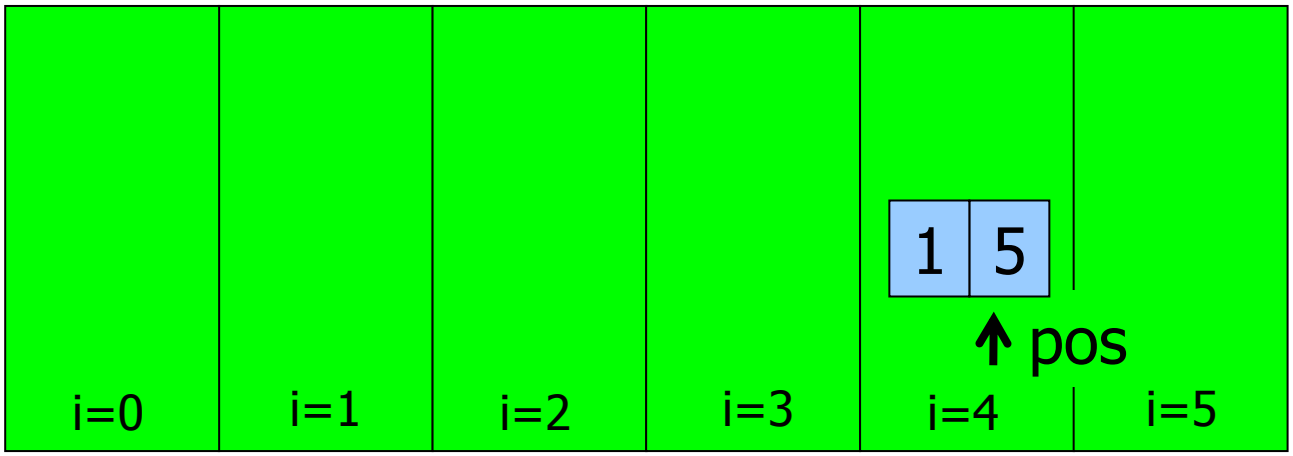
start=4



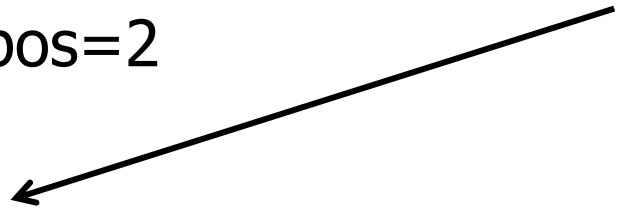
sol[pos] = val[i]

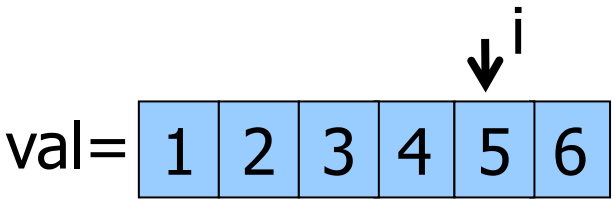


start=4

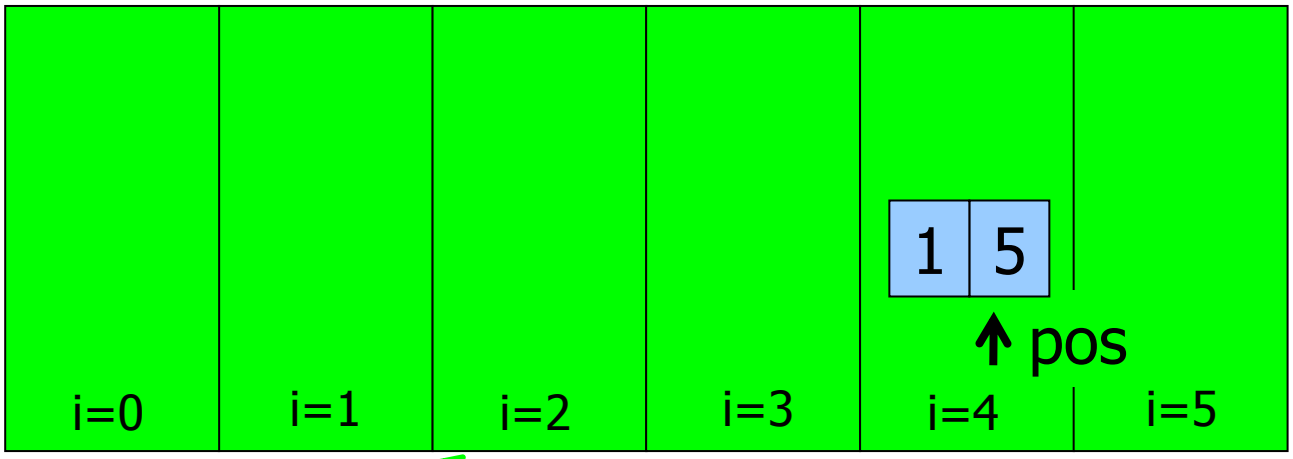


recursion with pos=2



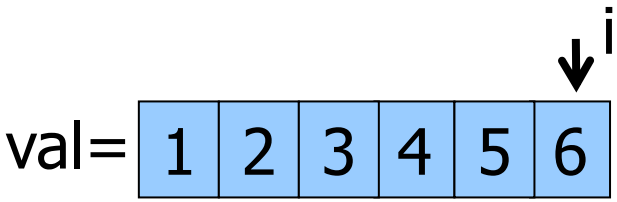


start=5

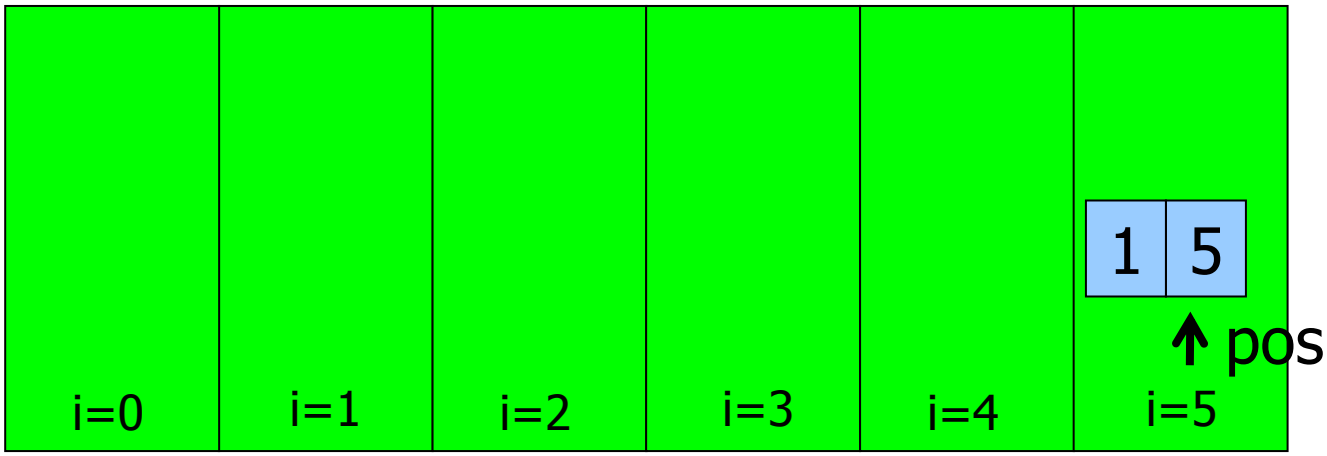


termination: display, update count
return and update start

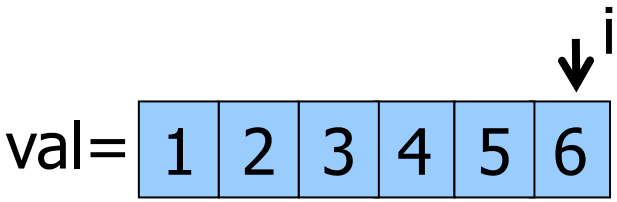
sol= 1 5



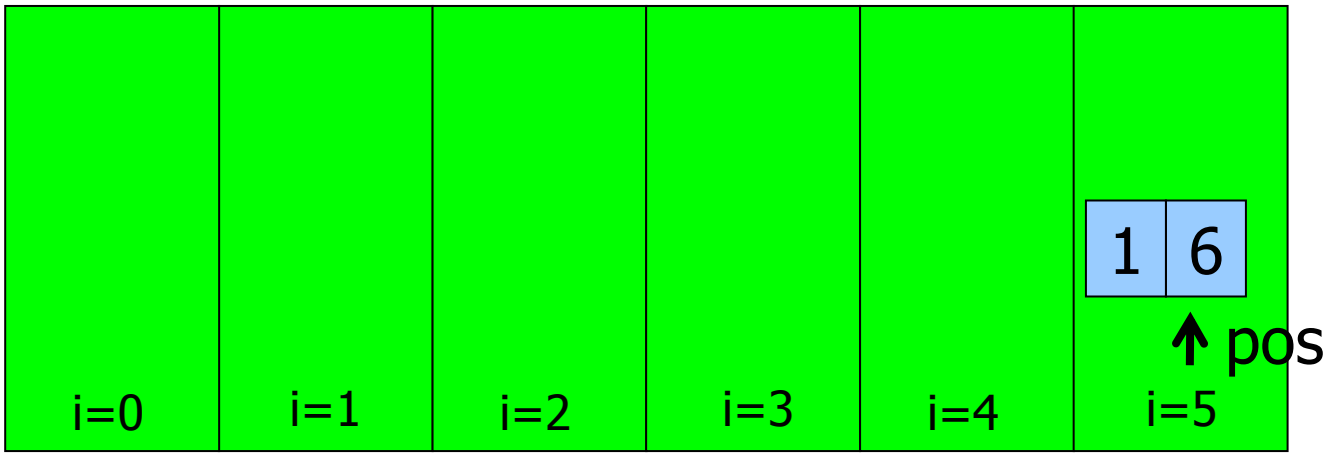
start=5



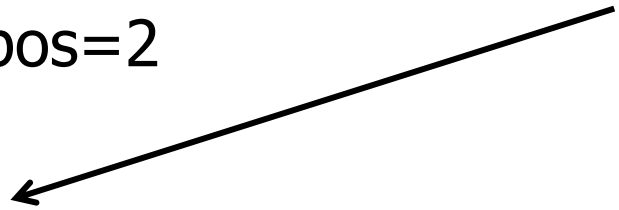
sol[pos] = val[i]

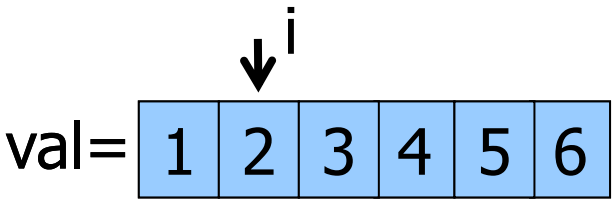


start=5

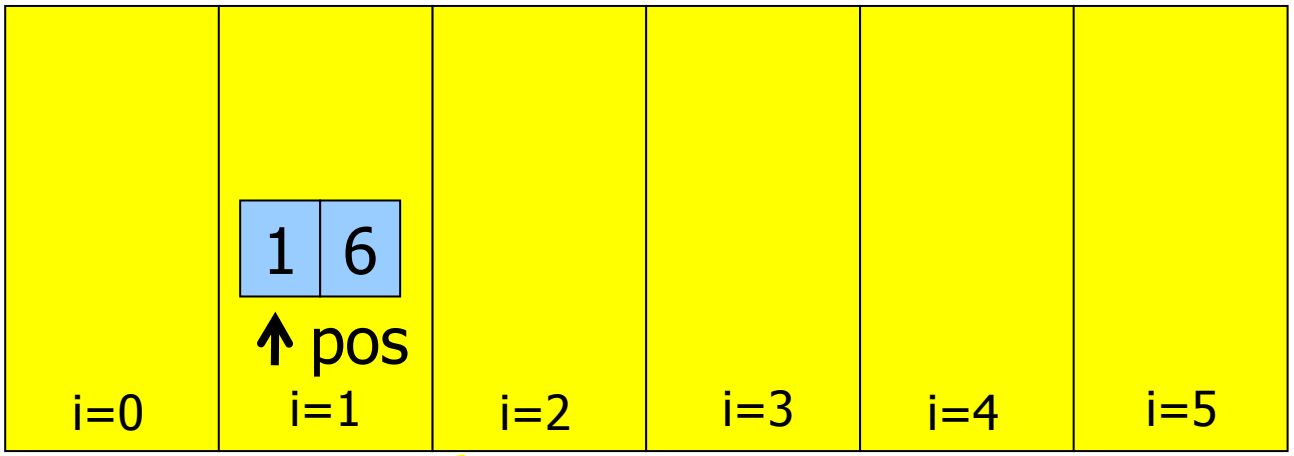


recursion with pos=2



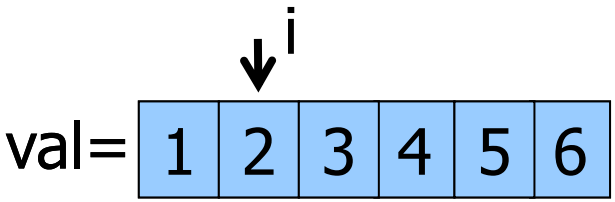


start=1

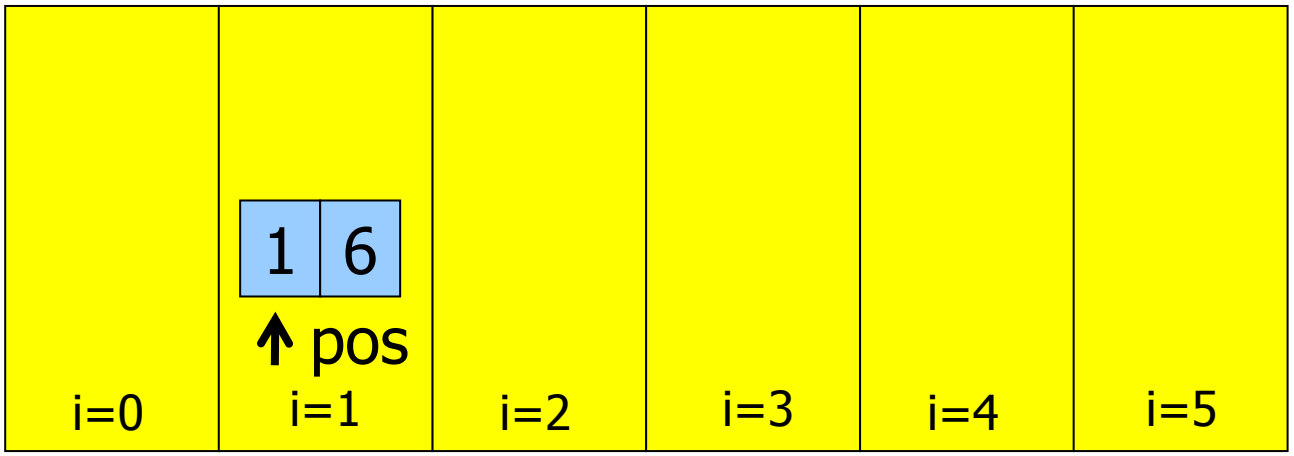


termination: display, update count
return and update start

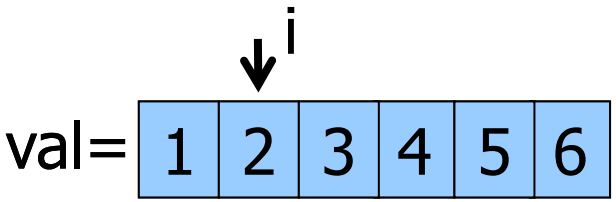
sol= 1 6



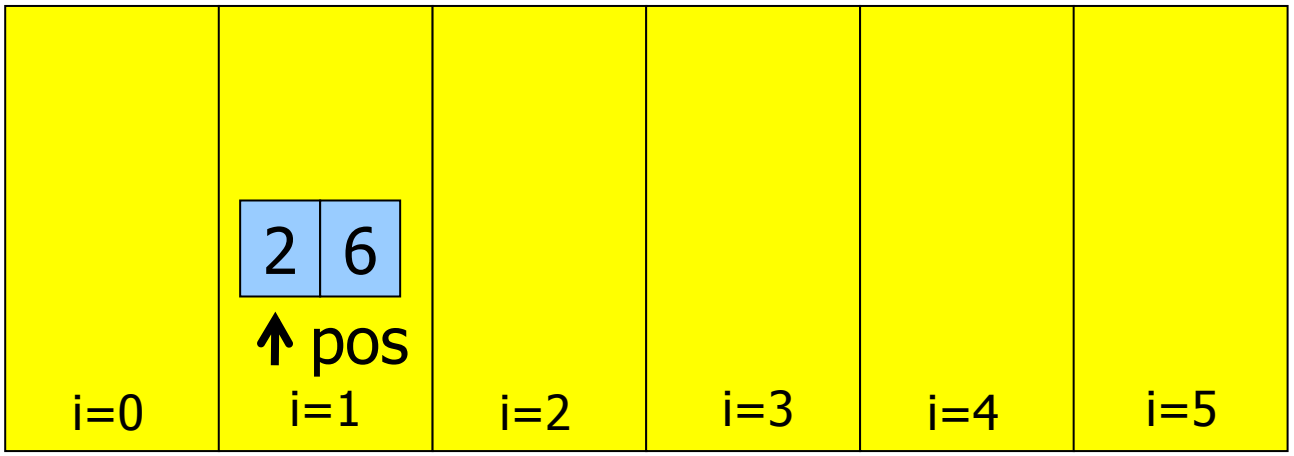
start=1



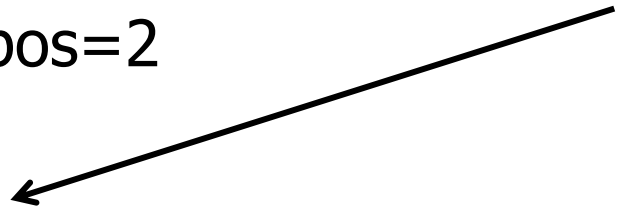
sol[pos] = val[i]



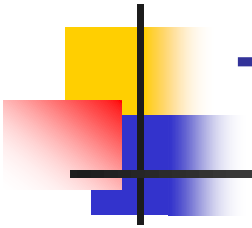
start=1



recursion with pos=2



etc. etc.



The Powerset

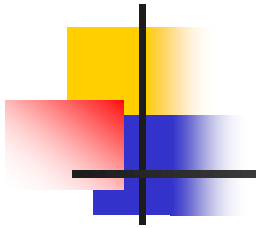
There are 3 models:

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

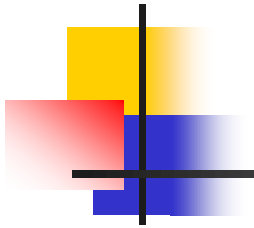
Divide and conquer

- 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

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



- 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.



termination: no more elements

```
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

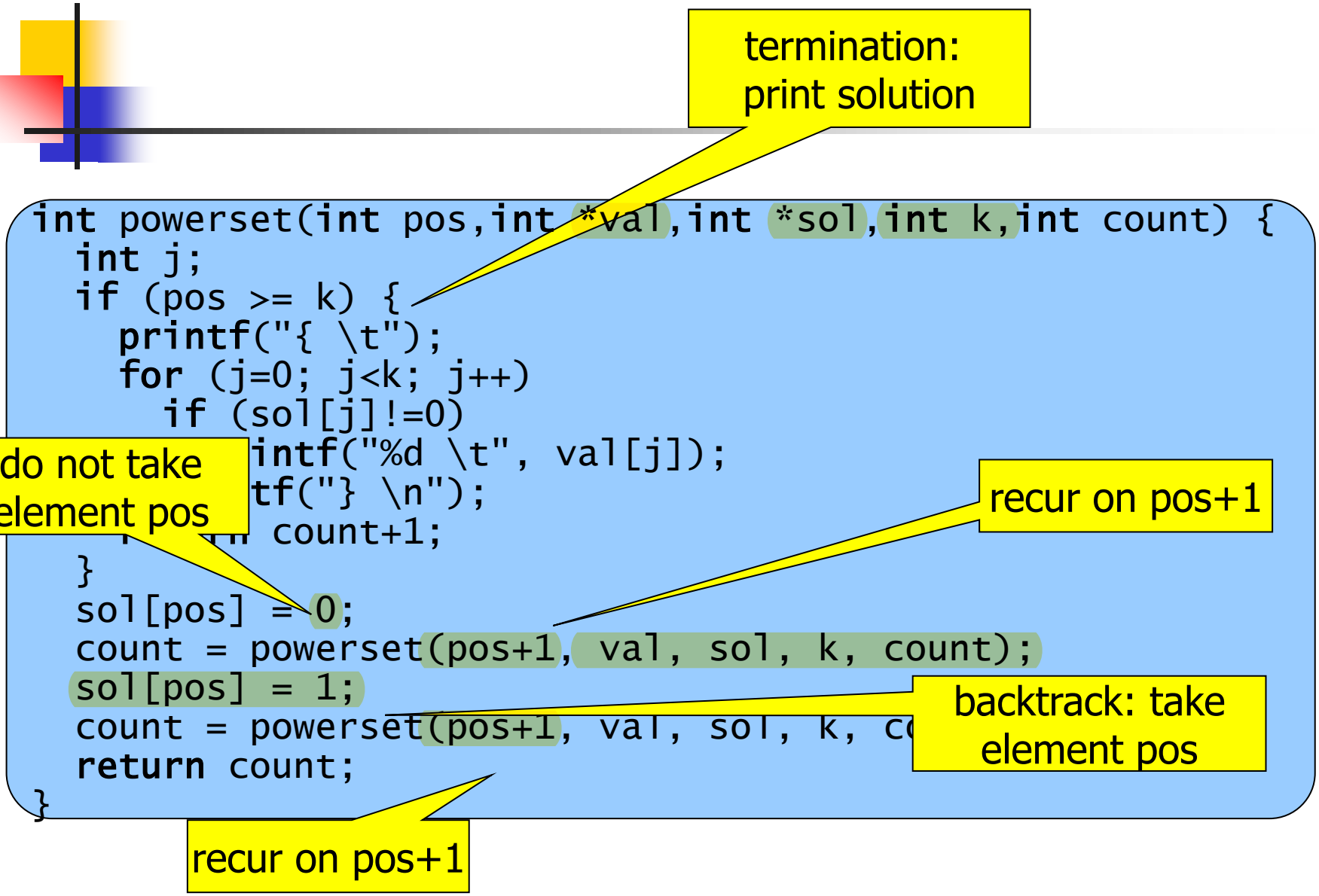
include element and recur

do not add and recur

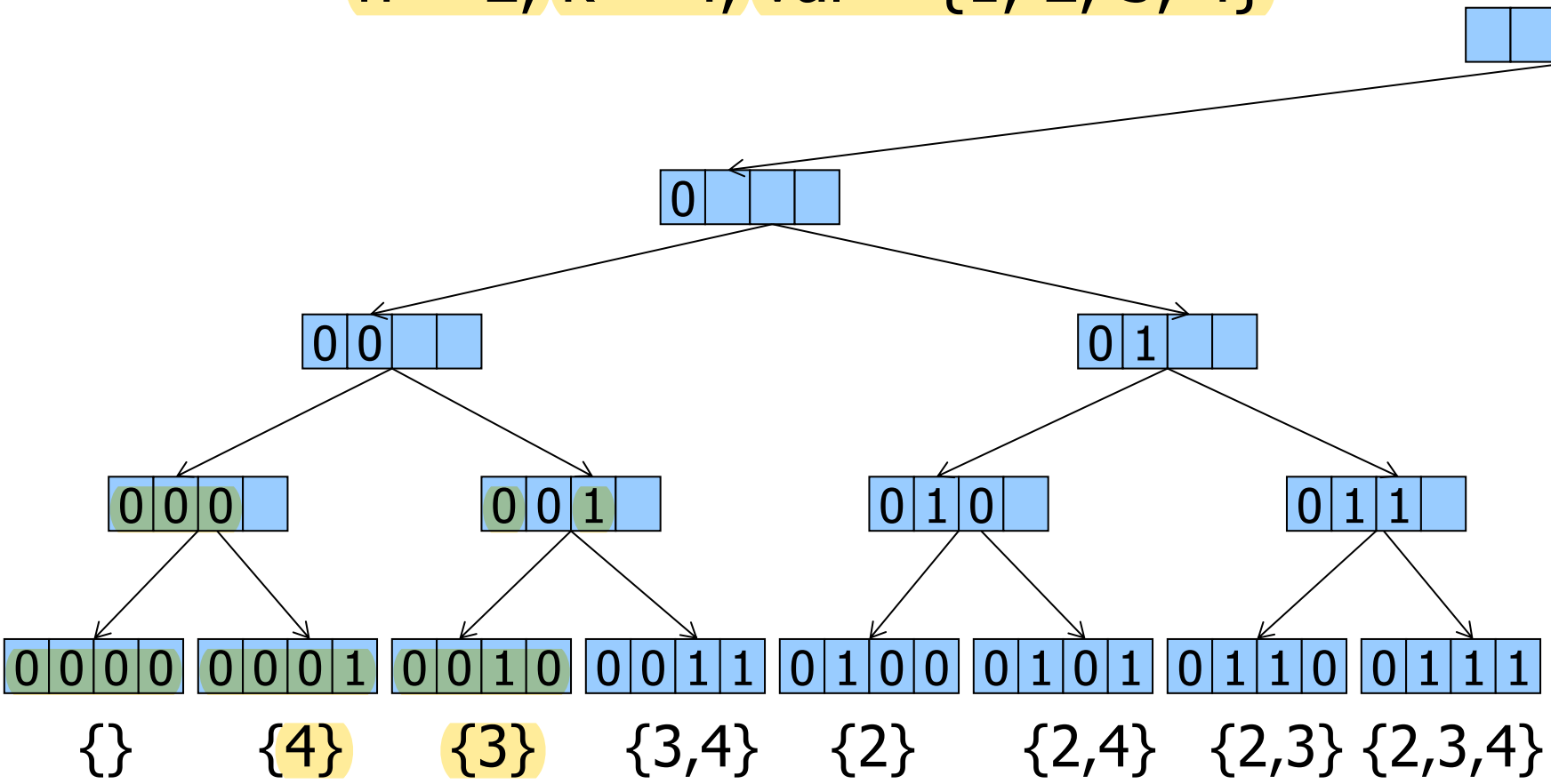
Arrangements with repetitions

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

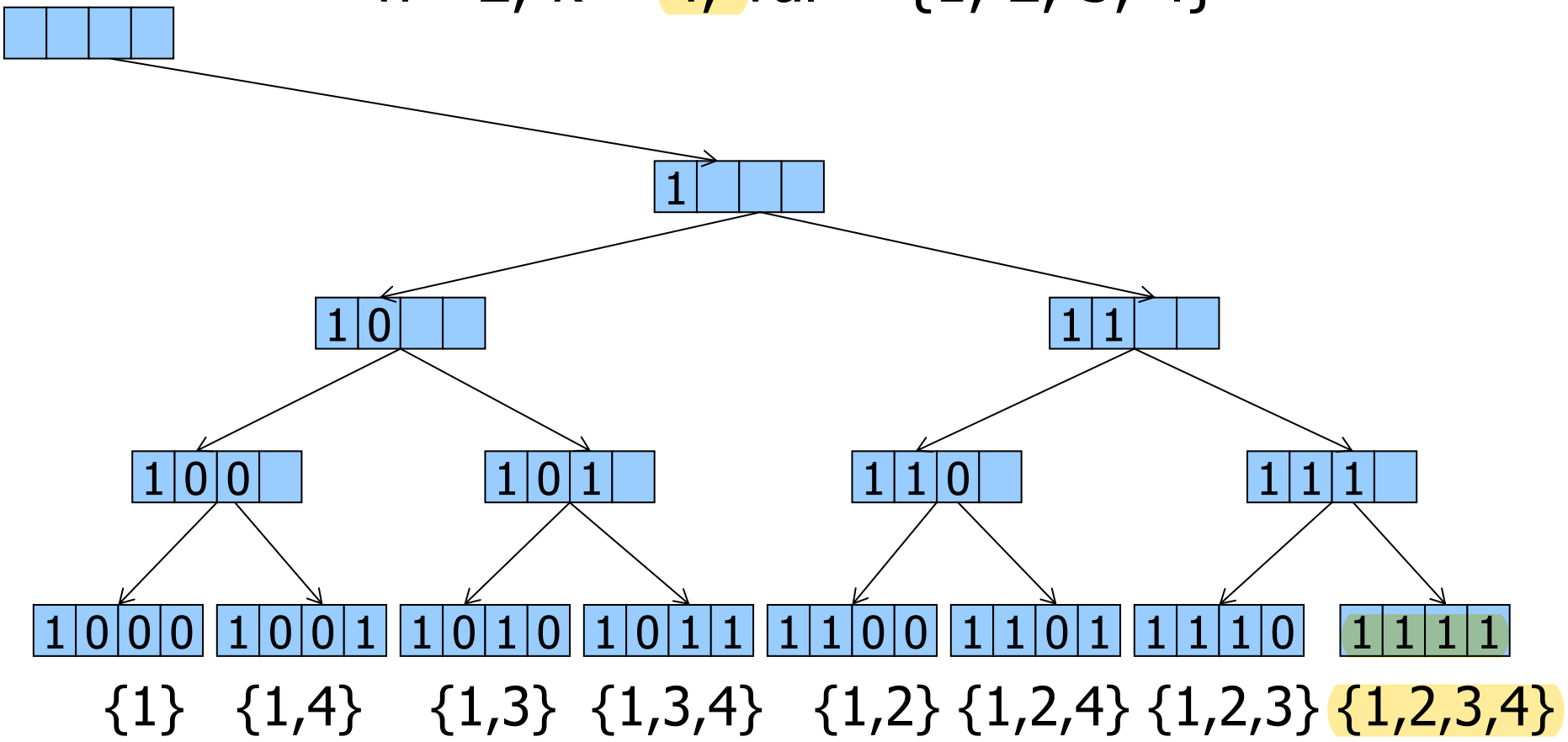
- the set of possible choices for each position in the array is {0, 1}, thus n = 2. The for loop is replaced by 2 explicit assignments
- `sol[pos]=0` if the pos-th object doesn't belong to the subset
- `sol[pos]=1` if the pos-th object belongs to the subset
- 0 and 1 may appear several times in the same solution.


termination:
print 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)
intf("%d \t", val[j]);
tf("} \n");
count+1;
}
sol[pos] = 0;
count = powerset(pos+1, val, sol, k, count);
sol[pos] = 1;
count = powerset(pos+1, val, sol, k, c
return count;
}
do not take
element pos
recur on pos+1
backtrack: take
element pos
recur on pos+1

n = 2, k = 4, val = {1, 2, 3, 4}



n =2, k = 4, val = {1, 2, 3, 4}

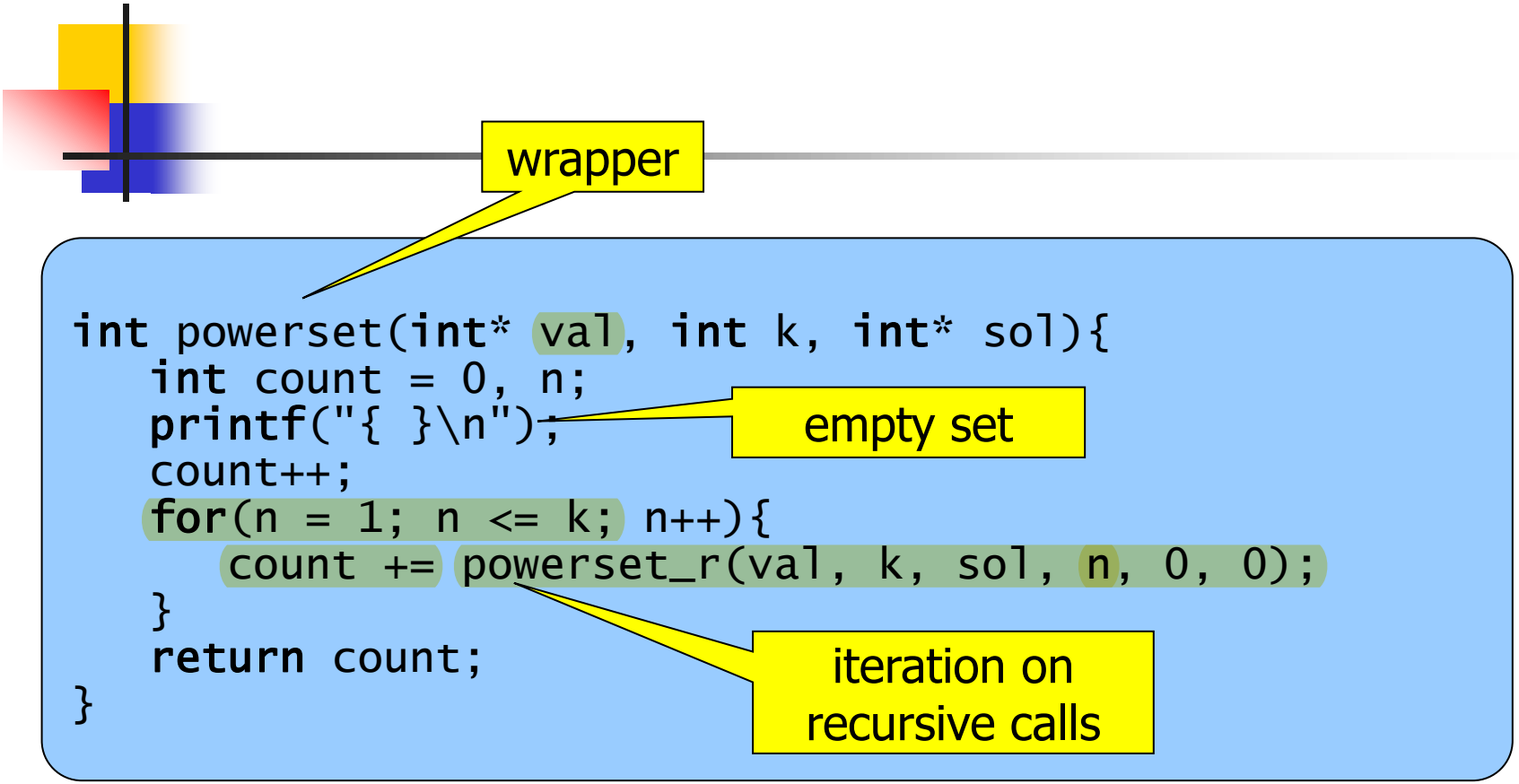


Simple combinations

- 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

$$\wp(S) = \{ \varnothing \} \cup U_{n=1}^{k} \binom{k}{n}$$

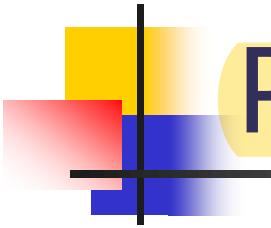
- the wrapper function takes care of the union of empty set (not generated as a combination) and of iterating the recursive call to the function computing combinations.


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

terminal case: predefined number of elements reached

```
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;
}
```

for all elements from start onwards

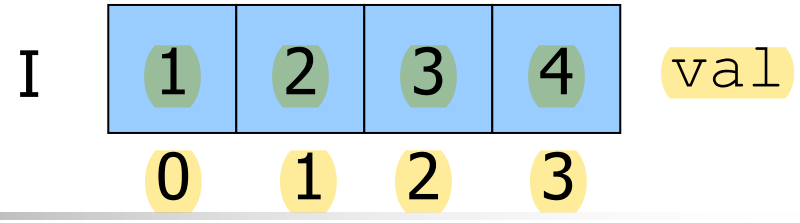


Partitions of a set S

Representing partitions:

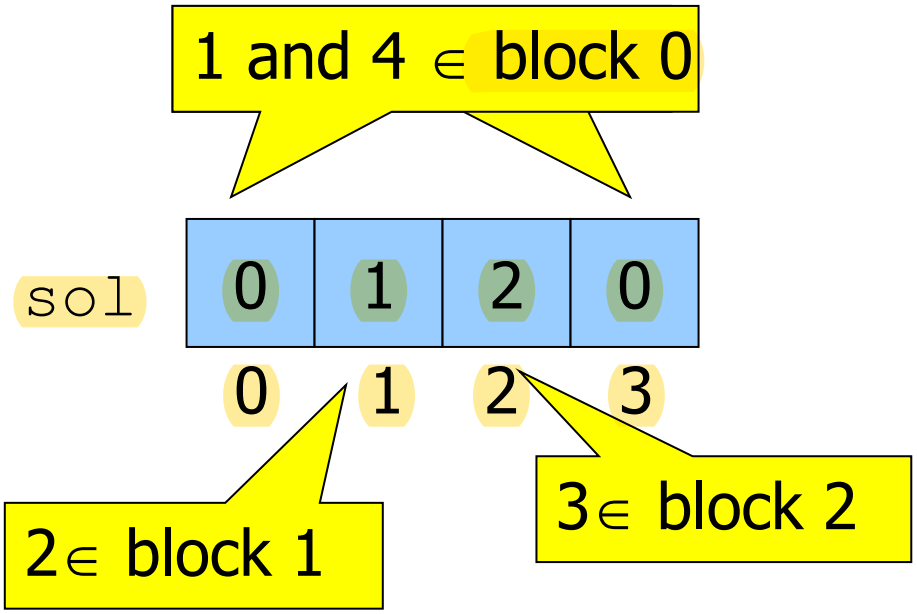
- given the element, identify the unique block it belongs to
- given the block, list the elements that belong to it.

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:



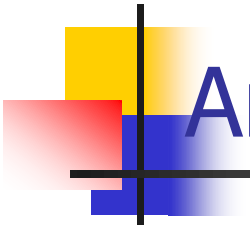
Problems

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.

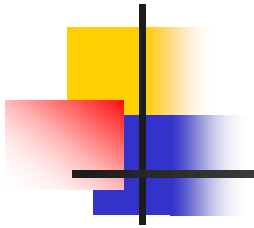
Arrangements with repetitions

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 `k`, i.e. the number of blocks
- `sol[i]` contains the block from 0 to `k-1` to which the current object of index `i` belongs



- 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;  disp_ripet(pos+1, val, sol, n, k);
  }
}
```

block occurrence array

occurrence computation

occurrence control

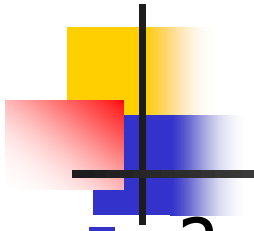
discarded solution

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



- 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.

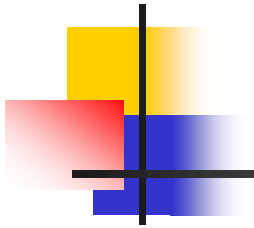
```
val = malloc(k*sizeof(int));
sol = malloc(k*sizeof(int));
```

```
void SP_rec(int n, int m, int pos, int *sol, int *val) {
  int i, j;
  if (pos >= k) {
    printf("partitione 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



Computing all the partitions of 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

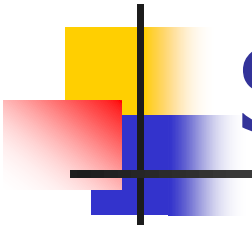
filter

recursion on objects

recursion on objects and blocks

Exhaustive search of the solutions space: single solution

Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino

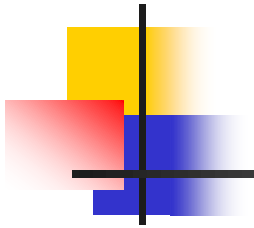


Single solution

Recursion is particularly suited to listing all the solutions and this is mandatory in optimization problems.

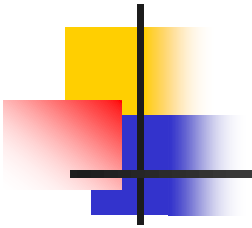
If one solution is enough, all recursive calls must terminate as soon as it is found.
Reminder: when a recursive call terminates, control returns to the calling function.

It is incorrect to imagine recursion that returns to the initial call, bypassing the chain of recursive calls.



Solution:

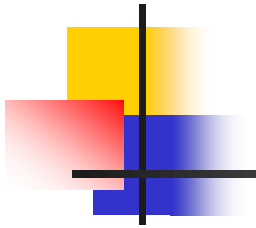
- flag:
  - global variable (not suggested)
  - parameter passed by reference
- recursive function that returns a success or failure value that is tested.



Use of a flag as a parameter by reference:

- define a `stop` flag (initialized to 0), pass the pointer to the flag to the recursive function:
  - in case of termination with success, `stop` is set to 1
  - the loop on choices includes in its condition the check if `stop==0`

```
/* main */
.....
int stop = 0;
.....
rec_funct(.....   , &stop);

void rec_funct(....., int *stop_ptr) {
  .....
  if (termination condition) {
    .....
    (*stop_ptr) = 1;
    return;
  }
  for (i=0; condition on i && (*stop_ptr)==0; i++) {
    .....
    rec_funct(....., stop_ptr);
    .....
  }
  return;
}
```



Recursive function that returns an integer value meaning success/failure (version without pruning):

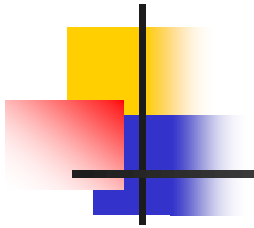
- in the terminal case, if the acceptance condition is satisfied, return 1, else return 0
- in the choice loop:
  - choose
  - check the result of the recursive call: if it is success, return 1
- choice loop finished: return 0.

in main

```
if (rec_funct(.....  )==0)
  printf("solution not found\n");

int rec_funct(.....) {
 if (termination condition)
   if (acceptance condition) {
      .....
      return 1;
   else
      return 0;
   }
 for (loop on choices) {
     choice;
     if (rec_funct(.....))
       return 1;
     .....
  }
  return 0;
}
```

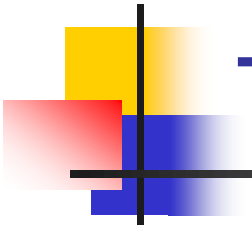
Optimization problems

Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino



While exhaustively exploring the solutions' space, keep track of the best solution found so fare and update it at each step if necessary.

It is required to generate all the solutions.

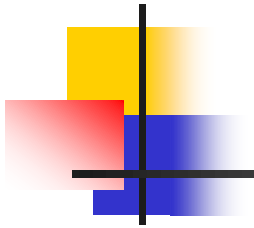


The bank account

Input: array of integers of known length n. Each integer represents a distinct operation on a nak account:

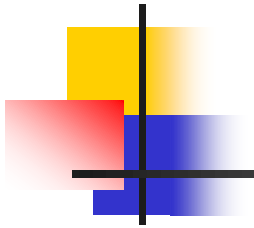
- >0: deposit
- <=0: withdrawal.

Given an ordering for the operations, the current balance on the account is obtained by algebraically adding together the current amount and the previous balance (initially set to 0).



For each ordering of the operations there will be a current maximum balance and a current minimum balance. The final balance will obviously be the same no matter the ordering.

Find the ordering that minimizes the difference between maximum current balance and minimum current balance.



Given n=10 and val={1,-2,3,14,-5,16,7,8,-9,120}, with the ordering

- {3,1,14,-2,-5,16,7,8,-9,120} the maximum balance is 153, the minimum balance is 3, the difference is 150
- {120,1,3,-2,14,-5,16,-9,7,8} the maximum balance is 153, the minimum balance is 120, the difference is 33 and it is a solution.

Model:

- simple permutations to enumerate orderings.

Algorithm:

- recursive algorithm to compute simple permutations
- when in the terminal case:
  - compute maximum and minimum balance and current difference
  - compare the current difference with the minimum one found so far
  - possibly update the solution.

Assumption:

- There is un upper bound on the maximum minimum difference (= INT_MAX).

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

global variable

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
int min_diff = INT_MAX;
void perm(int pos,int *val,int *sol,int *mark,int *fin,int n);
void check(int *sol, int *fin, int n);
int main(void) {
  int i, n, k, tot, *val, *sol, *mark, *fin;
  printf("Insert n: "); scanf("%d", &n);
  val=malloc(n*sizeof(int));
  sol=malloc(n*sizeof(int));
  mark=malloc(n*sizeof(int));
  fin=malloc(n*sizeof(int));
  for (i=0; i < n; i++)
    { sol[i]= -1; mark[i]= 0;}
  // READ VALUES IN val
  perm(0, val, sol, mark, fin, n);
  // PRINT RESULT FROM fin
  return 0;
}
```

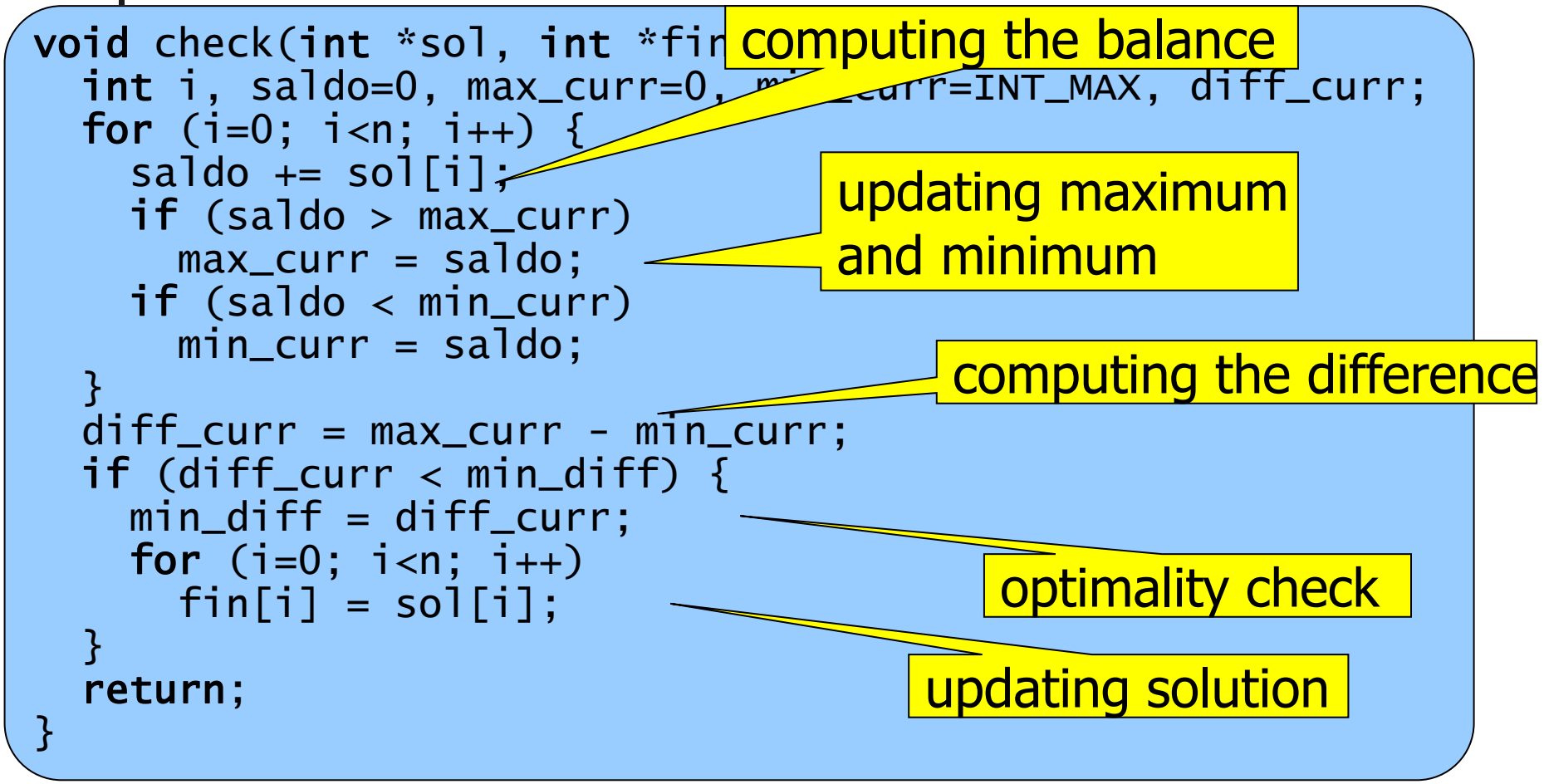
number of permutations not needed

```
void perm(int pos,int *val,int *sol,int *mark,int *fin,int n) {
  int i;
  if (pos >= n) {
    check(sol, fin, n);
    return;
  }
  for (i=0; i<n; i++)
    if (mark[i] == 0) {
      mark[i] = 1;
      sol[pos] = val[i];
      perm(pos+1, val, sol, mark, fin, n);
      mark[i] = 0;
    }
  return;
}
```

termination condition

solution optimality check

generating permutations

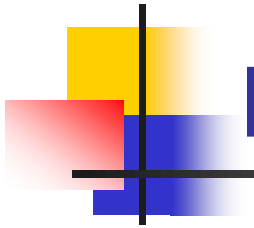

void check(int *sol, int *fin
int i, saldo=0, max_curr=0, m curr=INT_MAX, diff_curr;
for (i=0; i<n; i++) {
saldo += sol[i];
if (saldo > max_curr)
max_curr = saldo;
if (saldo < min_curr)
min_curr = saldo;
}
diff_curr = max_curr - min_curr;
if (diff_curr < min_diff) {
min_diff = diff_curr;
for (i=0; i<n; i++)
fin[i] = sol[i];
}
return;
}
computing the balance
updating maximum and minimum
computing the difference
optimality check
updating solution

The knapsack (discrete)

Given a set of N objects each having w_j and value v_j and given a maximum weightcap, find the subset S of objects such that:

- $\sum_{j \in S} w_j x_j \leq$ cap
- $\sum_{j \in S} v_j x_j =$ MAX
- $x_j \in \{0,1\}$

Each object is either taken ($x_j = 1$) or left ($x_j = 0$). There is only one instance of each object.



Example

N= 4

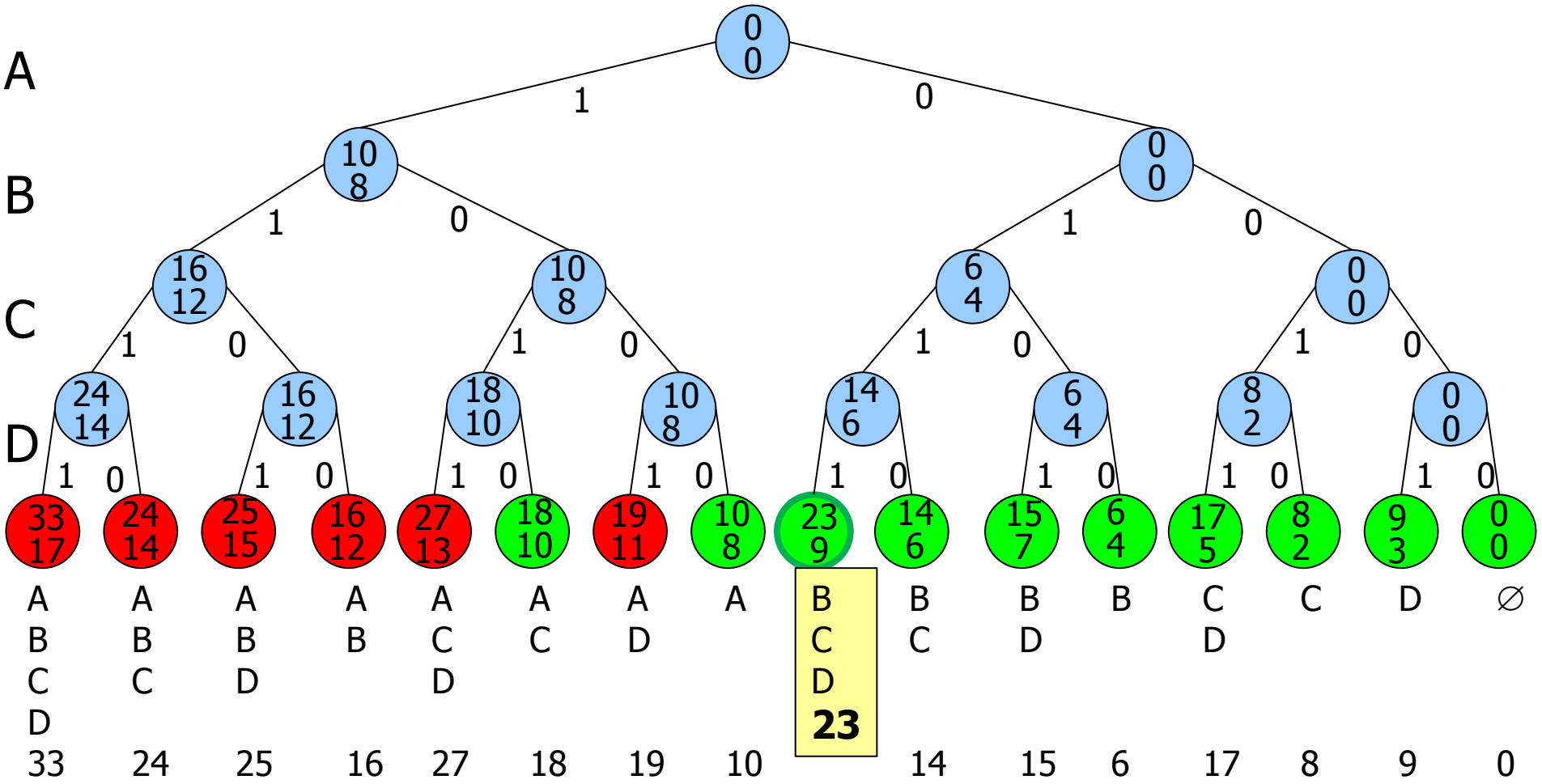
cap = 10

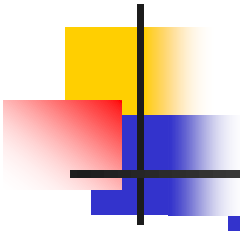
		A	B	C	D
Value	v_i	10	6	8	9
Weight	w_i	8	4	2	3

Solution:

set {B, C, D} with maximum value 23

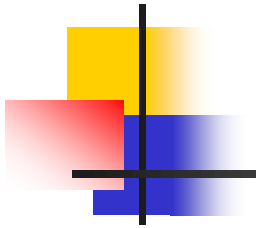
Model: arrangements with repetition (powerset)





Data structures

- Same data stuctures used for arrangements with repetitions +:
- integer variables:
  - `cap` knapsack capacity (max weight)
  - `curr_value` current value
  - `curr_cap` currently used capacity (weight) of knapsack
  - `bv` current best value
- array of integers `best_sol` for current best solution.

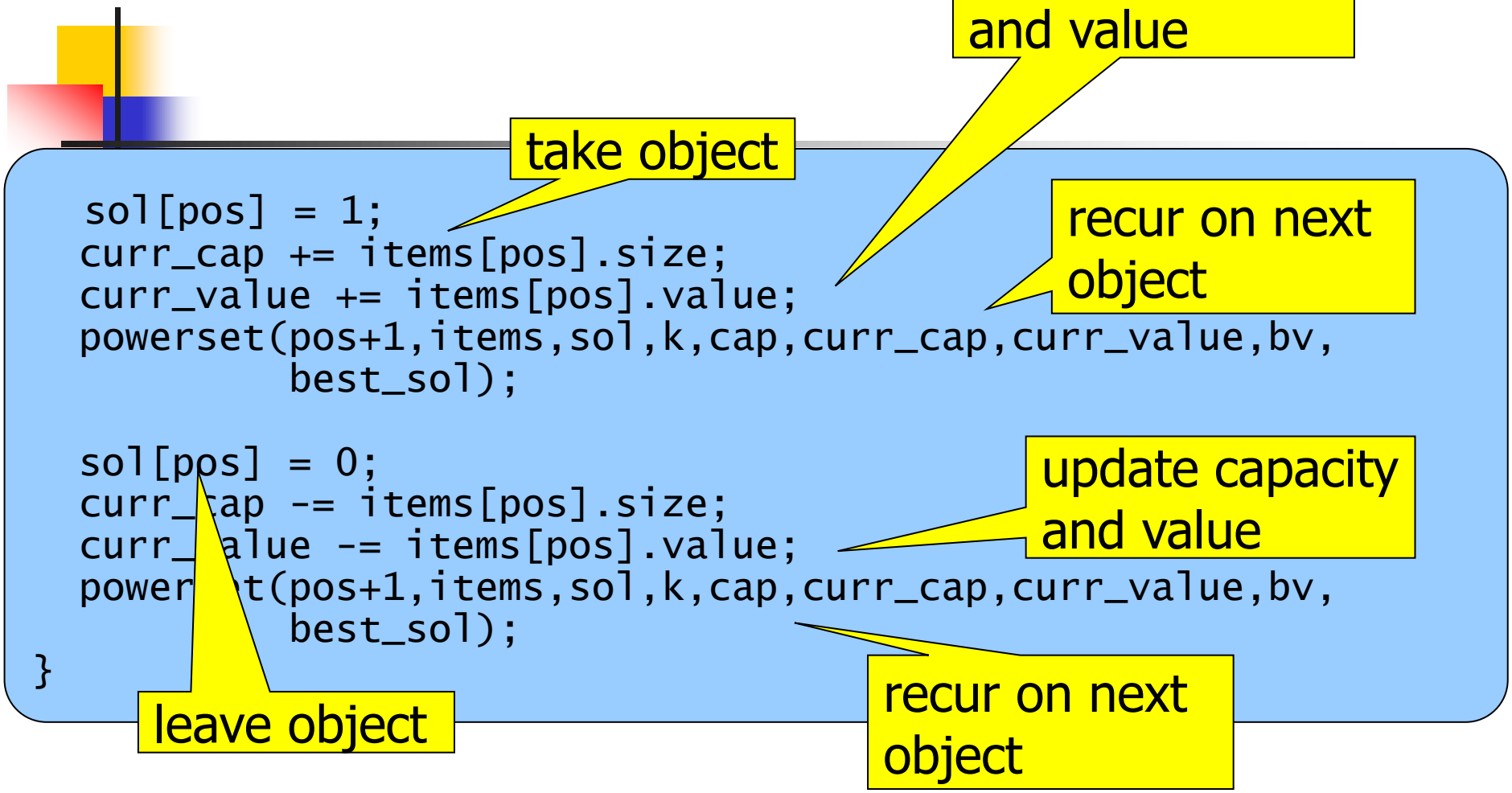


```
void powerset(int pos,Item *items,int *sol,int k, int cap,
    int curr_cap,int curr_value,int *bv, int *best_sol) {

  int j;

  if (pos >= k) {
    if (curr_cap <= cap) {
      if (curr_value > *bv) {
        for (j=0; j<k; j++)
          best_sol[j] = sol[j];
        *bv = curr_value;
      }
    }
    return;
  }
```

termination condition

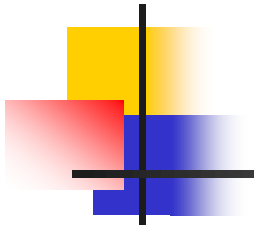
acceptance check

optimality check

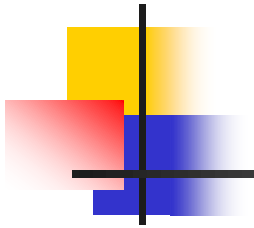

update capacity and value
take object
recur on next object
sol[pos] = 1;
curr_cap += items[pos].size;
curr_value += items[pos].value;
powerset(pos+1,items,sol,k,cap,curr_cap,curr_value,bv,
best_sol);
sol[pos] = 0;
curr_cap -= items[pos].size;
curr_value -= items[pos].value;
powerset(pos+1,items,sol,k,cap,curr_cap,curr_value,bv,
best_sol);
}
update capacity and value
leave object
recur on next object

Space pruning

Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino

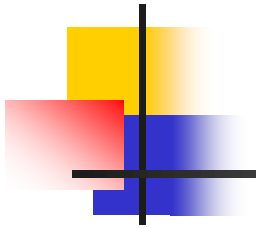


- Solution acceptance criterion expressed in terms of constraints
- Constraint evaluation:
  - directly in terminal cases, without using a specific data structure
  - at each recursive call using a dynamically updated data structure
- Very rapid growth of the solution space $\Rightarrow$ inability to apply enumeration



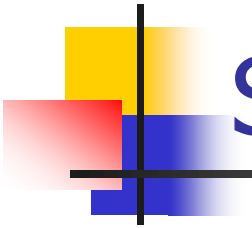
Pruning:

- search space reduction
- no loss in terms of completeness
- a priori cancellation of recursion tree branches that can't lead to valid/optimal solutions.



Constraints allow to:

- a priori rule out paths that lead to non acceptable solutions
- anticipate the acceptance test: instead of performing in the terminal case, test before recursive descent.



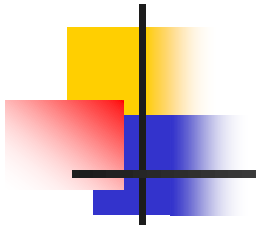
Sum of subsets

Given a set S of distinct positive integers and and integer X, find all subsets Y of S such that the sum of the elements in Y equals X.

Example: S = {2, 1, 6, 4} X = 7

Solution:

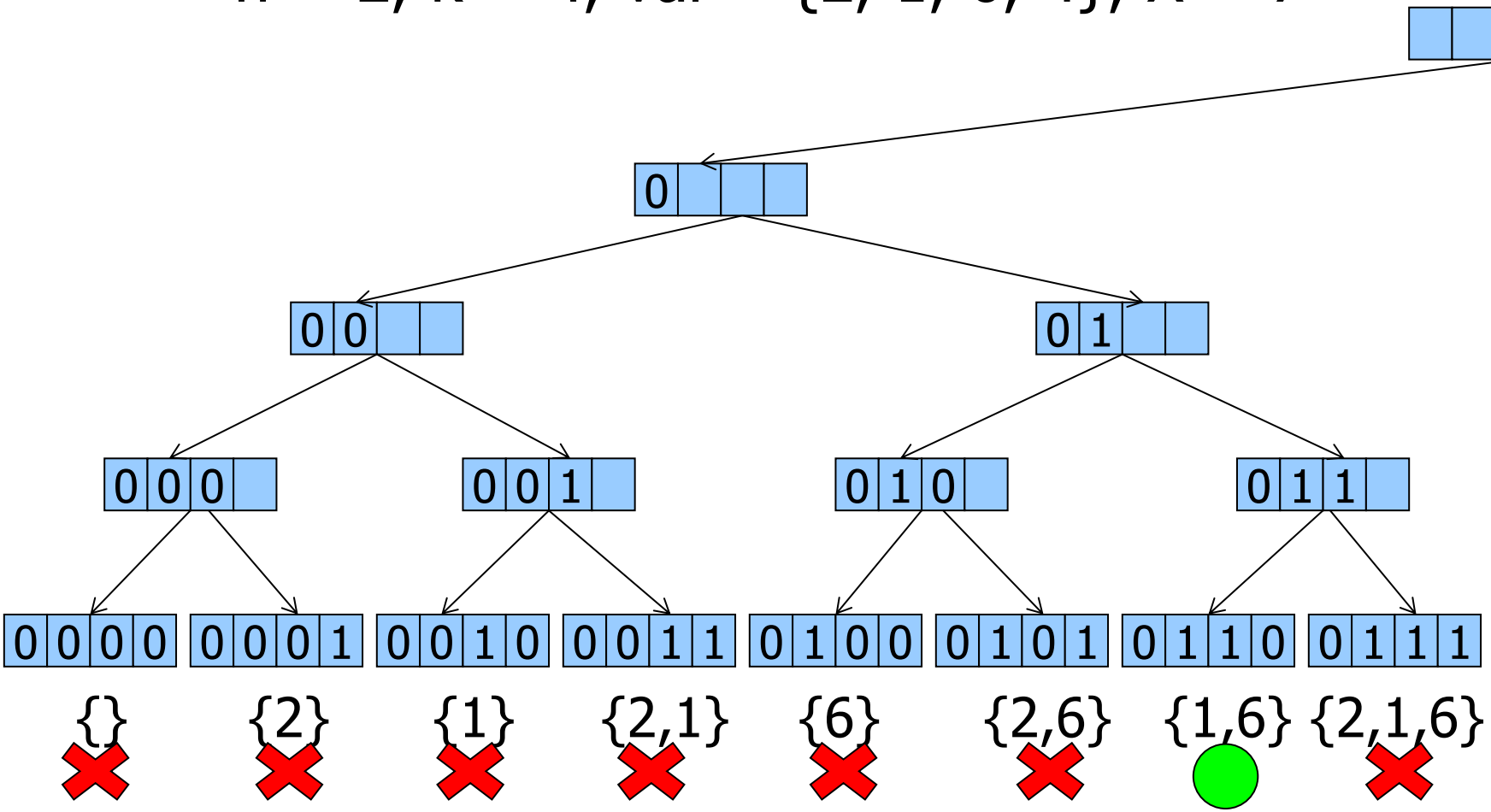
Y = { {1,2,4}, {1,6} }



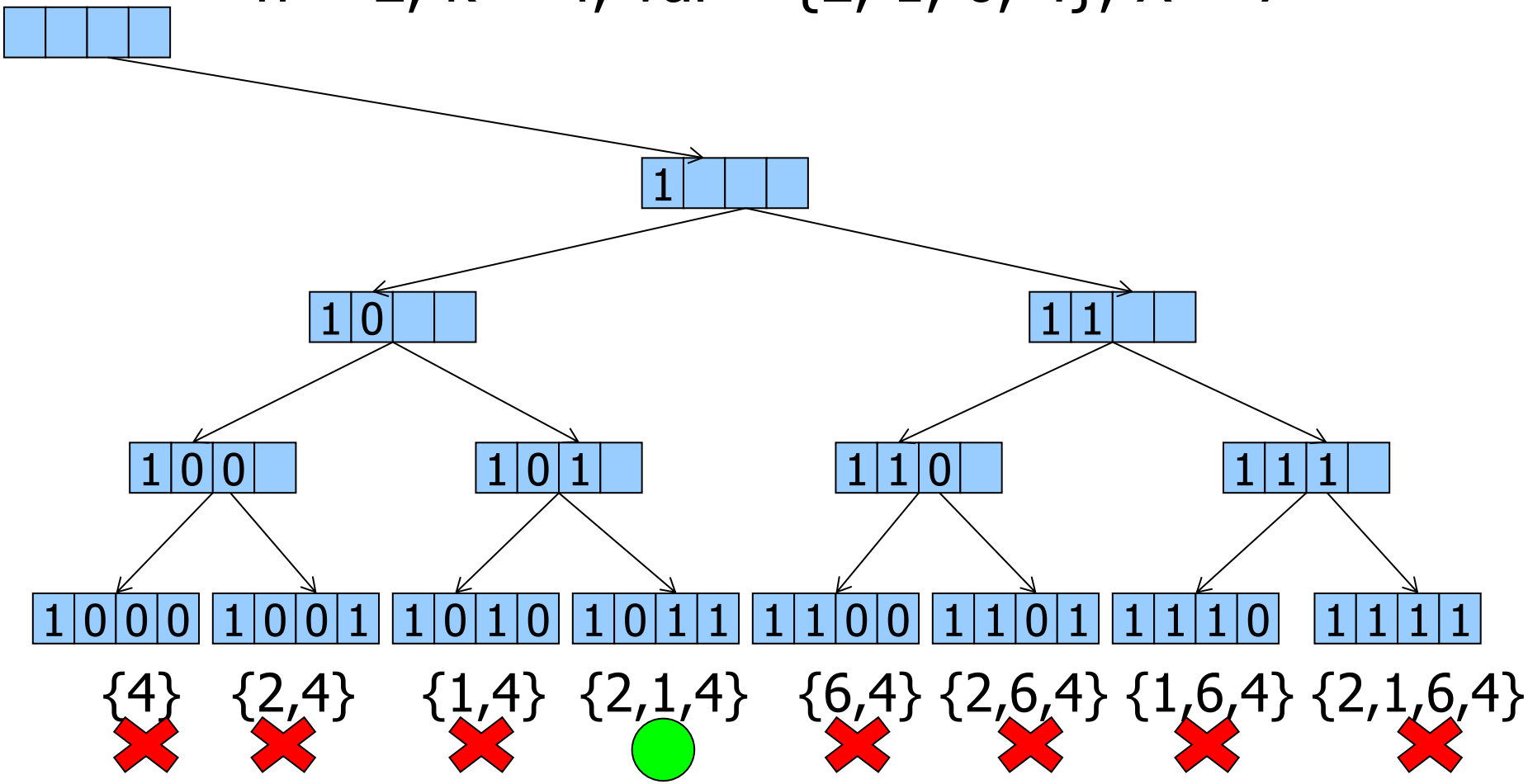
Enumerative approach (without pruning):

- compute the powerset $\wp$(val) (arrangements with repetitions)
- For each subset(termination condition), check if the sum of its elements equals X.

n = 2, k = 4, val = {2, 1, 6, 4}, X = 7



n = 2, k = 4, val = {2, 1, 6, 4}, X = 7



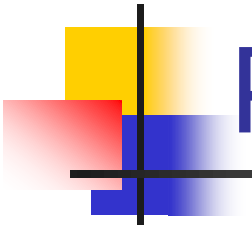
```
val = malloc(k * sizeof(int));
sol = malloc(k * sizeof(int));
```

```
void powerset(int pos,int *val,int *sol,int k,int X) {
  int j, out;
  if (pos >= k) {
    out = check(sol, val, X, k);
    if (out==1) {
}etc.etc.

int check(int *sol, int *val, int X, int k) {
  int j, tot=0;
  for (j=k-1; j>=0; j--)
    if (sol[j]!=0)
      tot += val[k-j-1];
  if (tot==X)
    return 1;
  else
    return 0;
}
```

termination

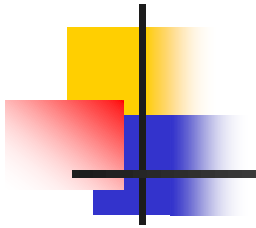
check solution

print solution

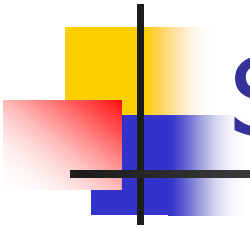


Pruning

- Early evaluation of constraints in an intermediate state
- No general methodology
- Typical cases:
  - static filter on choices: acceptance conditions that do not depend on past choices, but only on the problem (e.g., boundaries in a map)



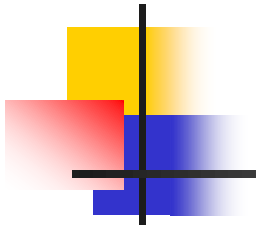
- dynamic filter on choices: acceptance conditions that depend on pst choices (e.g. position of other pieces in a game)
- partial solution validation: evaluation of the hope to reach a solution or sufficient condition to rule out that possibility.



Sum of subsets

Pruning-based approach: strategy based on hope evaluation:

- sort `val` in ascending order
- `p_sum` is the current sum, initially 0
- `r_sum`, initially set to the sum of all values in `val`, contains the sum of the not yet taken values (still available values)
- at each step consider an element of `val` only if "promising".



An element of `val` is promising if:

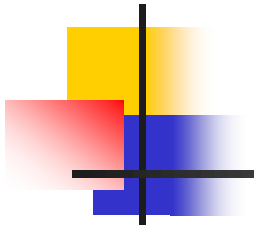
- the partial solution + still available values are >= the target sum

```
p_sum + r_sum >= X
```

&&

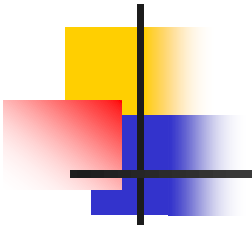
- the partial solution + the value of `val` is <= the target sum

```
p_sum + val[pos]<= X
```



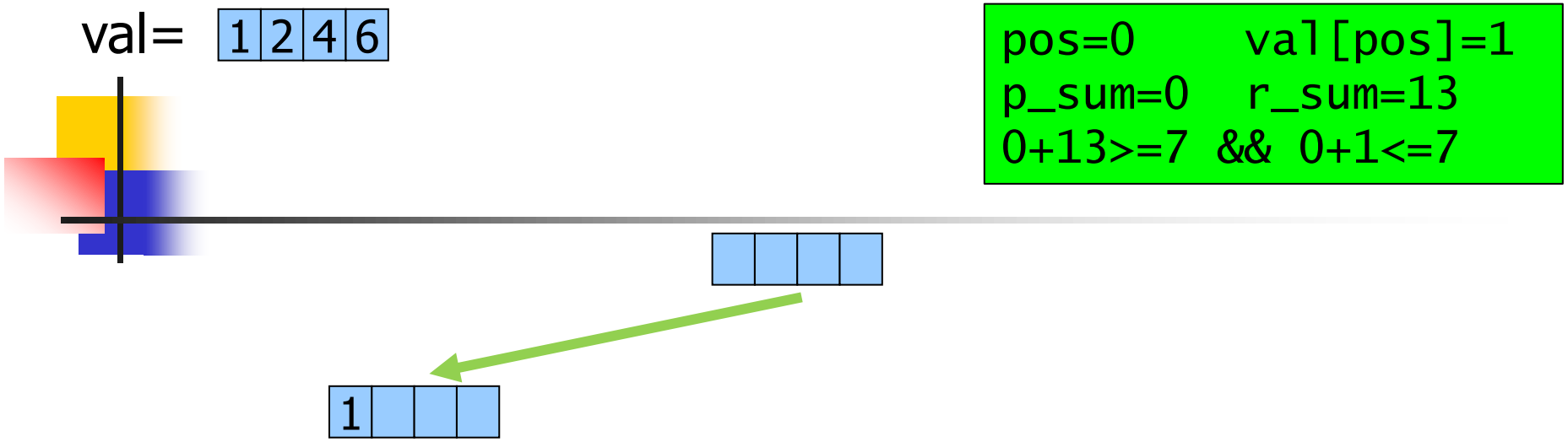
If the element is promising:

- take it (`mark[pos]=1`)
- recur on the next one (`pos+1`), updating `p_sum` (`p_sum+val[pos]`) and (`r_sum-val[pos]`)
- while backtracking, don't take it (`mark[pos]= 0`)
- recur on the next one (`pos+1`), `p_sum` is unchanged, `r_sum` is updated (`r_sum - val[pos]`)

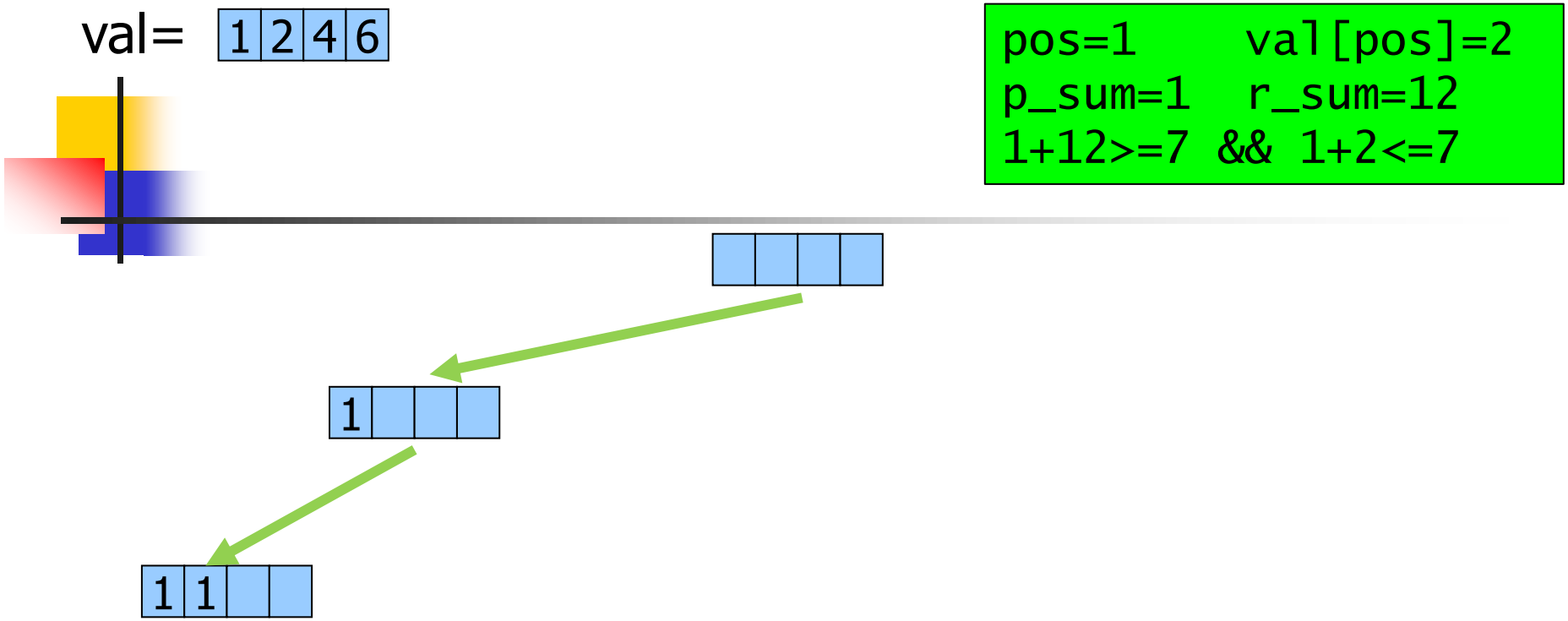


If an element is not promising, as the `val` array is sorted, all the elements that follow are not promising:

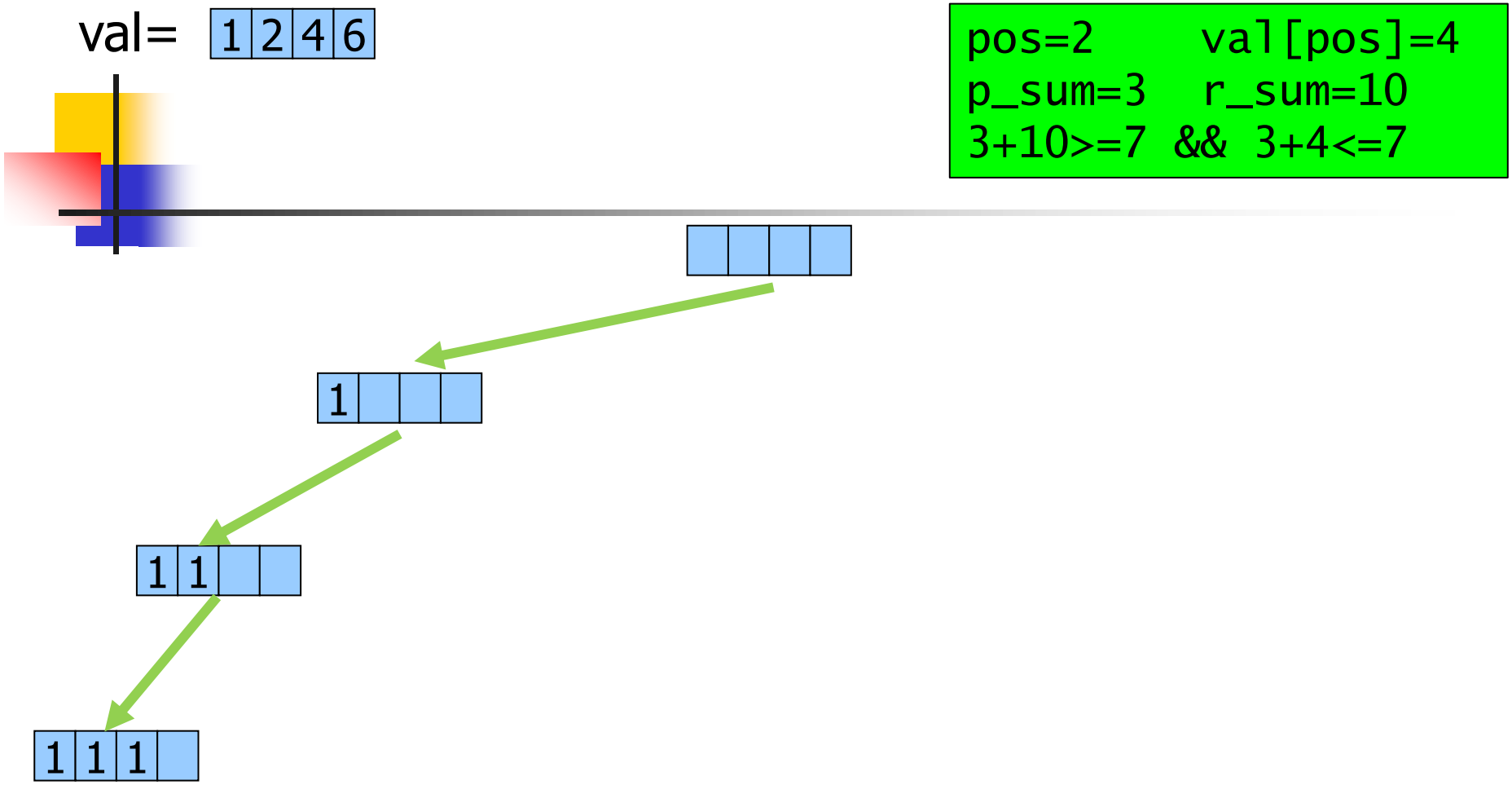
- If `p_sum + r_sum < X` such a sum will not increase, considering the following element, independently of ordering
- If `p_sum + val[pos] > X`, as the following element is > than the current one, condition ≤X will never be satisfied.



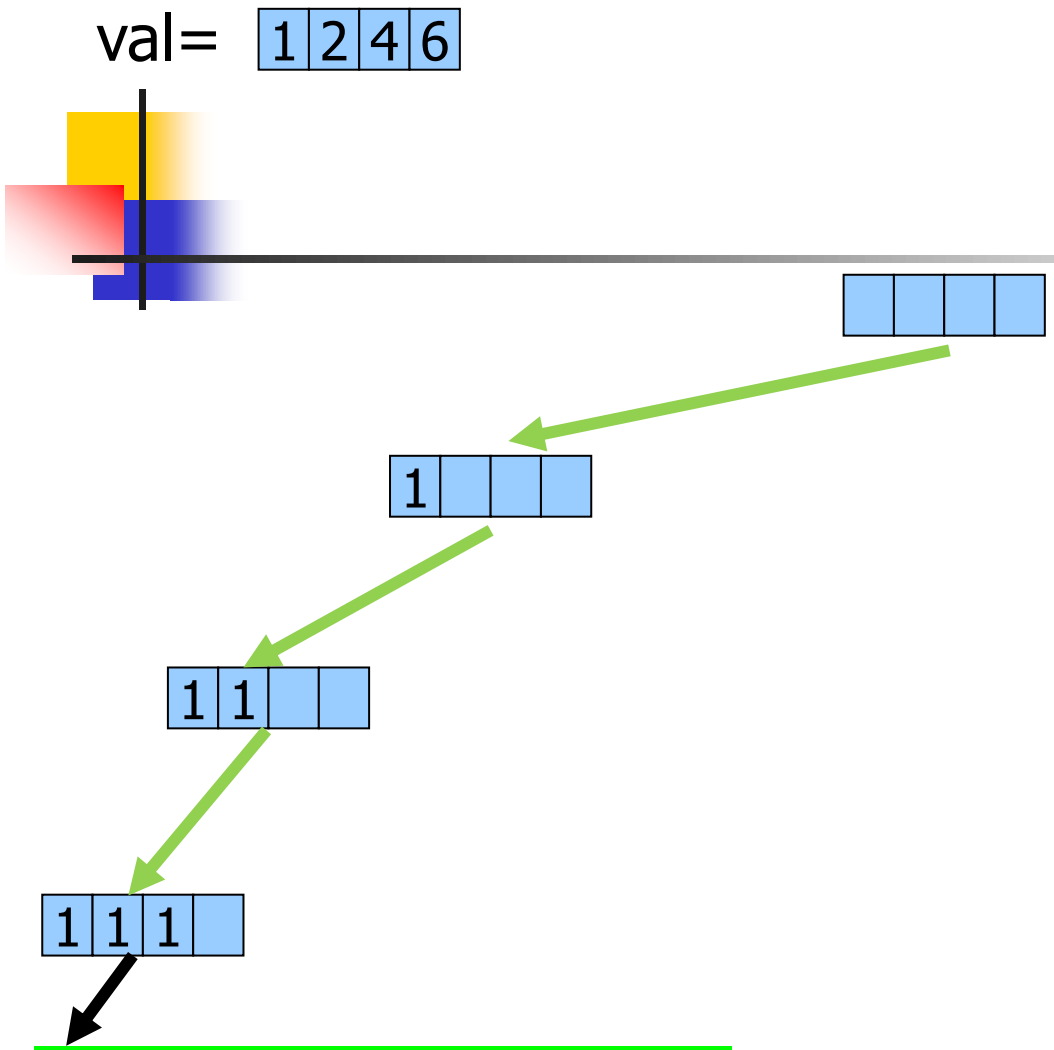
Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`



Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`



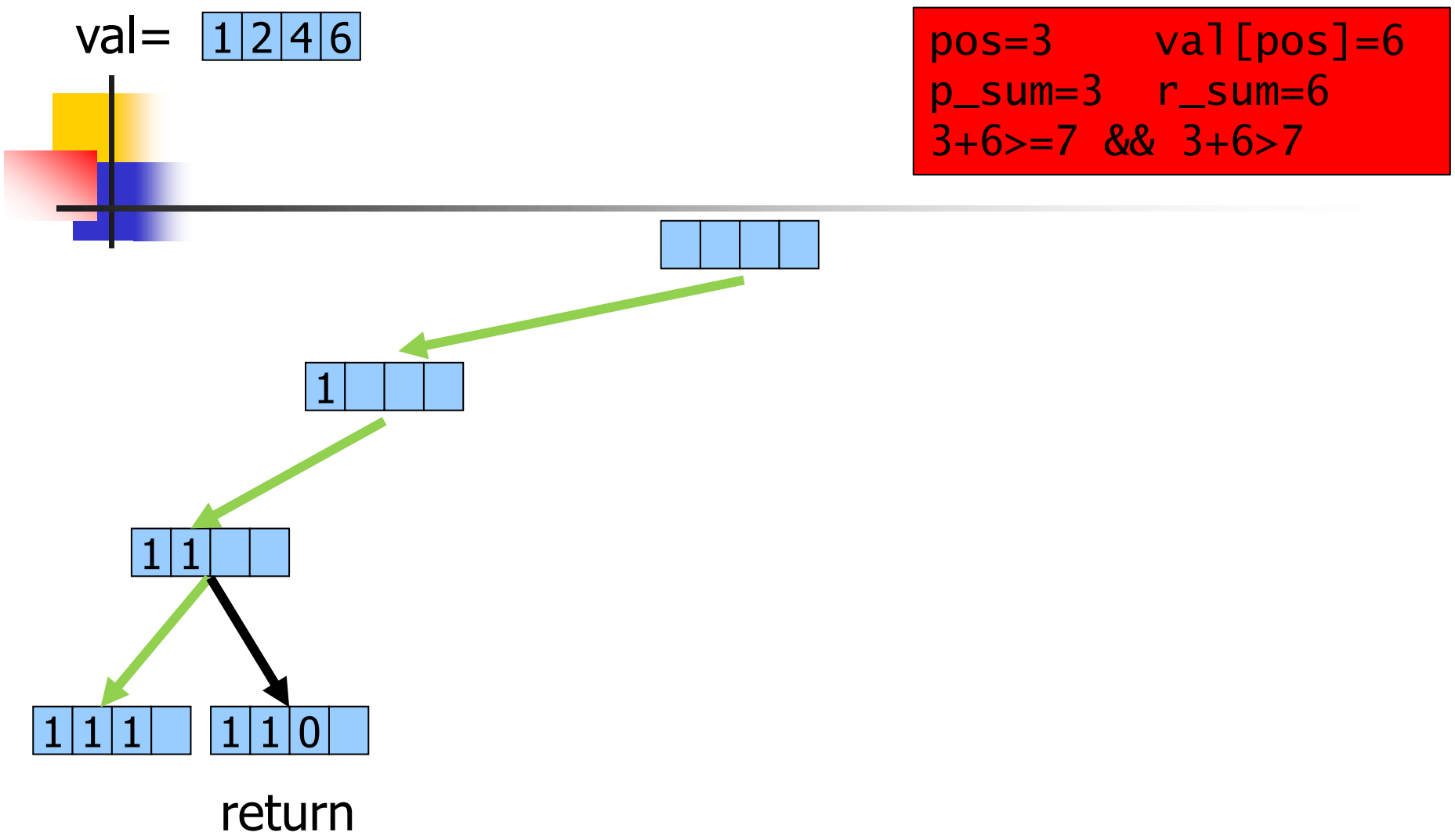
Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`



termination: `p_sum=7=X`

return

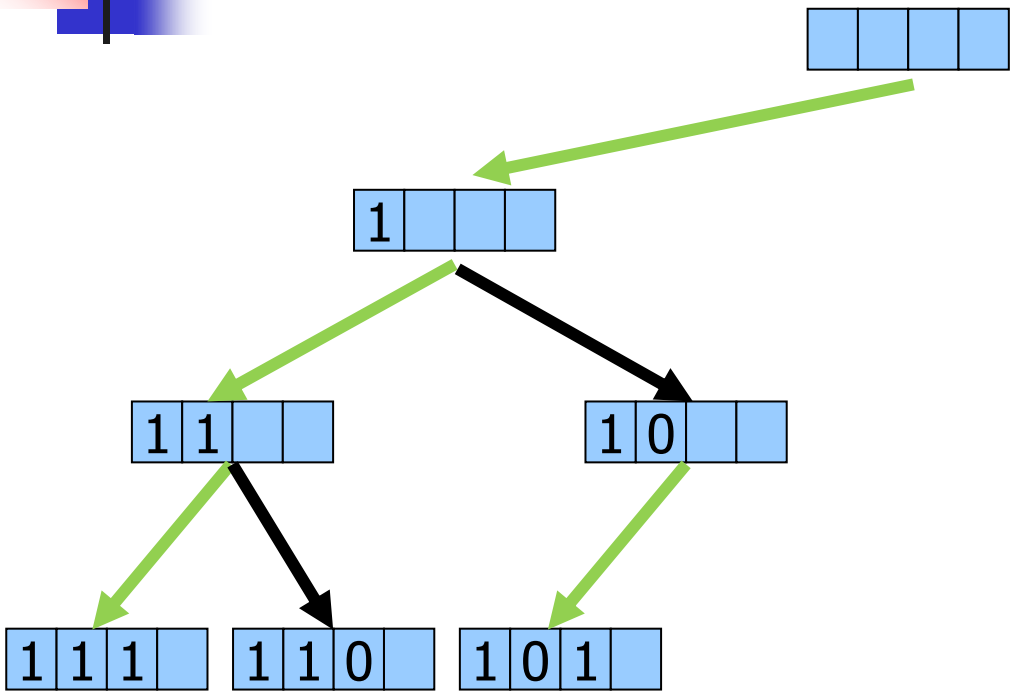
Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`



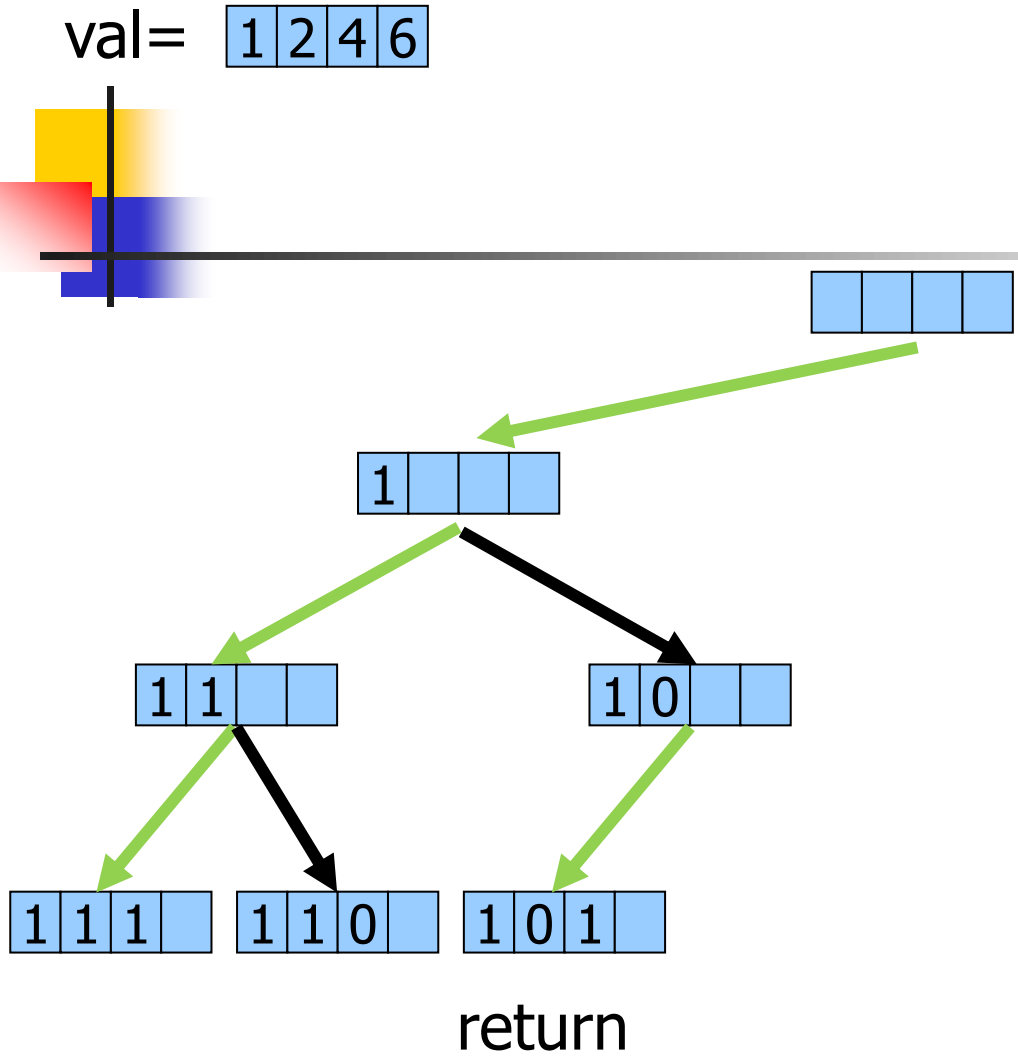
Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`

val= 1 2 4 6

```
pos=2      val[pos]=4
p_sum=1    r_sum=10
1+10>=7  && 1+4<=7
```



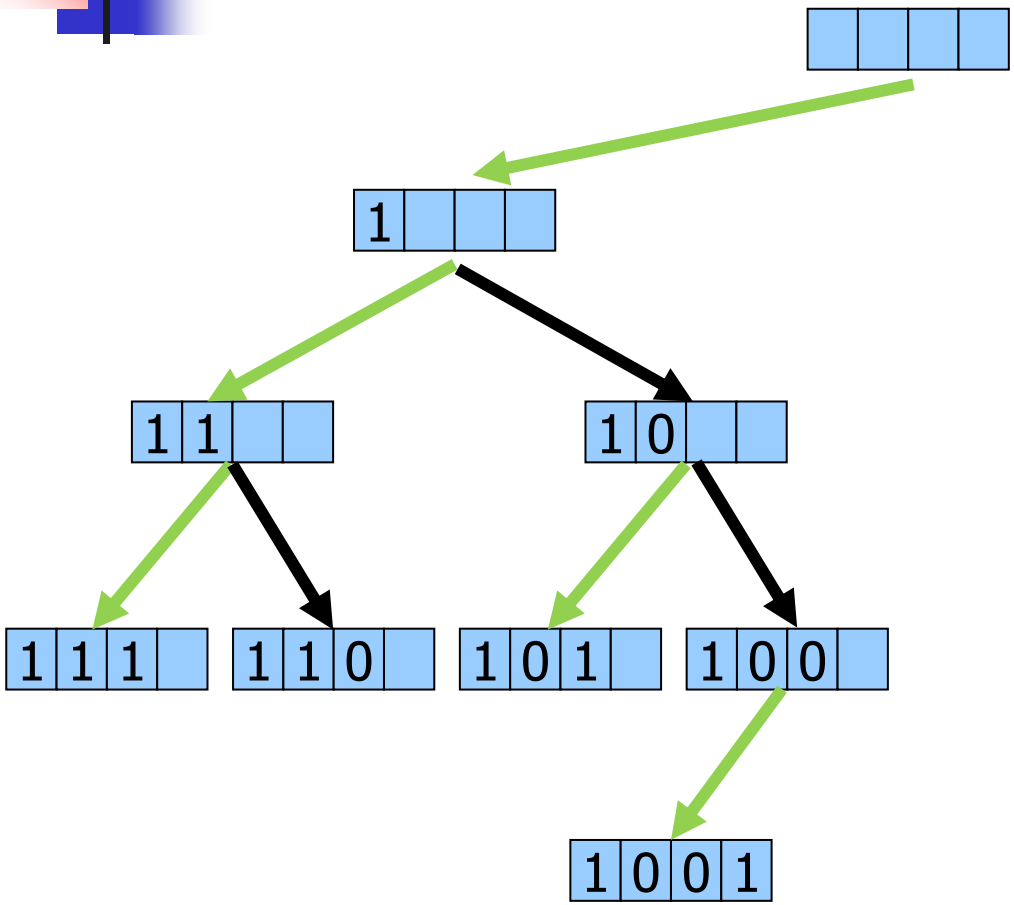
Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`



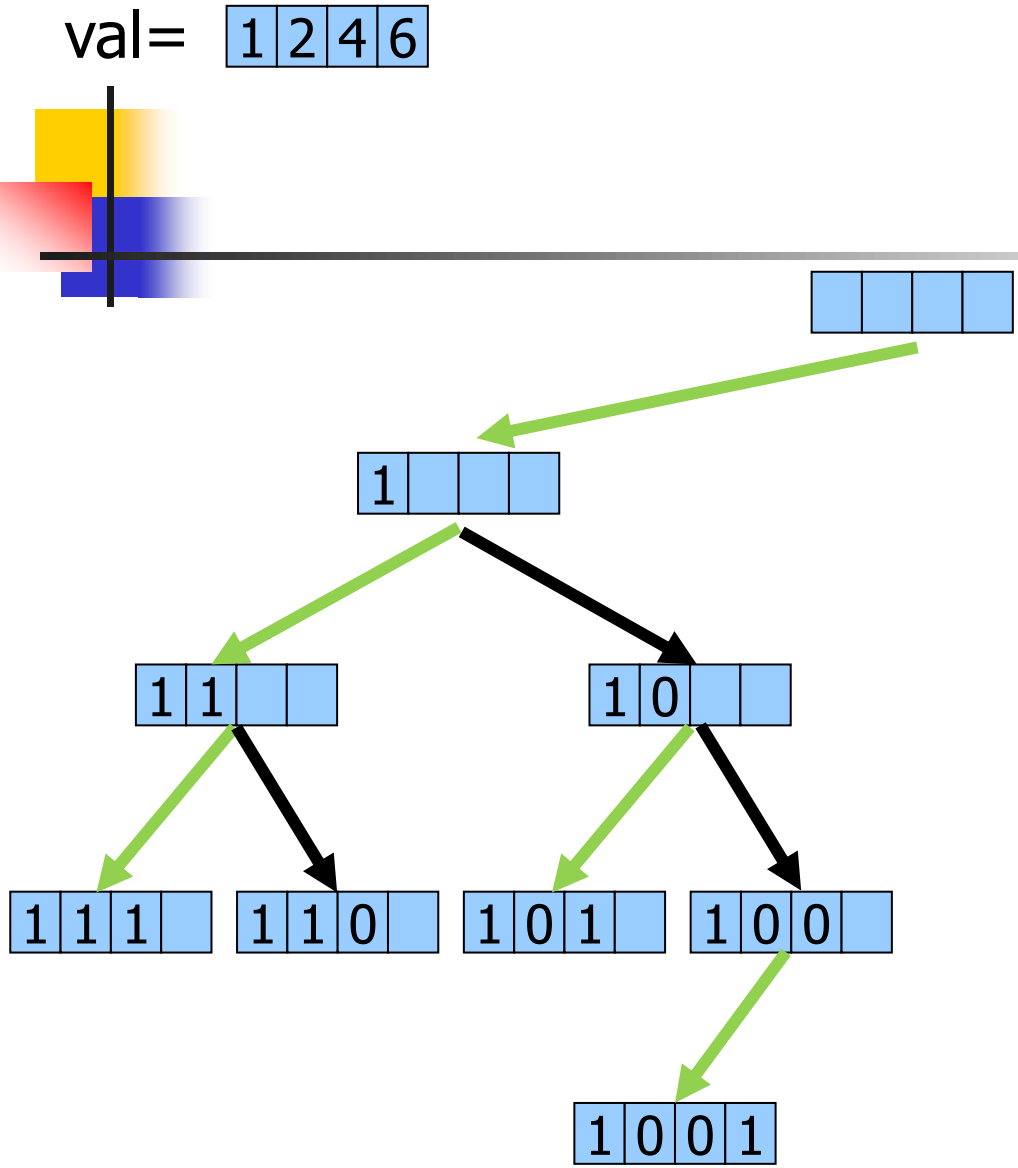
Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`

val= 1 2 4 6

```
pos=3     val[pos]=6
p_sum=1   r_sum=6
1+6>=7 && 1+6<=7
```



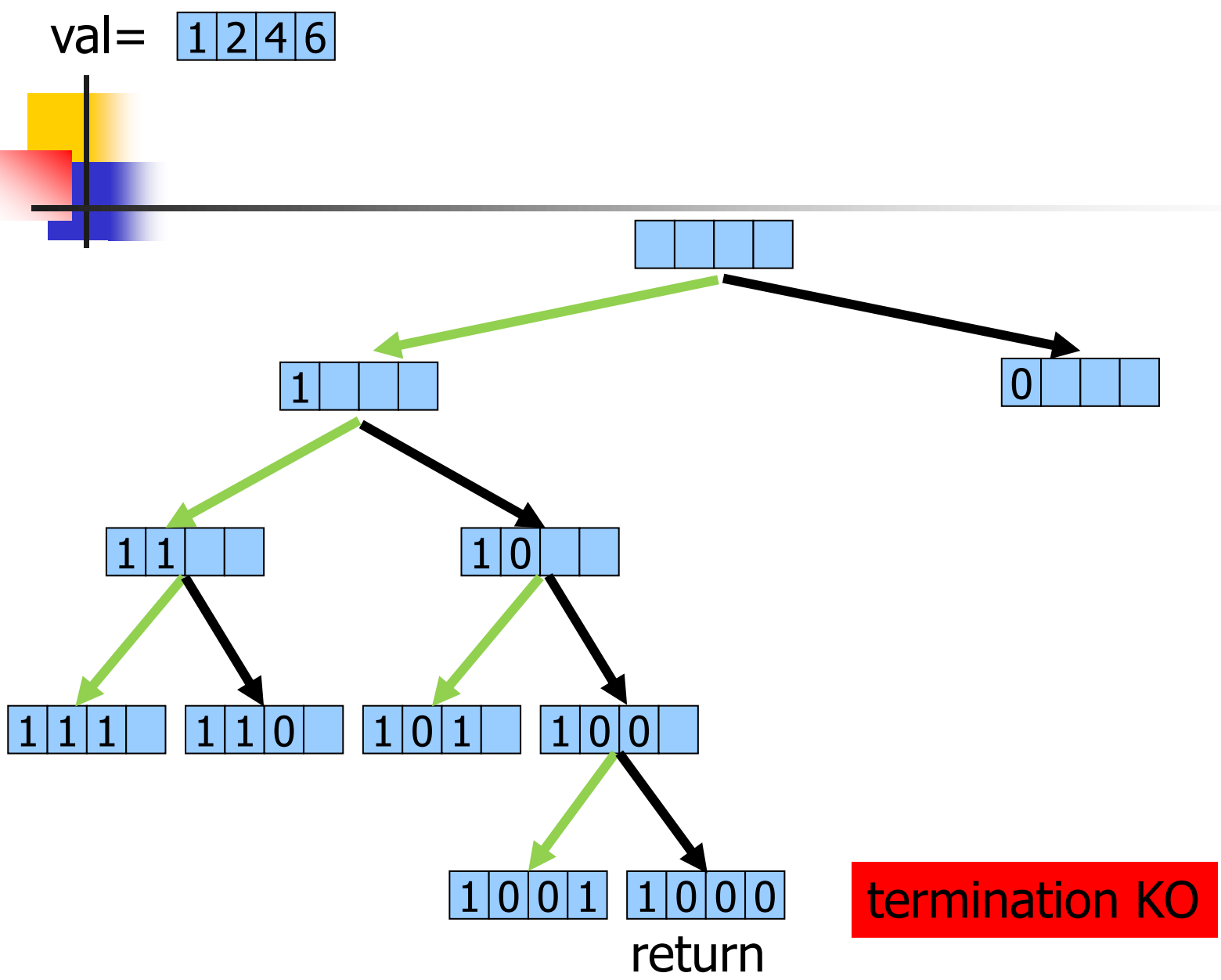
Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`



return

termination: `p_sum=7=X`

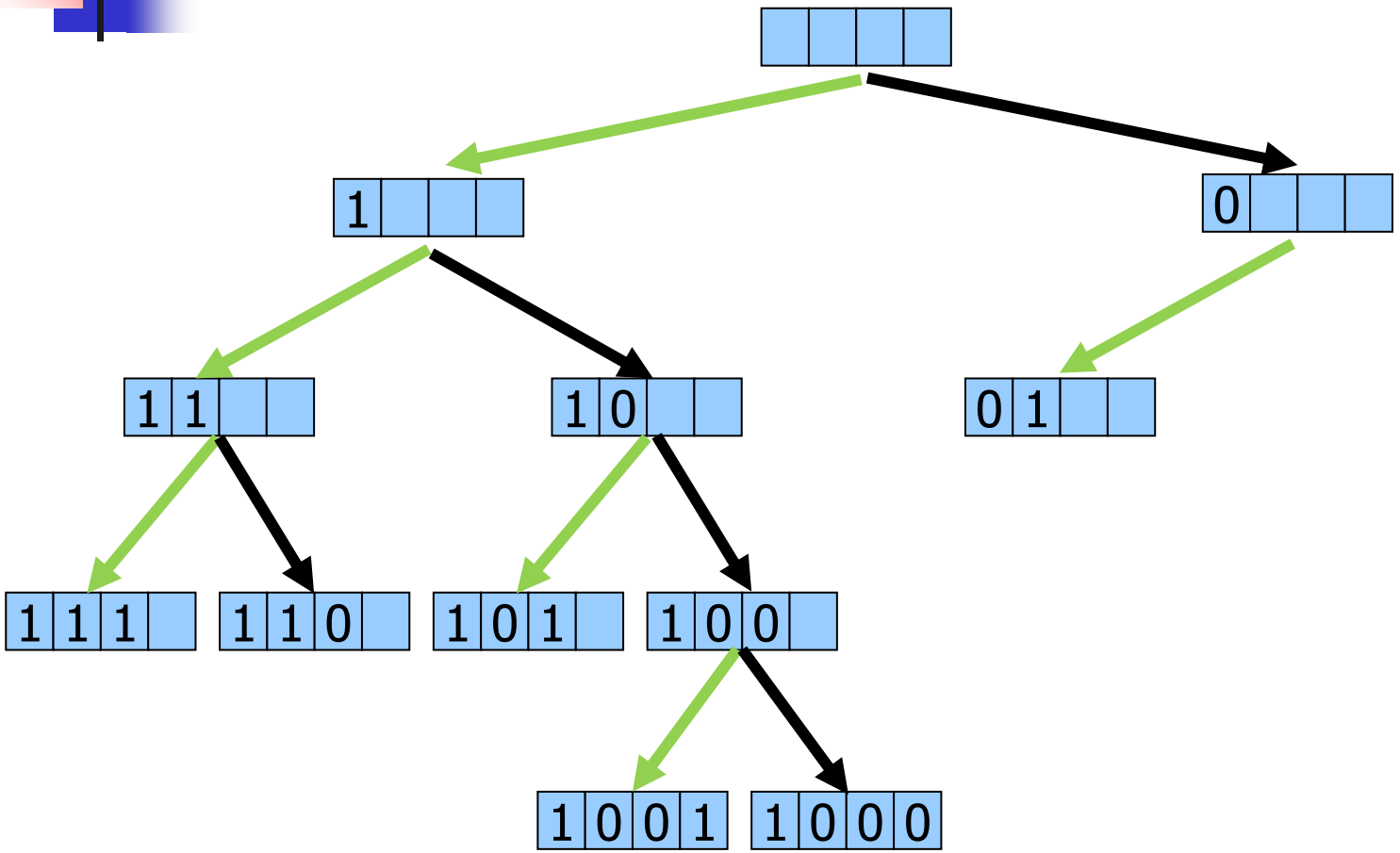
Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`



Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`

val= 1 2 4 6

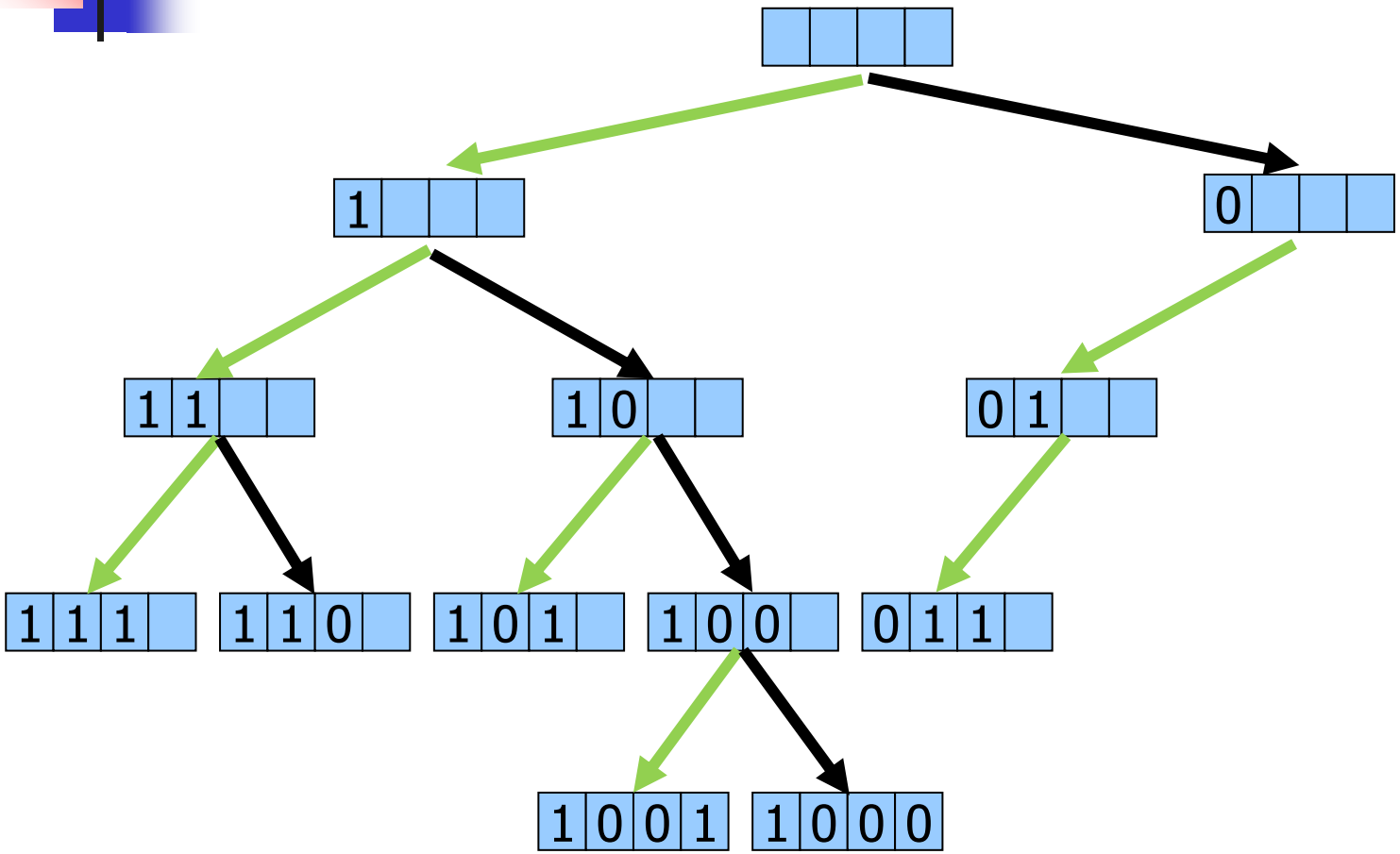
```
pos=1      val[pos]=2
p_sum=0    r_sum=12
0+12 >= 7 && 0+2<=7
```



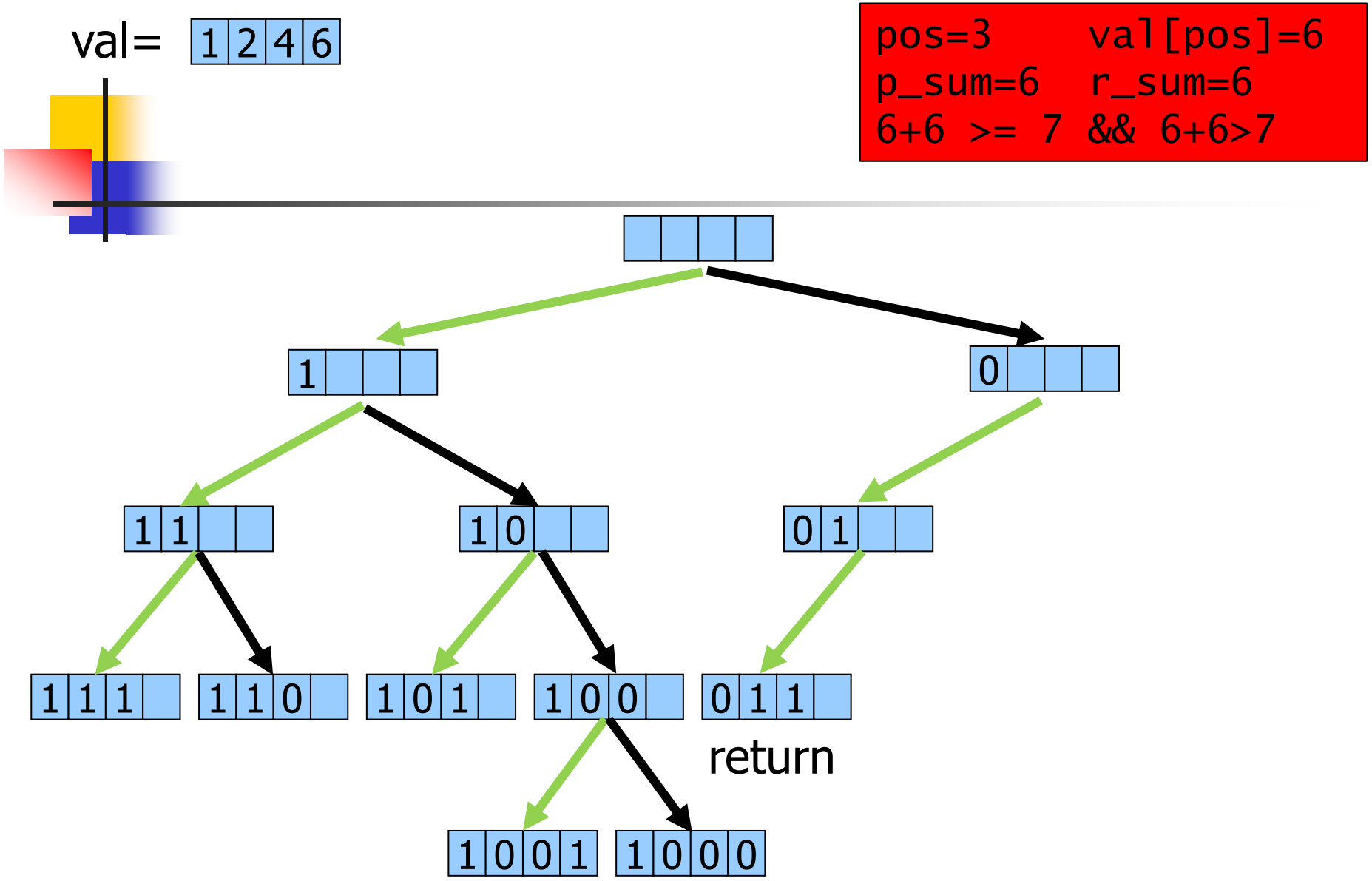
Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`

val= 1 2 4 6

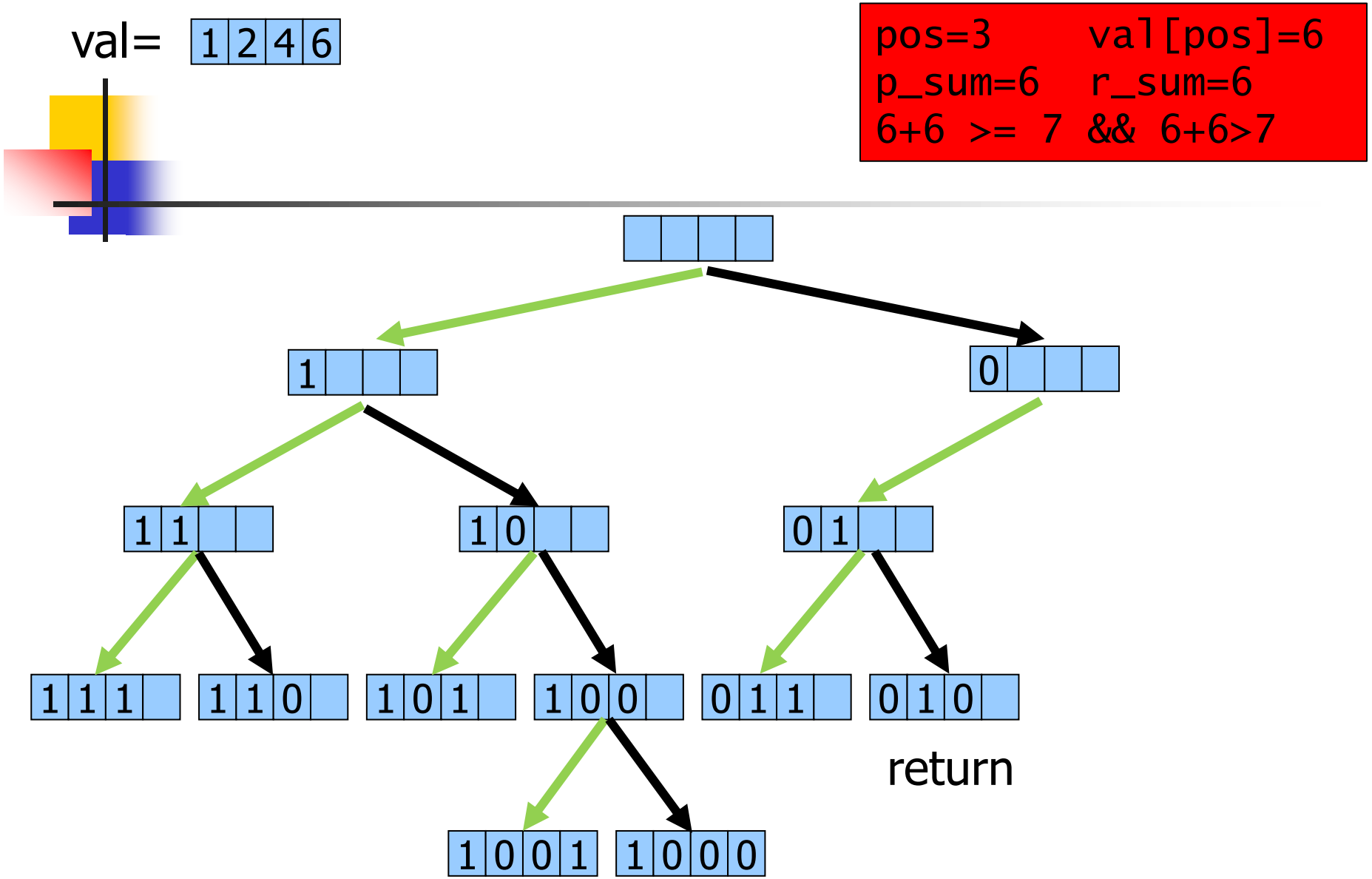
```
pos=2      val[pos]=4
p_sum=2    r_sum=10
2+10 >= 7 && 2+4<=7
```



Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`



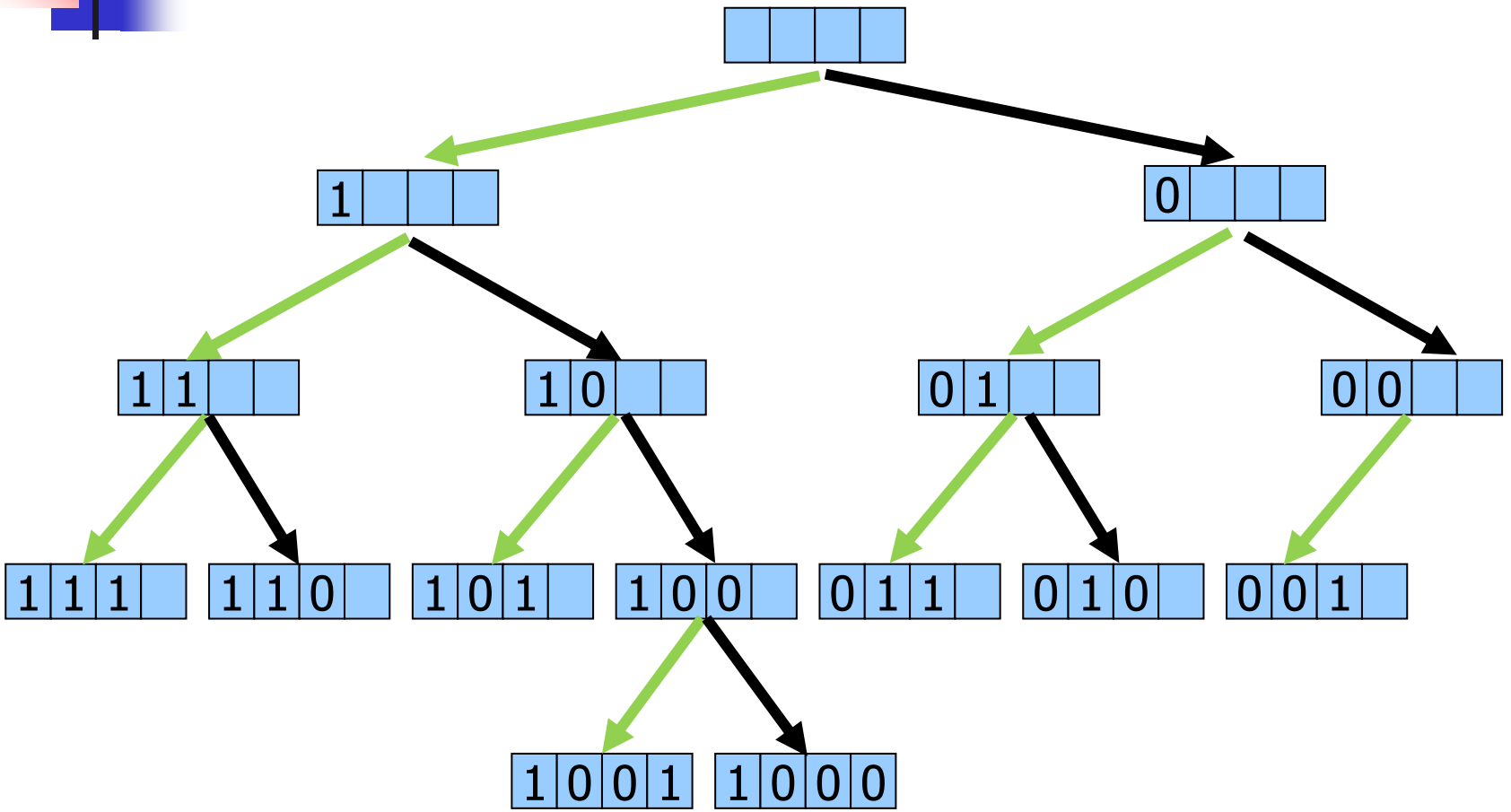
Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`



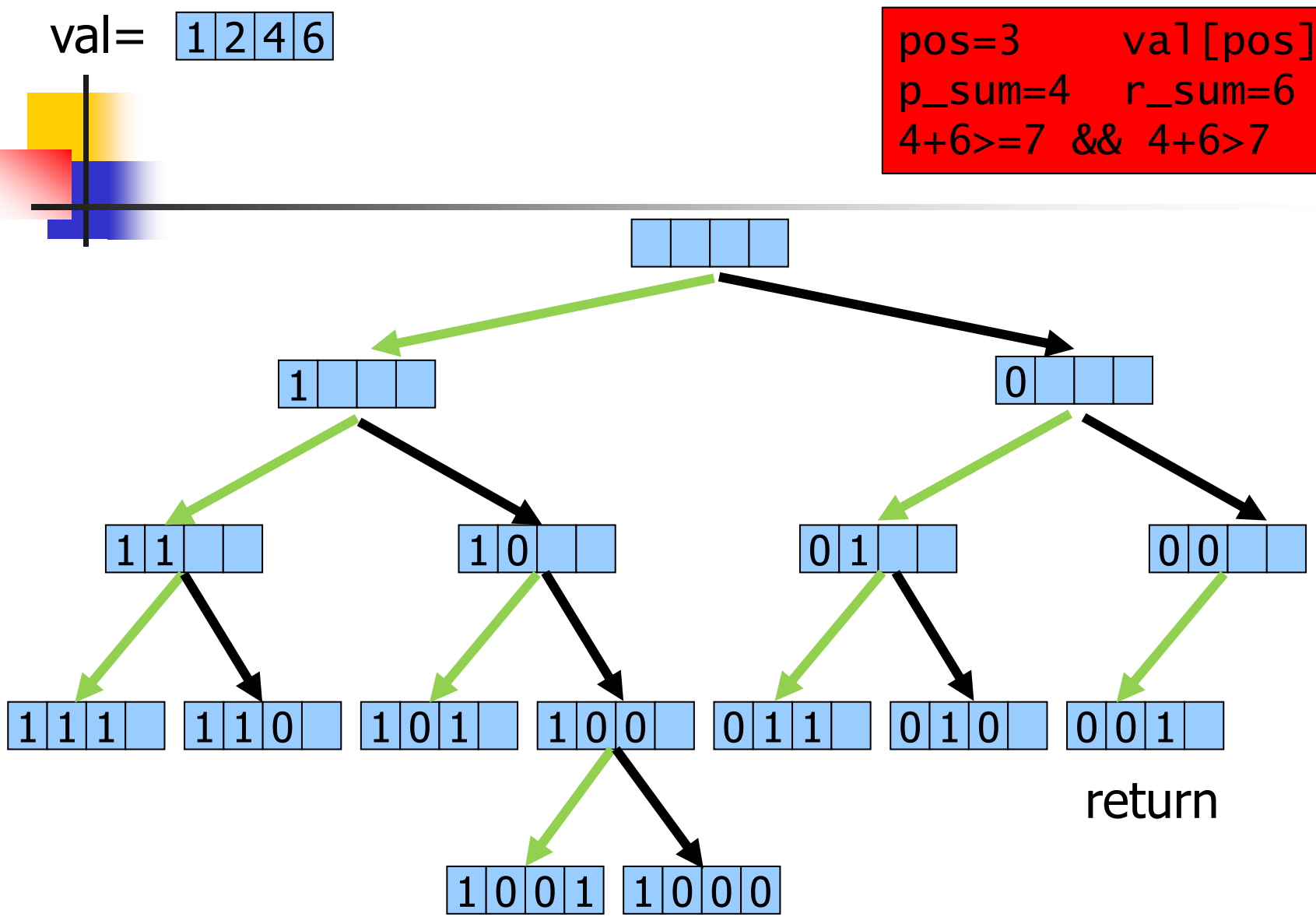
Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`

val= 1 2 4 6

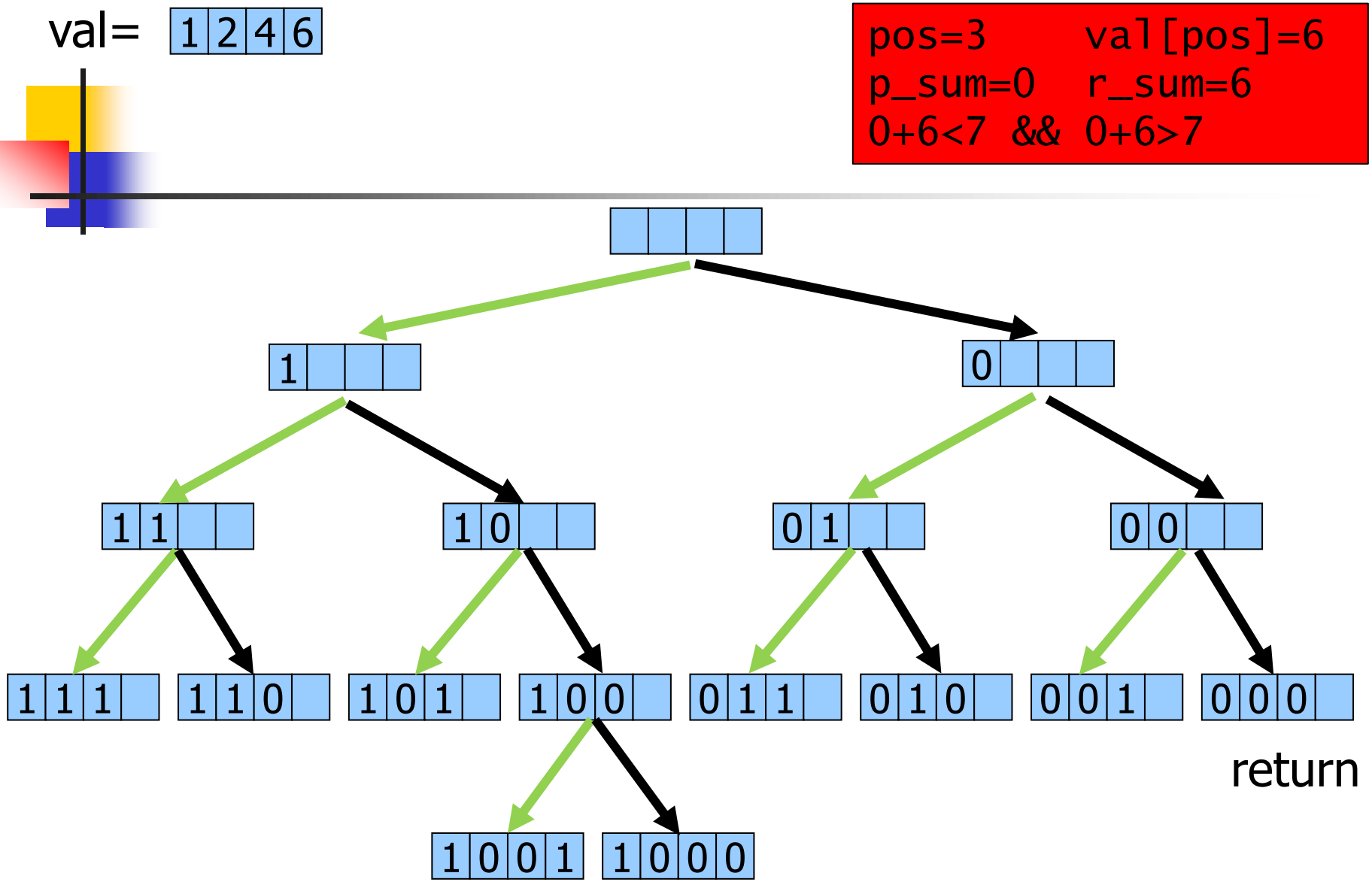
```
pos=2      val[pos]=4
p_sum=0    r_sum=10
0+10>=7  &&  0+4<=7
```



Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`



Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`



Promising: `p_sum + r_sum >= X && p_sum + val[pos]<= X`

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

```
void sumset(int pos,int *val,int *mark,int p_sum,
            int r_sum,int X) {
  int j;
  if (p_sum==X) {
    printf("\n{\t");
    for(j=0;j<pos;j++)
      if(mark[j])
        printf("%d\t",val[j]);
    printf("}\n");
    return;
  }
  if(promising(val,pos,p_sum,r_sum,X)){
    mark[pos]=1;
    sumset(pos+1,val,mark,p_sum+val[pos],r_sum-val[pos],X);
    mark[pos]=0;
    sumset(pos+1,val,mark,p_sum, r_sum-val[pos],X);
  }
}
```

termination

print solution

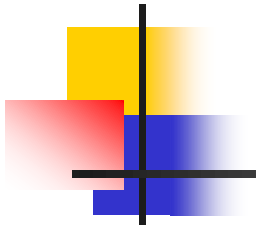
check if promising

take

recur

do not take

recur

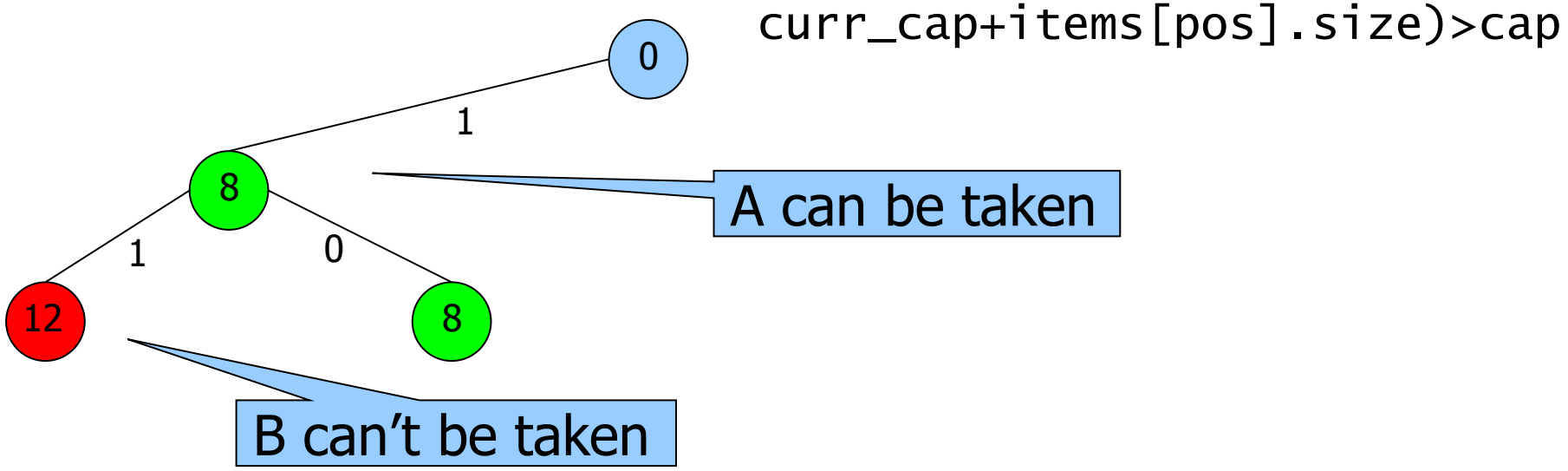


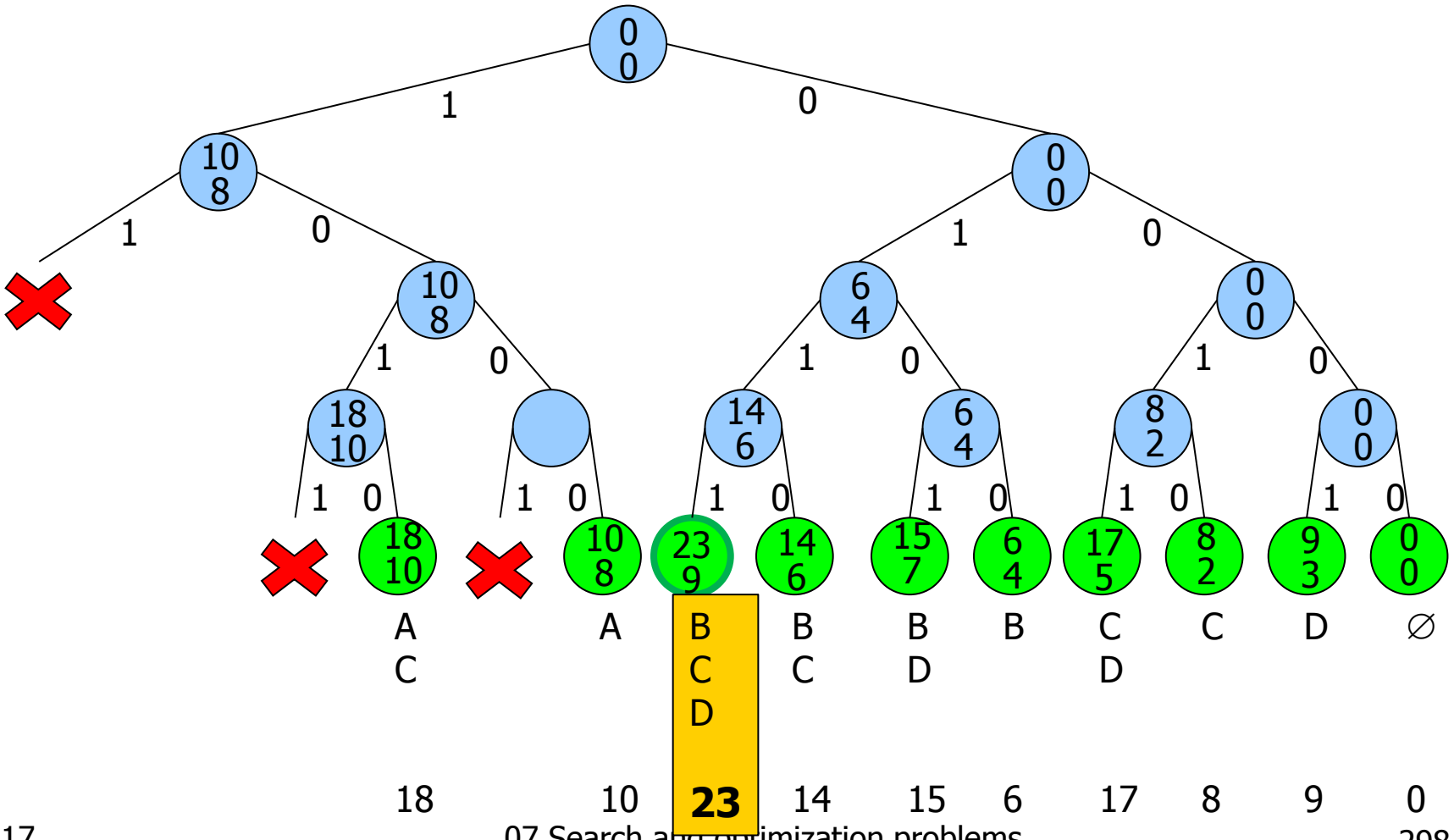
```
int promising(int *val,int pos,int p_sum,int r_sum,int X) {
    return (p_sum+r_sum > =X)&&(p_sum+val[pos]<=X);
}
```

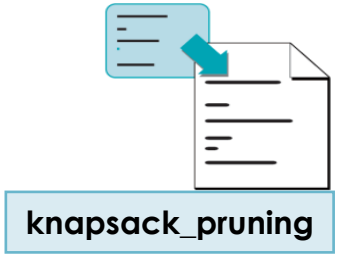
The knapsack (discrete)

Pruning-based approach: powerset

Pruning function: if, taking an object, the consumed capacity exceeds the maximum one, the object is not taken




0 0
1
0
10 8
0 0
1
0
1
0
10 8
6 4
0 0
1
0
1
0
1
0
18 10
14 6
6 4
8 2
0 0
1
0
1
0
1
0
1
0
1
0
18 10
10 8
23 9
14 6
15 7
6 4
17 5
8 2
9 3
0 0
A C
A
B C D
B C
B D
B
C D
C
D
∅
18
10
23
14
15
6
17
8
9
0




```
void powerset(int pos,Item *items,int *sol,int k,int cap,
     int curr_cap,int curr_value,int *bv,int *best_sol) {
  int j;
  if (pos >= k) {
    if (curr_value > *bv) {
      for (j=0; j<k; j++)
        best_sol[j] = sol[j];
      *bv = curr_value;
    }
  }
        return;
}
  if ((curr_cap + items[pos].size) > cap) {
    sol[pos] = 0;
    powerset(pos+1,items,sol,k,cap,curr_cap,curr_value,
             bv,best_sol);
    return;
  }
```

termination condition

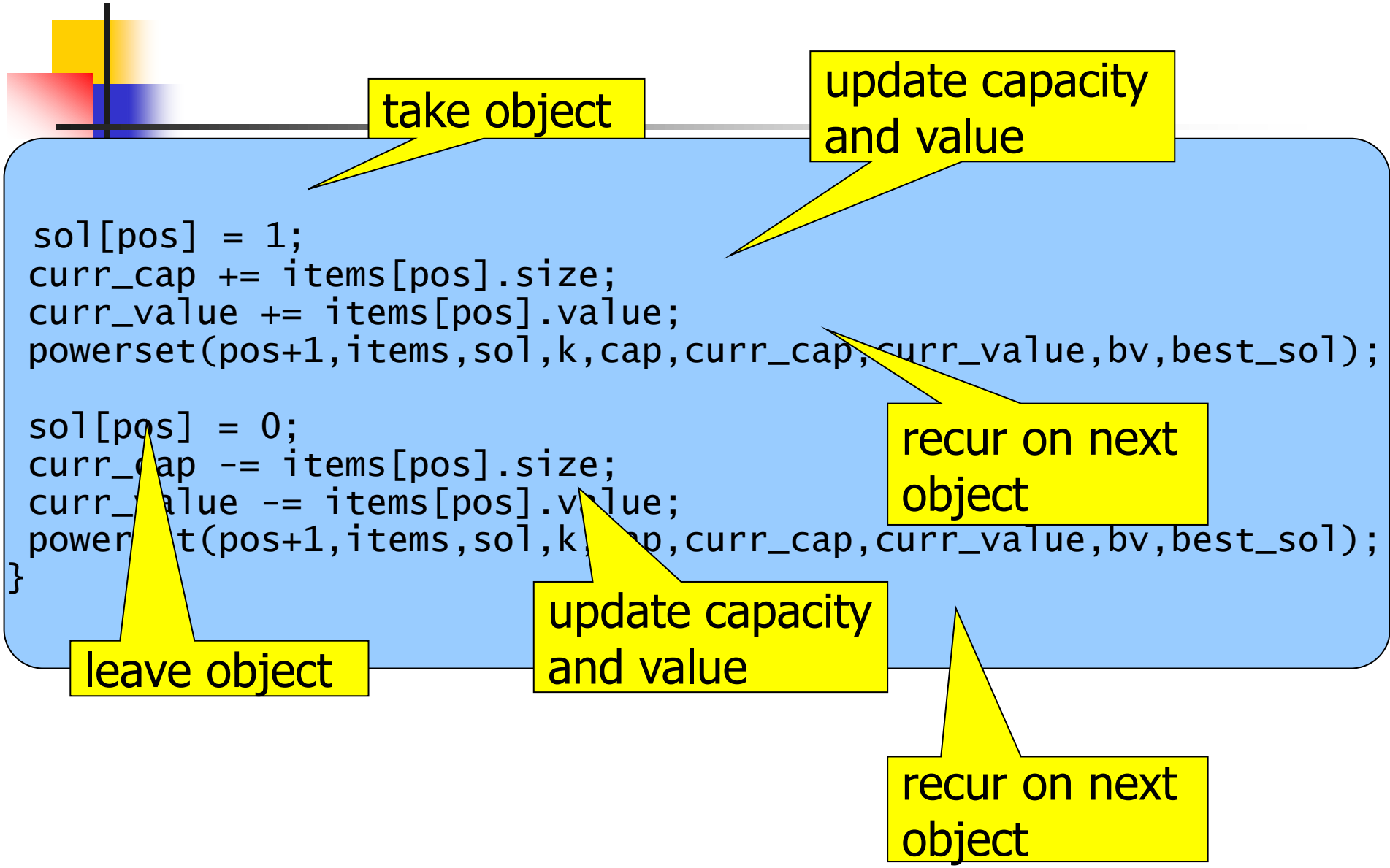
optimality check

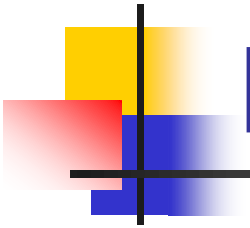
pruning check

leave object

recur on next object

return


take object
update capacity and value
sol[pos] = 1;
curr_cap += items[pos].size;
curr_value += items[pos].value;
powerset(pos+1,items,sol,k,cap,curr_cap,curr_value,bv,best_sol);
sol[pos] = 0;
curr_cap -= items[pos].size;
curr_value -= items[pos].value;
powerset(pos+1,items,sol,k,cap,curr_cap,curr_value,bv,best_sol);
}
recur on next object
leave object
update capacity and value
recur on next object



References

- Backtracking
  - Bertossi 16
- Permutations and subsets
  - Bertossi 16.3