Algorithms (3) : Permutation, Combination, & Brute Force Search

Use recursion to explore all possible solutions!

Outline

1. **Motivation – Recursion for Permutation**
2. Basic: Permutation using Recursion
3. Permutation without repetition
4. Permutation without repetition & order
5. Recursion & Brute Force Search

Permutation – 2 alphabets

Given a set of 2 alphabets: { a , b }

- **<u>Question 1</u>**:
  - How many **<u>different words of 2 characters (or slots)</u>** can we create with the set?

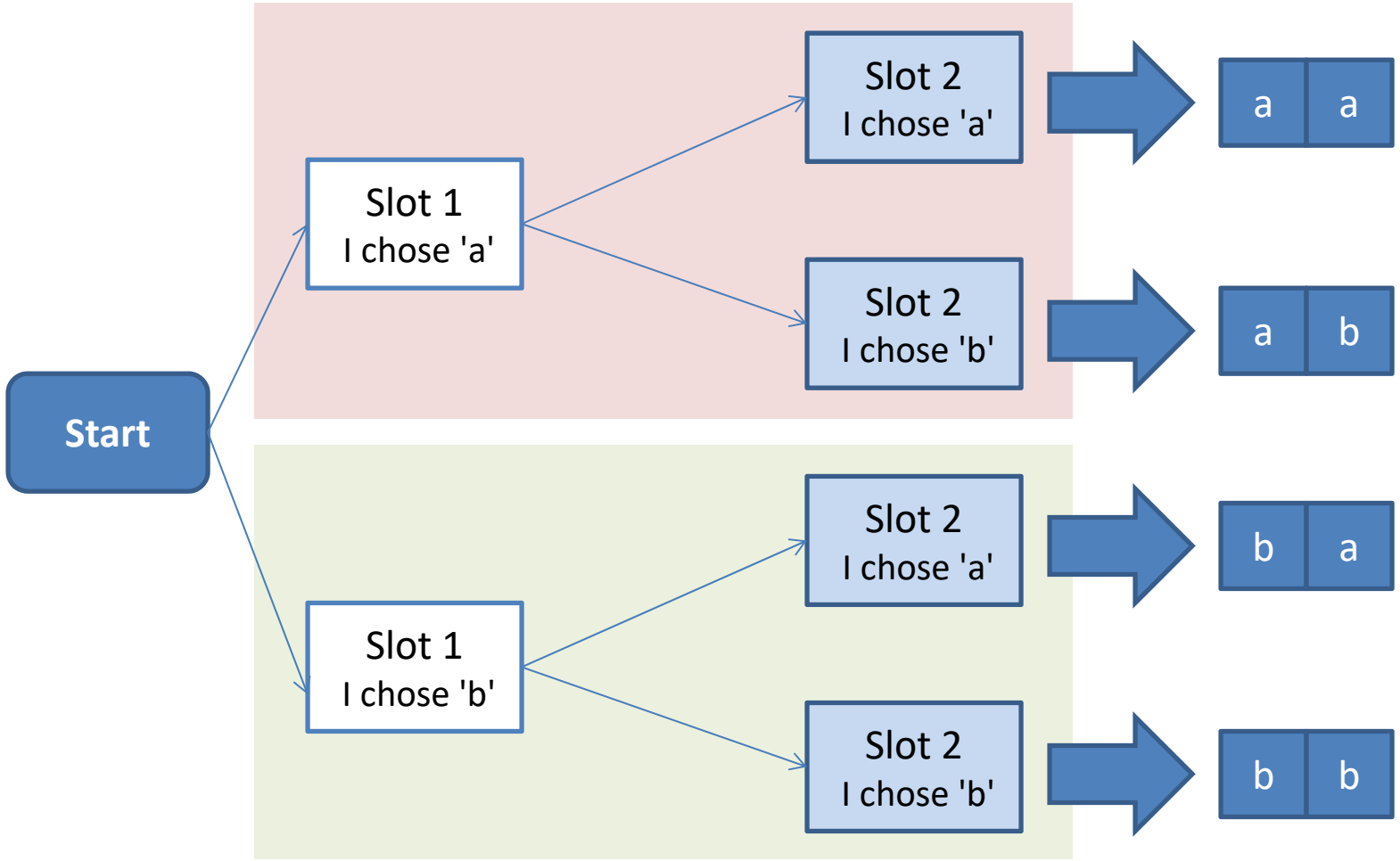
Slot 1 { a , b }	Slot 2 { a , b }
2 choices	2 choices

= 2 x 2 = 4 different words (or 4 permutations)

  - Also, print all the words out

Permutation – 2 alphabets

- Print all the words out:



Case 1: Permutation with Repetition

- We call the above scenario – **permutation with repetition**.

- Can we write a program to generate that permutation?
  - Initial Idea: may be using loops

Permutation – Using nested loops

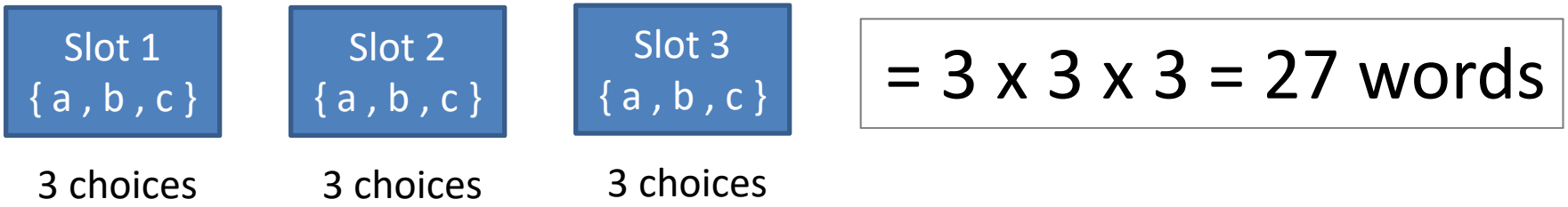
```
1  void perm( char alphabets[] , int a_len , char result[] )
2  {
3      for ( int i = 0 ; i < a_len ; i++ )
4      {
5          result[ 0 ] = alphabets[ i ] ;
6          for ( int j = 0 ; j < a_len ; j++ )
7          {
8              result[ 1 ] = alphabets[ j ] ;
9              result[ 2 ] = '\0' ;
10             printf( "%s\n" , result ) ;
11         }
12     }
13 }
14
15 int main( void )
16 {
17     char alphabets [ 2 ] = { 'a' , 'b' } ;
18     char result    [ 3 ] ;
19     perm( alphabets , 2 , result );
20 }
```

Loop 1 for slot 1

Loop 2 for slot 2

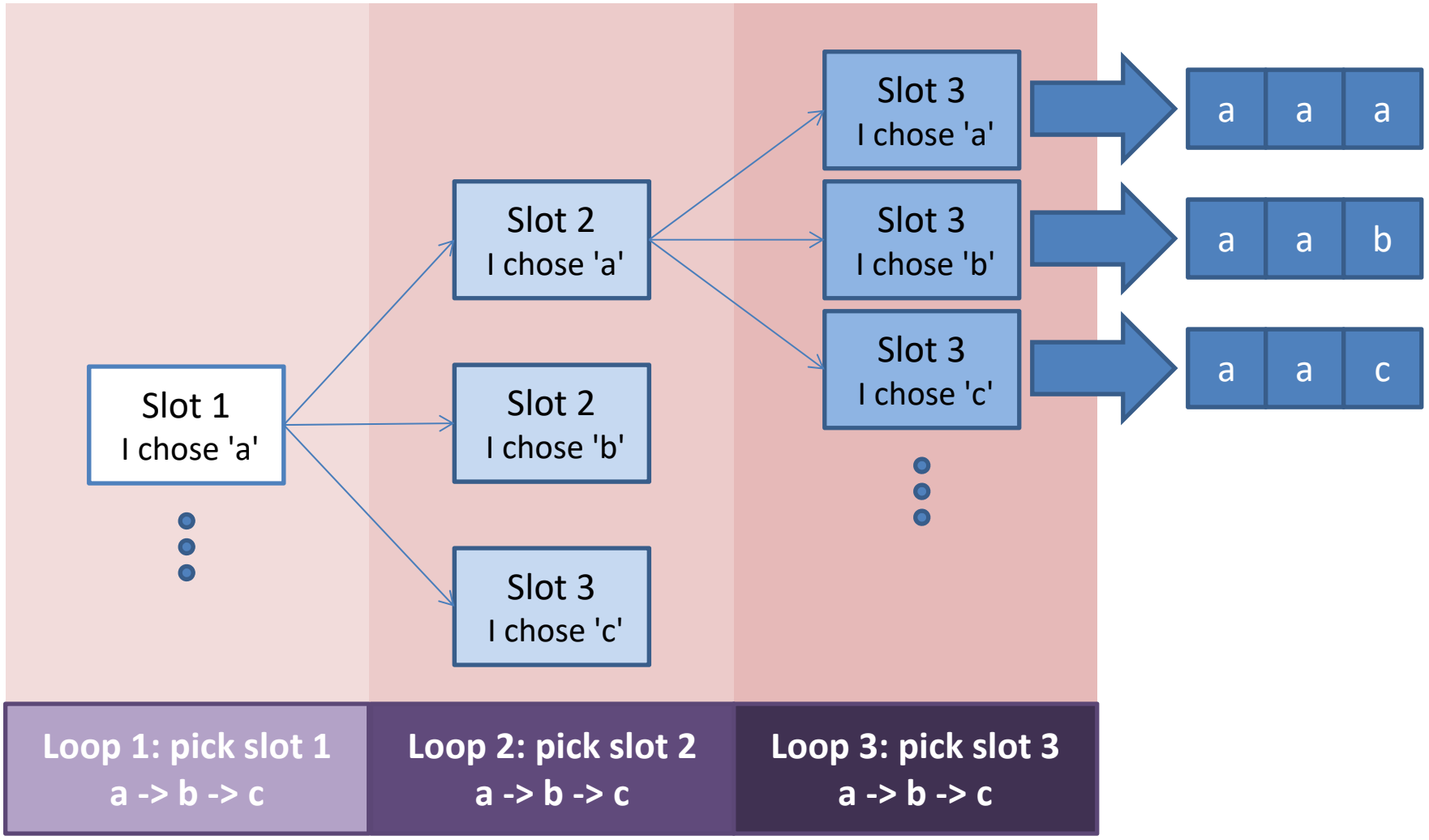
Permutation – 3 alphabets

- **Question 2**:
  - How many **different words of 3 characters (or slots)** can we create with a set of 3 alphabets?



  - Again, print all the words out

Permutation – 3 alphabets



Permutation – 3 alphabets

```
1  void perm( char alphabets[] , int a_len , char result[] )
2  {
3      for ( int i = 0 ; i < a_len ; i++ )
4      {
5          result[ 0 ] = alphabets[ i ] ;
6          for ( int j = 0 ; j < a_len ; j++ )
7          {
8              result[ 1 ] = alphabets[ j ] ;
9              for ( int k = 0 ; k < a_len ; k++ )
10             {
11                 result[ 2 ] = alphabets[ k ] ;
12                 result[ 3 ] = '\0' ;
13                 printf( "%s\n" , result );
14             }
15         }
16     }
17 }
18 int main( void ) {
19     char alphabets[ 3 ] = { 'a' , 'b' , 'c' } , result[ 4 ] ;
20     perm( alphabets , 3 , result );
21 }
```

Loop 1 for slot 1

Loop 2 for slot 2

Loop 3 for slot 3

Permutation – Variations

- **Question 3**:
  - How many **different words of 3 characters (or slots)** can we create with a set of **2 alphabets**?
    - Your answer?
  - Print all the words out…
    - How many loops do you need?
    - How to perform the looping?
    - Let's discuss in the class.

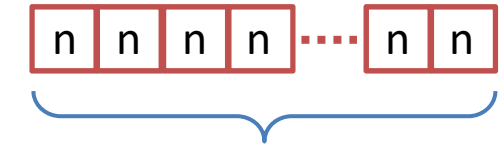
Permutation – Variations

- **Question 4**:
  - How many **different words of 2 characters (or slots)** can we create with a set of **3 alphabets**?
    - Your answer?
  - Print all the words out…
    - How many loops do you need?
    - How to perform the looping?
    - Let's discuss in the class.

Permutation – General case

- **Final Question**:
  - How many **different words of m characters (or slots)** can we create with the set of **n alphabets**?

n^m permutations!



  - Print all the words out…
    - How many layers of loops do you need? **m layers**
    - What if m is an input (i.e., a variable) from the users?

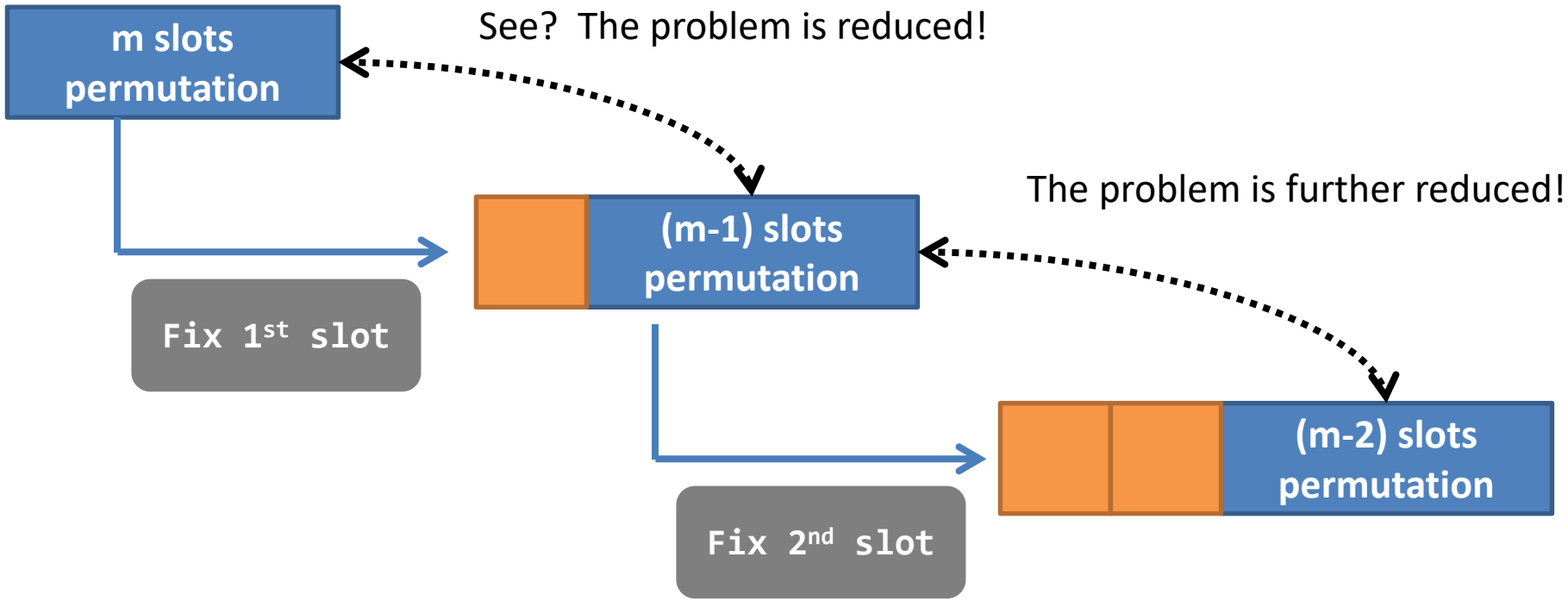
Can your program **produce m layers of loops** while it is running?!

Outline

1. Motivation – Recursion for Permutation
2. **Basic: Permutation using Recursion**
3. Permutation without repetition
4. Permutation without repetition & order
5. Recursion & Brute Force Search

Permutation – Recursion

- Define sub-problems in the permutation process



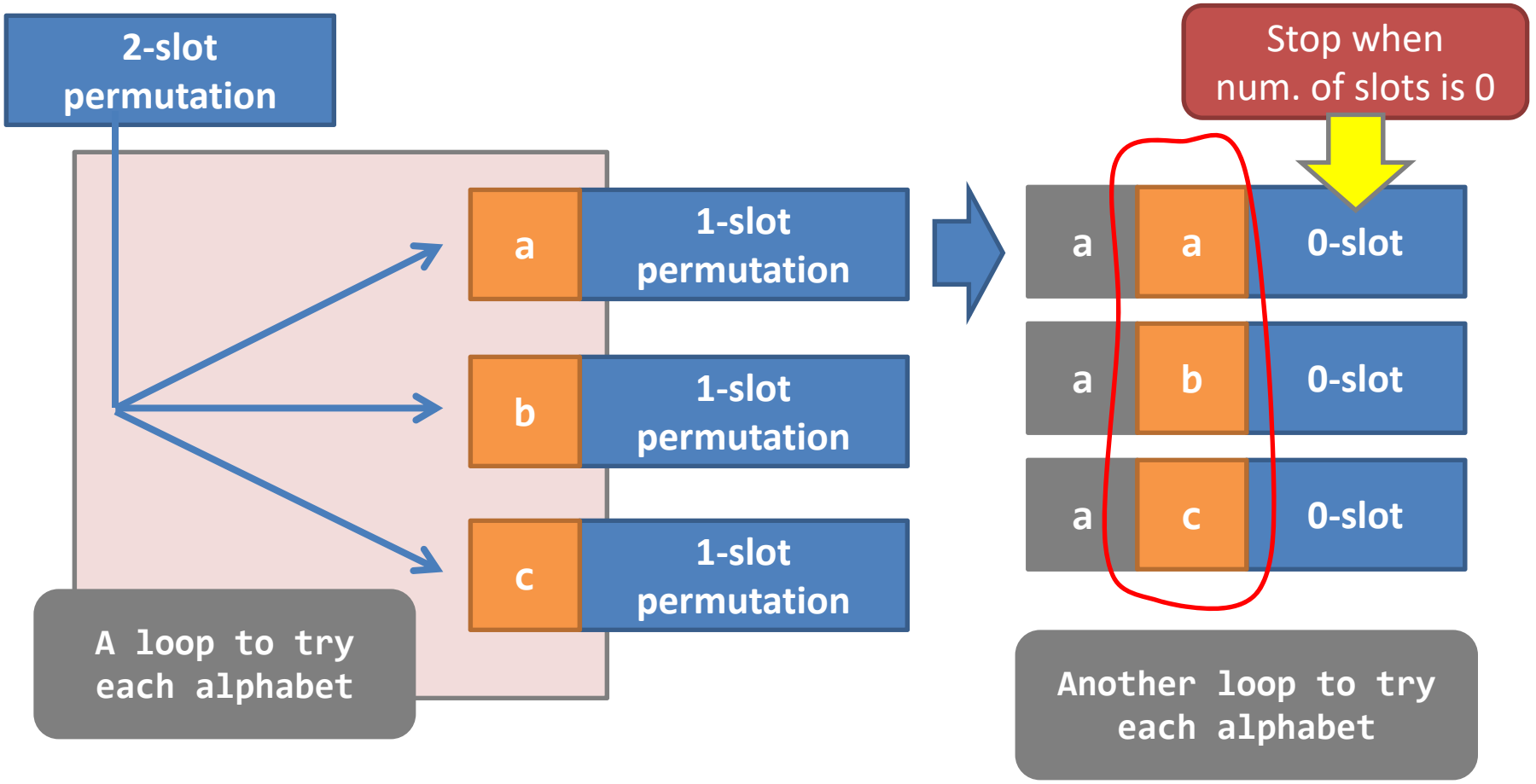
Permutation – Recursion

```
1  void perm( int num_slots , char alphabets [] , int a_len ,
2                             char result    [] , int r_len )
3  {
4      // Q1: what is the termination condition? Where to print?
5
6      result[ ?? ] = ?? ; // Q2: Which position? Which alphabet?
7
8      perm( num_slots - 1 , alphabets , a_len , result , r_len );
9  }
10 int main( void )
11 {
12     char alphabets [ 3 ] = { 'a' , 'b' , 'c' };
13     char result    [ 3 ] ;   // 2 slots + '\0'
14
15     result[2] = '\0' ;
16     perm( 2 , alphabets , 3 , result , 2 );
17 }
```

Every time, reduce it to a smaller sub-problem!

Permutation – Recursion

- Example: words of **2 characters** from **3 alphabets**:



Permutation – Final code

```
1  void perm( int num_slots , char alphabets [] , int a_len ,
2                             char result    [] , int r_len )
3  {
4     if ( num_slots == 0 )  // end?
5     {
6        printf( "%s\n" , result );
7        return ;
8     }
9
10    for ( int i = 0 ; i < a_len ; i++ )
11    {
12       result[ r_len - num_slots ] = alphabets[ i ];
13       perm( num_slots - 1 , alphabets , a_len , result , r_len );
14    }
15 }
16
17 int main( void )
18 {
19    // just the same as the previous code; don't repeat here
20 }
```

Termination condition

Target slot contents varies using a loop!

Permutation – Final code #2

```
1  void perm( int num_slots , char alphabets [] , int a_len ,
2                             char result    [] , int r_len )
3  {
4     if ( num_slots == 0 )  // end?
5        printf( "%s\n" , result );
6     else
7     {
8        for ( int i = 0 ; i < a_len ; i++ )
9        {
10          result[ r_len - num_slots ] = alphabets[ i ];
11          perm( num_slots-1 , alphabets , a_len , result , r_len );
12       }
13    }
14 }
15
16 int main( void )
17 {
18    // just the same as the previous code; don't repeat here
19 }
```

Termination condition

Target slot contents varies using a loop!

Outline

1. Motivation – Recursion for Permutation
2. Basic: Permutation using Recursion
3. **Permutation without repetition**
4. Permutation without repetition & order
5. Recursion & Brute Force Search

Permutation: To repeat or not to repeat?

- Strings of **m characters** from **n alphabets**:

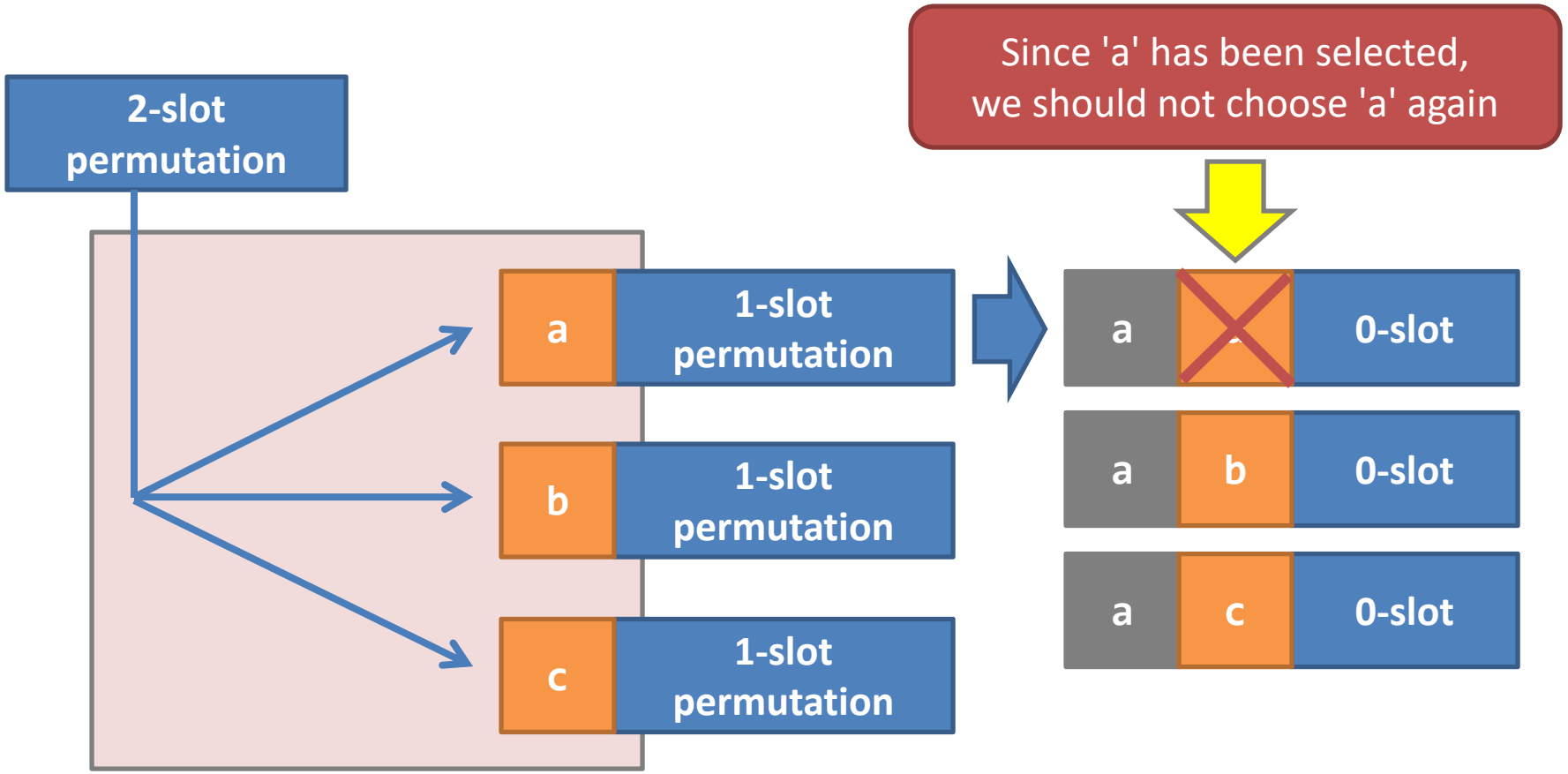
# of Strings = n^m

Permutation: m = 2 and n = 3	
with repetition	without repetition
a a	~~a a~~
a b	a b
a c	a c
b a	b a
b b	~~b b~~
b c	b c
c a	c a
c b	c b
c c	~~c c~~

# of Strings = n!

Case 2: Permutation without repetition

- A new problem!



Permutation without repetition

- After each recursion level, reduce the alphabet set by one!

- To **memorize** the selected alphabets:
  - Let's introduce a new array called "`selected`".
  - All elements of "`selected`" are initially **false** (0).
  - At each recursion, we select the alphabet at index `i` only when "`selected[i] == 0`", i.e., not selected.
  - When the recursion selects the alphabet at index `i`, set "`selected[i] = 1`".

Previous code: Perm. with repetition

```
1  void perm( int num_slots , char alphabets [] , int a_len ,
2                             char result    [] , int r_len )
3
4  {
5     if ( num_slots == 0 )  // end?
6     {
7        printf( "%s\n" , result );
8        return ;
9     }
10    for ( int i = 0 ; i < a_len ; i++ )
11    {
12
13
14
15          result[ r_len - num_slots ] = alphabets[ i ];
16          perm( num_slots - 1 , alphabets , a_len ,
17                                result    , r_len );
18
19    }
20 }
```

How to change this code to avoid repetition?

Permutation without repetition

```
1  void perm( int num_slots , char alphabets [] , int a_len ,
2                             char result    [] , int r_len ,
3                             int  selected  [] )
4  {
5     if ( num_slots == 0 )  // end?
6     {
7        printf( "%s\n" , result );
8        return ;
9     }
10    for ( int i = 0 ; i < a_len ; i++ )
11    {
12       if ( selected[ i ] == 0 )
13       {
14          selected[ i ] = 1 ;
15          result[ r_len - num_slots ] = alphabets[ i ];
16          perm( num_slots - 1 , alphabets , a_len ,
17                                result    , r_len , selected );
18       }
19    }
20 }
21
```

selected[] has the same length as **alphabets[]**, i.e., **a_len**

Use only the alphabets that have not been selected before.

Any error in this code?

Permutation without repetition

```
1  void perm( int num_slots , char alphabets [] , int a_len ,
2                             char result    [] , int r_len ,
3                             int  selected  [] )
4  {
5     if ( num_slots == 0 )  // end?
6     {
7        printf( "%s\n" , result );
8        return ;
9     }
10    for ( int i = 0 ; i < a_len ; i++ )
11    {
12       if ( selected[ i ] == 0 )
13       {
14          selected[ i ] = 1 ;
15          result[ r_len - num_slots ] = alphabets[ i ];
16          perm( num_slots - 1 , alphabets , a_len ,
17                                result    , r_len , selected );
18          selected[ i ] = 0 ; // common mistake: don't miss it!
19       }
20    }
21 }
```

selected[] has the same length as **alphabets[]**, i.e., **a_len**

Use only the alphabets that have not been selected before.

We need to de-select after coming back from perm()

Permutation without repetition

```
1  void perm( int num_slots , char alphabets [] , int a_len ,
2                             char result    [] , int r_len ,
3                             int  selected  [] )
4  {
5     if ( num_slots == 0 ) // end?
6        printf( "%s\n" , result );
7     else
8     {
9        for ( int i = 0 ; i < a_len ; i++ )
10       {
11          if ( selected[ i ] == 0 )
12          {
13             selected[ i ] = 1 ;
14             result[ r_len - num_slots ] = alphabets[ i ];
15             perm( num_slots - 1 , alphabets , a_len ,
16                                   result    , r_len , selected );
17             selected[ i ] = 0 ; // common mistake: don't miss it!
18          }
19       }
20    }
21 }
```

selected[] has the same length as **alphabets[]**, i.e., **a_len**

Use only the alphabets that have not been selected before.

So, we need to de-select it here

Outline

1. Motivation – Recursion for Permutation
2. Basic: Permutation using Recursion
3. Permutation without repetition
4. **Permutation without repetition & order**
5. Recursion & Brute Force Search

Case 3: Combination "n choose m"

- Strings of **m characters** from **n alphabets**:

Permutation: m = 2 and n = 3		
with repetition	without repetition	without repetition and ordered
a a	a b	a b
a b	a c	a c
a c	b a	~~b a~~
b a	b c	b c
b b	c a	~~c a~~
b c	c b	~~c b~~
c a		
c b		
c c		

remove duplication

$_9C_3$ or 9 choose 3

Combination m = 2 and n = 3
a b
a c
b c

# of combinations: $_nC_m$

Any idea to achieve this? How to change the code on previous page?

Previous code: Perm. without repet.

```
1  void perm( int num_slots , char alphabets [] , int a_len ,
2                             char result    [] , int r_len ,
3                             int  selected  [] )
4  {
5     if ( num_slots == 0 ) // end?
6        printf( "%s\n" , result );
7     else
8     {
9        for ( int i = 0 ; i < a_len ; i++ )
10       {
11          if ( selected[ i ] == 0 )
12          {
13             selected[ i ] = 1 ;
14             result[ r_len - num_slots ] = alphabets[ i ];
15             perm( num_slots - 1 , alphabets , a_len ,
16                                   result    , r_len , selected );
17             selected[ i ] = 0 ; // common mistake: don't miss it!
18          }
19       }
20    }
21 }
```

How to change this code to avoid repetition?

Permutation without repetition & order

```
1  void perm( int num_slots , char alphabets [] , int a_len ,
2                             char result    [] , int r_len ,
3                             int  selected  [] , int prev_i )
4  {
5     if ( num_slots == 0 ) // end?
6        printf( "%s\n" , result );
7     else {
8        for ( int i = prev_i+1 ; i < a_len ; i++ )
9        {
10          if ( selected[ i ] == 0 )
11          {
12             selected[ i ] = 1 ;
13             result[ r_len - num_slots ] = alphabets[ i ];
14             perm( num_slots - 1 ,
15                   alphabets , a_len ,
16                   result    , r_len , selected , i );
17             selected[ i ] = 0 ; // common mistake: don't miss it!
18          }
19       }
20    }
21 }
```

prev_i to remember the previous index

Note: in main(), initialize prev_i as -1

But then, can we simplify the code?

Permutation without repetition & order

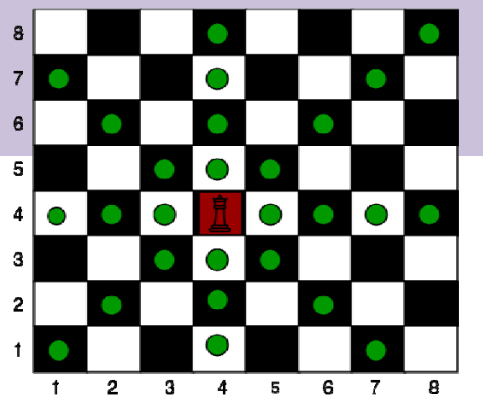
```
1  void perm( int num_slots , char alphabets [] , int a_len ,
2                             char result    [] , int r_len ,
3                             int  prev_i )
4  {
5     if ( num_slots == 0 ) // end?
6        printf( "%s\n" , result );
7     else {
8        for ( int i = prev_i+1 ; i < a_len ; i++ )
9        {
10          result[ r_len - num_slots ] = alphabets[ i ];
11          perm( num_slots - 1 ,
12                alphabets , a_len ,
13                result    , r_len , i );
14       }
15    }
16 }
```

No need the “selected” array any more, right?

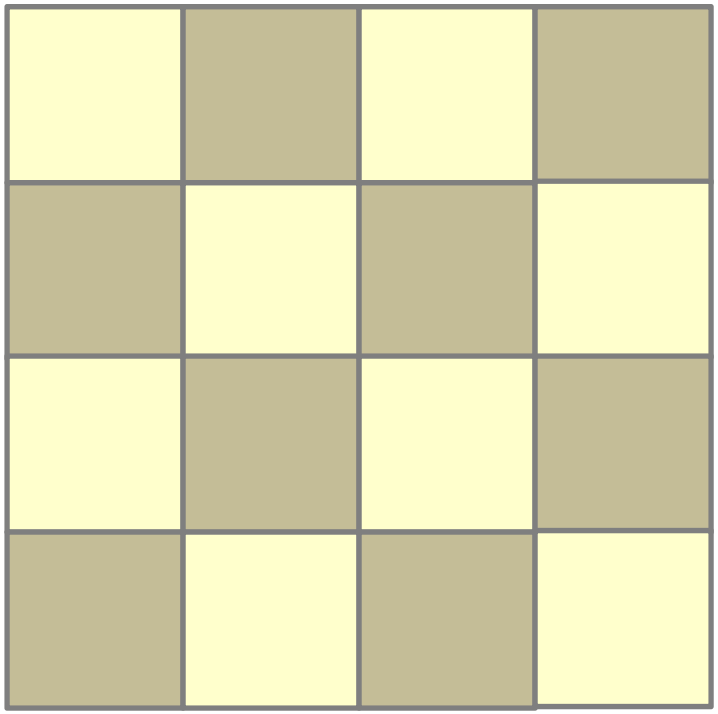
Outline

1. Motivation – Recursion for Permutation
2. Basic: Permutation using Recursion
3. Permutation without repetition
4. Permutation without repetition & order
5. **Recursion & Brute Force Search**

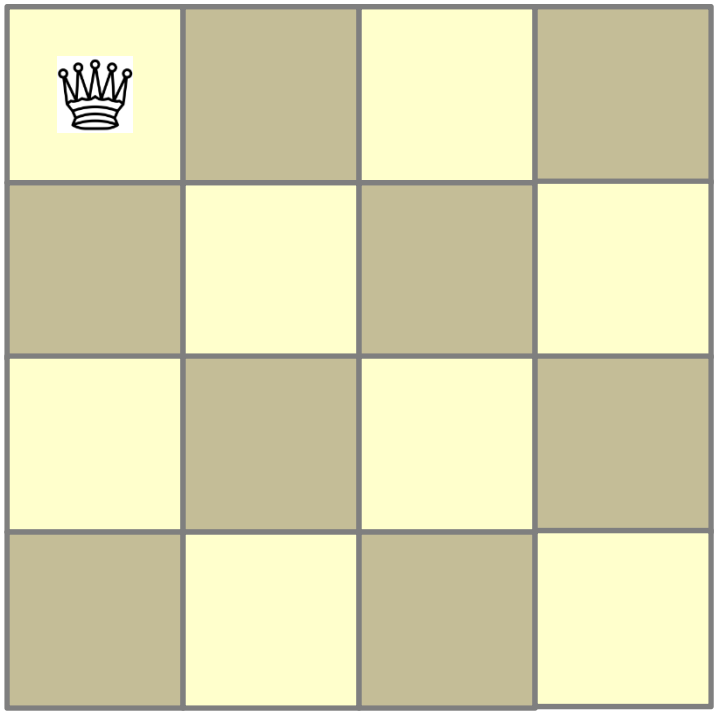
Decision problem



- On a 4x4 grid, can you place ***4 queens***
  - So that they cannot attack one another?



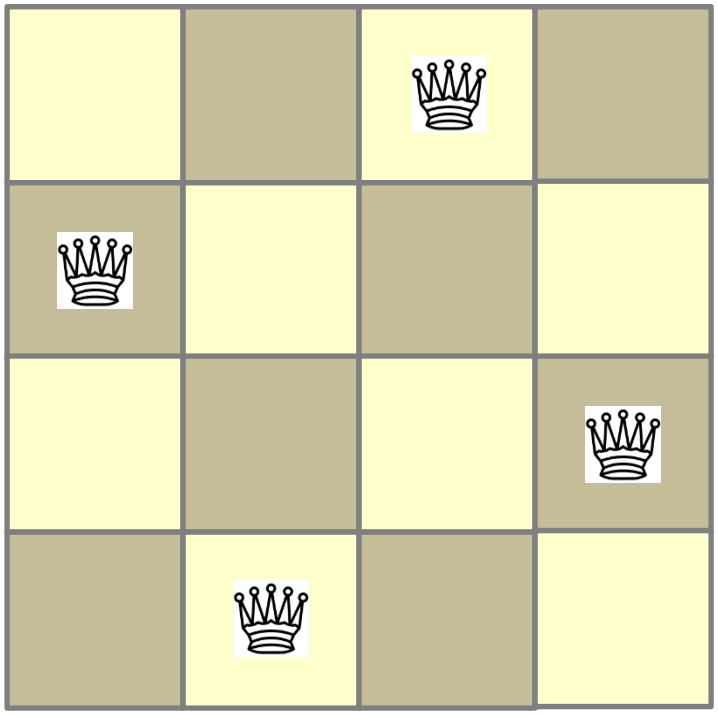
Start with an empty grid



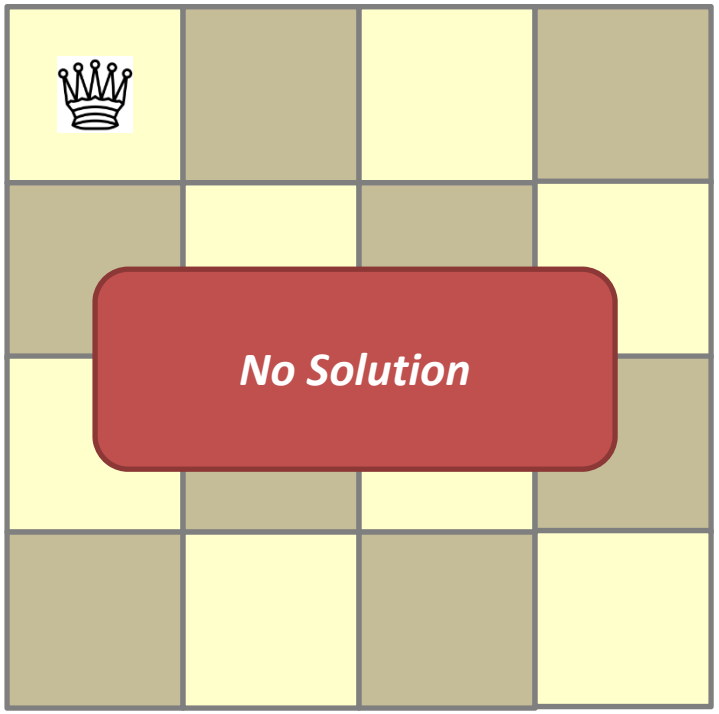
Start with a partial solution

Decision problem

- On a 4x4 grid, can you place ***4 queens***
  - So that they cannot attack one another?



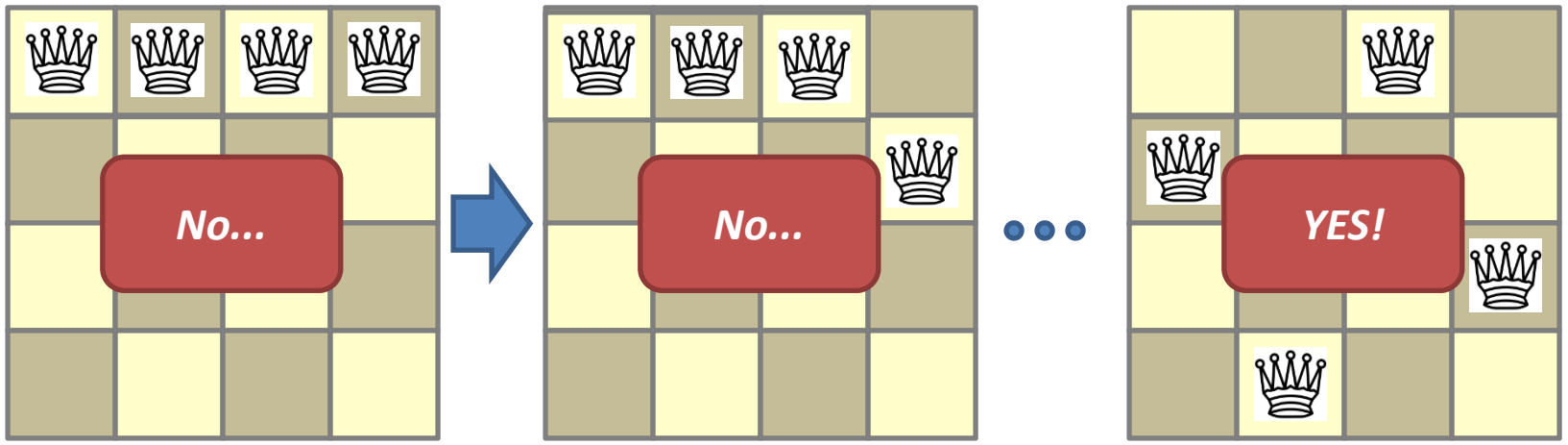
Start with an empty grid



Start with a partial solution

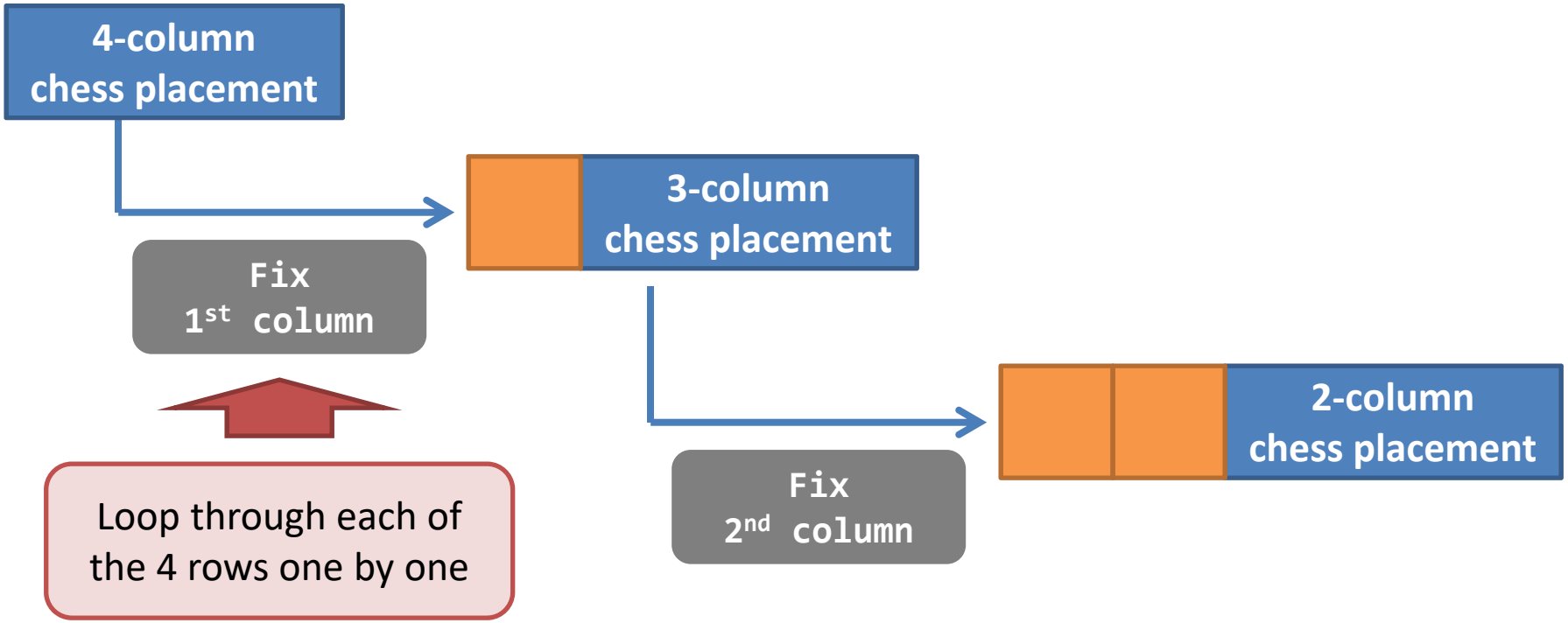
Decision problem

- This is a classical problem called the **N-queen problem**.
  - One approach to solve it is by **exhaustively trying every possible layout** (or the solution candidate).
  - We call such approach ***brute-force search***.



How to perform brute force?

- We may define a **recursive solution** as "*filling a queen in each column one at a time*"



How to perform brute force?

```
1  void brute_force( int column , char map[ 4 ][ 4 ] )
2  {
3      if ( column == 4 )
4      {
5          if ( is_valid_solution( map ) )
6              print_map( map );
7          return ;
8      }
9
10     for ( int i = 0 ; i < 4 ; i++ )
11     {
12         map[ i ][ column ] = 1 ;
13         brute_force( column + 1 , map );
14         map[ i ][ column ] = 0 ;
15     }
16 }
```

Termination Condition

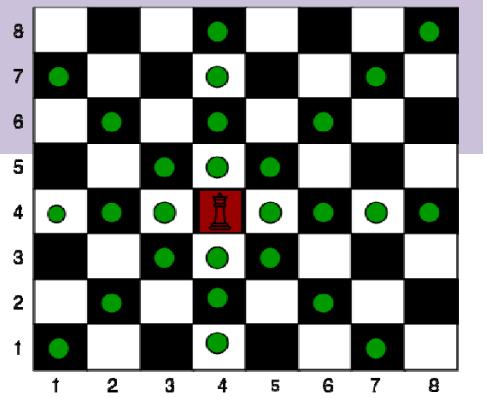
At "**column**", try each row!
To place a queen, set
map[i][column] = 1 ;

Any idea to improve?

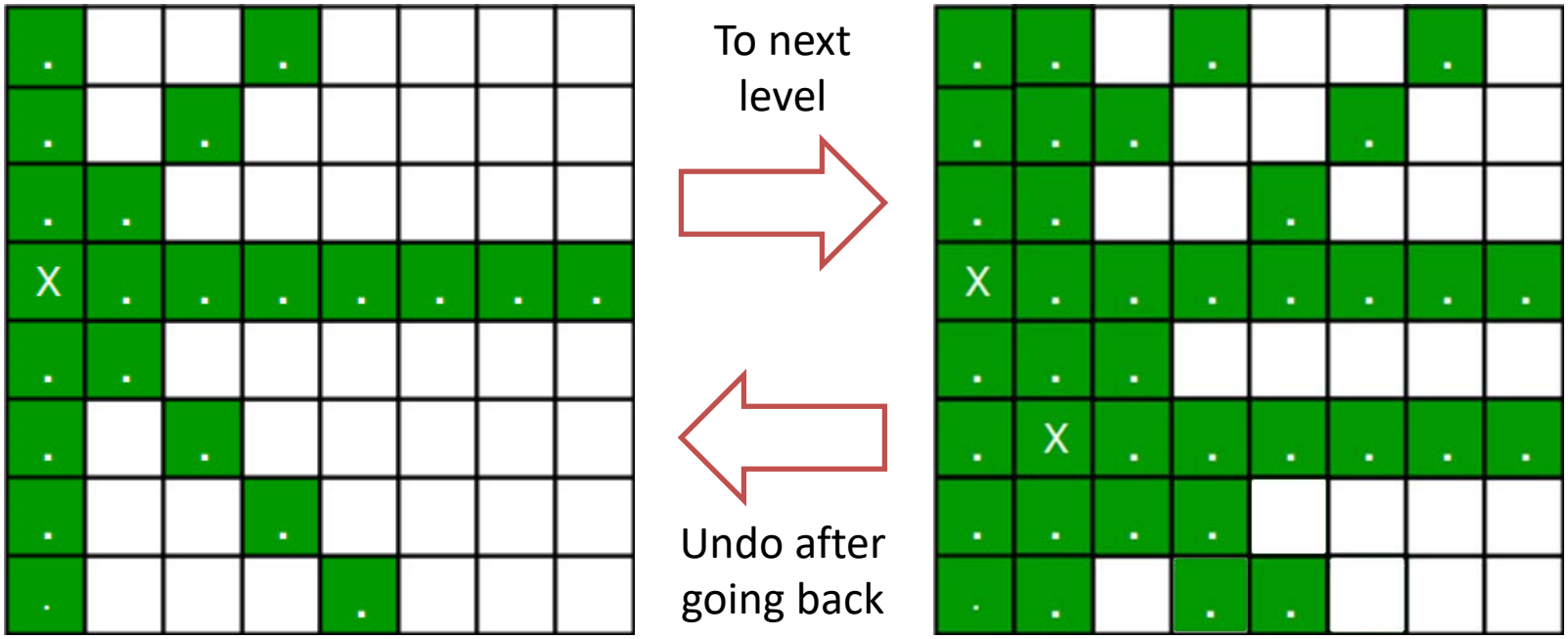
E.g., use “extra variables” to denote “feasible placement locations” after each move! Similar to the “Sieve of Eratosthenes” algorithm in Lec 5

Start the solution: "brute_force(0 , map);"

By means of “memorization”



Use an extra 2D array (all zeros initially) to keep track of all attackable positions after each move! **Add one to these locations** and **subtract one when “undo”**



Summary

- Hope you enjoyed the beauty and the simplicity of recursive algorithms.
- Three basic cases:
  - Permutation with repetitions: ab, aa, ba, bb
  - Permutation without repetitions: ab, ba
  - Combinations (no repeat and ordered): ab

  The keys are **how to define the subproblem**, **how to pass and return data**, and the **termination condition**