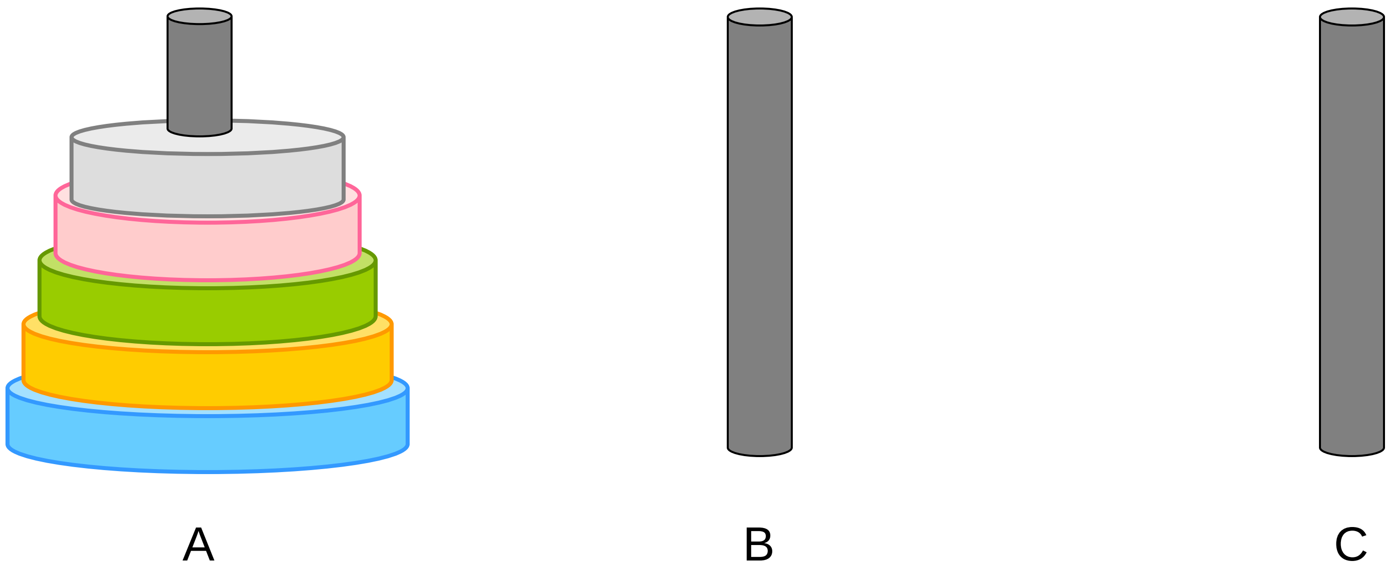
EE2331 Data Structures and Algorithms

Recursion

Towers of Hanoi

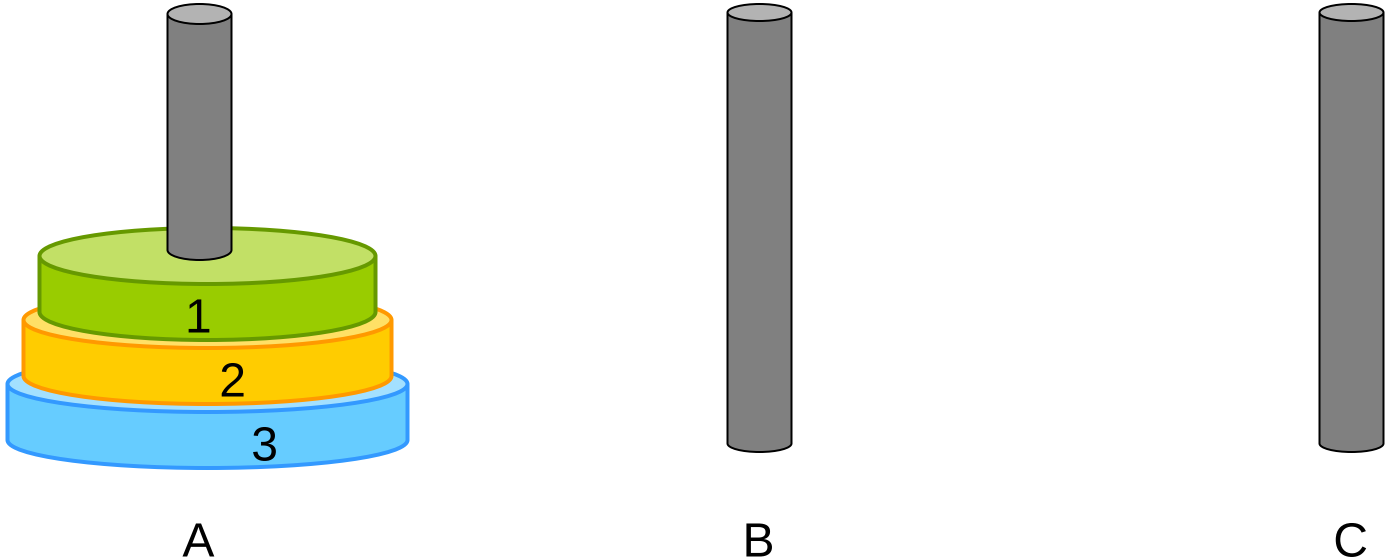


The initial setup of the Towers of Hanoi

Towers of Hanoi

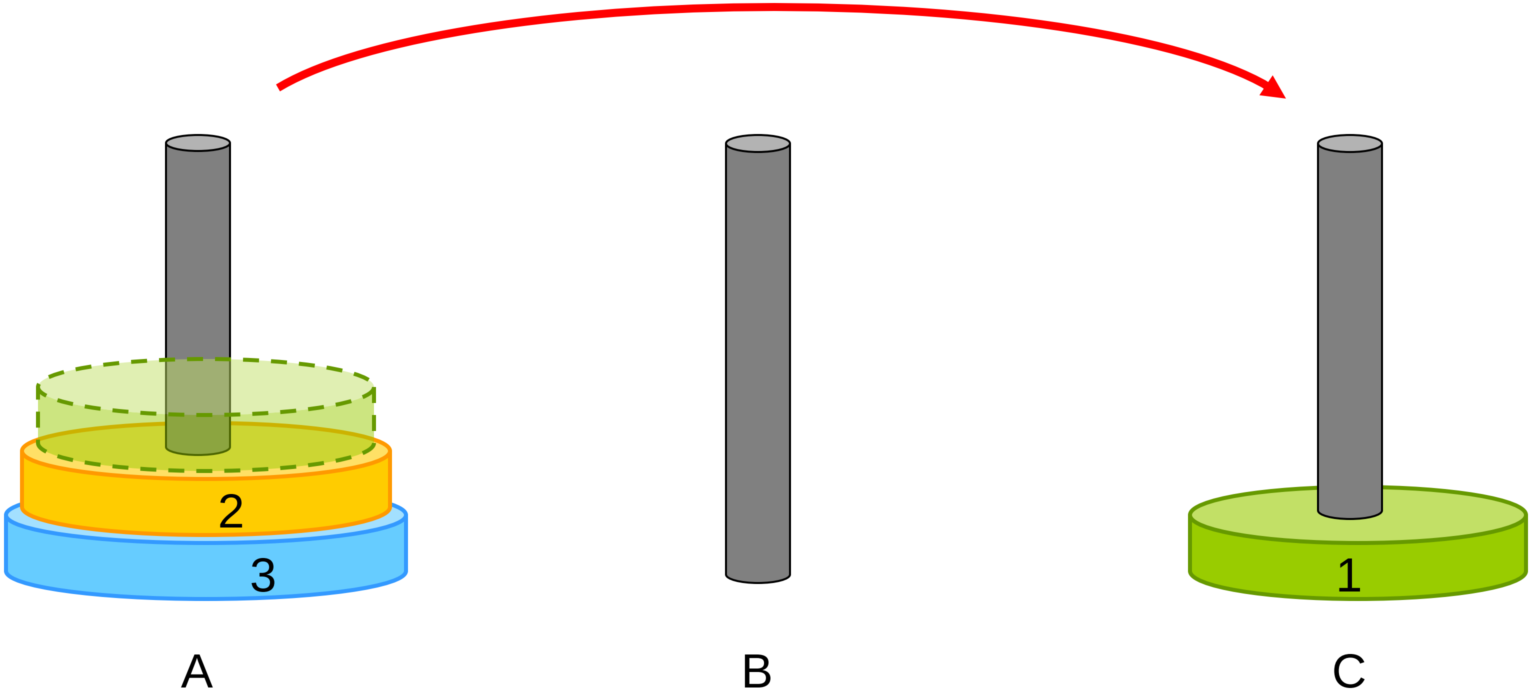
- Three pegs, named as *A*, *B*, and *C*.
- *n* disks each with different diameters, stacked in order of diameter.
- Disk with larger diameter always below disk with smaller diameter
- Only the top disk on a peg can be moved to another peg (but a larger disk cannot never rest on a smaller disk)
- The disks are initially placed in peg *A*
- The task is: to move all *n* disks to peg *C* (in smallest no. of move)

Towers of Hanoi



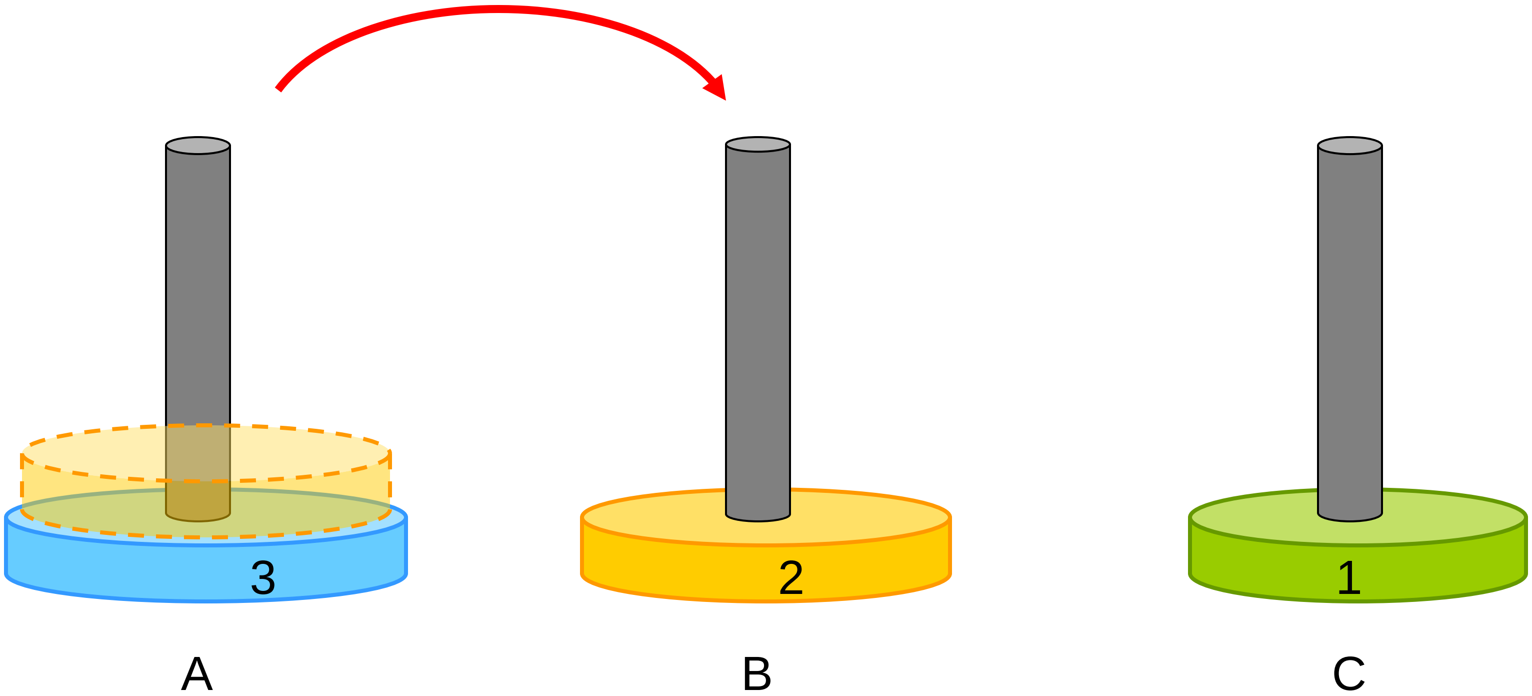
Let’s try a simple case first

Towers of Hanoi



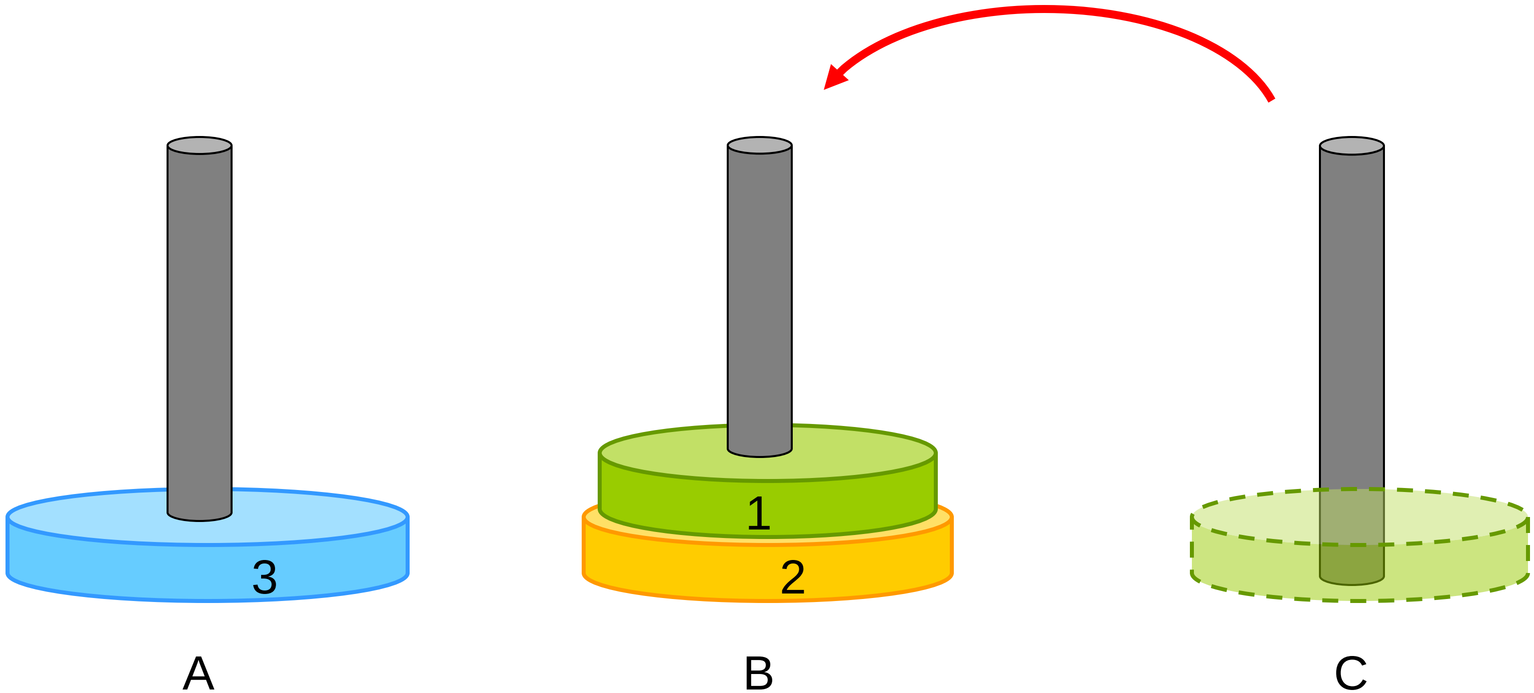
Move disk 1 from A to C

Towers of Hanoi



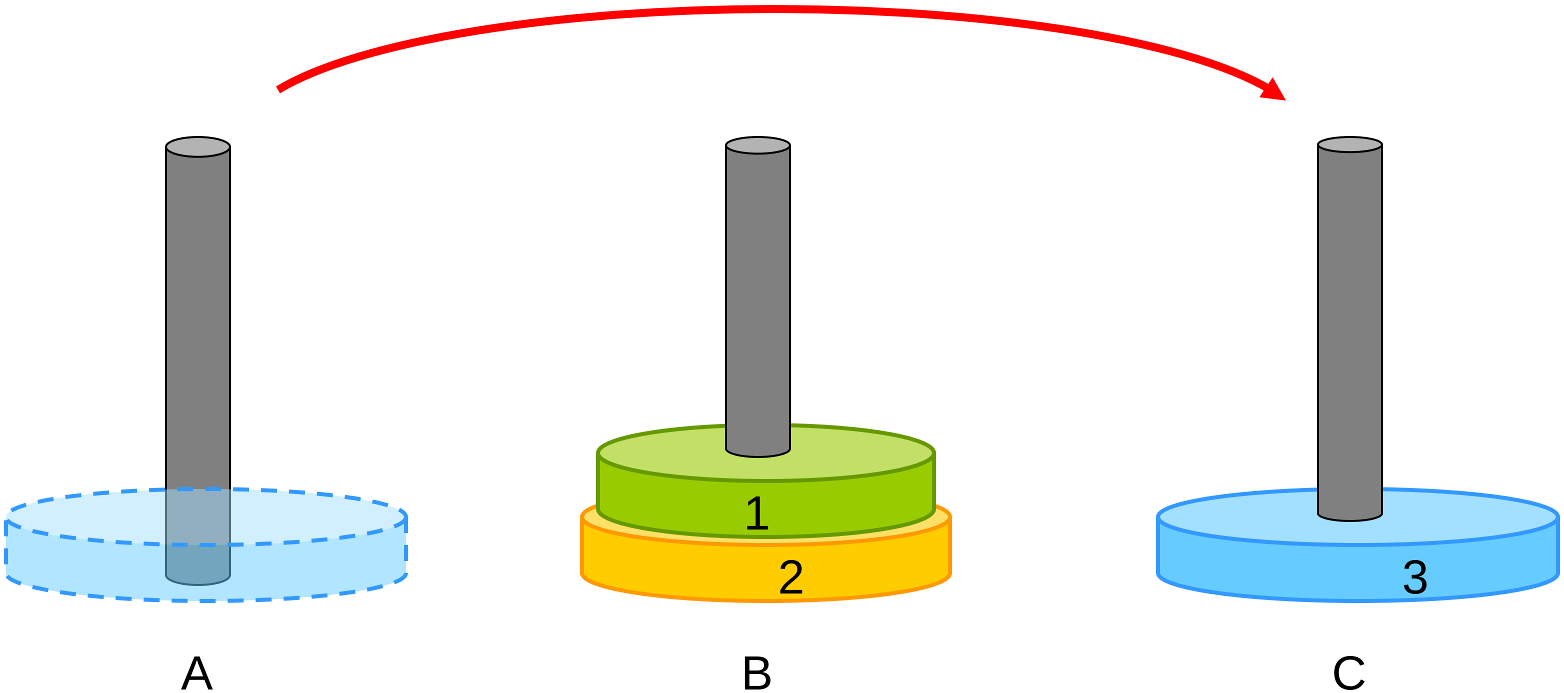
Move disk 2 from A to B

Towers of Hanoi



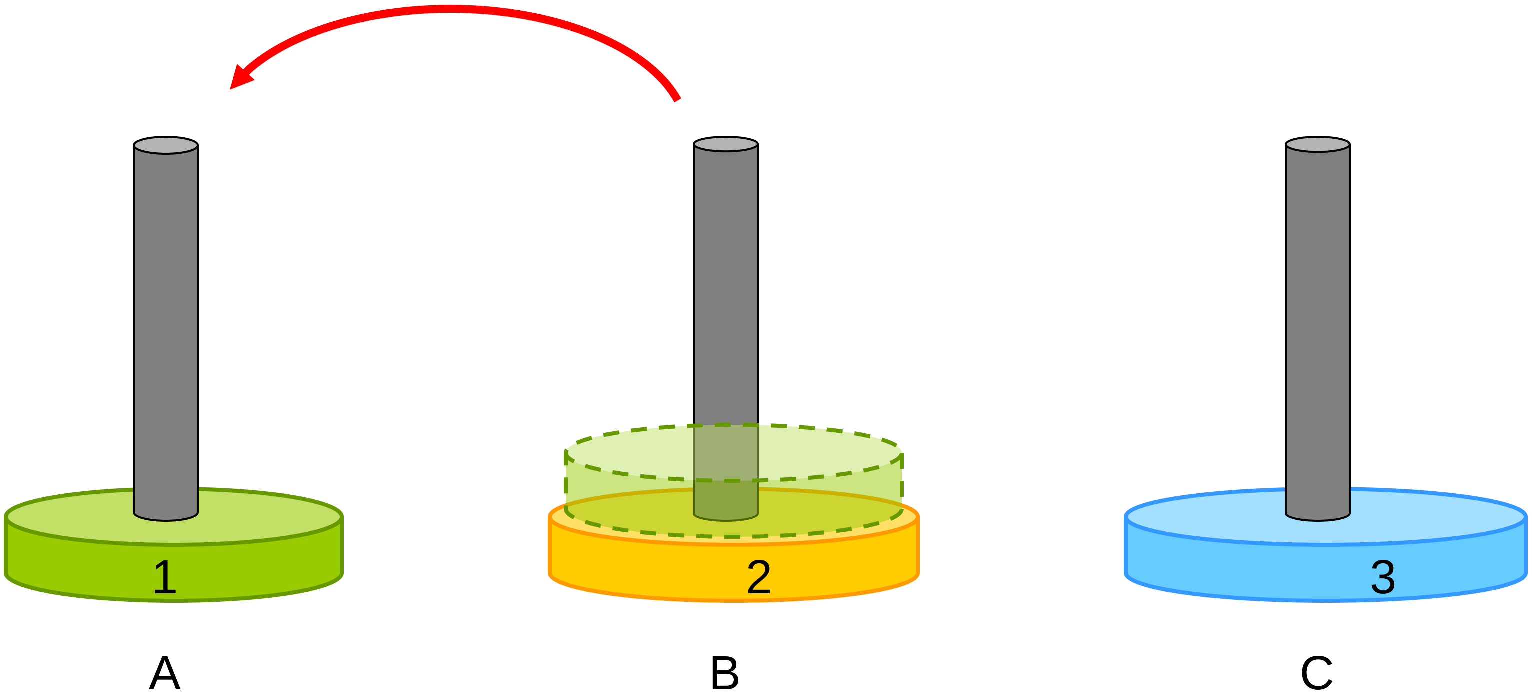
Move disk 1 from C to B

Towers of Hanoi



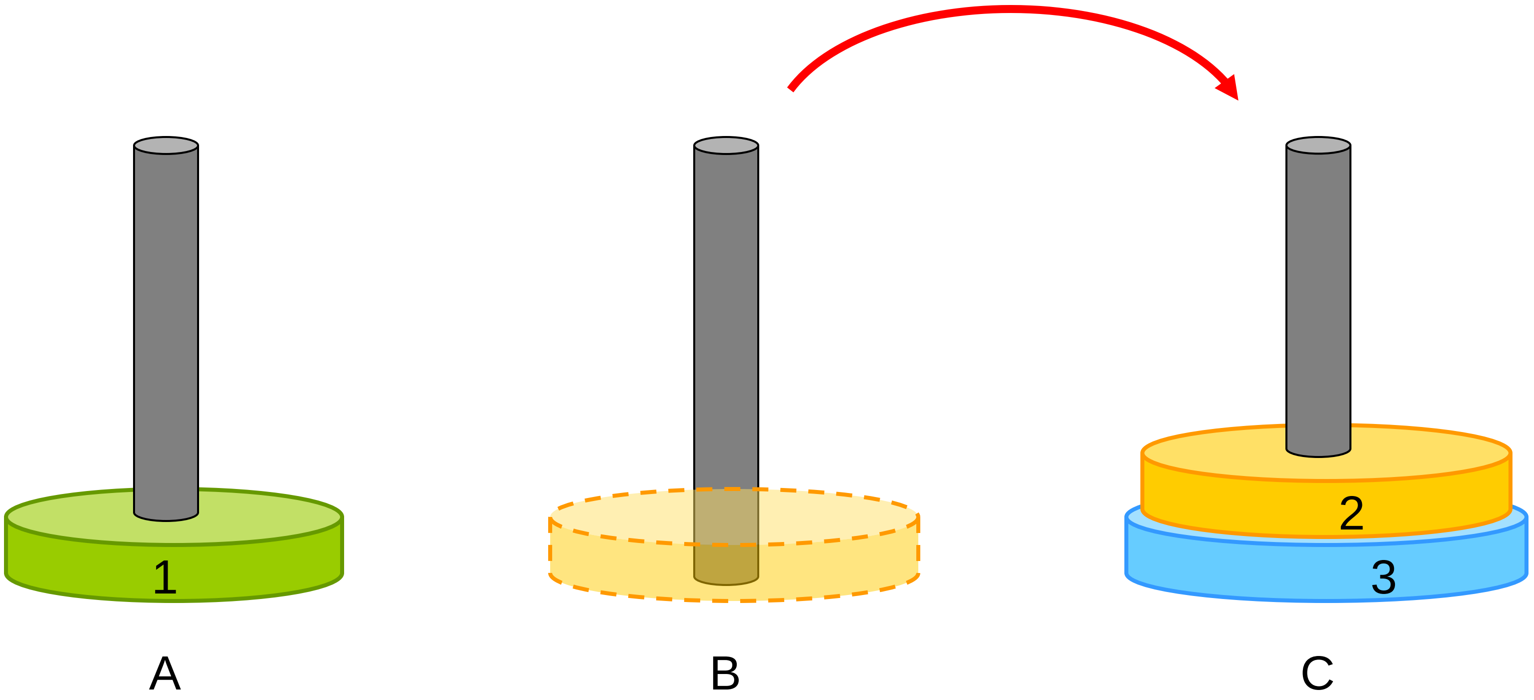
Move disk 3 from A to C

Towers of Hanoi



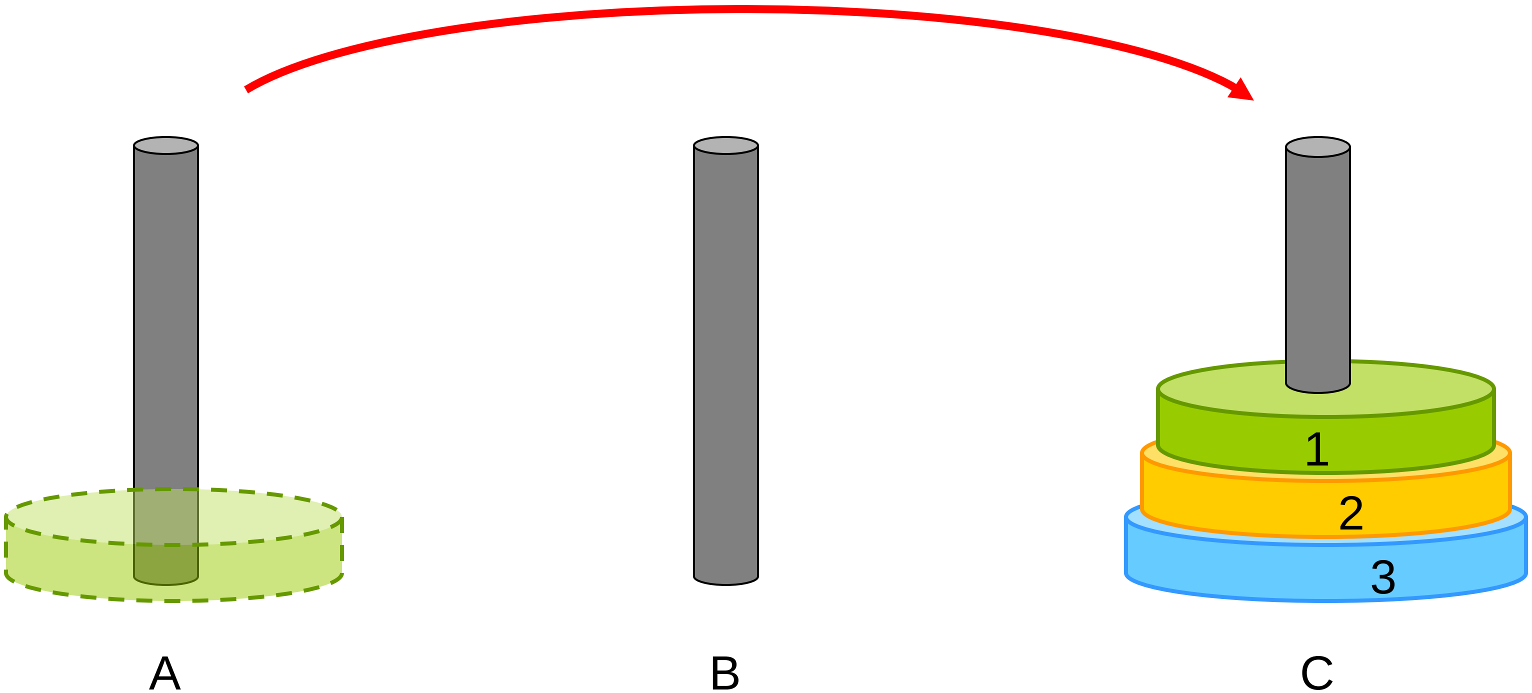
Move disk 1 from B to A

Towers of Hanoi



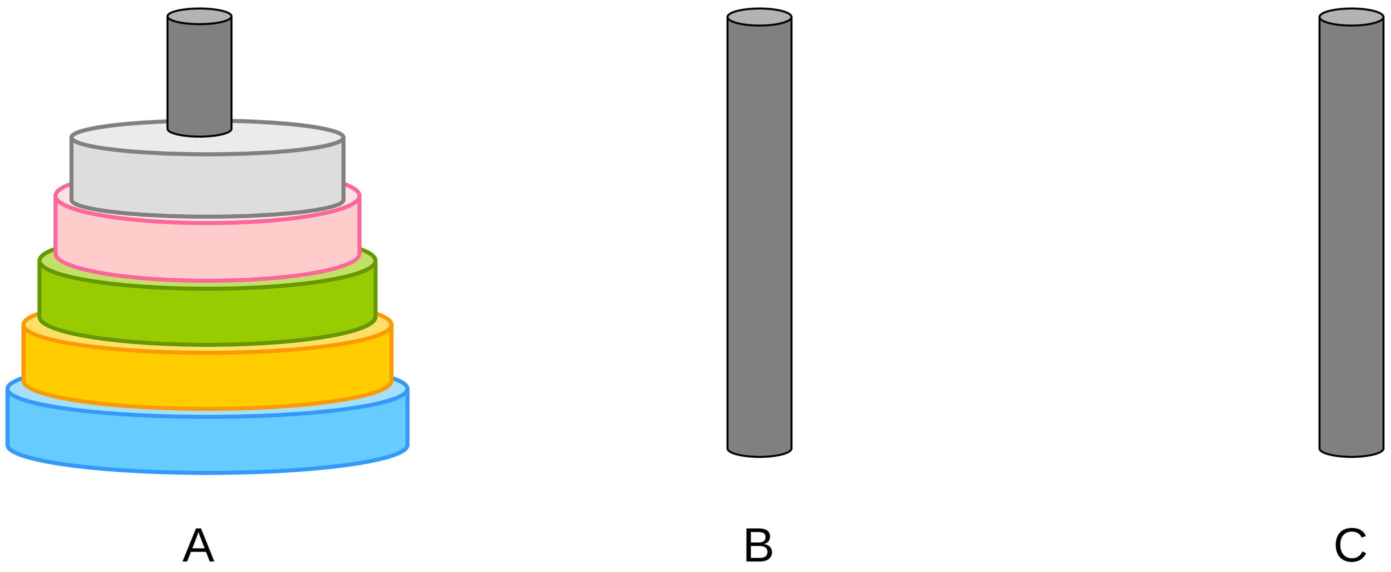
Move disk 2 from B to C

Towers of Hanoi



Finally, move disk 1 from A to C

Towers of Hanoi



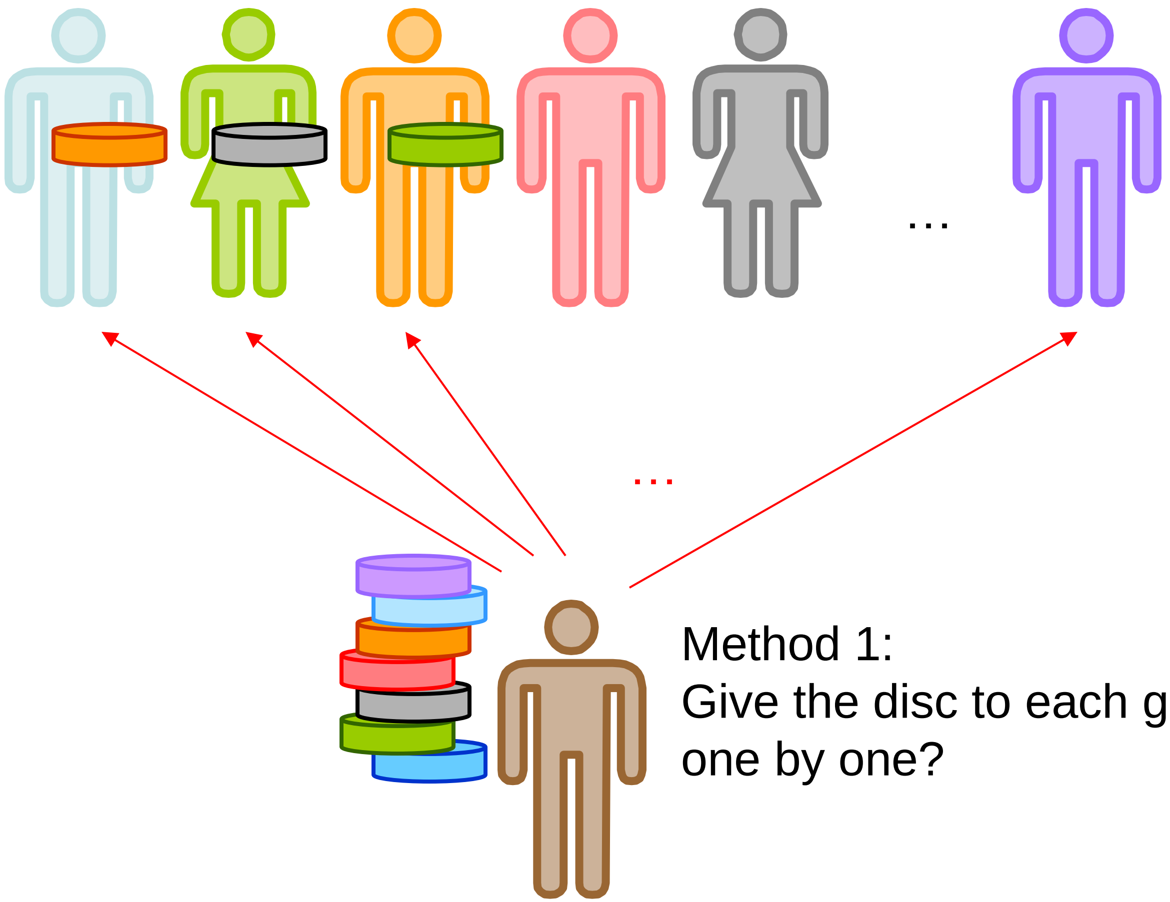
How about if the no. of disk increased?

Any systemic procedure?
(revisit this problem at the end of this topic)

Introduction

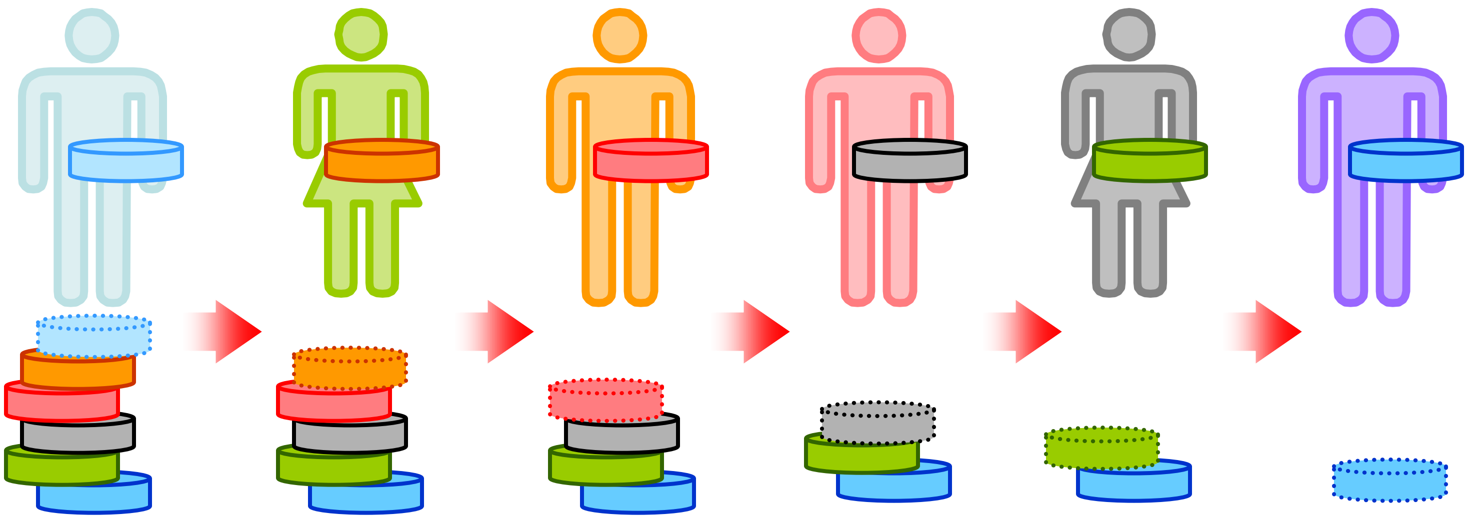
- Recursion is a powerful and elegant algorithm in solving complex problems. It usually results in more “clean” code that is easier to understand
- For simple problem, recursion is a kind of overkill
- Daily life problems solved by recursion
  - Puzzles
  - Distributing quiz papers
    - Imagine that in a party, there are many guests. You are the host and you have to give a paper disc to each guests.
  - You may:
    - Give the disc to each guest one by one, or
    - Pick up a disc and pass the rest to one of the guest. That guest then pick up a disc and pass the rest to another guest until everyone has a disc finally

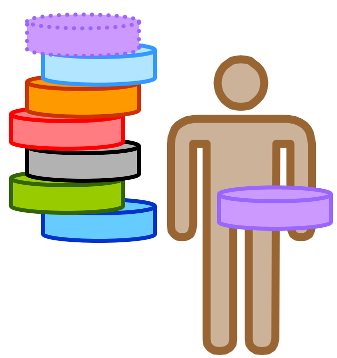
Introduction



Method 1:
Give the disc to each guest one by one?

Introduction





Method 2:
Pick up a disc and pass the rest (n-1) to the next guest?

Introduction

- Method 1 – Iteration (Pseudo Code):

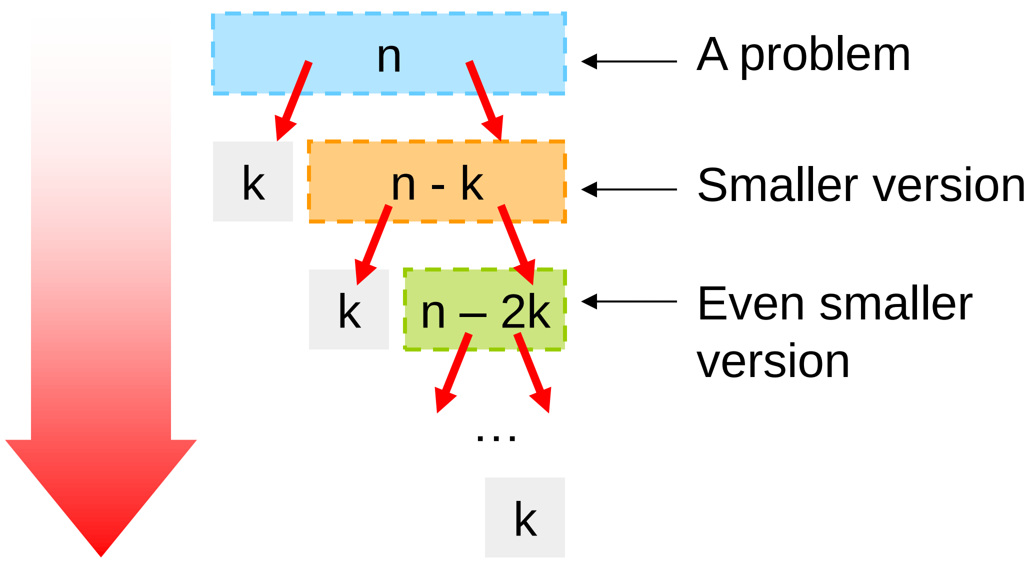
```
distributeDiscs(guests[], discs[]) {
    for each guest in guests[]
        take one disc from discs[]
        give disc to guest
}
```

- Method 2 – Recursion (Pseudo Code):

```
distributeDiscs(guests[], discs[]) {
    if (all guests have disc)
        return

    take one disc from discs[]
    give disc to guest
    distributeDiscs(guests[], discs[])    // with one less guest and disc
}
```

Recursion

- Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first
- Recursion is a technique that solves a problem by solving a smaller problem of the same type



Recursion

- In C++, a function may call itself either directly or indirectly through other calls.
- When a function call itself recursively, each invocation gets a fresh set of all automatic (local) variables, independent of the previous set.
- Recursion is good when the problem is recursively defined, or when the data structure that the algorithm operates on is recursively defined.

Two Essential Steps

- Creating a recursive procedure essentially requires defining two things:
- **Base Case**
  - You must have some base cases, which can be solved without recursion
- **Recursive Case**
  - The cases that are to be solved recursively, the recursive call must always be to a case that makes progress toward a base case

Factorial Function

- *n factorial*, $n!$, is defined as the product of all integers between n and 1

- $n! = n \times (n - 1) \times (n - 2) \times \ldots \times 1$
- $0! = 1$ (the base case)
- $1! = 1$
- $2! = 2 \times 1 = 2$
- $3! = 3 \times 2 \times 1 = 6$
- …

Factorial Function

- $n! = n \times (n-1) \times (n-2) \times \ldots \times 1$

```
int factorial(int n) {
    int result = 1;
    while (n > 1)
        result *= n--;
    return result;
}
```

Loop n-1 times

Time Complexity:

O(n)

Space Complexity:

2 variables (n and *result*) throughout the whole function

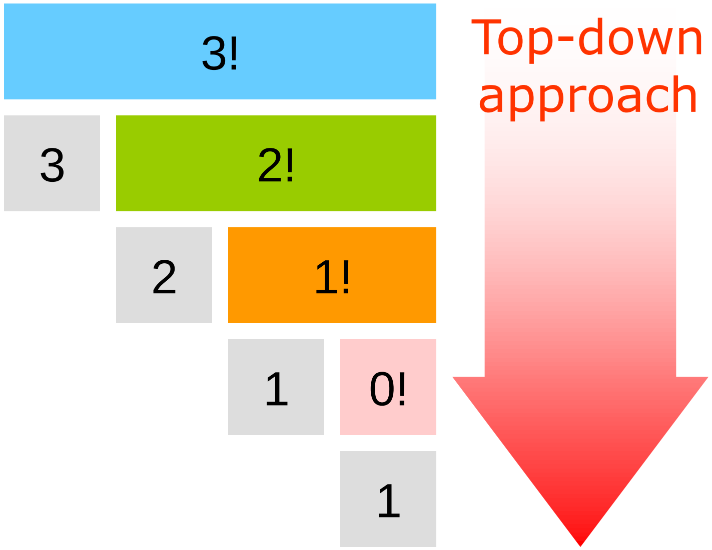
= O(1)

(i.e. independent of the size of n)

Factorial Function

- $n! = n \times (n - 1) \times (n - 2) \times \ldots \times 1$ (closed-form)
- $n! = n \times (n - 1)!$ (recursive form)

- 3! = 3 x 2!
- 2! = 2 x 1!
- 1! = 1 x 0!
- 0! = 1 (base case)



Factorial Function

- $n! = n \times (n - 1)!$

```
int factorial(int n) {
    //precondition: n >= 0
    if (n == 0) return 1;
    return (n * factorial(n - 1));
}
```

Terminate condition (base case, not solved by recursion)

Invariant: as $n > 0$, so $n - 1 \geq 0$
Therefore, factorial(n-1) returns (n-1)! correctly

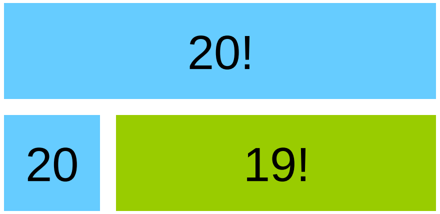
```
int factorial(int n) {
    return (n == 0? 1: n * factorial(n - 1));
}
```

Factorial Function: Example

- Calling factorial(20)

```
int factorial(int n) { n = 20
    if (n == 0) return 1;
    return (n * factorial(n – 1));
}
```

Space requirement: allocated one integer (int n) through out the whole function



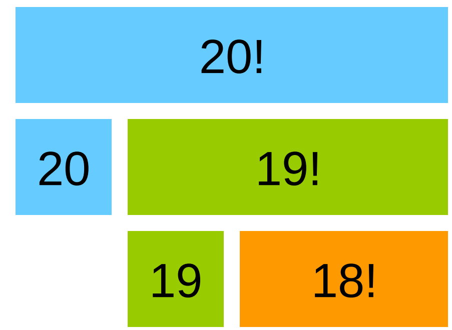
Factorial Function: Example

- Calling factorial(20)

```
int factorial(int n) {  n = 20
    if (n == 0) return 1;
    return (n * factorial(n – 1));
}
```

```
int factorial(int n) {  n = 19
    if (n == 0) return 1;
    return (n * factorial(n – 1));
}
```

Another integer (int n) being allocated in this function (i.e. totally 2 integers in memory)



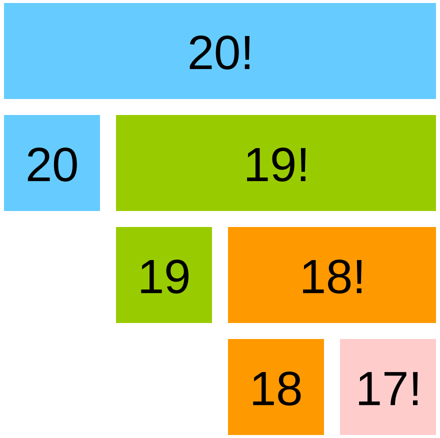
Factorial Function: Example

- Calling factorial(20)

```
int factorial(int n) {    n = 20
    if (n == 0) return 1;
    return (n * factorial(n – 1));
}
```

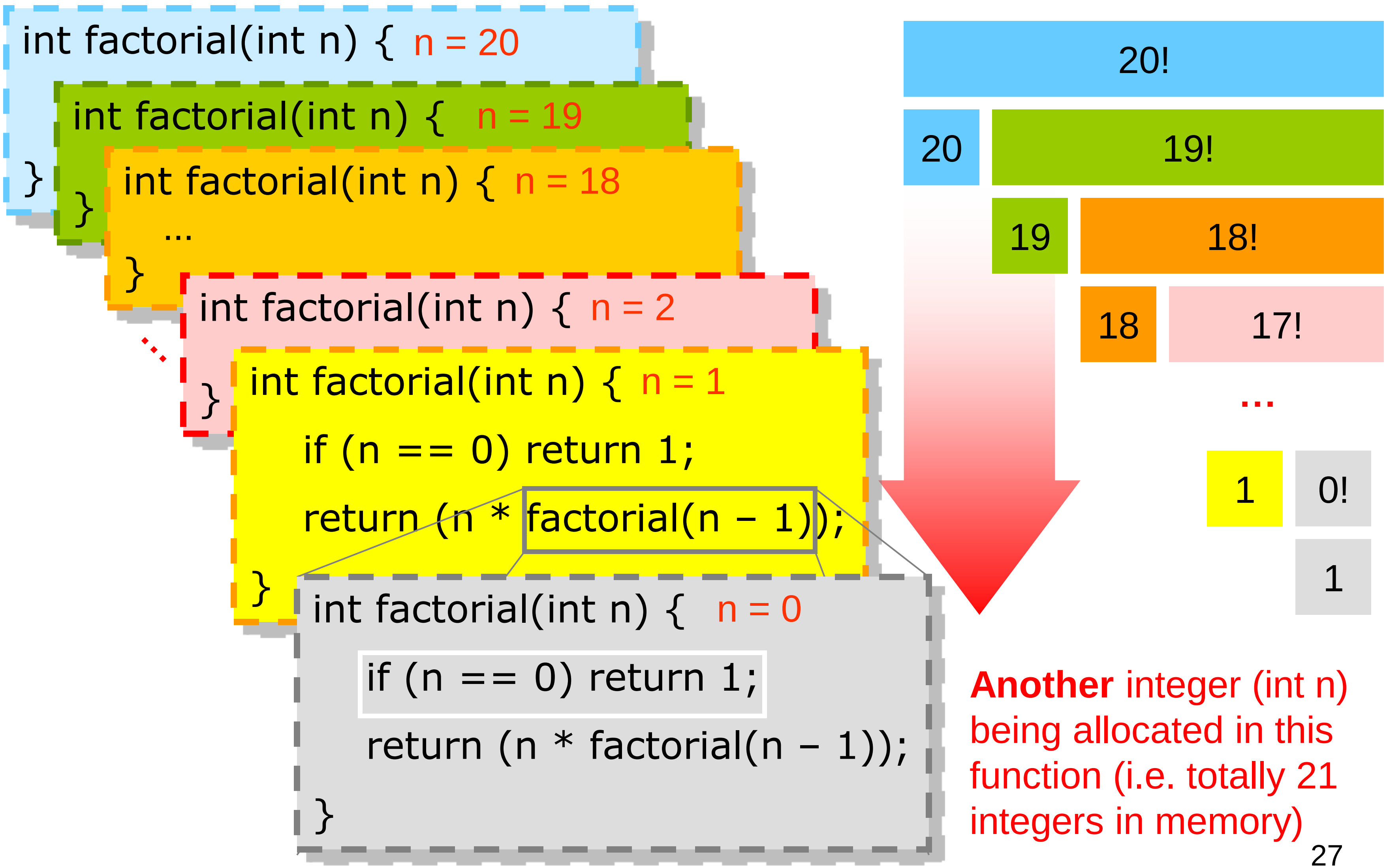
```
int factorial(int n) {    n = 19
    if (n == 0) return 1;
    return (n * factorial(n – 1));
}
```

```
int factorial(int n) {    n = 18
    if (n == 0) return 1;
    return (n * factorial(n – 1));
}
```



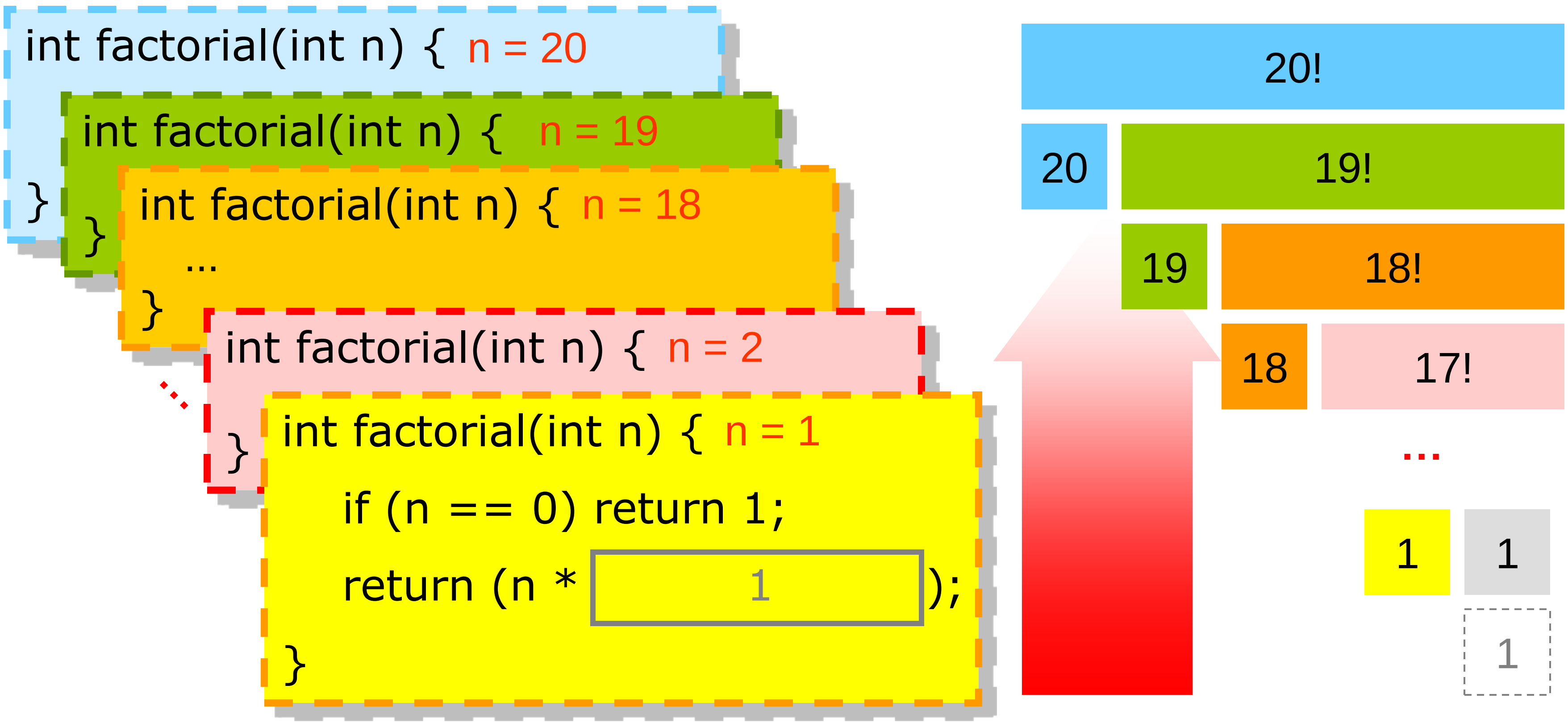
Another integer (int n) being allocated in this function (i.e. totally 3 integers in memory)

Factorial Function: Example



Another integer (int n) being allocated in this function (i.e. totally 21 integers in memory)

Factorial Function: Example

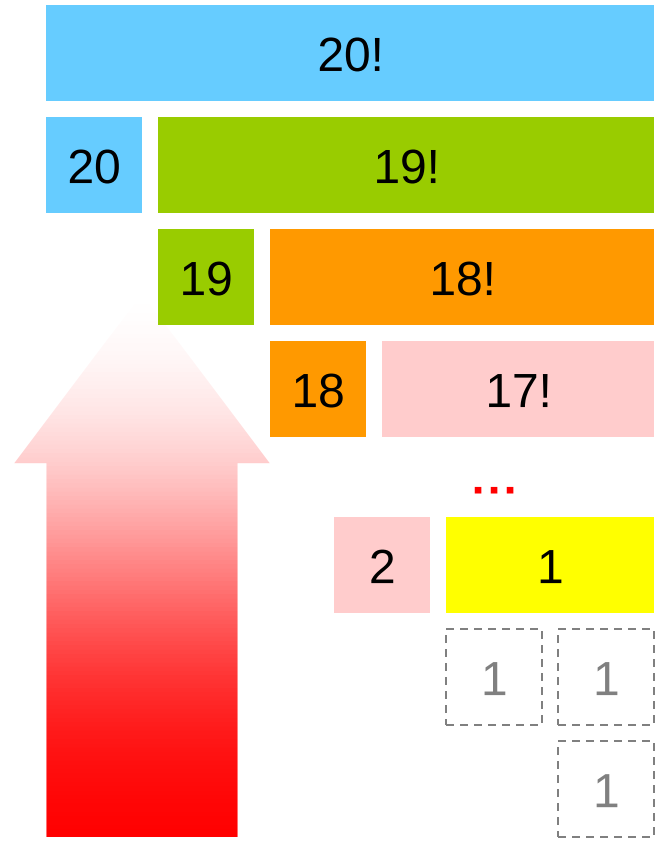


The function of n = 0 returns 1
and now totally only 20 integers
in memory

Factorial Function: Example

```
int factorial(int n) {   n = 20
}
  int factorial(int n) {   n = 19
  }
    int factorial(int n) {   n = 18
      ...
    }
      int factorial(int n) {   n = 2
        if (n == 0) return 1;
        return (n * 1);
      }
```

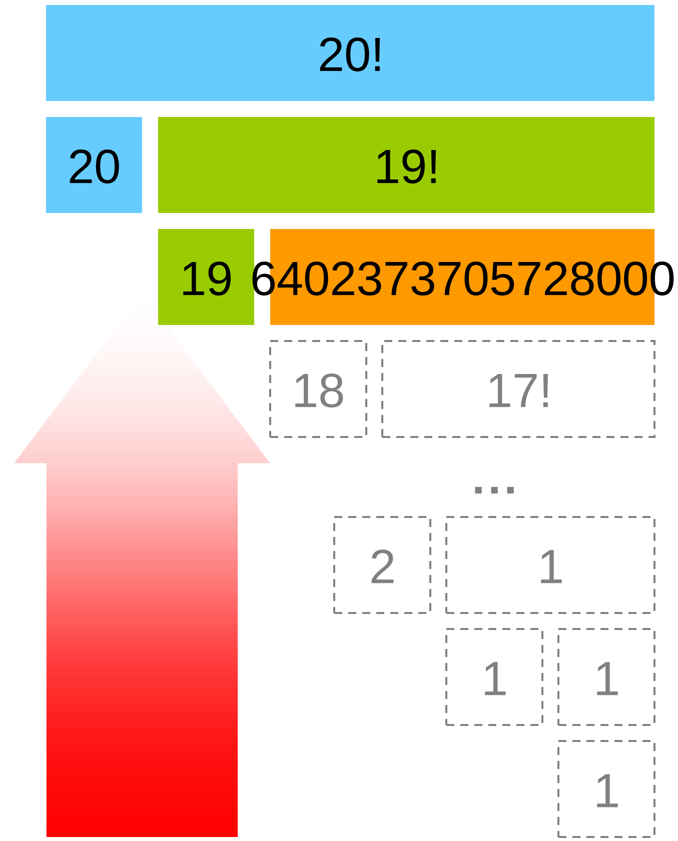
The function of n = 1 returns 1
and now totally only 19 integers
in memory



Factorial Function: Example

```
int factorial(int n) {   n = 20
}
```

```
int factorial(int n) {   n = 19
    if (n == 0) return 1;
    return (n * 6402373705728000);
}
```

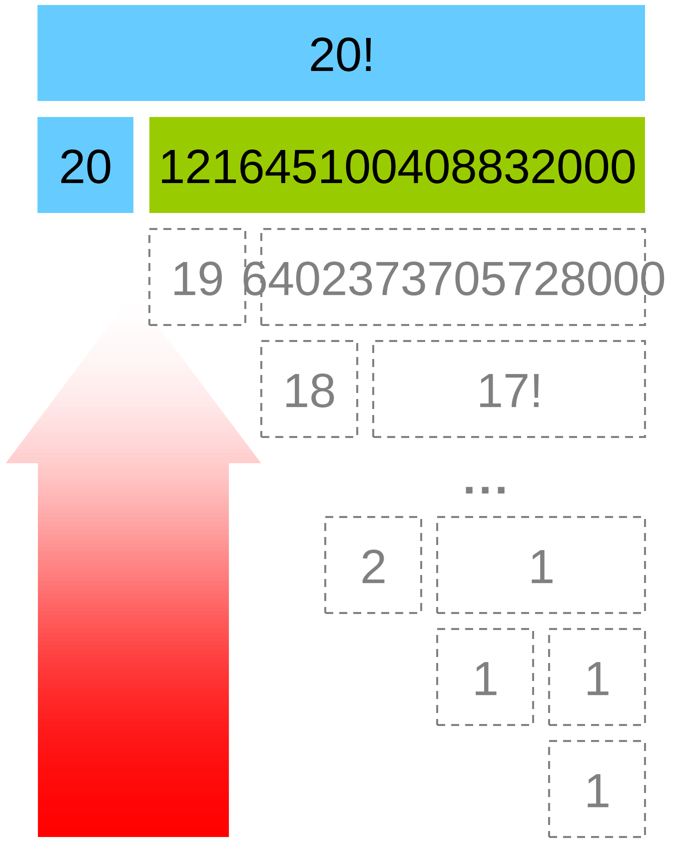


The function of n = 18 returns 6402373705728000 and now totally only 2 integers in memory

Factorial Function: Example

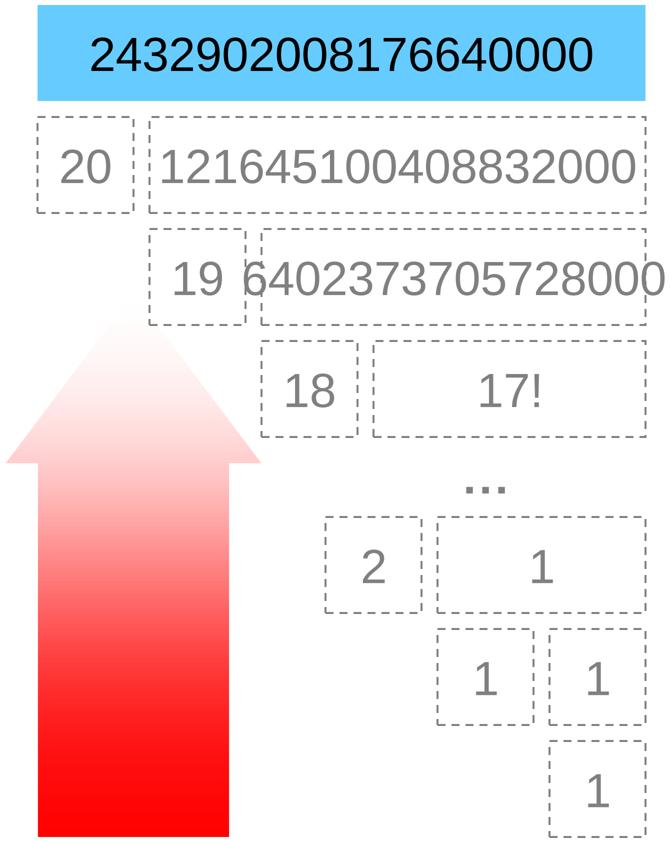
```
int factorial(int n) {   n = 20
    if (n == 0) return 1;
    return (n * 121645100408832000);
}
```

The function of n = 19 returns 121645100408832000 and now totally only 1 integer in memory



Factorial Function: Example

The function of n = 20 returns
2432902008176640000 and
now no integers in memory



Space Complexity =
Time Complexity =

Time Complexity

```
int factorial(int n) {
1    if (n == 0) return 1;
2    return (n * factorial(n – 1));
}
```

- Let T(n) be the running time of input size equals to n
- The running time for line 1 is a constant c_1
- The running time for line 2 is equal to a constant c_2 + T(n-1)

$$T(n) = \begin{cases} c_1 & , n = 0 \\ c_2 + T(n-1) & , n \geq 1 \end{cases}$$

$$For\ n \geq 1,$$

$$T(n) = 2c_2 + T(n-2)$$

$$T(n) = 3c_2 + T(n-3)$$

$$\vdots$$

$$T(n) = n \cdot c_2 + T(0)$$

$$T(n) = n \cdot c_2 + c_1$$

$$T(n) = O(n)$$

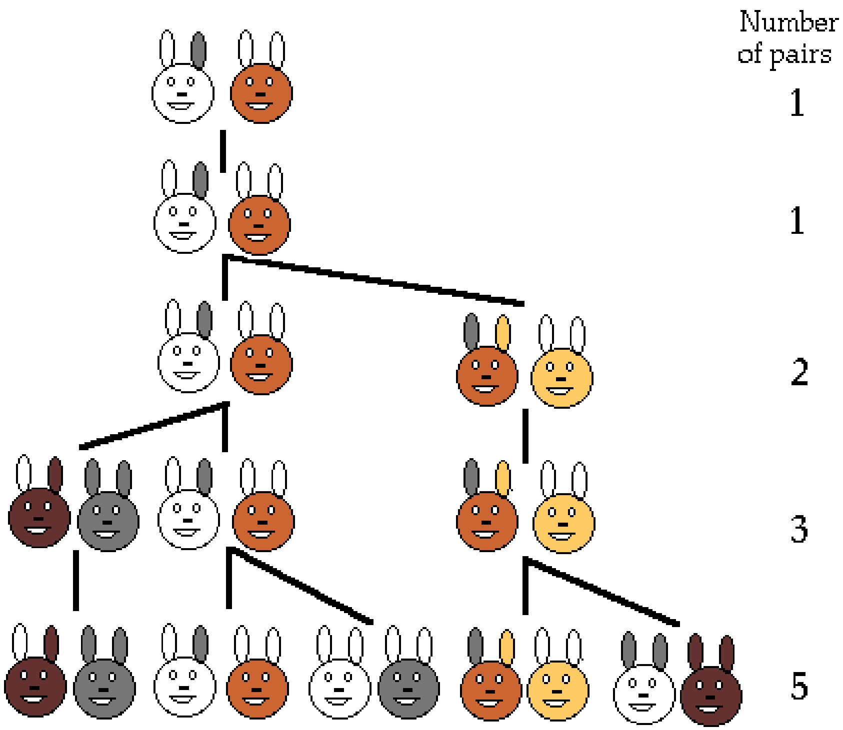
Rabbit Breeding Problem

- In the year 1202, Fibonacci became interested in the reproduction of rabbits.
- He created an imaginary set of ideal conditions under which rabbits could breed, and posed the question:
  - How many pairs of rabbits will there be a year from now?
  - The ideal set of conditions was a follows:
    - You begin with one male rabbit and one female rabbit. These rabbits have just been born.
    - A rabbit will reach sexual maturity after one month.
    - The gestation period of a rabbit is one month.
    - Once it has reached sexual maturity, a female rabbit will give birth every month.
    - A female rabbit will always give birth to one male rabbit and one female rabbit.
    - Rabbits never die.

Leonardo Fibonacci

Rabbit Breeding Problem

1. At the end of the first month, they mate, but there is still one only 1 pair.
2. At the end of the second month the female produces a new pair, so now there are 2 pairs of rabbits in the field.
3. At the end of the third month, the original female produces a second pair, making 3 pairs in all in the field.
4. At the end of the fourth month, the original female has produced yet another new pair, the female born two months ago produces her first pair also, making 5 pairs.



Rabbit Breeding Problem

- Did you notice the pattern?
  - rabbit(2_{nd} month) = rabbit(1_{st} month) + rabbit(0_{th} month)
  - rabbit(3_{rd} month) = rabbit(2_{nd} month) + rabbit(1_{st} month)
  - rabbit(4_{th} month) = rabbit(3_{rd} month) + rabbit(2_{nd} month)
  - …

- Let the population at month n be $fib(n)$. At this time, only rabbits who were alive at month $n - 2$ are fertile and able to produce offspring, so $fib(n - 2)$ pairs are added to the current population of $fib(n - 1)$.
- Thus the total is $fib(n) = \underbrace{fib(n - 1) + fib(n - 2)}_{\text{Recursively defined}}$

Fibonacci Sequence

- By definition, the first two Fibonacci numbers are 0 and 1, and each remaining number is the sum of the previous two.
- In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relation:

  $F_n = F_{n-1} + F_{n-2}$ where $F_0 = 0$ and $F_1 = 1$
- So the Fibonacci number sequence is as follows:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89…

Fibonacci Sequence

- $fib(n) = fib(n - 1) + fib(n - 2)$
- e.g. Compute $fib(4)$

$= fib(3) + fib(2)$

$= [fib(2) + fib(1)] + [fib(1) + fib(0)]$

$= [\ \{fib(1) + fib(0)\} + fib(1)] + [fib(1) + fib(0)]$

$= [\ \{1 + 0\} + 1] + [1 + 0]$

$= \underline{3}$

Fibonacci Sequence

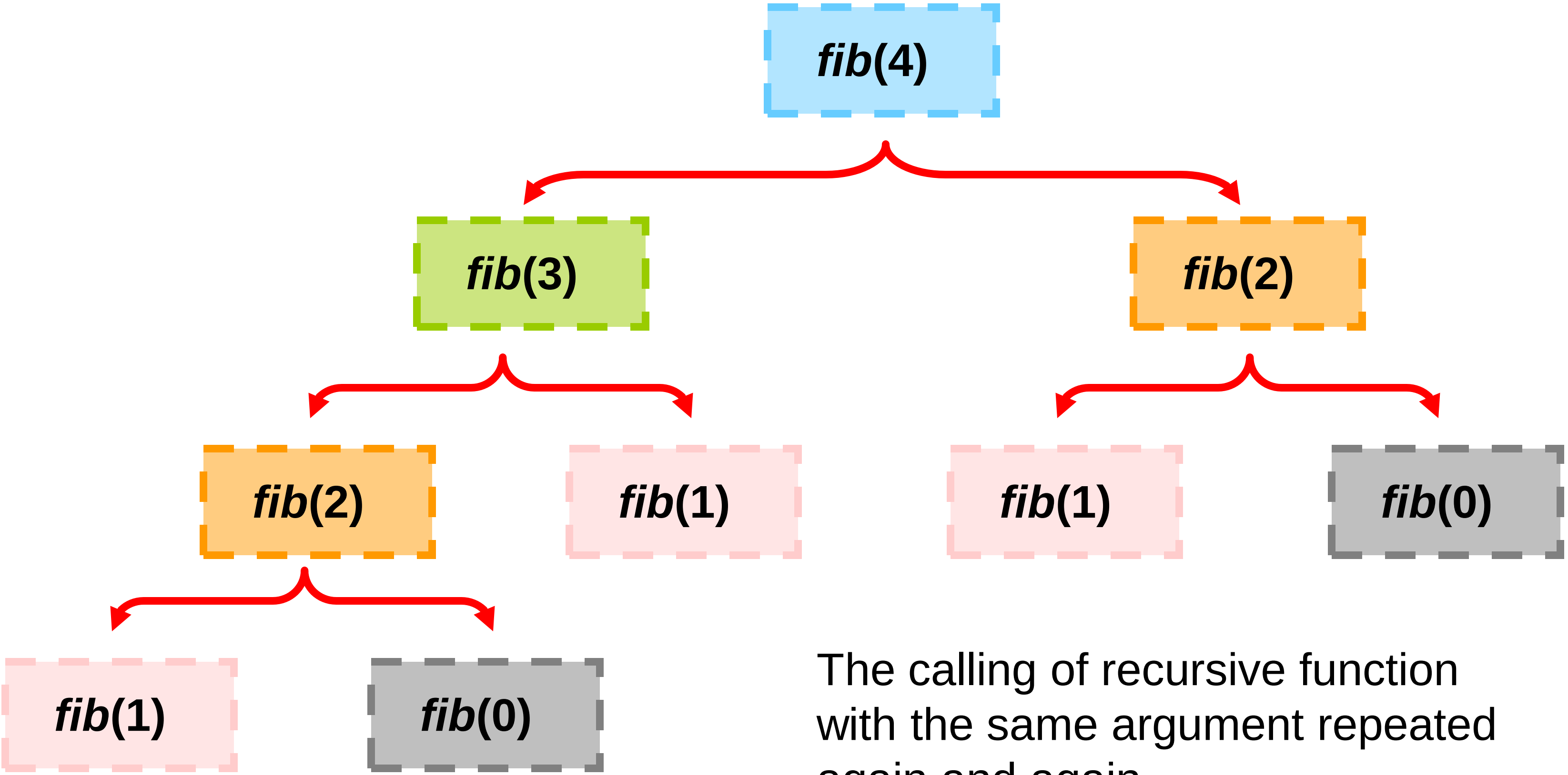
- *fib*(*n*) = *fib*(*n* - 1) + *fib*(*n* - 2)

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return (fib(n - 1) + fib(n – 2));
}
```

Check base cases before recursion

Calling itself (recursion)

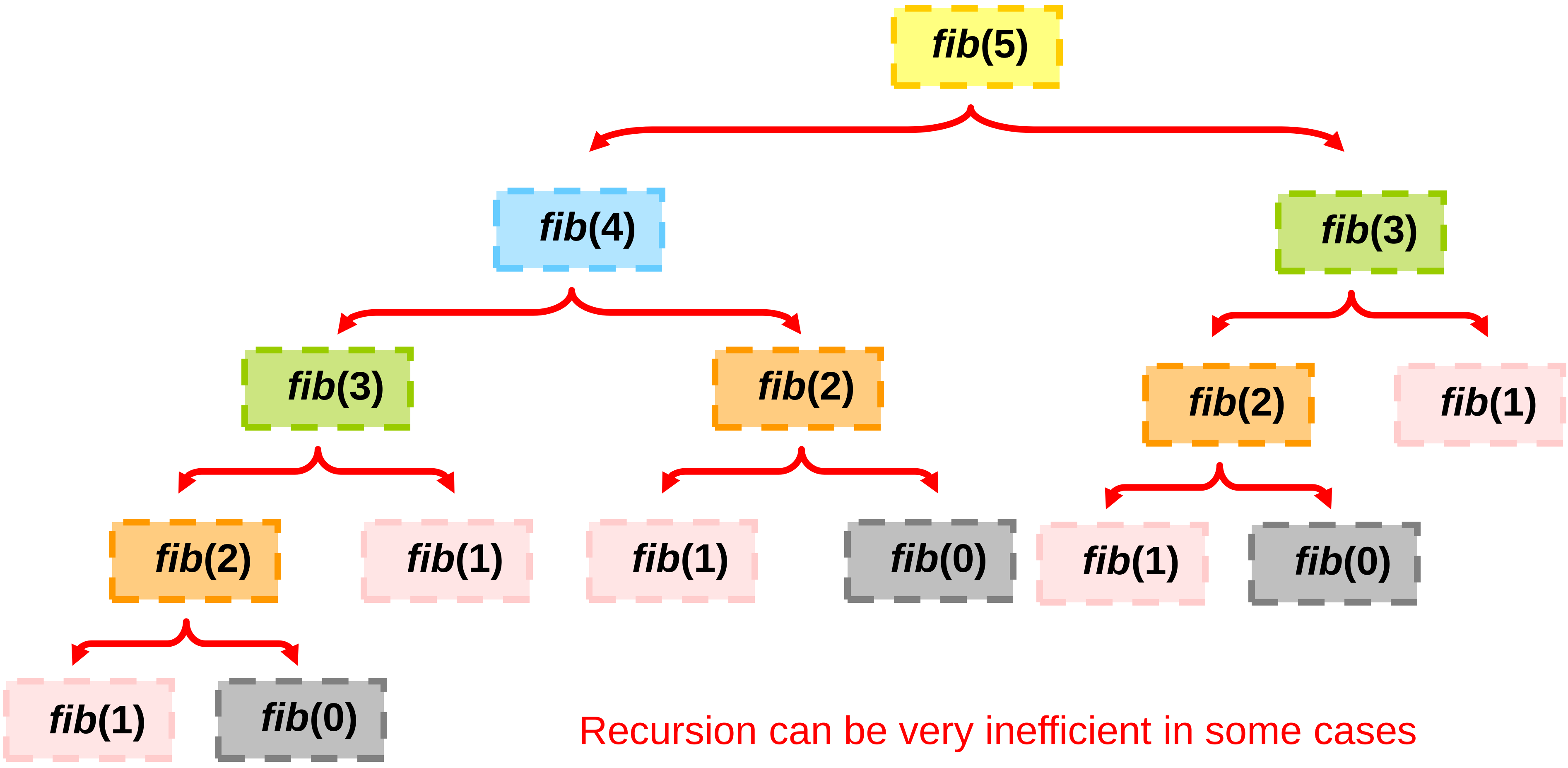
Fibonacci Sequence



The calling of recursive function with the same argument repeated again and again

e.g. *fib*(2) being called twice, *fib*(1) being called 3 times

Fibonacci Sequence



Recursion can be very inefficient in some cases

Ackermann Function

- The Ackermann function $Ack(m, n)$ is defined recursively for non-negative integers m and n as follows:

- $$Ack(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ Ack(m-1, 1) & \text{if } m>0 \text{ and } n=0 \\ Ack(m-1, Ack(m, n-1)) & \text{if } m>0 \text{ and } n>0 \end{cases}$$

Ackermann Function

- e.g. Compute $Ack(1, 2)$
- $= Ack(0, Ack(1, 1))$
- $= Ack(0, Ack(0, Ack(1, 0)))$
- =
- =
- =
- =

Ackermann Function

- $$Ack(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ Ack(m-1, 1) & \text{if } m>0 \text{ and } n=0 \\ Ack(m-1, Ack(m, n-1)) & \text{if } m>0 \text{ and } n>0 \end{cases}$$

```
int Ack(int m, int n) {

}
```

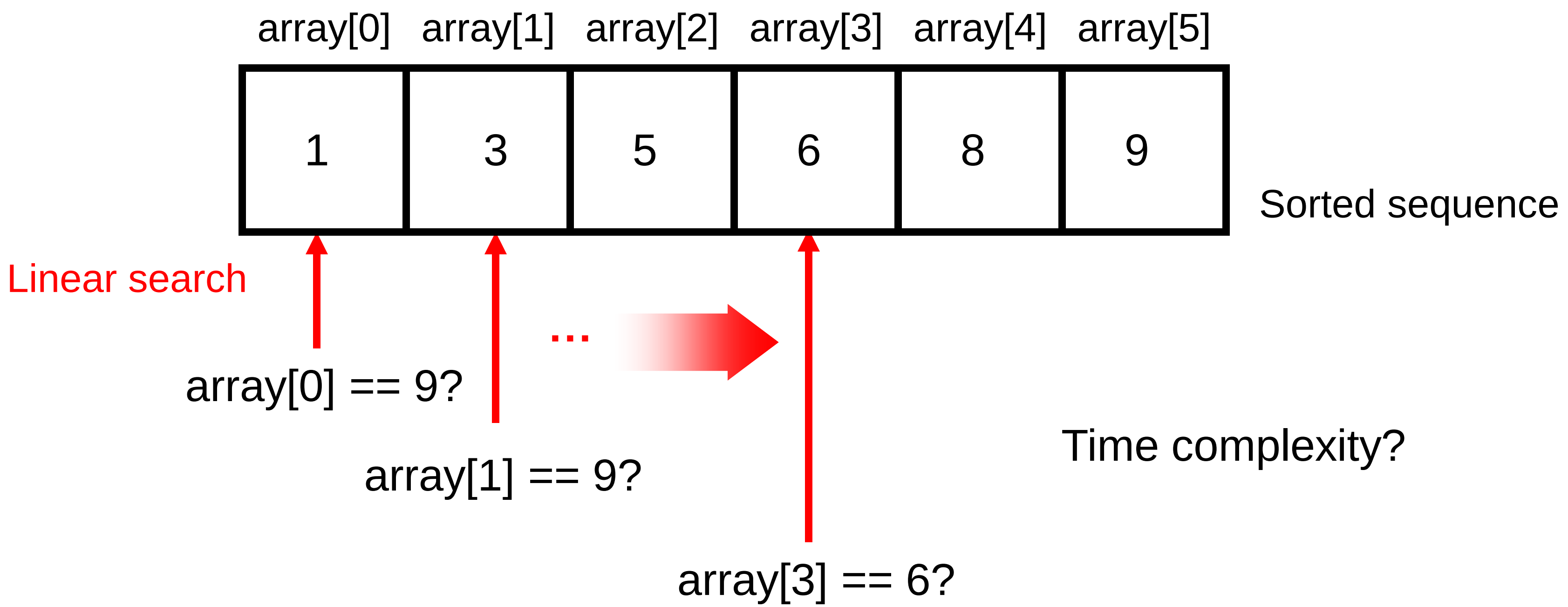
Recursive Searching

array[0]	array[1]	array[2]	array[3]	array[4]	array[5]
1	3	5	6	8	9

Sorted sequence

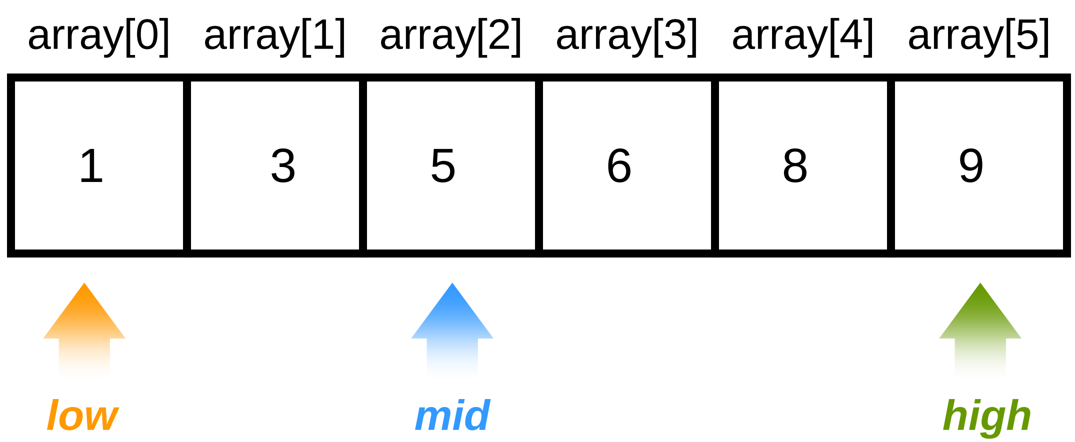
To look for a certain element in the array, e.g. 6

Linear Search



To look for a certain element in the array, e.g. 6

Recursive Binary Search



Sorted sequence

Recursive approach

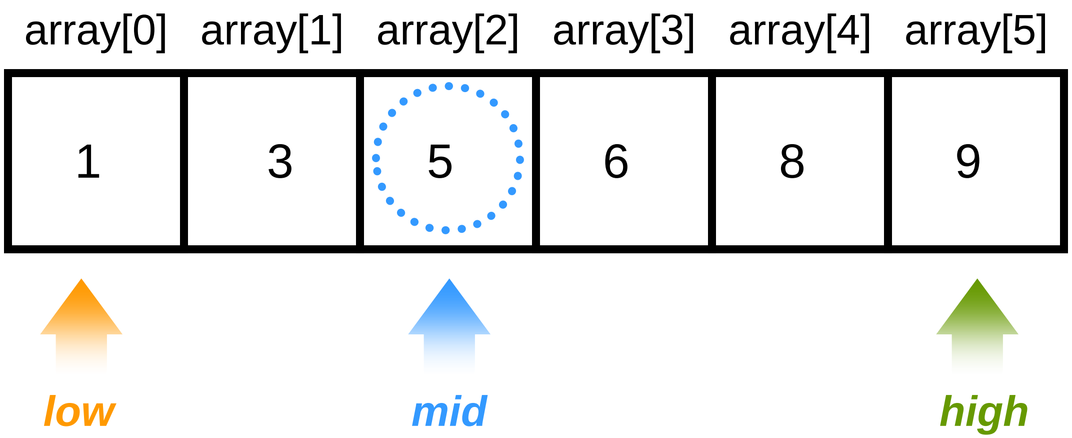
***low* = left most element**

***high* = right most element**

***mid* = (*high* + *low*) / 2**

To look for a certain element in the array, e.g. 6

Recursive Binary Search



Sorted sequence

Recursive approach

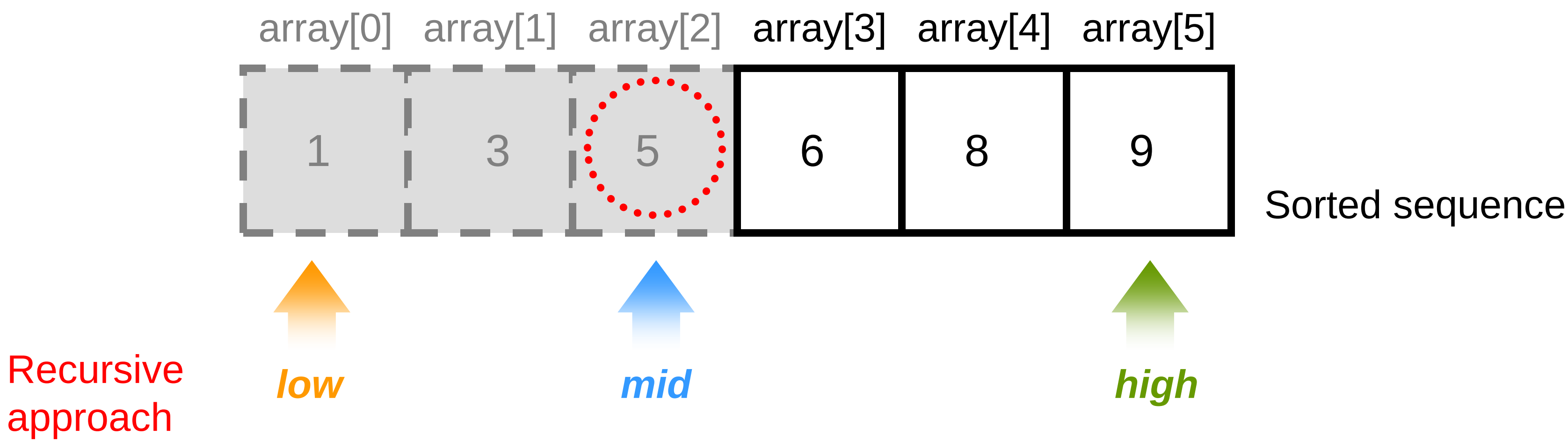
Compare array[*mid*] with 6

array[*mid*] > 6 : search left sub-sequence

array[*mid*] == 6 : the answer!

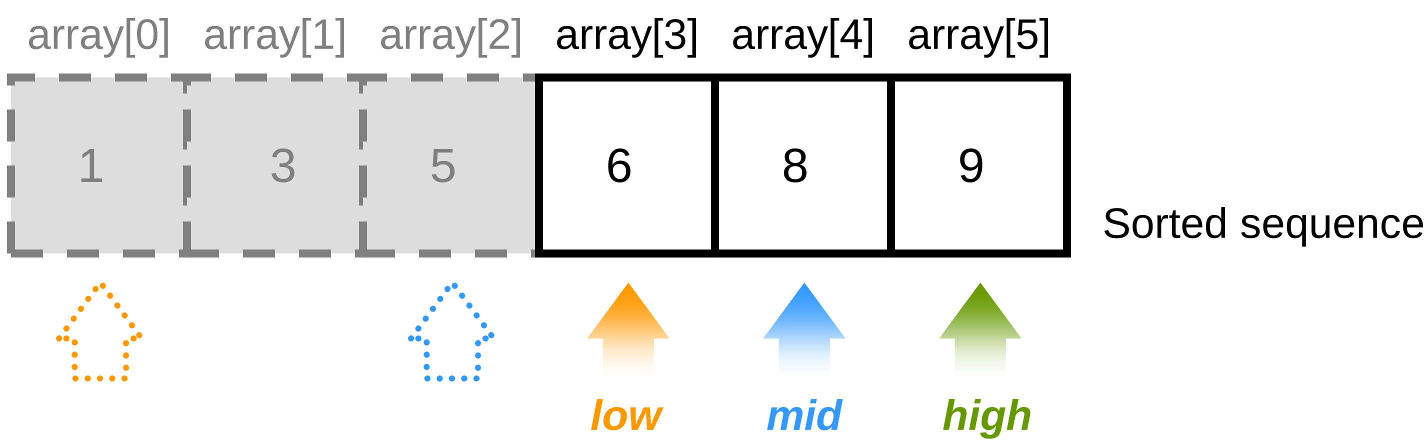
array[*mid*] < 6 : search right sub-sequence

Recursive Binary Search



Since 5 < 6, the answer must be in the right sub-sequence

Recursive Binary Search

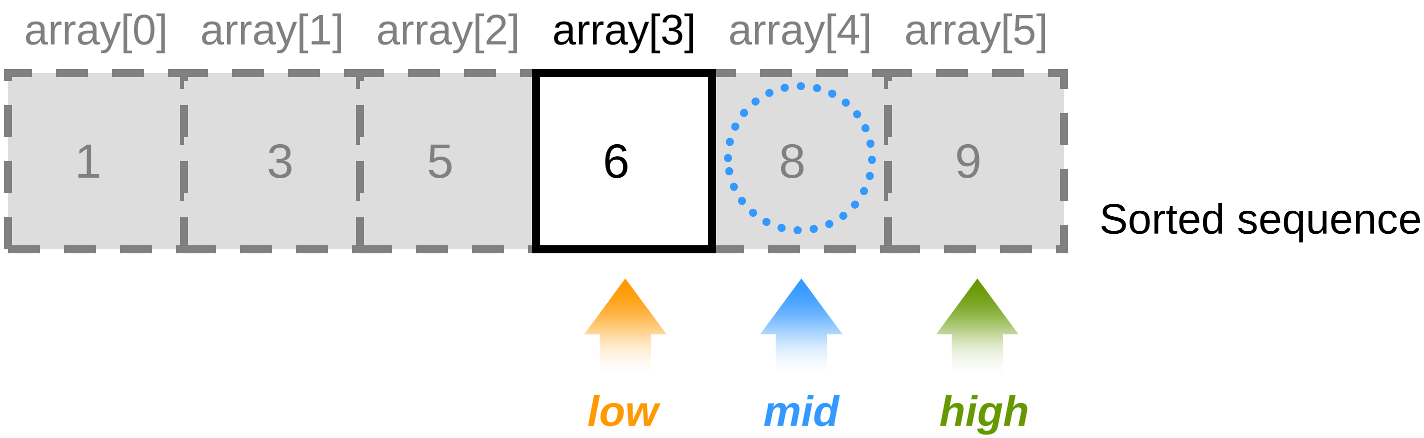


Recursive
approach

The new search windows is [*mid* + 1, *high*]

update *low* and recalculate *mid* pointers

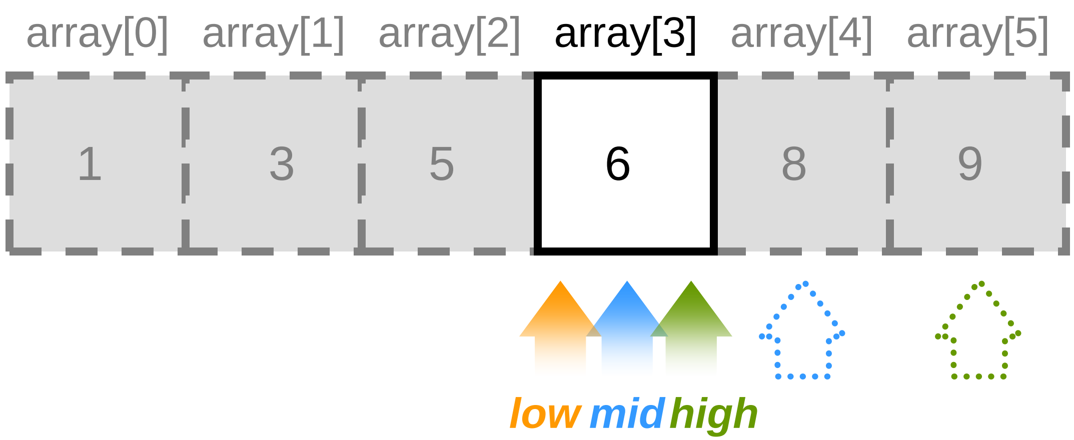
Recursive Binary Search



Recursive approach

Since 8 > 6, the answer must be in the left sub-sequence

Recursive Binary Search



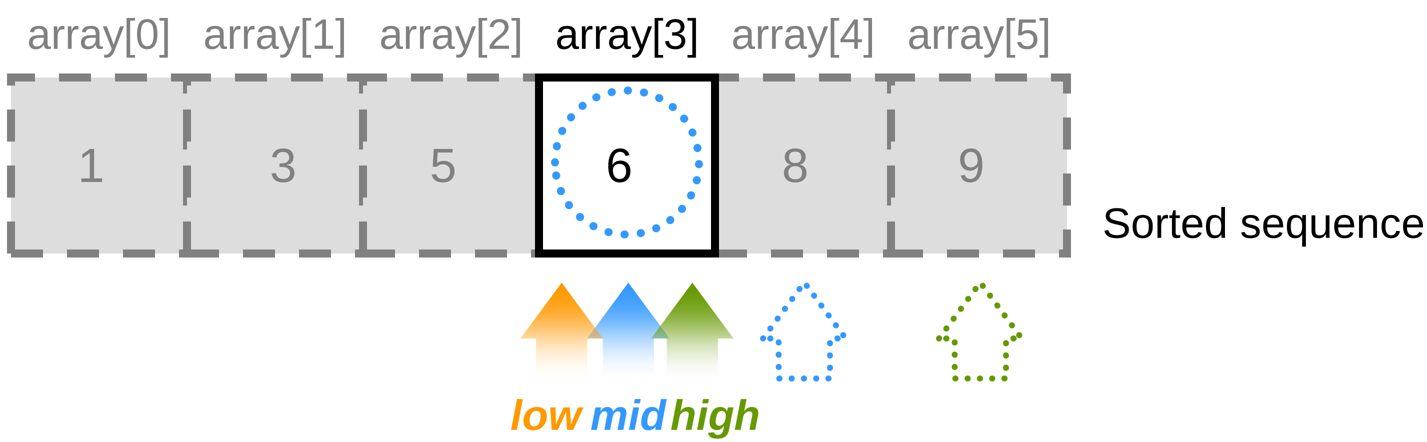
Sorted sequence

Recursive approach

The new search window is [*low*, *mid* – 1]

Update *high* and recalculate *mid* pointers

Recursive Binary Search



Recursive approach

array[3] == 6

The answer is 3

Recursive Binary Search

- Each time the routine is called recursively, the no. of elements to be searched is halved
- The expected number of elements to be searched is $\log_2 n + 1$, where n is total number of elements

array[0]	array[1]	array[2]	array[3]	array[4]	array[5]
1	3	5	6	8	9
		1	3	2	

Non-recursive Binary Search

- A non-recursive approach
  - Use a loop approach instead of recursive calling
  - Update either the *low* or *high* pointer in each iteration
  - Loop until *low > high* (the failure condition)
  - Time: O(log n) / Space: O(1)

```
int binsch(int array[], int low, int high, int x) {
    int mid;
    while (low <= high) {
        mid = (high + low) / 2;
        if (array[mid] == x) return mid;  //x has been found
        if (array[mid] > x) high = mid - 1;
        if (array[mid] < x) low = mid + 1;
    }
    return -1;  //cannot find x in the array
}
```

Recursive Binary Search

```
int binsch(int array[], int low, int high, int x) {

}
```

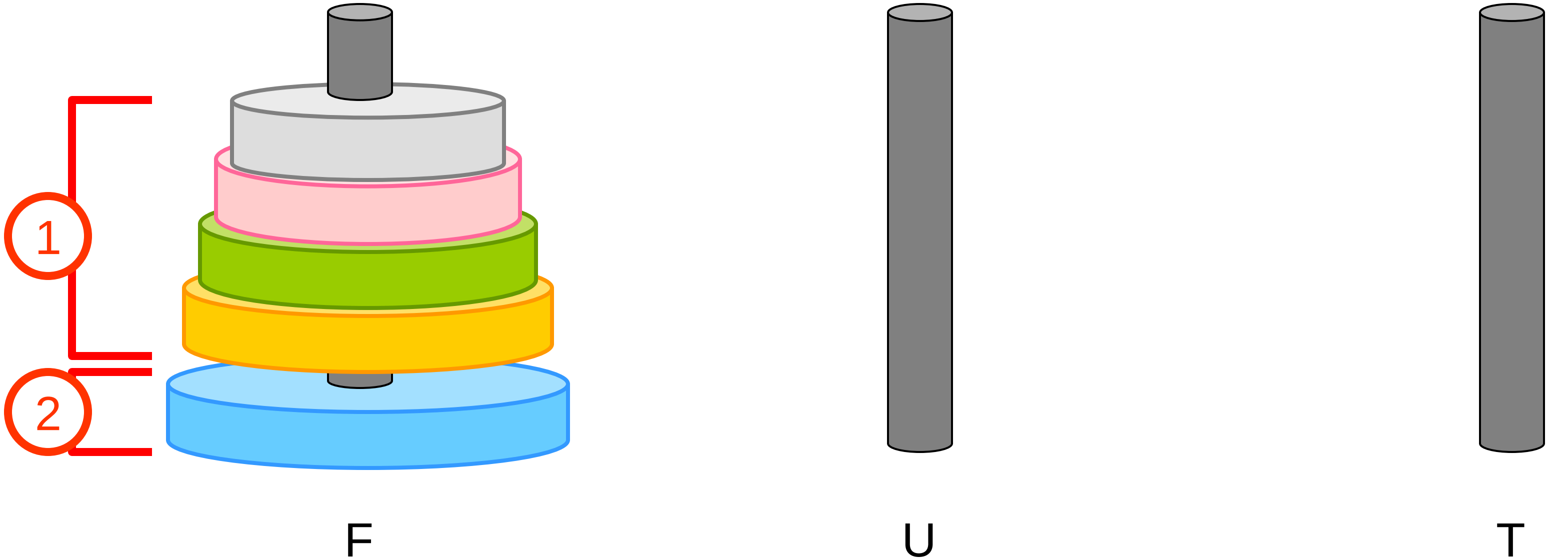
Time complexity? Space complexity?

Recursion vs. Iteration

- Iteration sometimes can be used in place of recursion
  - An iterative algorithm uses a <u>looping structure</u>
  - A recursive algorithm uses a <u>branching structure</u>
- Recursive solutions are often less efficient, in terms of both <u>time</u> and <u>space</u>, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code

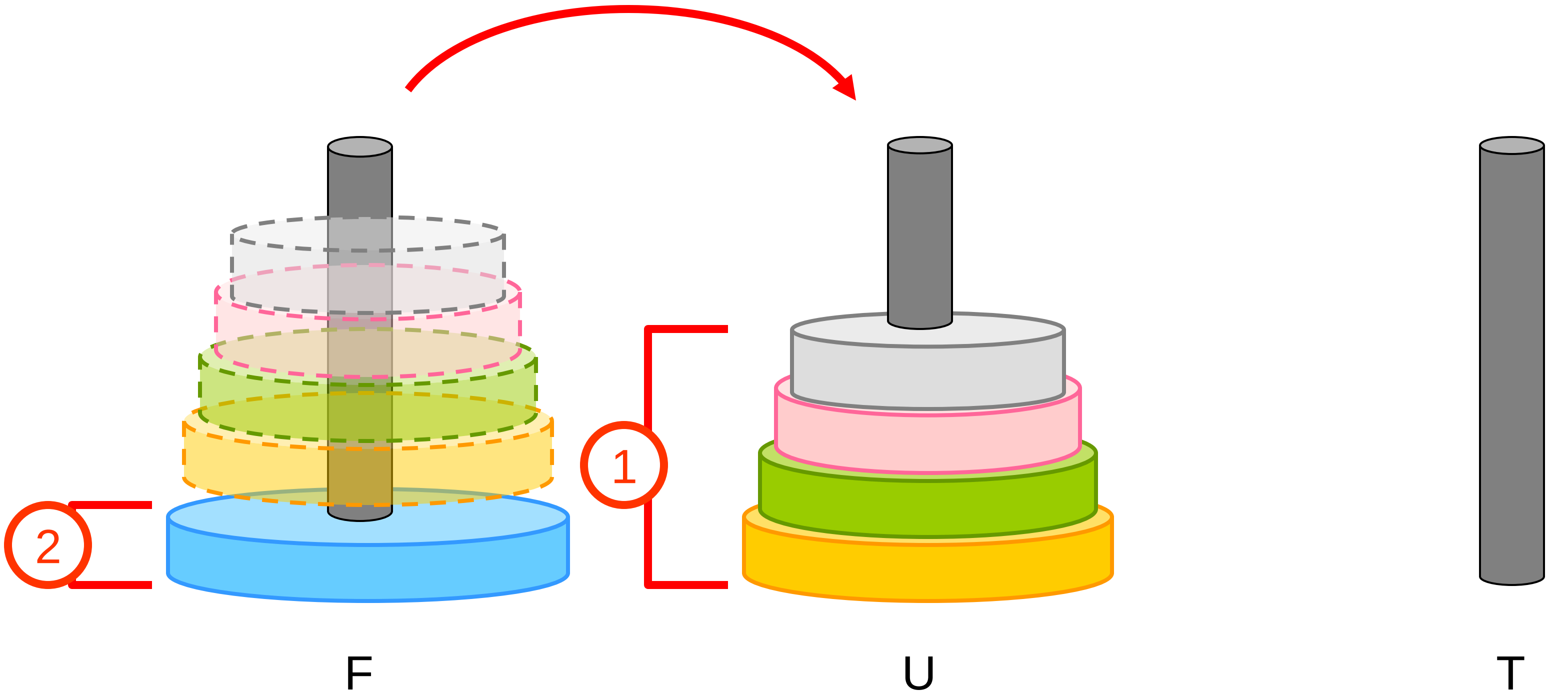
Towers of Hanoi

Towers of Hanoi



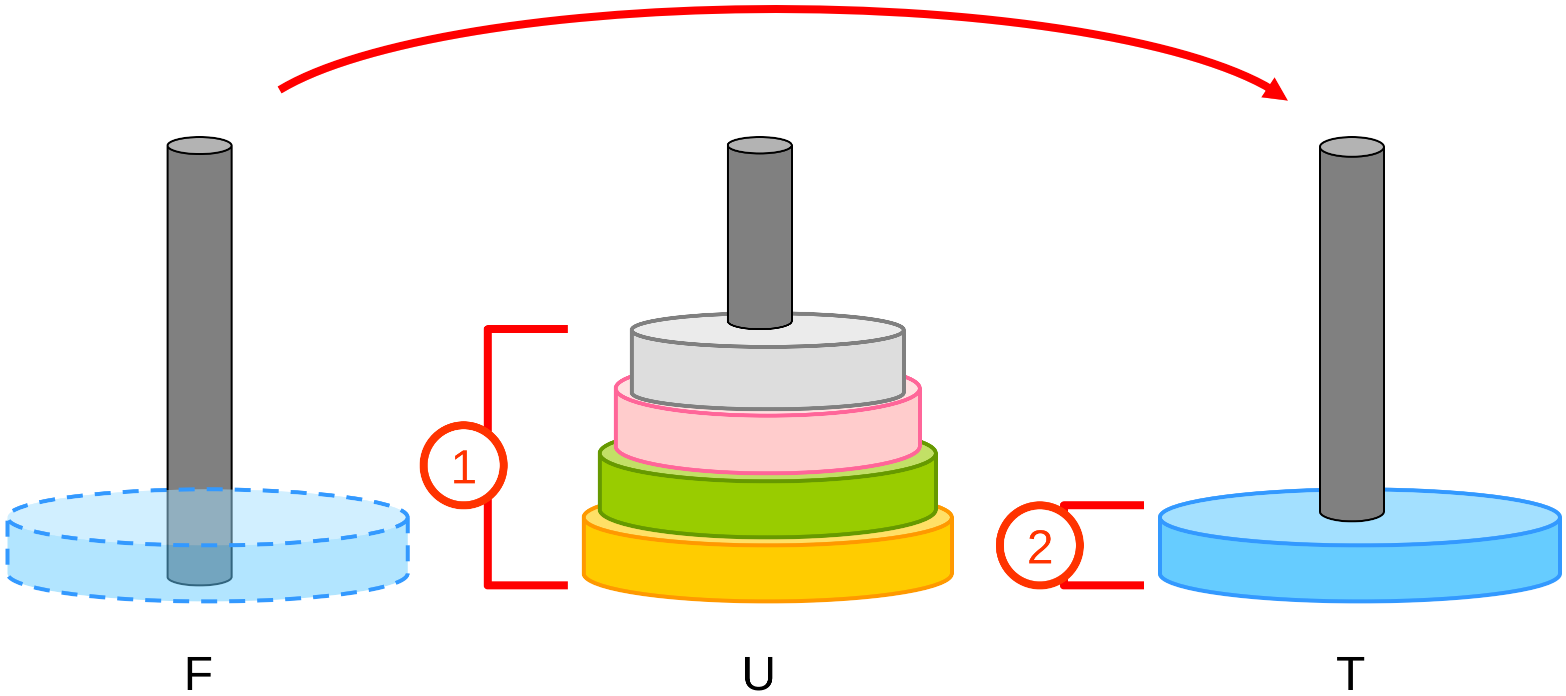
Imagine that the disks are divided into 2 groups

Towers of Hanoi: Step 1



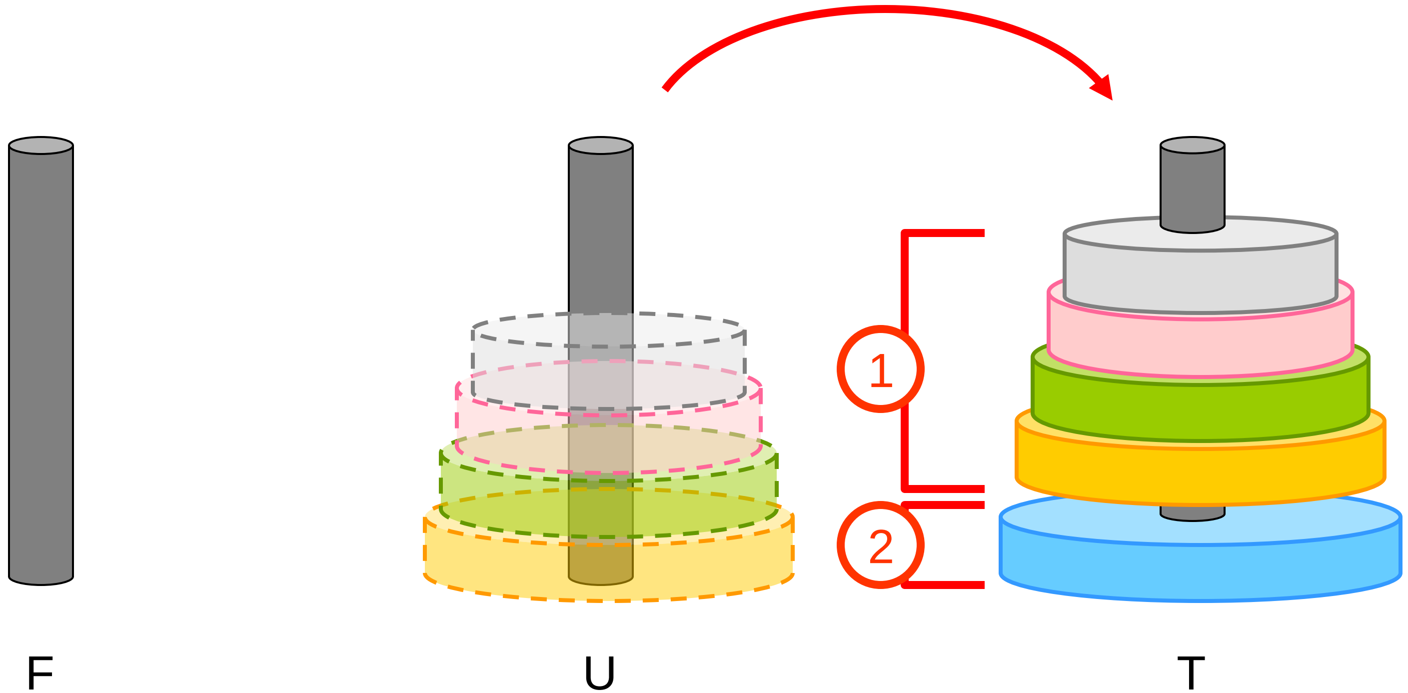
Move group 1 (*n*-1 disks) from F to U
(involves a lot of steps)

Towers of Hanoi: Step 2



Move group 2 (1 disk) from F to T
(just one step)

Towers of Hanoi: Step 3



Move group 1 from U to T
(involves a lot of steps)

The Procedure

- To move n disks from F to T, using U as buffer
  1. Move the top n-1 disks from F to U, using T as auxiliary (recursion step)
  2. Move the remaining largest disk from F to T
  3. Move the n-1 disks from U to T, using F as auxiliary (recursion step)

- Rmove(n) = Rmove(n-1) + move(1) + Rmove(n-1)

- The problem is reduced to moving n-1 disks
- If n == 1, simply move the single disk from F to T (base case)

Towers of Hanoi (Stack Version)

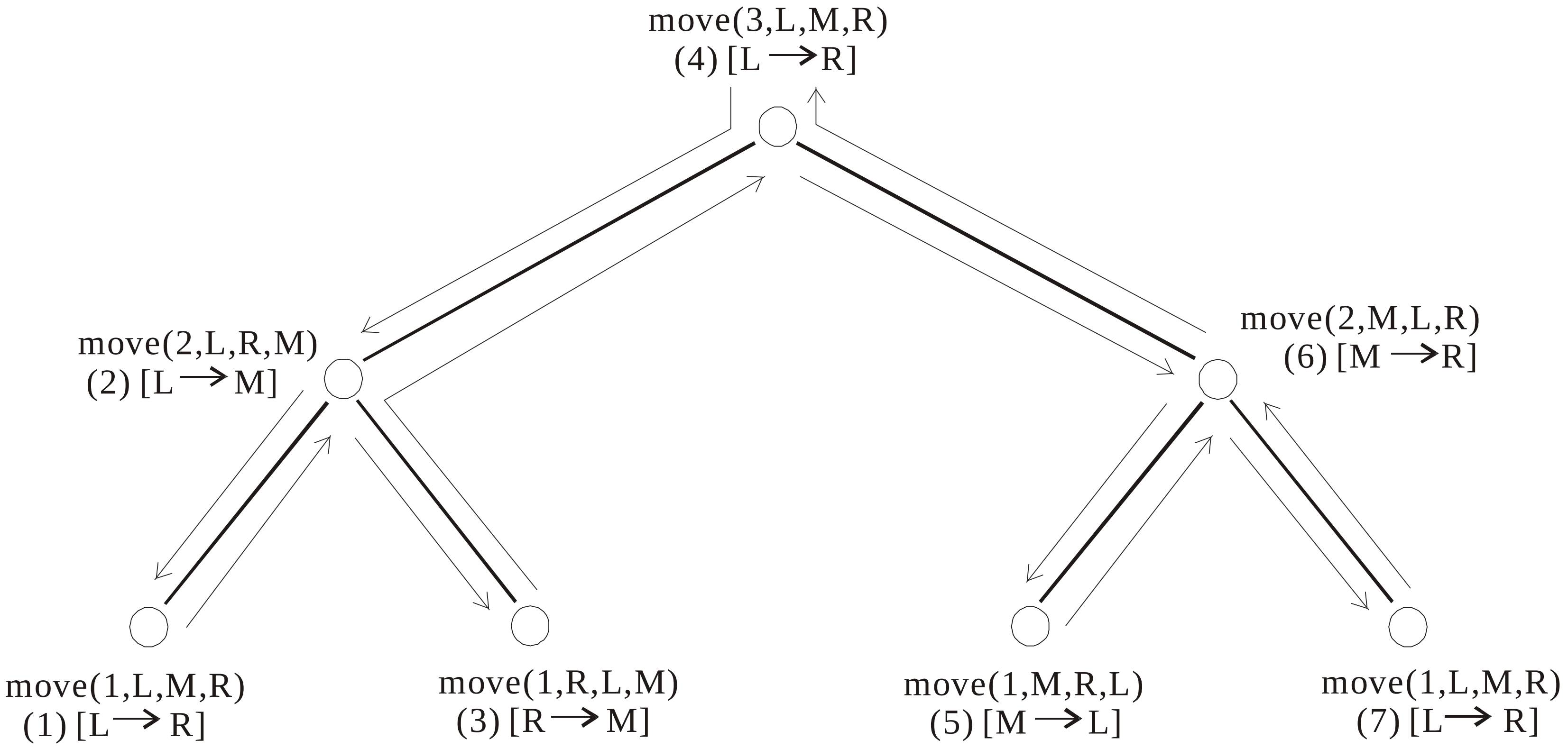
```
// Let  the function below denotes the operation of moving N
// disks from tower F to tower T using tower U.

void move(int N, stack<disk>& F, stack<disk>& U, stack<disk>& T) {

    if (N == 1) {                    // base case

        T.push(F.top());

        F.pop();

    } else if(N > 1) {

        move(N-1, F, T, U);          // F → U

        T.push(F.top());             // F → T

        F.pop();

        move(N-1, U, F, T);          // U → T

    }

}
```

Towers of Hanoi (Print Version)

```
// Let  the function below denotes the operation of moving N
// disks from tower F to tower T using tower U.

//version 2, without the else statement

void move(int N, char F, char U, char T) {

   if (N > 0) {

      move(N-1, F, T, U);

      cout << F << “ -> ” << T << endl;

      move(N-1, U, F, T);

   }

}
```

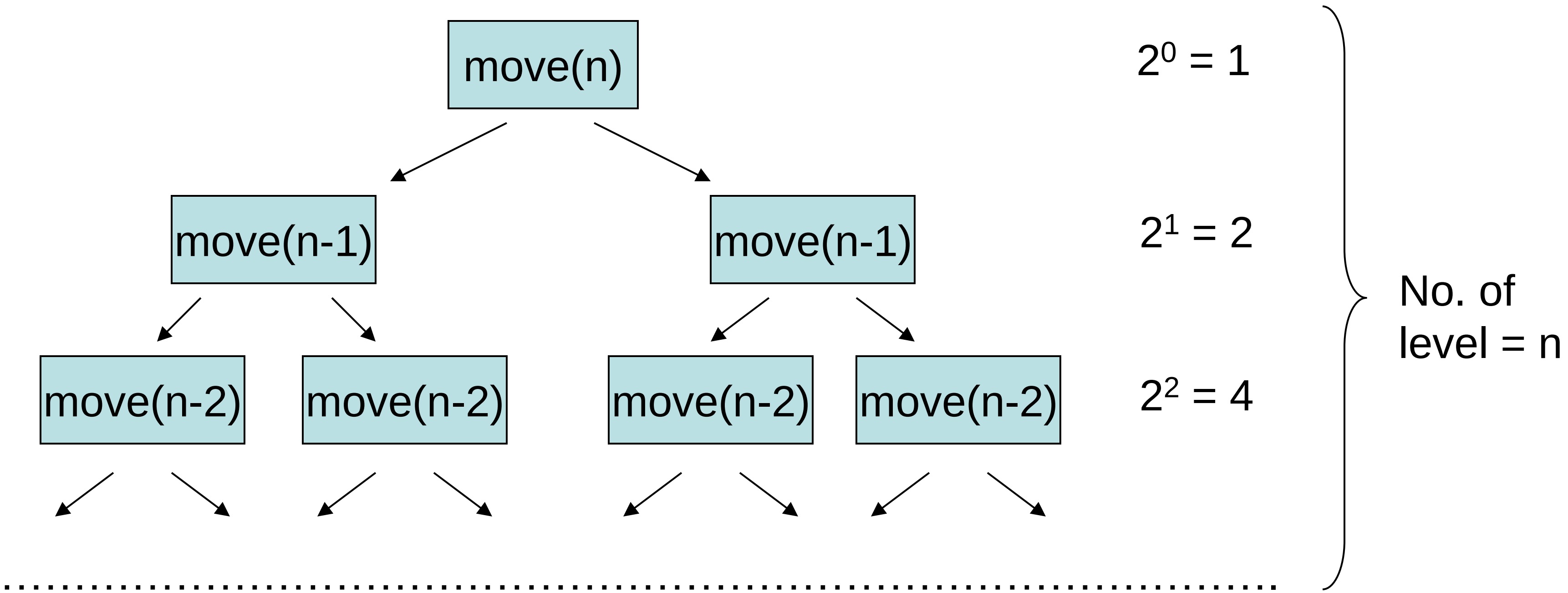
Recursion Tree

- For moving 3 disks, from L to R using M



Time Complexity

- Each call makes one move of disk



The total no. of move is:

$2^0 + 2^1 + 2^2 + \ldots + 2^{n-2} + 2^{n-1} = 2^n - 1$

Analysis of Recursion

Two Classes of Recursion

- Direct recursion
  - The procedure call themselves before they are done
- Indirect recursion
  - The procedure may call other procedure that again invoke the calling procedure

```
int functionA(…) {
    …
    functionB(…);
    …
}
```

```
int functionB(…) {
    …
    functionA(…);
    …
}
```

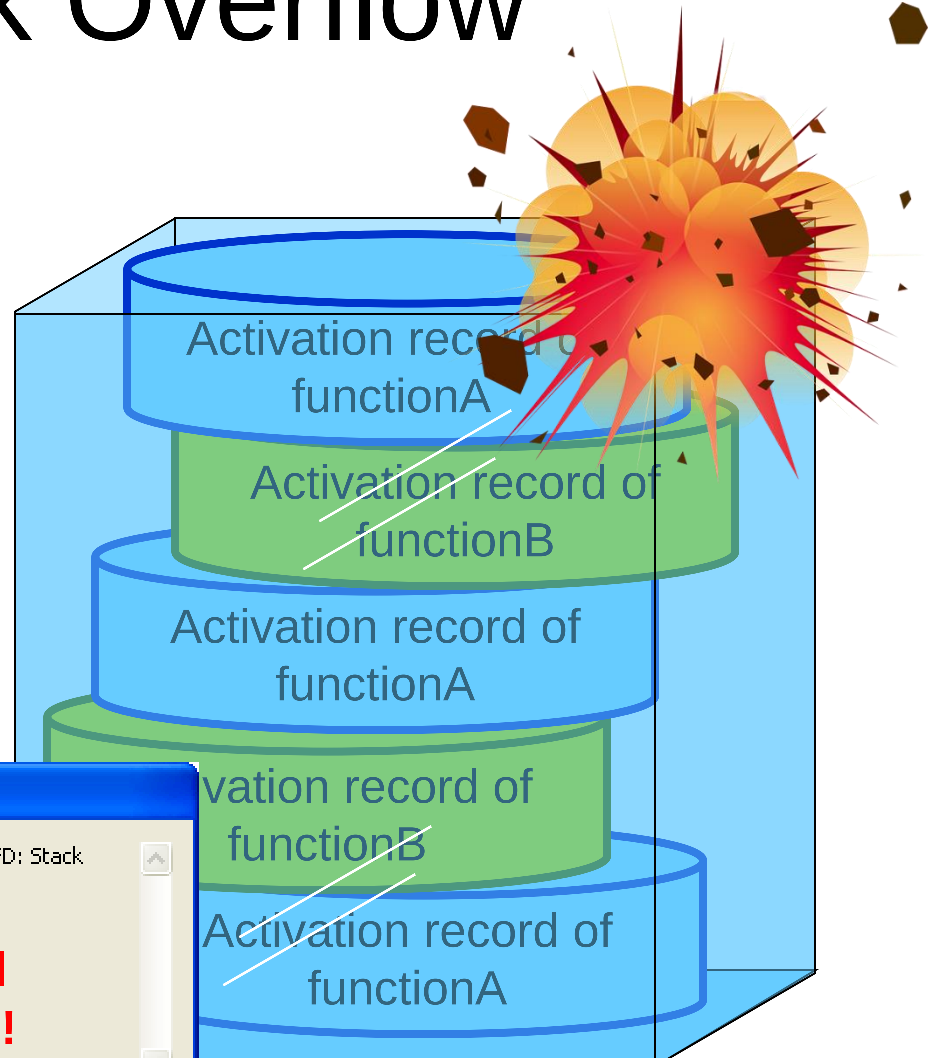
Activation Record

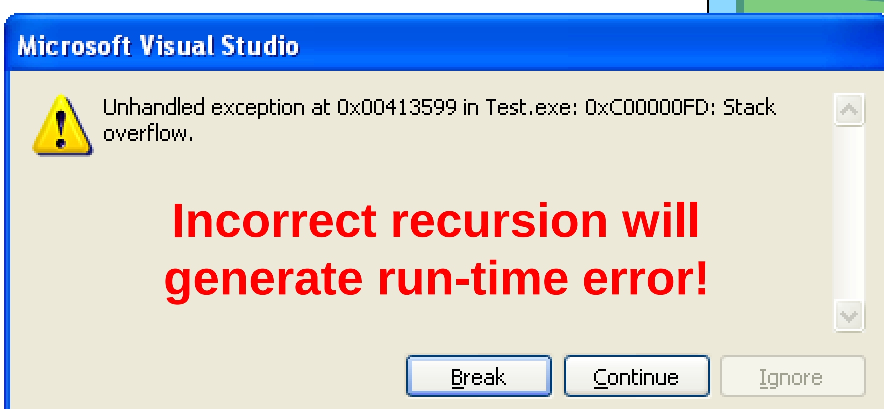
- The number of recursive calls is unknown in compile time
- When a function is being called, it is said to be activated. A fresh set of local variables and parameters are allocated for each call. They will be released when the function finishes
- These automatic variables, parameters and return address (back to the caller) are stored collectively into a call stack, known as an activation record.
- The record is removed (pop from stack) when the function returns.
- Since each call creates a separate record, a subroutine can be reentrant, and recursion is automatically supported

Call Stack Overflow

```
void functionA() {
    functionB();
}
```

```
void functionB() {
    functionA();
}
```





Develop Recursion Algorithm

1. Express the problem in the form of recurrence
2. Define base cases
   – Make sure the case can be solved without recursion
3. Define recursive cases
   – Make sure each recursive call makes progress to the base case

Pros of Using Recursion

- Natural and elegant way of solving problems
- Logical simplicity
  - e.g. Fibonacci sequence, Ackerman function
- Self-documentation, increase readability
  - e.g. factorial, recursive binary search
- Handle complicated problems
  - e.g. towers of Hanoi
- Programming efficiency

Cons of Using Recursion

- Often more expensive than non-recursive solution, in terms of time and space
- Space:
  - Activation record and stack
  - Recursive algorithm may need space proportional to the number of nested calls to the same function.
- Time:
  - Introduced overhead
  - The operations involved in calling a function - allocating, and later releasing, local memory, copying values into the local memory for the parameters, branching to / returning from the function

Class Exercise

- Compute the Greatest Common Divisor (GCD) of two numbers.
  - Let M > N, and M = CN + R such that 0 <= R < N
  - If R = 0, then M = CN, and gcd(M, N) = N
  - If R > 0, then N > R, and gcd(M, N) = gcd(N, R)
- Write a recursive function to compute the GCD of *M* and *N* where *M* >= *N* > 0

```
int gcd(int M, int N) {

}
```

Backtracking

Introduction

- *Search space* is a set of possible right answers to be explored.
- *Backtracking* is a technique whereby the search space is explored by systematically trying each possible path, backing up to try another whenever a dead end is encountered.
- Examples
  - Cross River Problem
  - Maze Solver
  - N Queens Problem

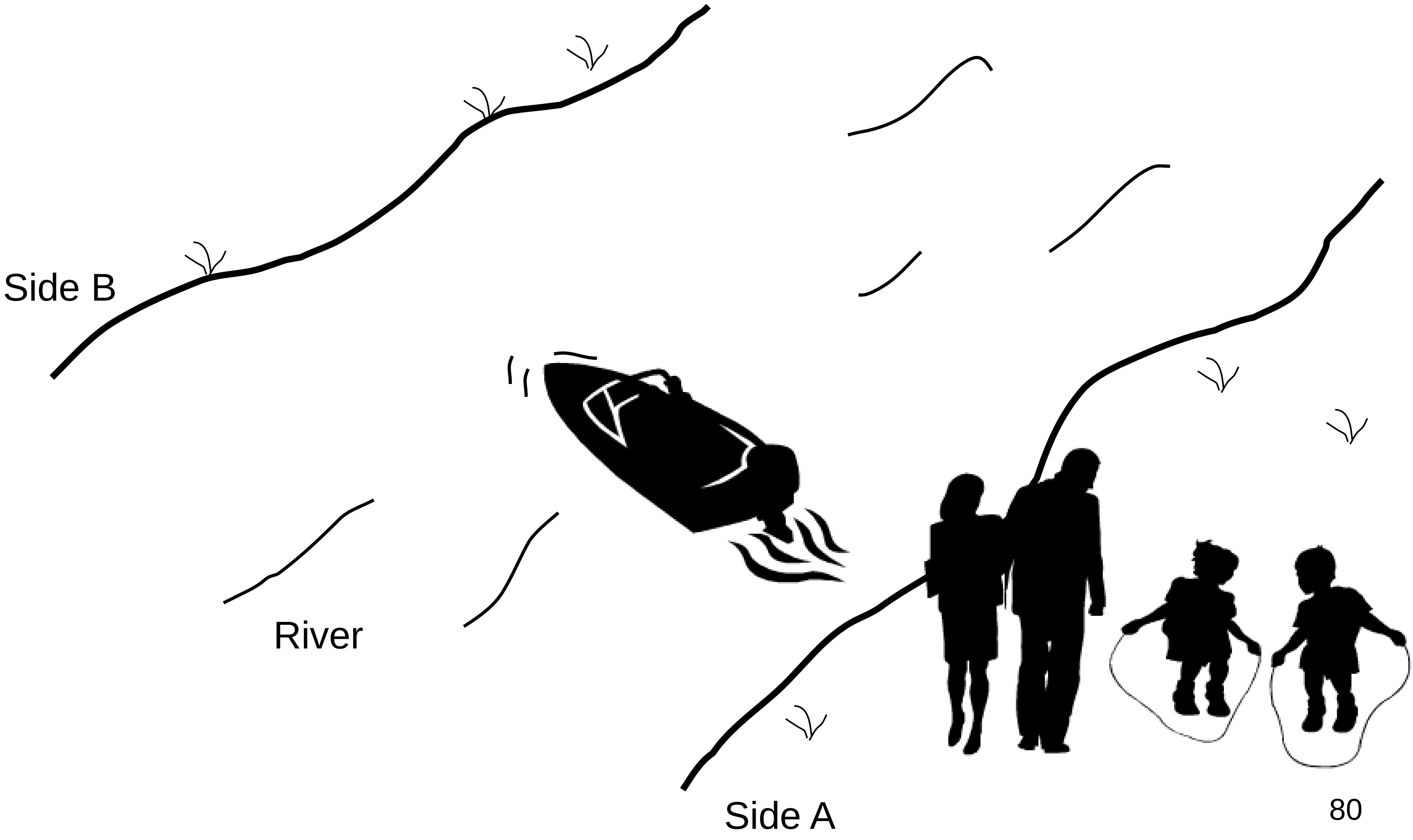
The First Example

Cross River Problem

Cross River Problem

- A family of 4 members (father, mother, son and daughter) go to picnic
- On their way to the destination, there is a river and a boat
- The boat can only take 2 people and only adult can sail the boat
- But
  - When father is not around, mother will hit her son
  - When mother is not around, father will hit his daughter
  - When both parents are not around, the brother will hit his sister
- How can they cross the river without letting any family member get injured?

How to Solve this Problem using Computer Program?



Trial-and-Error

	Side B	River	Side A
0	{}		{*father*, *mother*, *son*, *daughter*}
	{}	{*father*, *son*} ←	{*mother*, *daughter*}
1	{*father*, *son*}		{*mother*, *daughter*}
	{*son*}	{*father*} →	{*mother*, *daughter*}
2	{*son*}		{*father*, *mother*, *daughter*}
	{*son*}	{*mother, daughter*} ←	{*father*}
3	{*mother*, *son, daughter*} !		{*father*}

Retrace if Meet Any Errors

	Side B	River	Side A
0	{}		{*father*, *mother*, *son*, *daughter*}
	{}	{*father*, *son*}	{*mother*, *daughter*}
1	{*father*, *son*}		{*mother*, *daughter*}
	{*son*}	{*father*}	{*mother*, *daughter*}
2	{*son*}		{*father*, *mother*, *daughter*}
	~~{*son*}~~	~~{*mother*, *daughter*}~~	~~{*father*}~~
3	~~{*mother*, *son*, *daughter*}~~ !		~~{*father*}~~
	{*son*}	{*father*, *mother*}	{*daughter*}
3	{*father*, *mother*, *son*}		{*daughter*}

Continue

	Side B	River	Side A
3	{*father*, *mother*, *son*}		{*daughter*}
	{*father*, *son*}	{*mother*} →	{*daughter*}
4	{*father*, *son*}		{*mother*, *daughter*}
	{*father*, *son*}	← {*mother, daughter*}	{}
	{*father*, *mother*, *son*, *daughter*}		{}

5 Got the solution!

Is it a systemic way to solve this kind of problem?

Do you observe anything?

Do You Observe…

- Actually we have used the tree concept to search all possible moves
- If we meet an error, we trace back to pervious step and continue all other untried moves
- Let’s redraw the steps using a tree diagram

The Solution Tree

Initial setup: {*father*, *mother*, *son*, *daughter*}
{ }

1st move:

(1a) {*mother*, *daughter*}
{*father*, *son*}

(1b) {*father*, *son*}
{*mother*, *daughter*}

The first move has 2 choices only

From the game rule, *father* cannot stay with *daughter* alone and *mother* also cannot stay with *son* alone

So, obviously these 4 situations won't appeared:

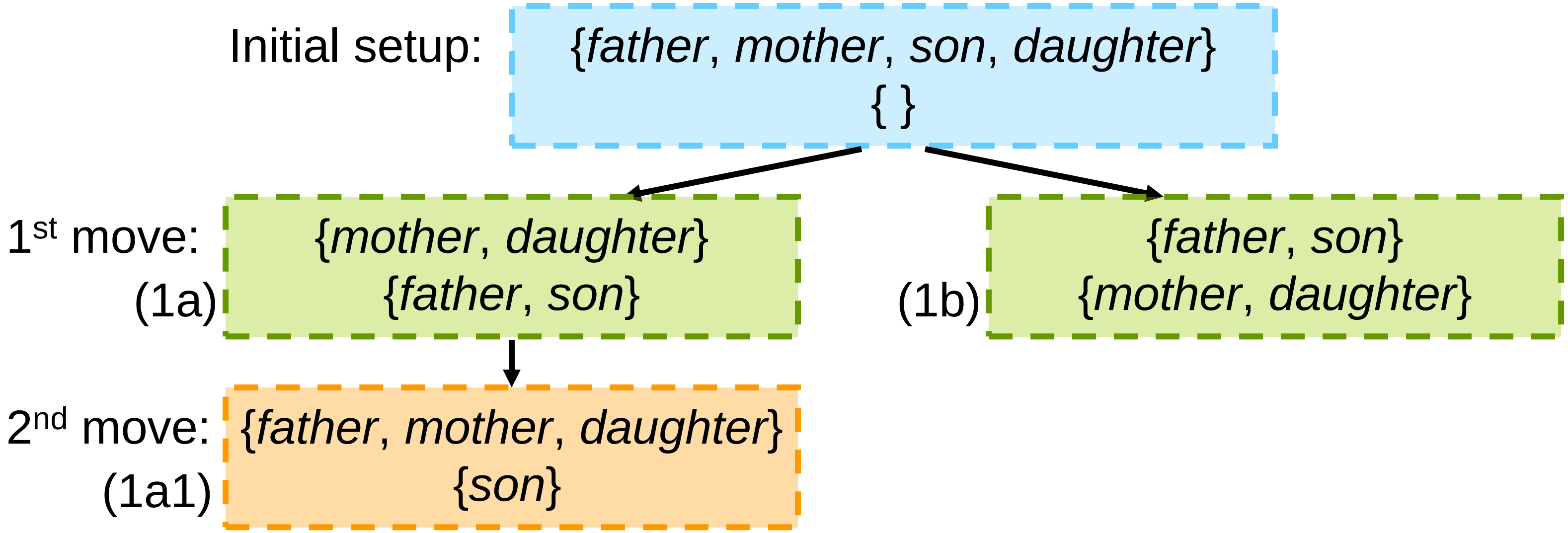
{*father*, *daughter*}
{*mother*, *son*}

{*father*, son, *daughter*}
{*mother*}

{*mother*, *son*}
{*father*, *daughter*}

{*mother*, *son*, *daughter*}
{*father*}

The Solution Tree



Consider move (1a) first

The boat is on side B. Since only *father* can sail the boat, the only move is *father* back to side A.

The Solution Tree

Initial setup: {*father*, *mother*, *son*, *daughter*}
{ }

1st move:

(1a) {*mother*, *daughter*}
{*father*, *son*}

(1b) {*father*, *son*}
{*mother*, *daughter*}

2nd move:

(1a1) {*father*, *mother*, *daughter*}
{*son*}

3rd move:

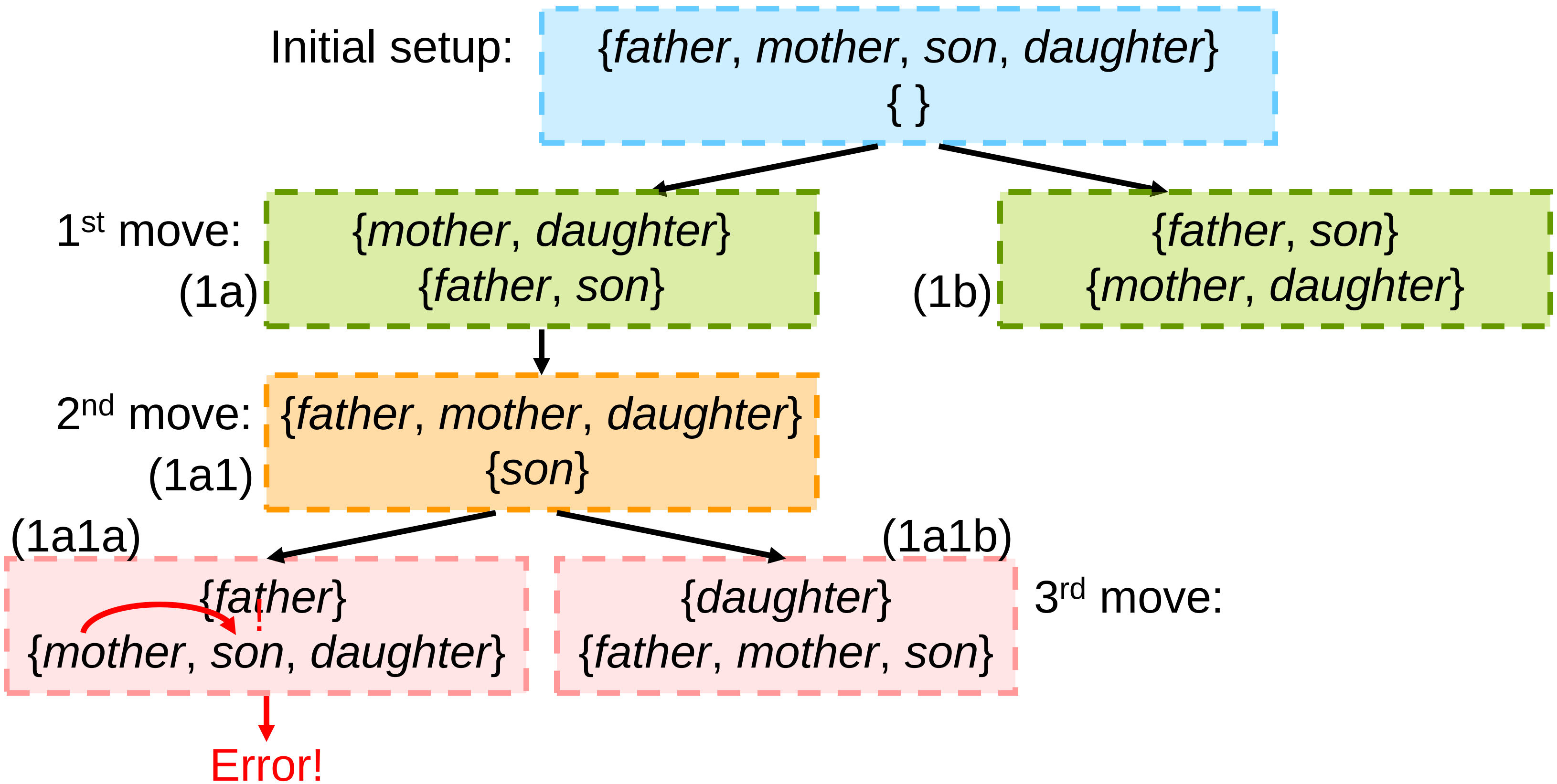
(1a1a) {*father*}
{*mother*, *son*, *daughter*}

(1a1b) {*daughter*}
{*father*, *mother*, *son*}

Now consider (1a1)

Since *father* cannot sail with *daughter*, only 2 moves left.
Either *mother* take the boat with *daughter*, or
father take the boat with *mother*

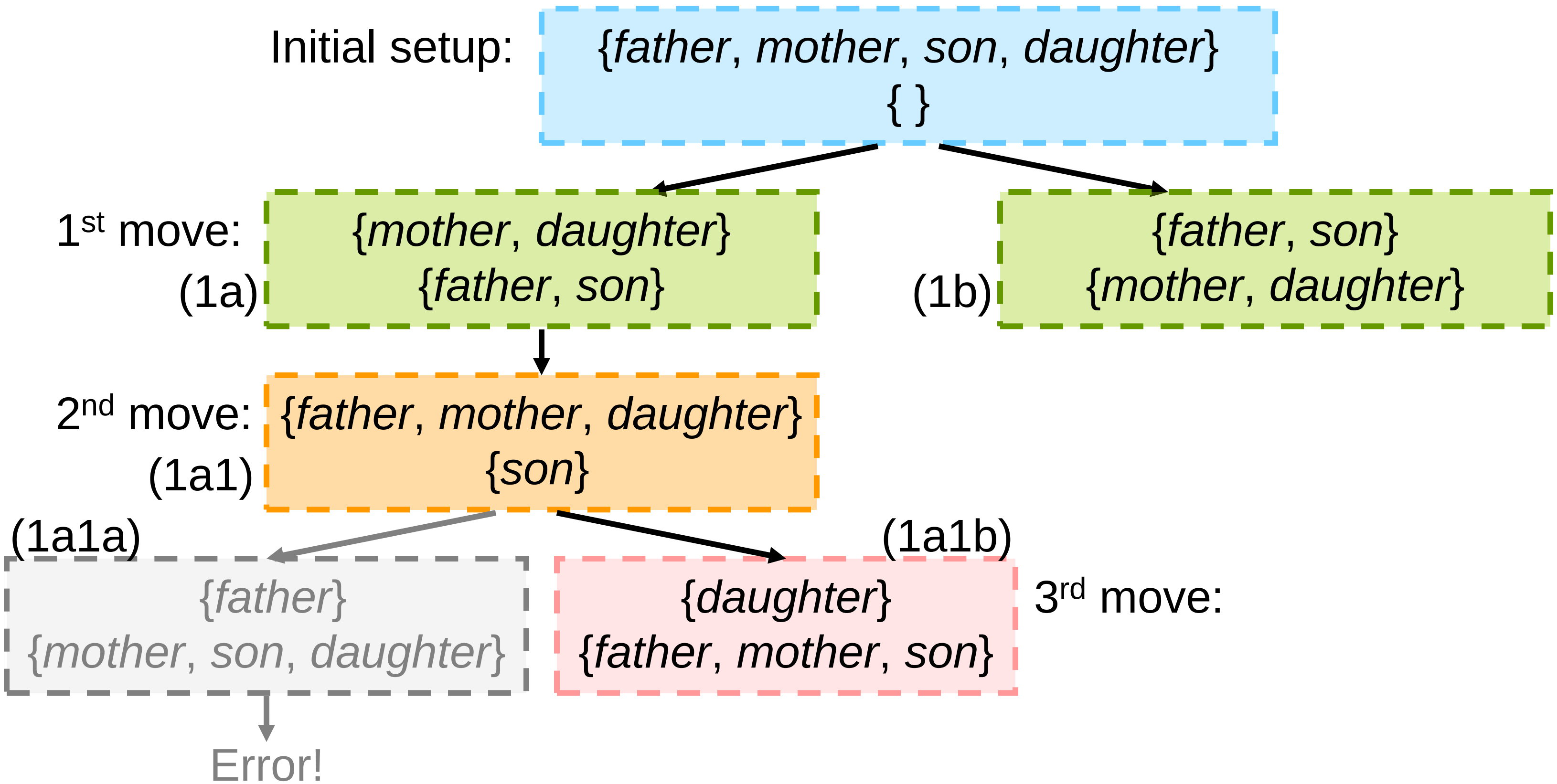
The Solution Tree



Consider (1a1a) first:

Now *mother* and *son* are on side B but *father* is not around! It violates the game rule!

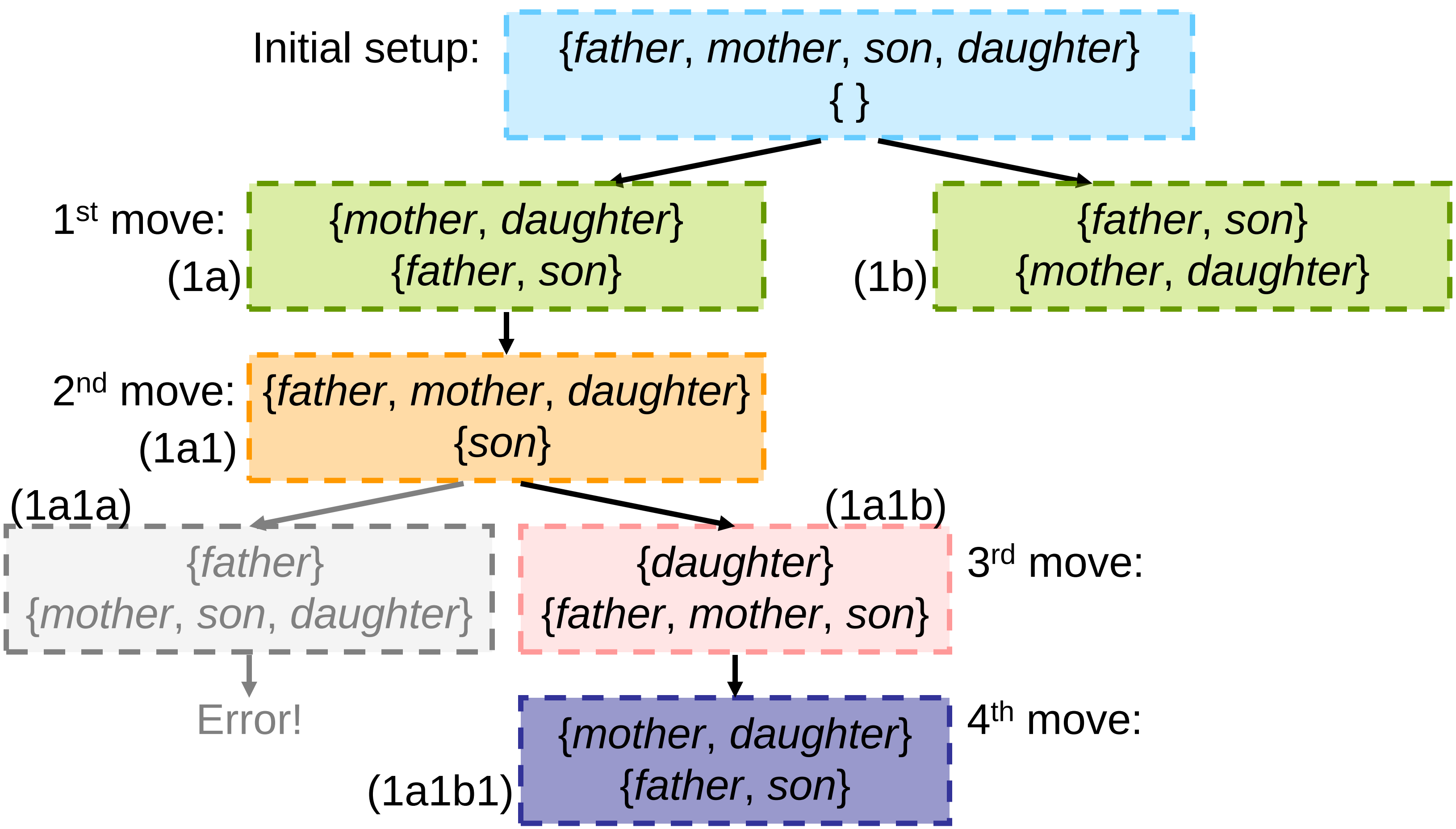
The Solution Tree



Since (1a1a) cannot lead to the solution, we backtrack to its previous step (1a1).

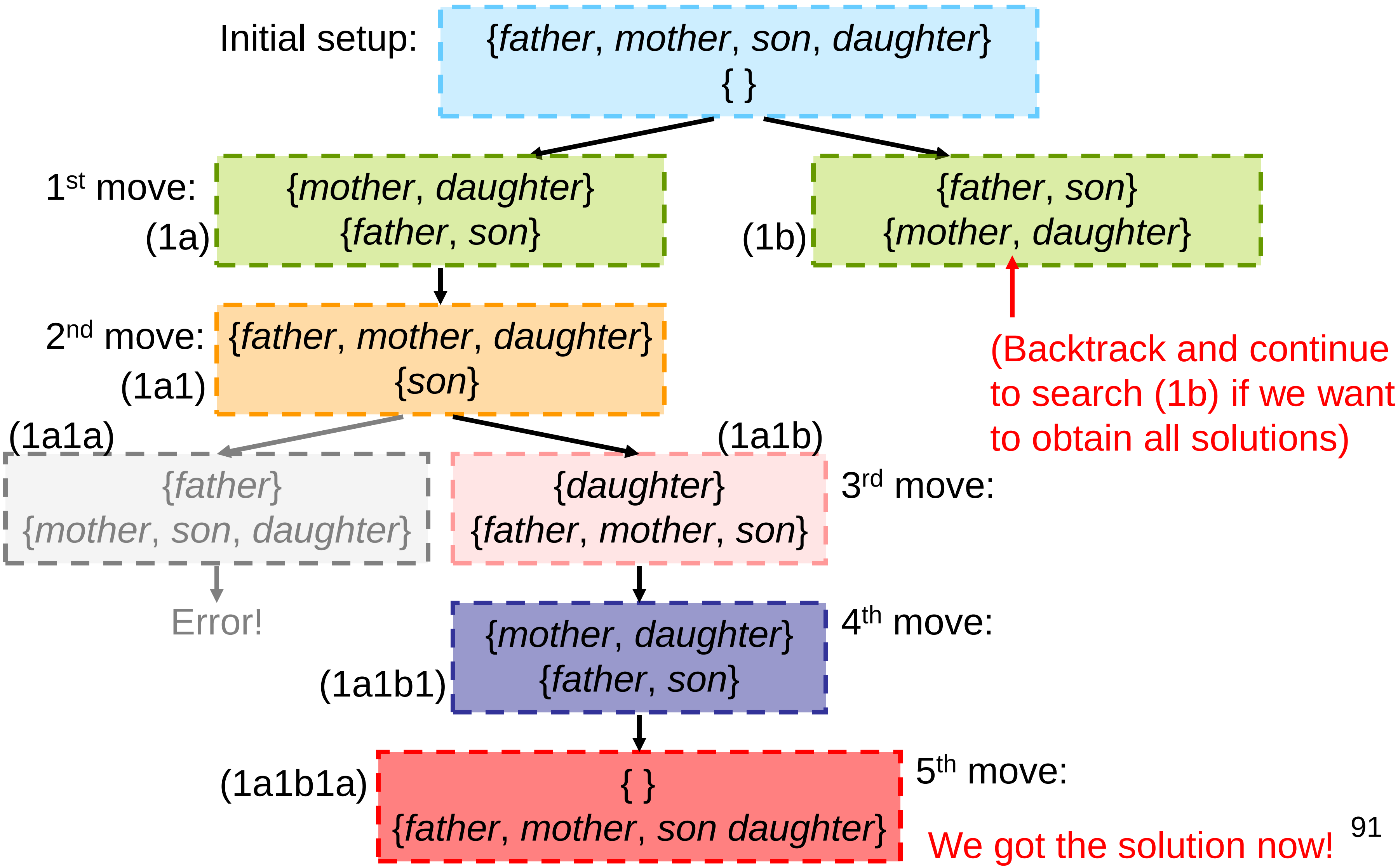
As (1a1) still have another move, we now consider (1a1b)

The Solution Tree



Using the same analysis, we know that (1a1b) has 1 move only

The Solution Tree



Backtracking Algorithm

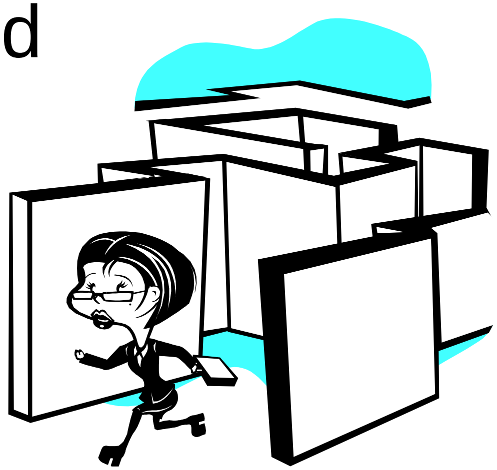
- The previous strategy is called **backtracking**
- You try each branch in the solution tree orderly in a systematic manner.
- You can abandon a branch (sub-tree) due to the imposed constraints. If this happens, go back to try another branch
- Used to find solutions to specific problems by trail-and-error
- Involves both **recursion** and a sequence of guesses that ultimately lead to a solution

The Second Example

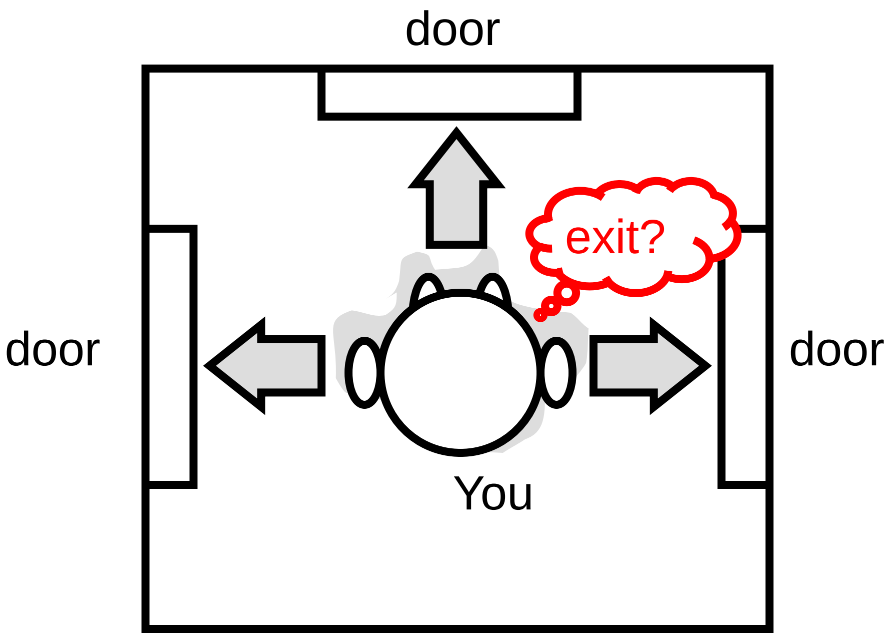
Maze Solver

Maze Solver

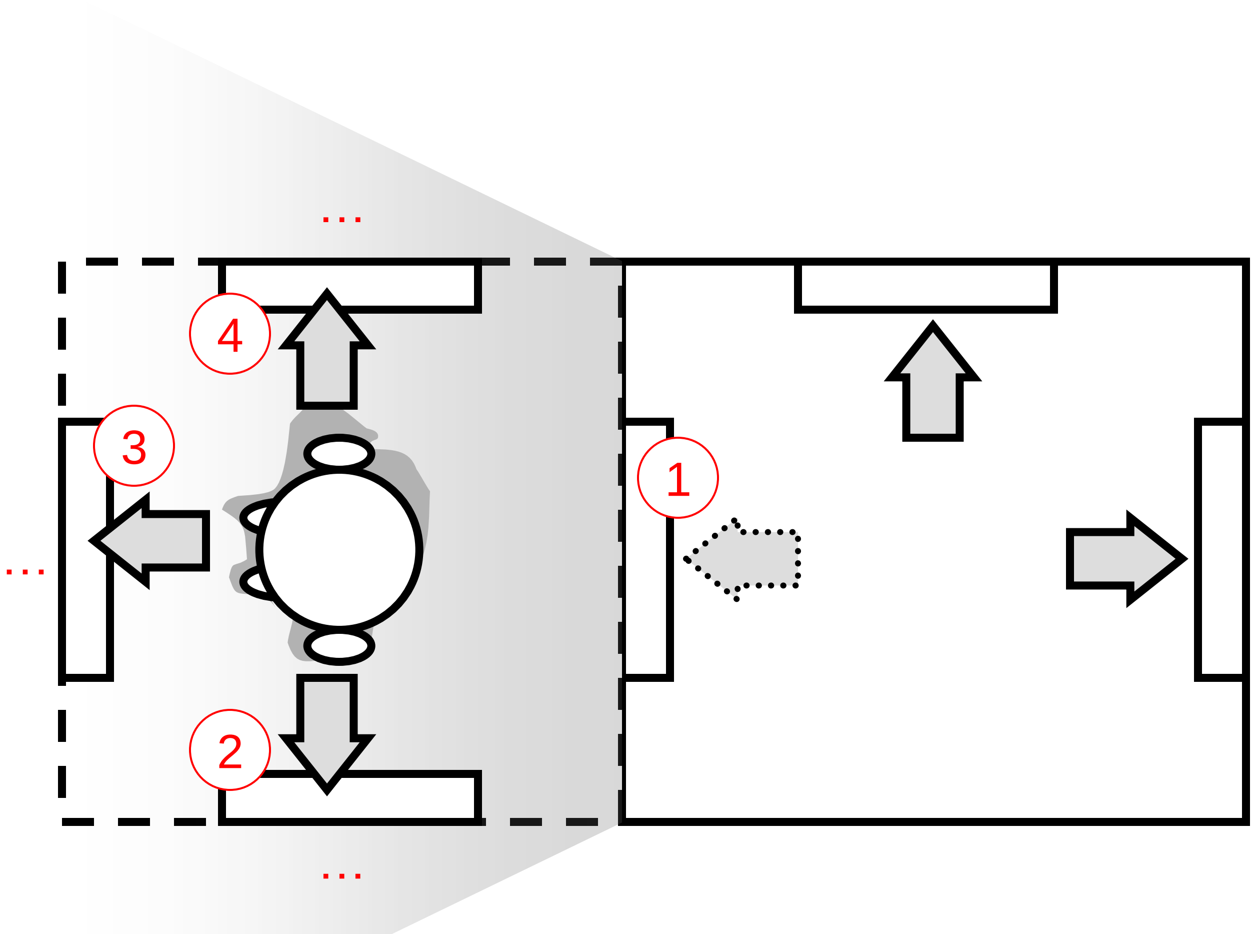
- How do you escape from a maze in a Role Playing Game (RPG)?
- e.g. use left or right convention
  - You probably try every way (always left or right first) and retrace your path (i.e. backtrack) when you reach a dead-end



How to Exit the Maze?



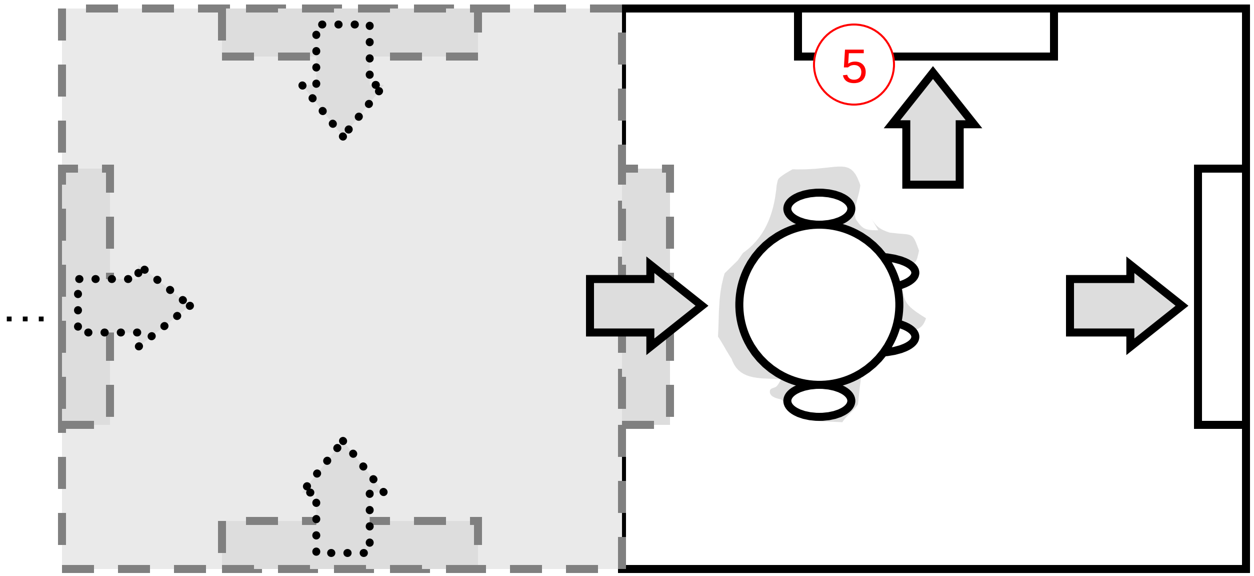
Try the Left Room?



Come Back If Left Room Cannot Reach the Exit

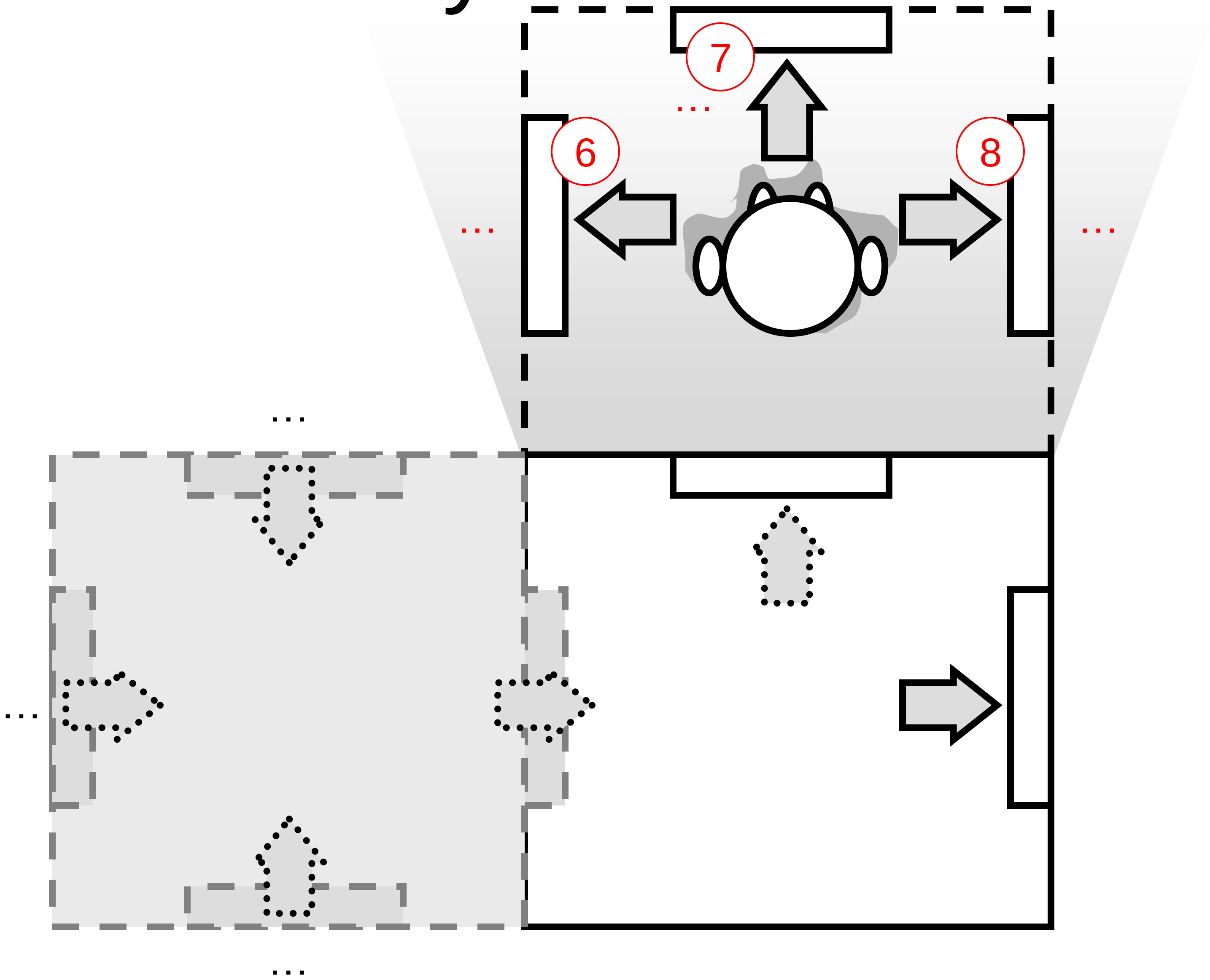
Come back eventually when every possible paths in left room are tried but still cannot reach the goal

…

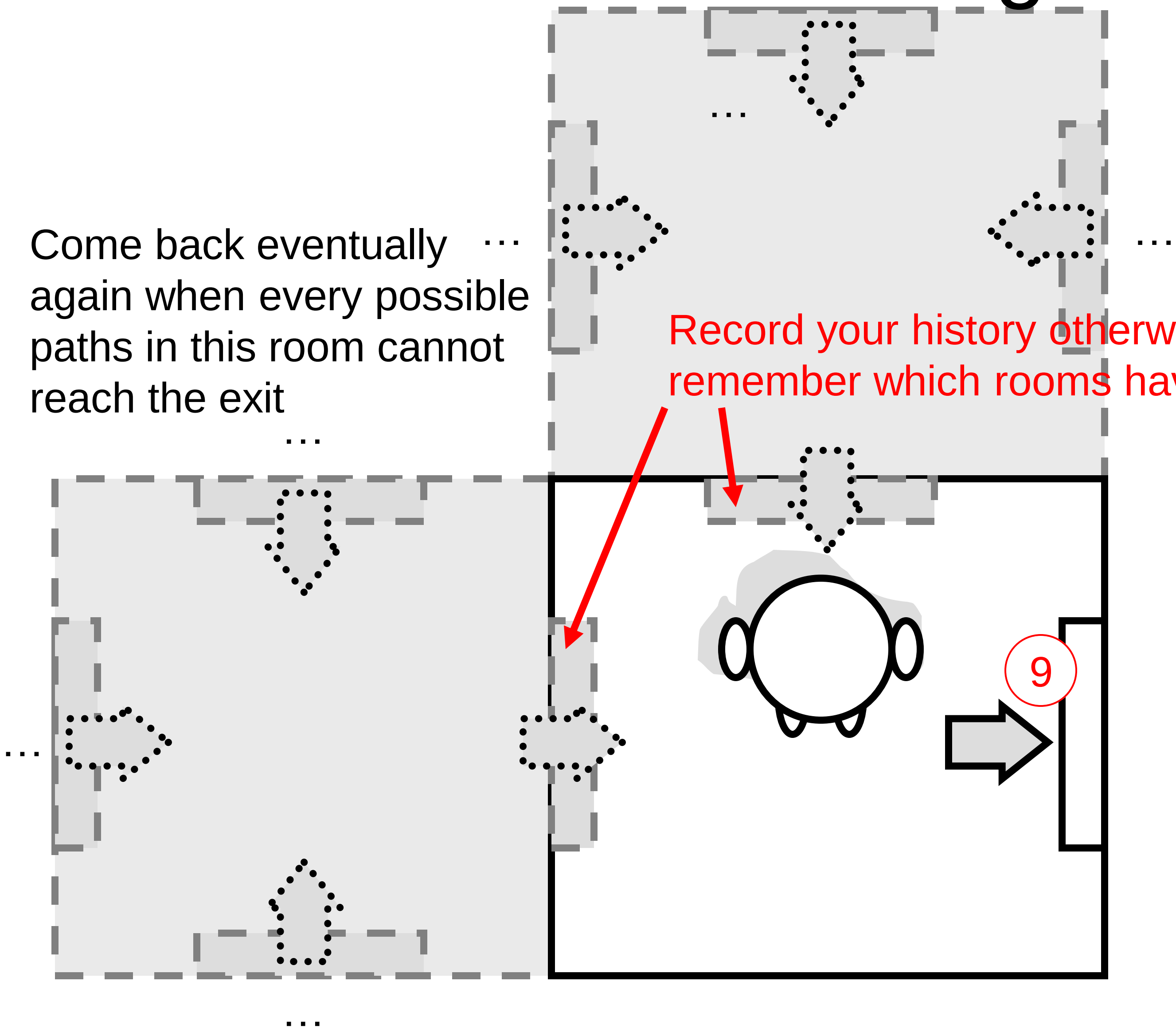


…

Try Another Room?



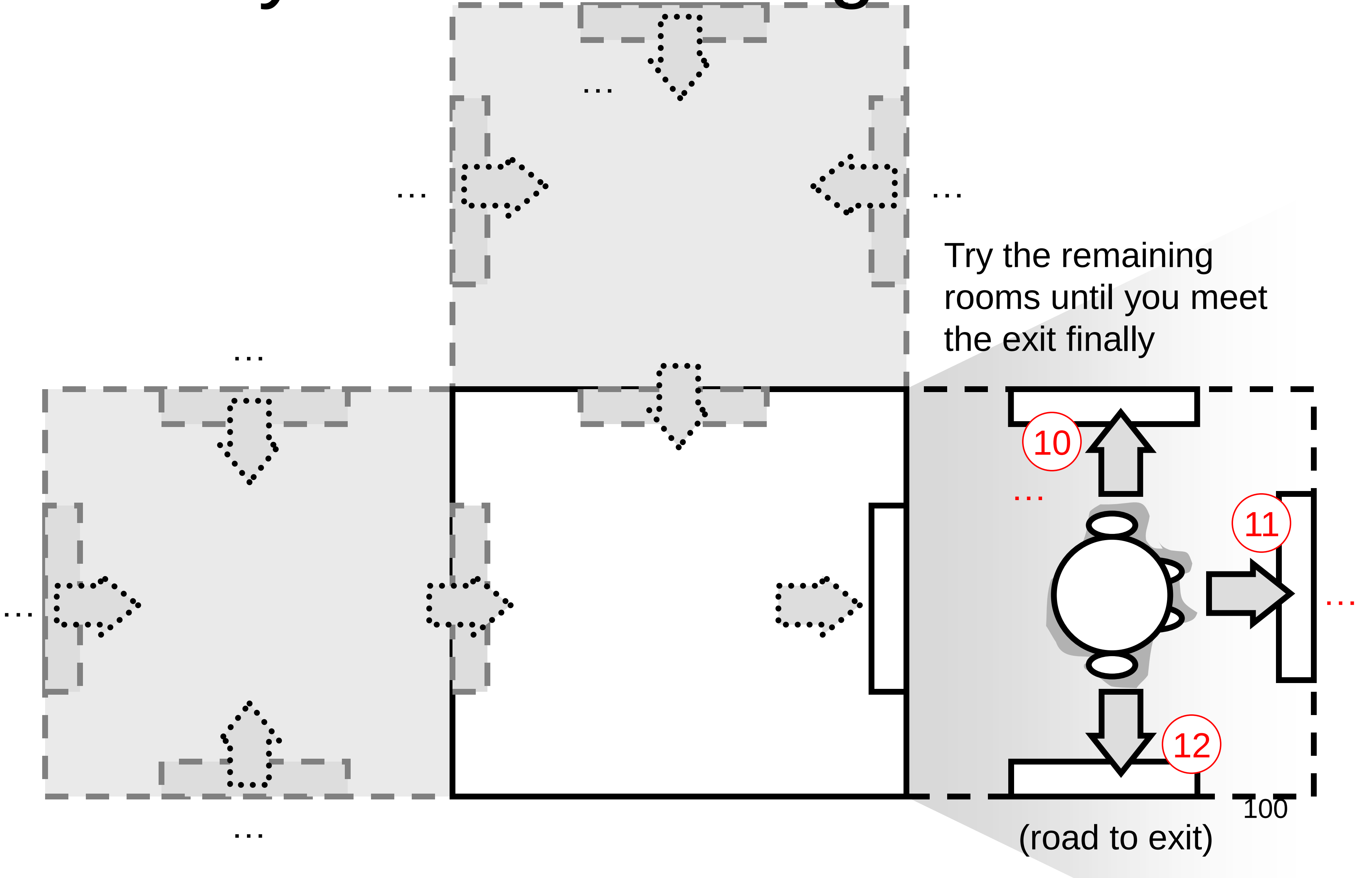
Come Back Again?



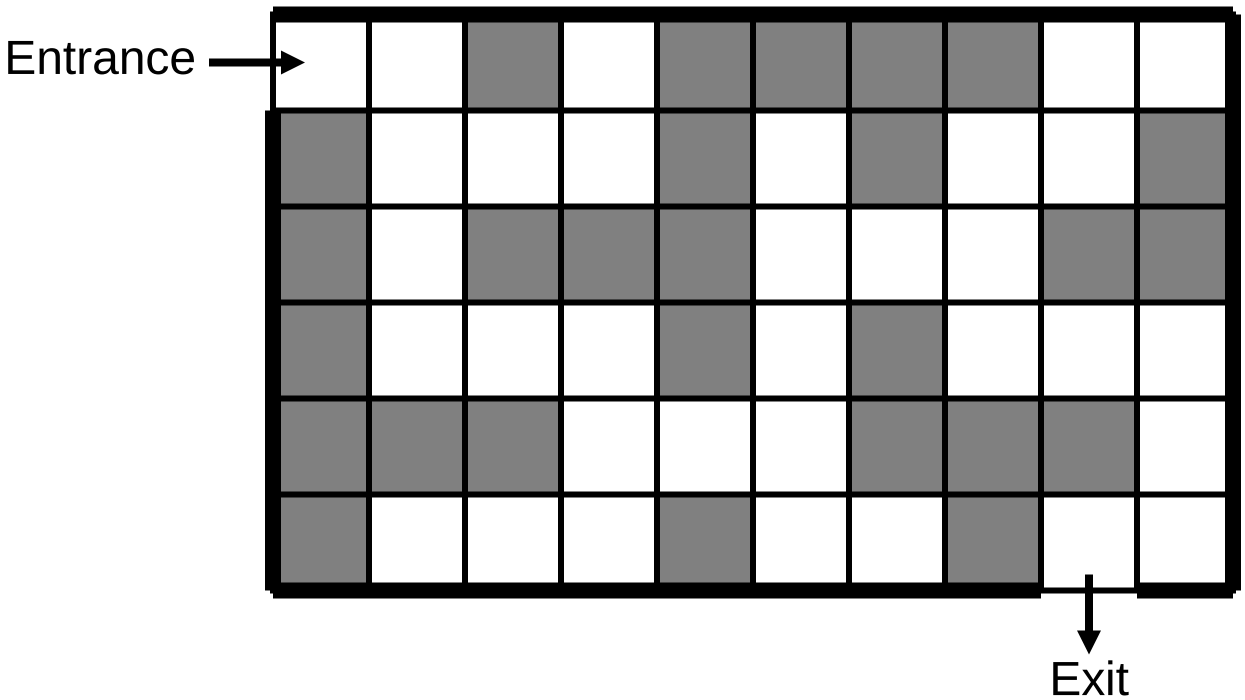
Come back eventually again when every possible paths in this room cannot reach the exit

Record your history otherwise cannot remember which rooms have been visited

Try the Remaining Rooms



Move from Entrance to Exit



Pseudocode

function try (Room *r*)
    if *r* is exit, the maze has been solved, return (*success*)
    if *r* has been visited or is a dead end, return (*fail*)

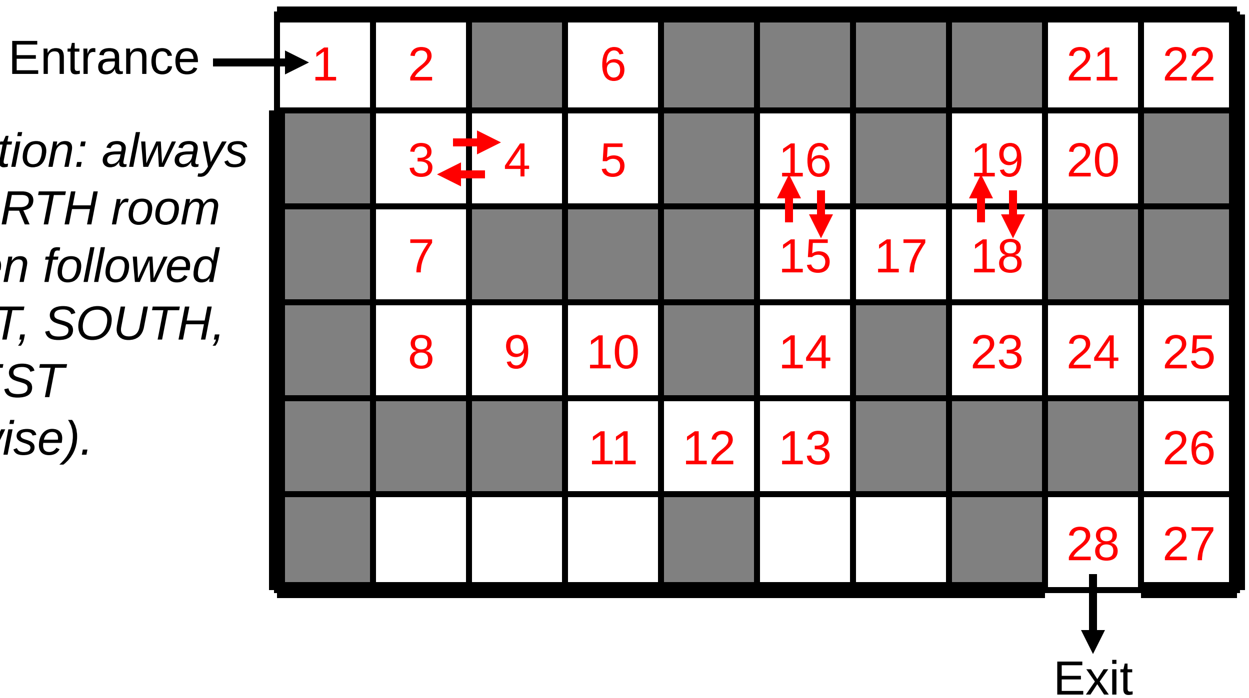
    for each connected room r_i {
        result = try(r_i)
        if *result* is *success*, record the move r_i, return (*success*)
    }
    return (*fail*)   // all connected rooms fail
end function

Base case

Recursive case

Move from Entrance to Exit

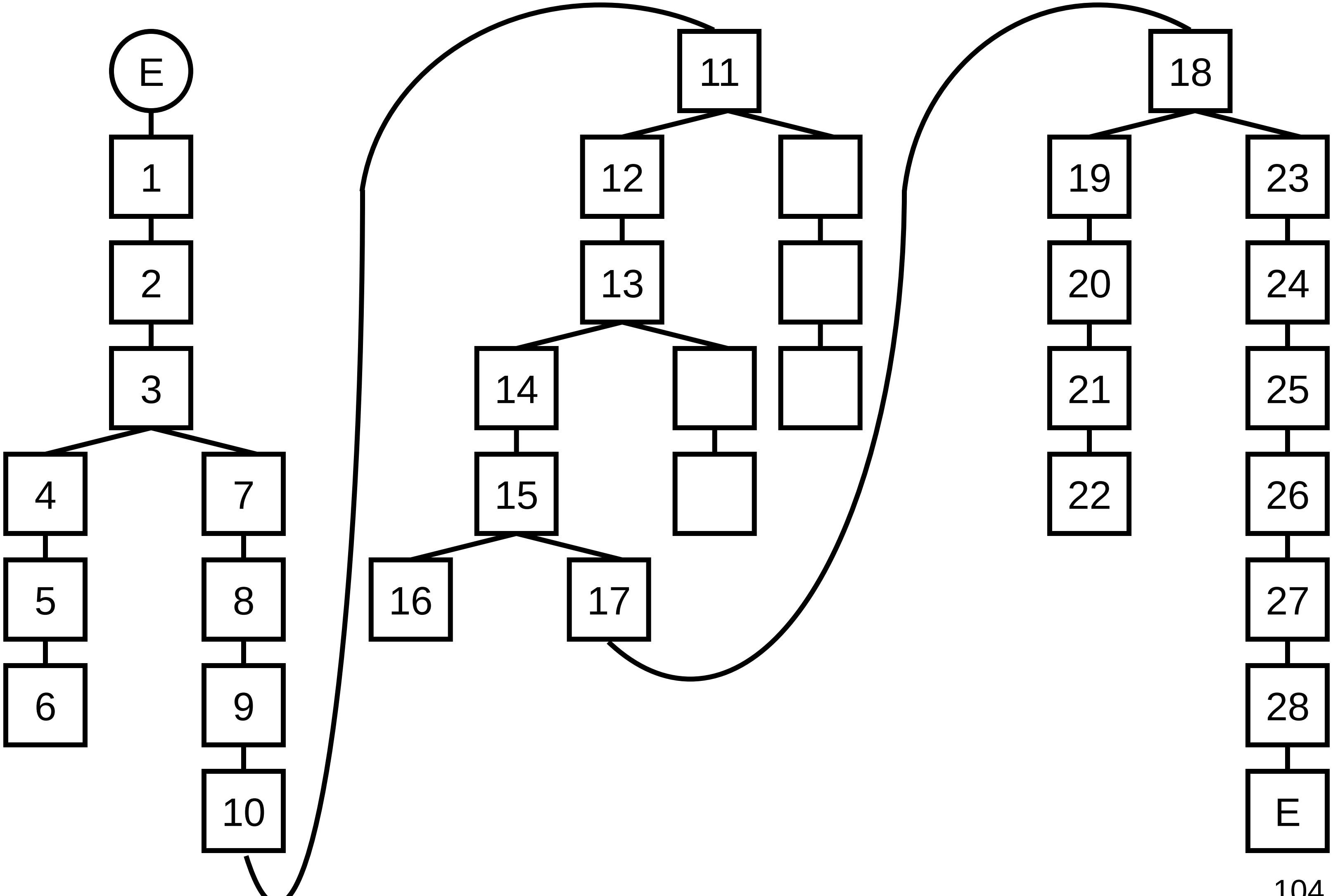
Convention: always visit NORTH room first, then followed by EAST, SOUTH, and WEST (Clockwise).



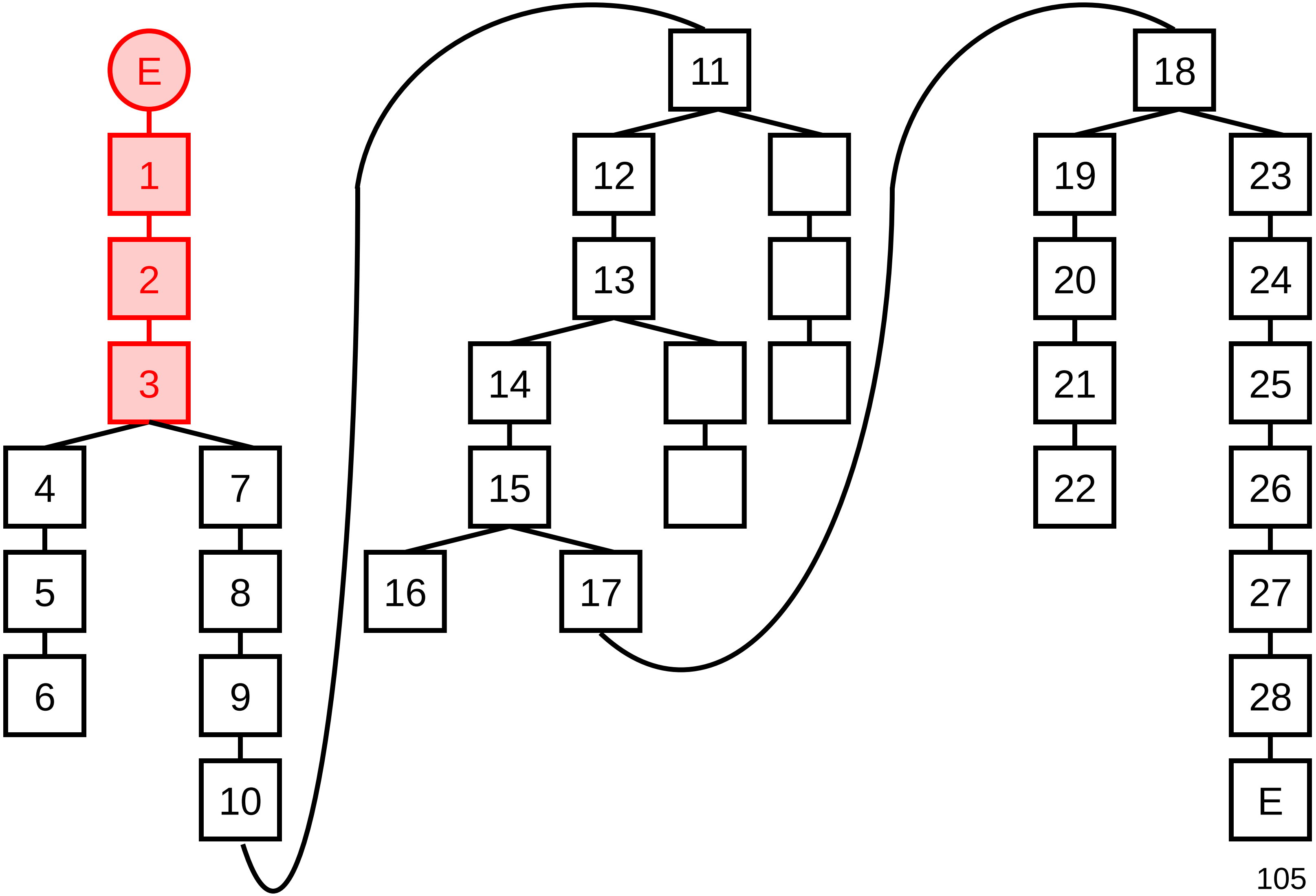
Note: for backtracking, 6 back to 5, then back to 4, then back to 3 and start over at 3 again

It's commonly misunderstand as 6 backtrack to 3 directly!

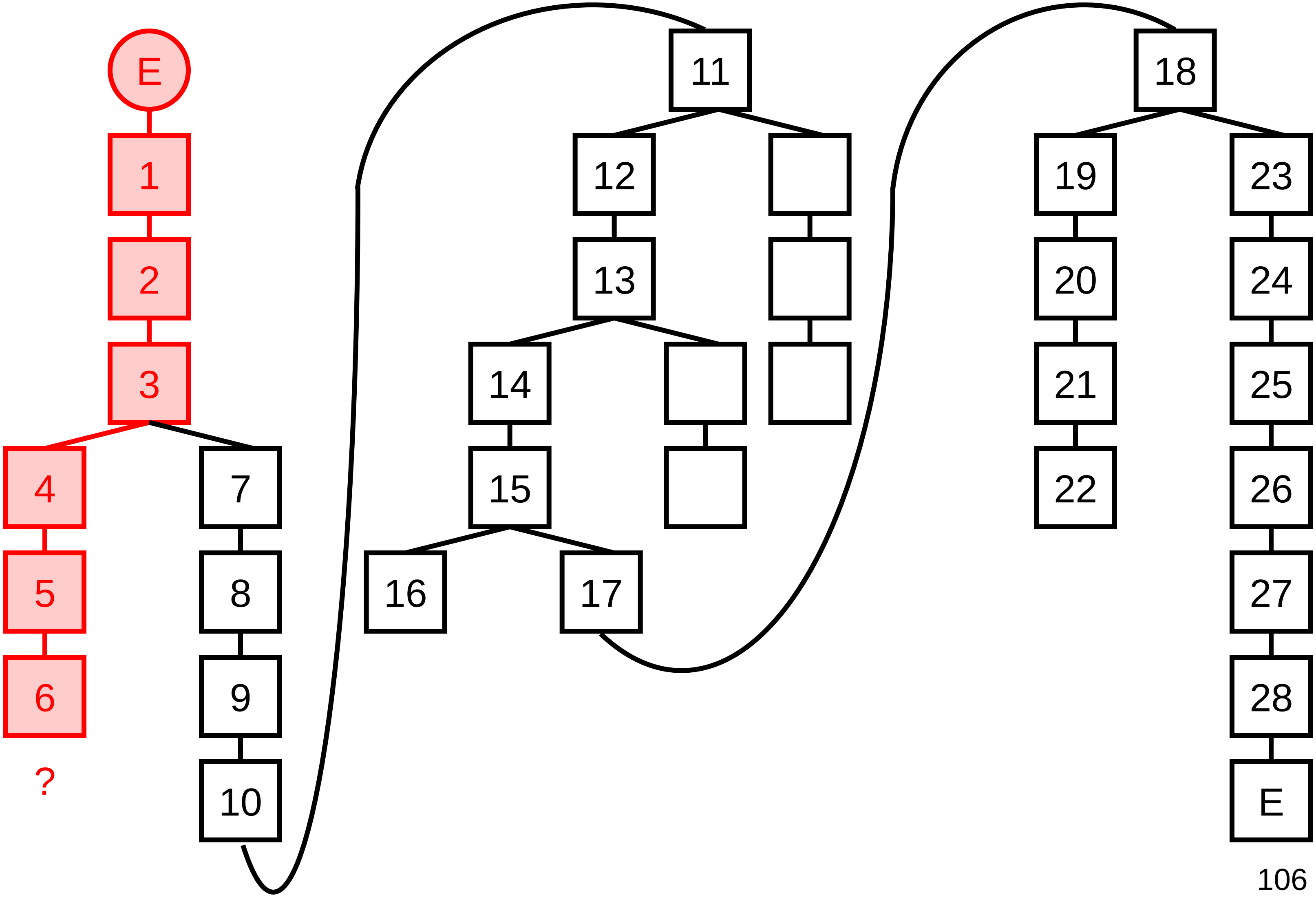
Transform into a Tree



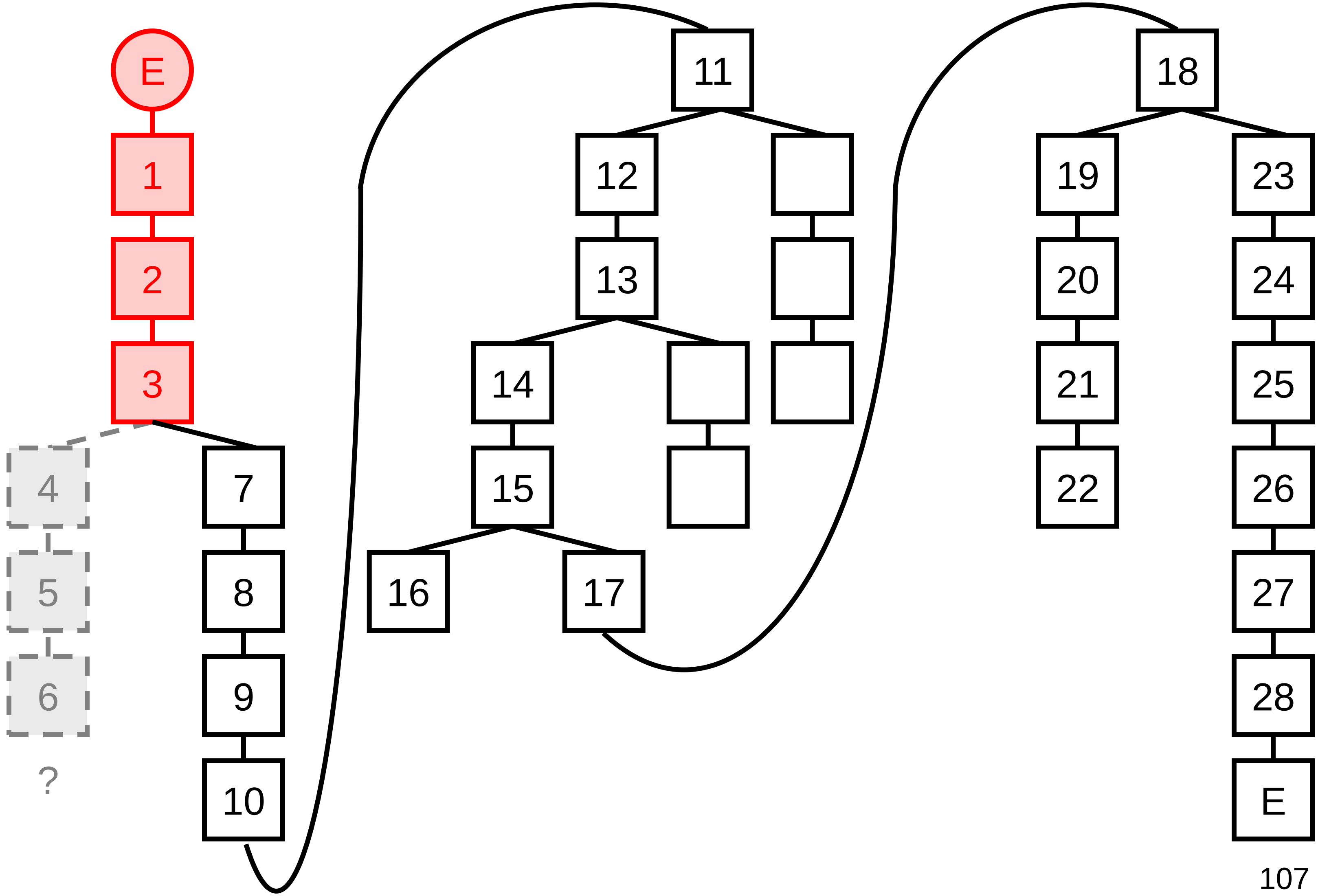
How Backtracking Works



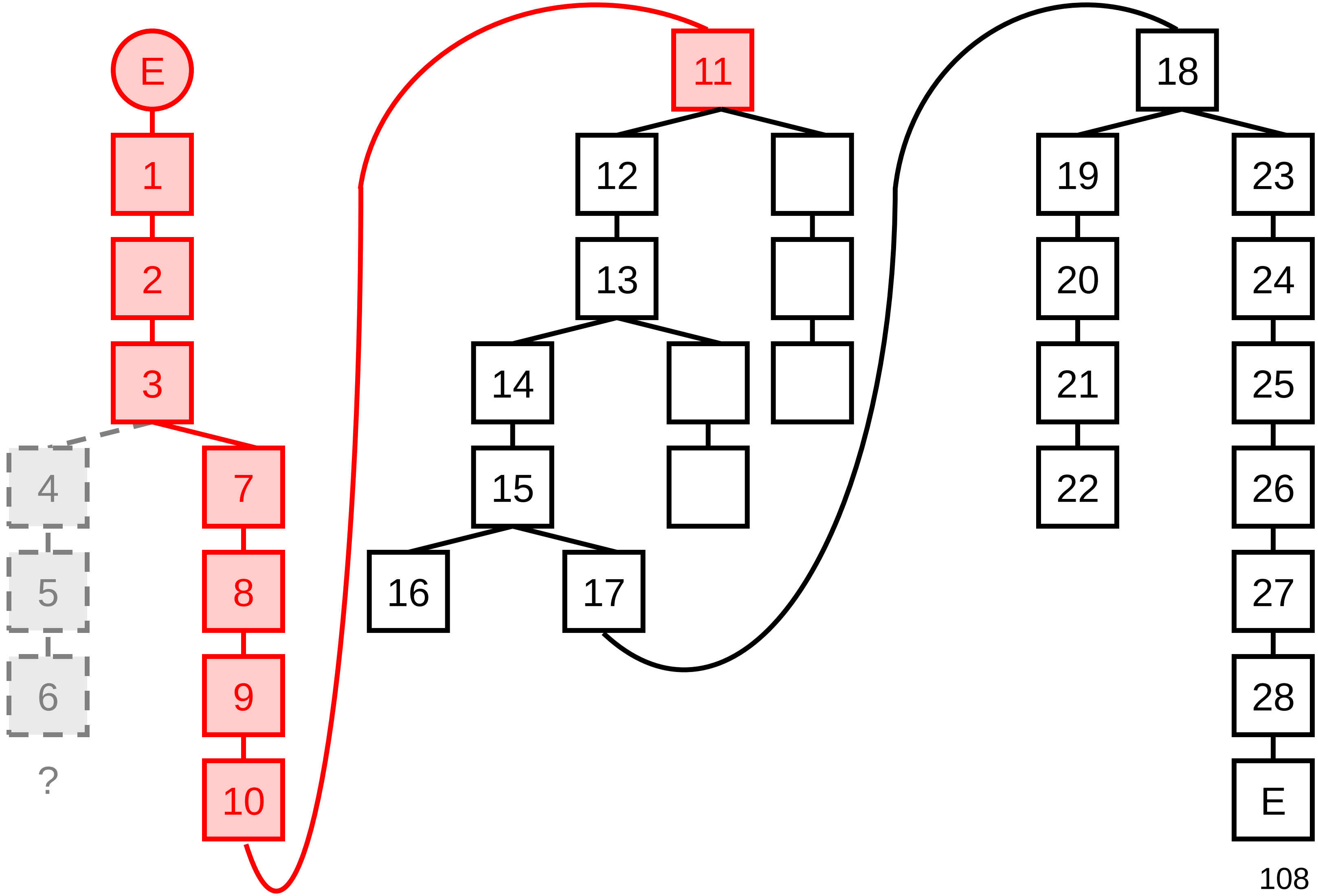
How Backtracking Works



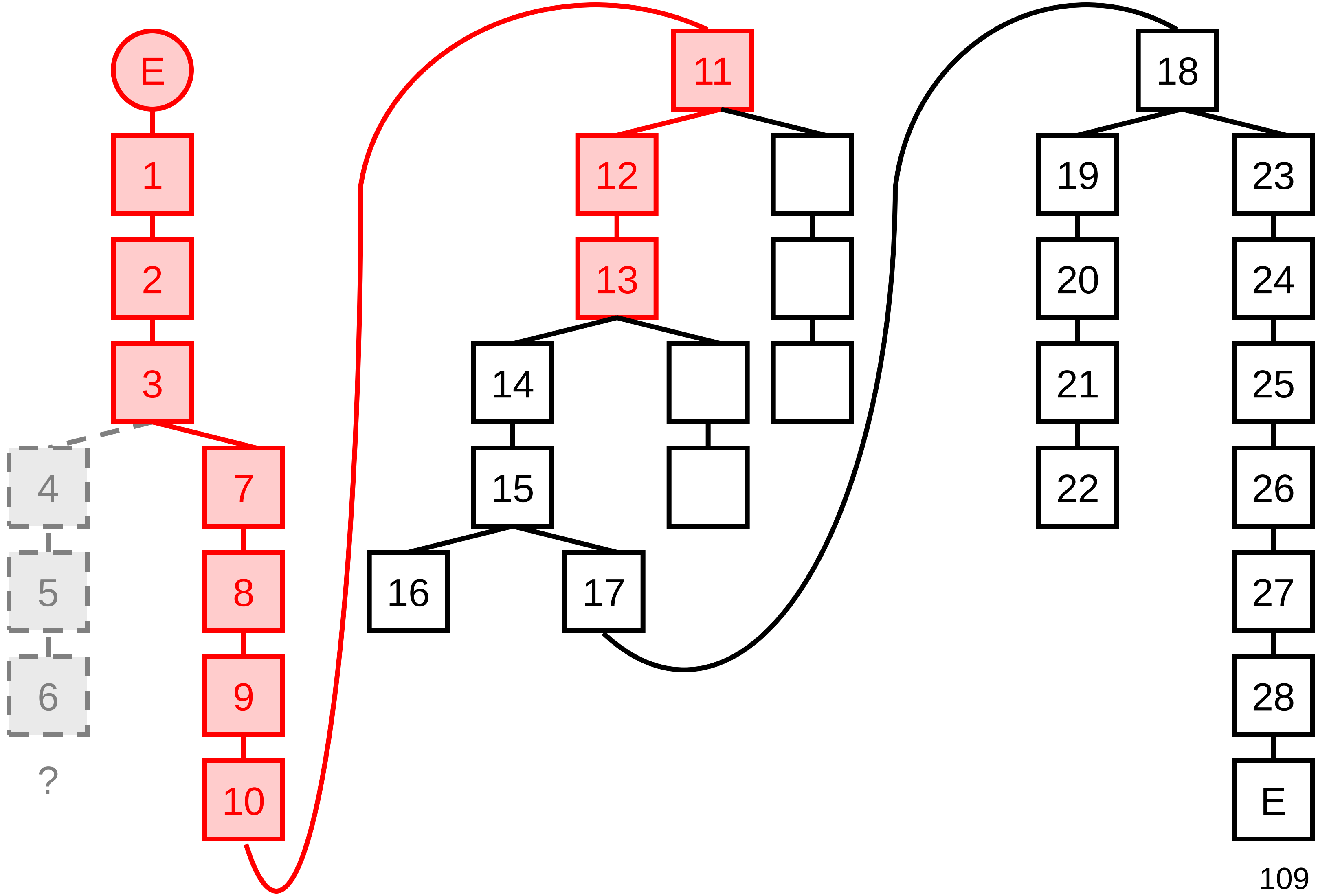
How Backtracking Works



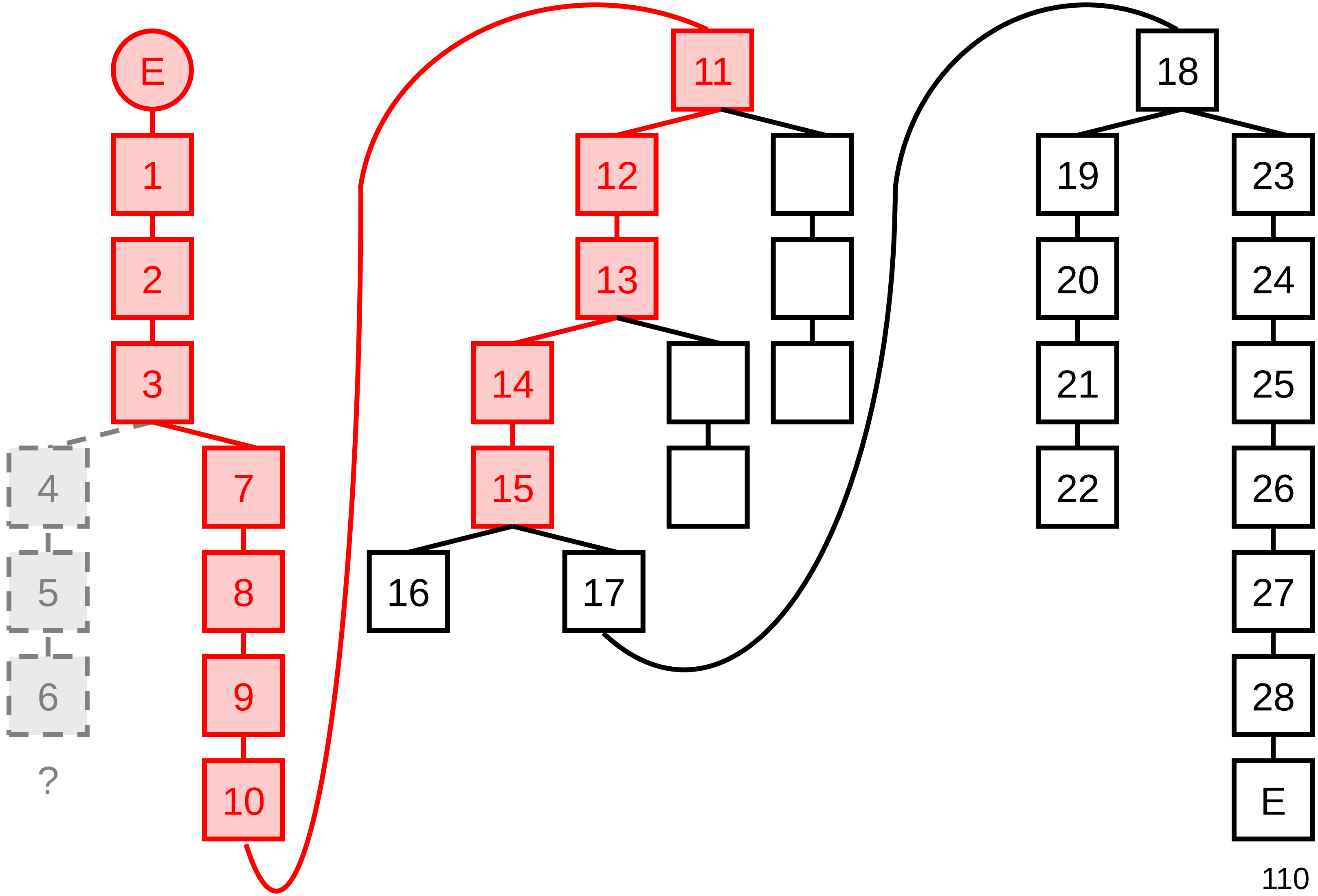
How Backtracking Works



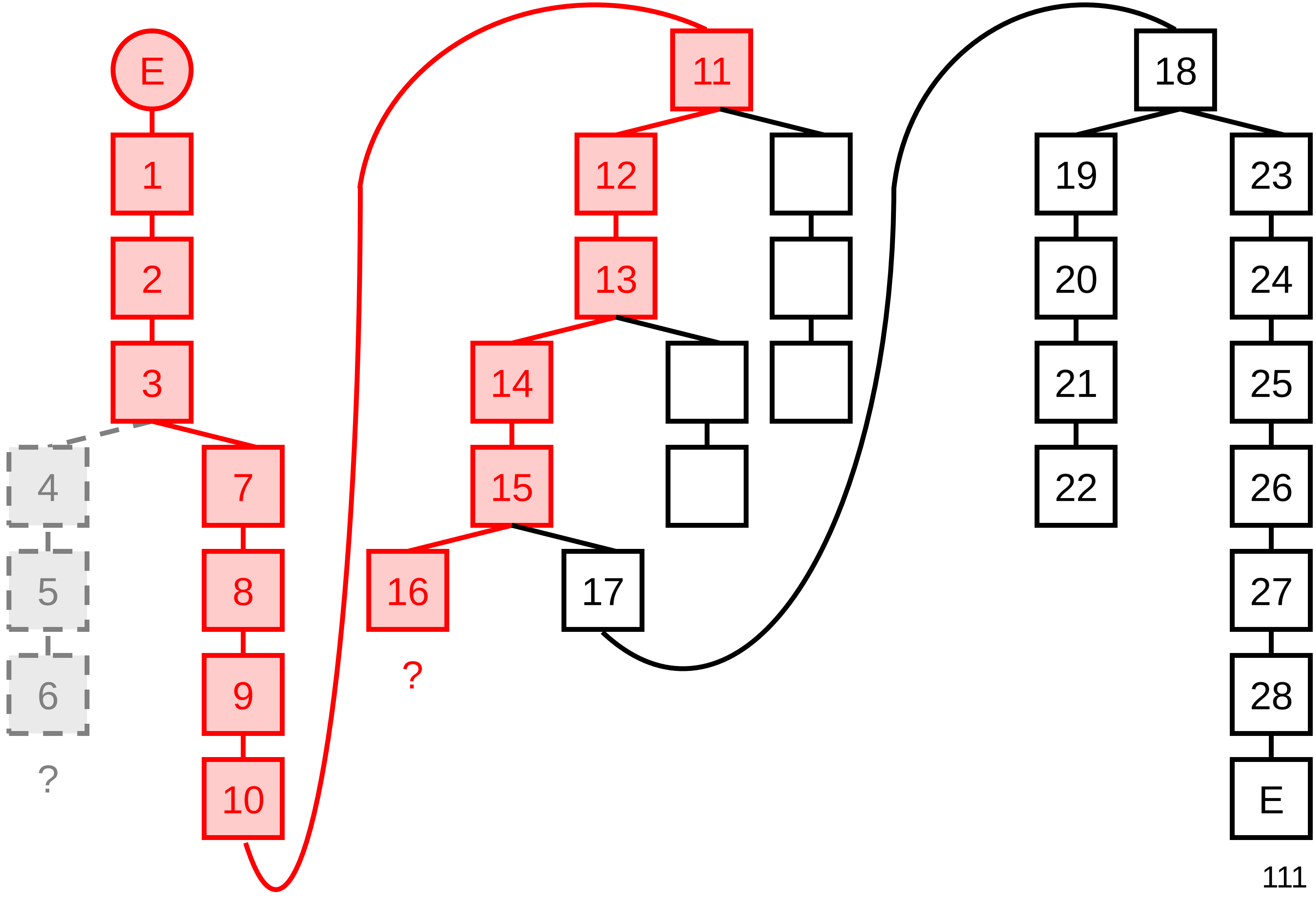
How Backtracking Works



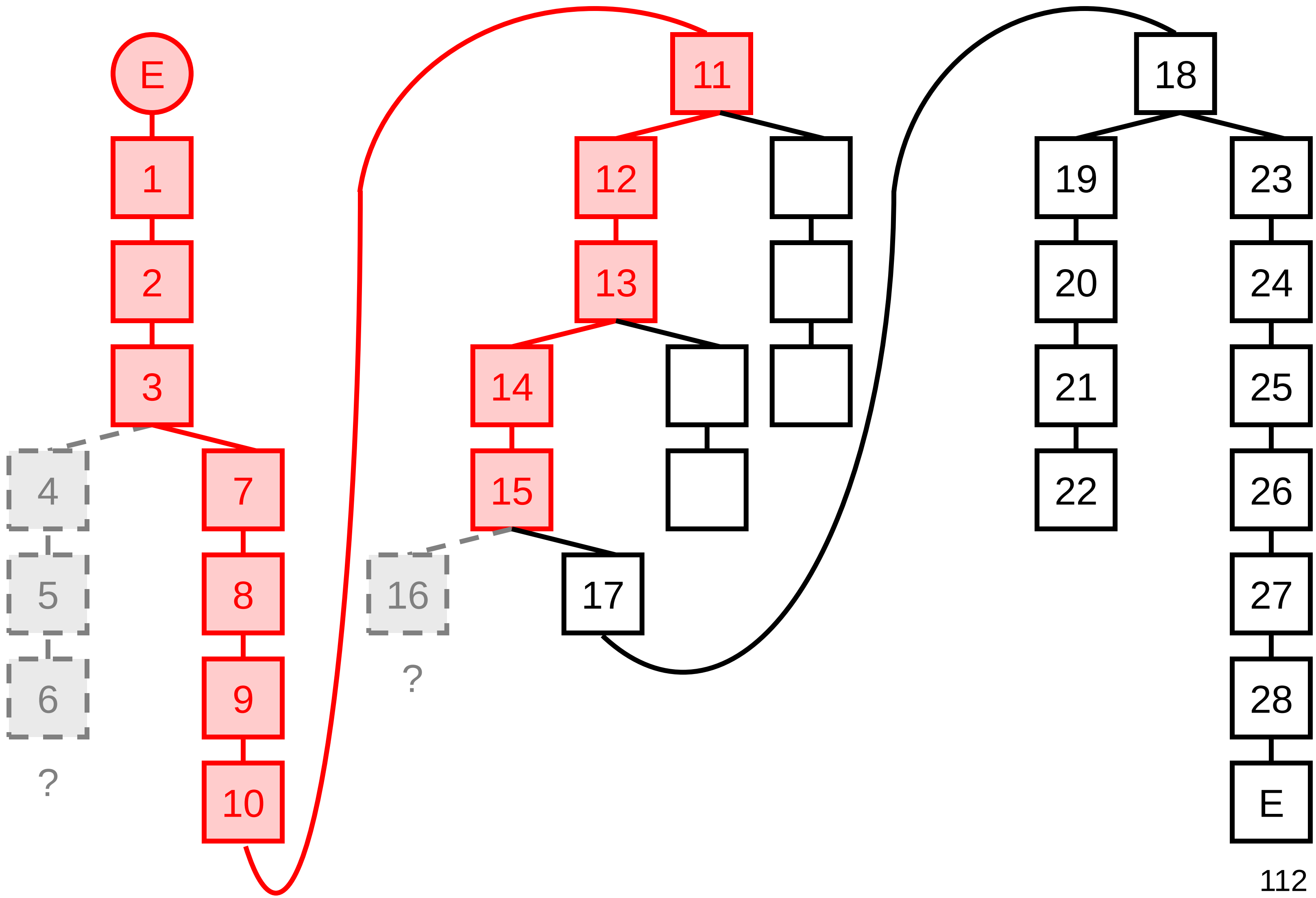
How Backtracking Works



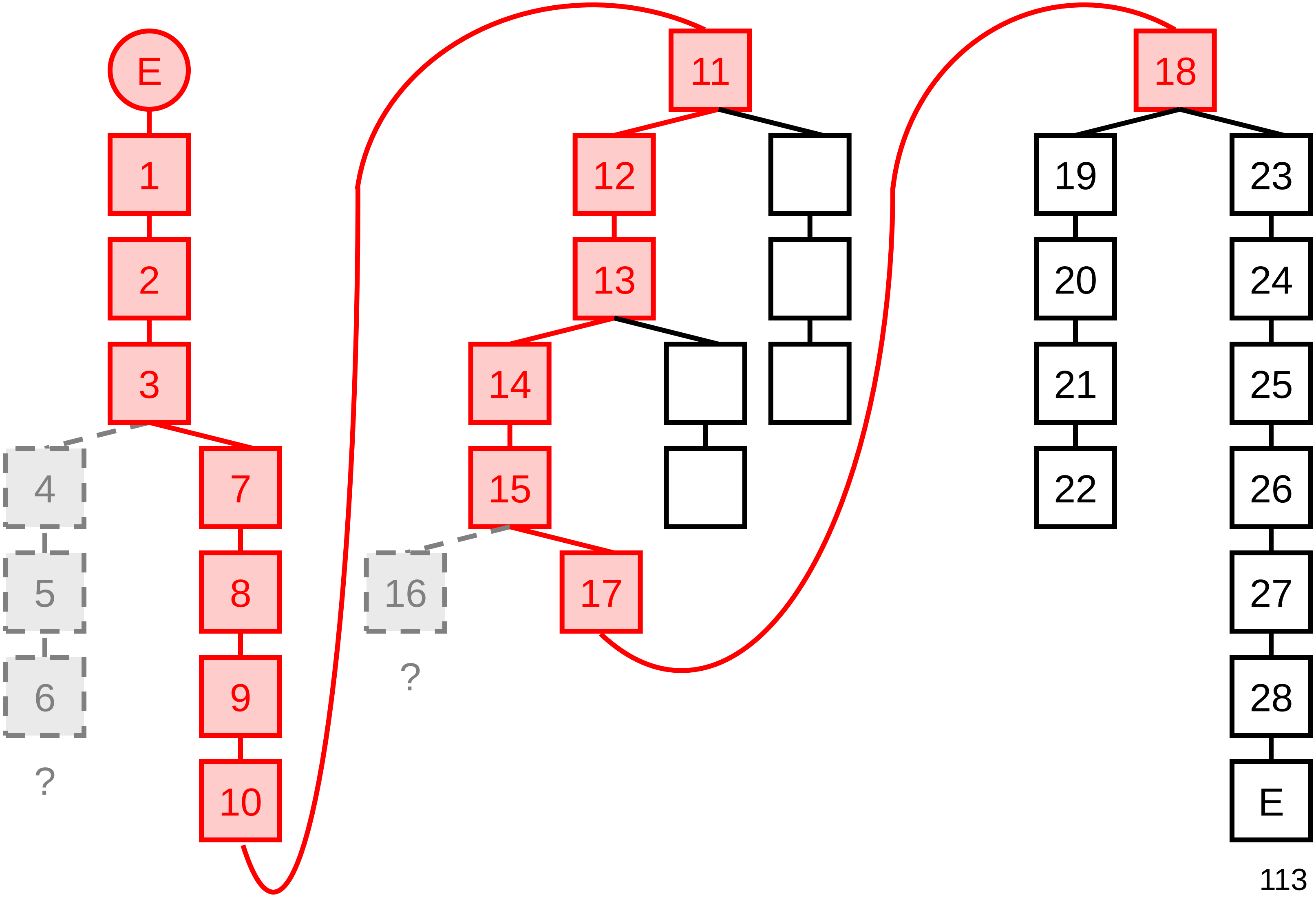
How Backtracking Works



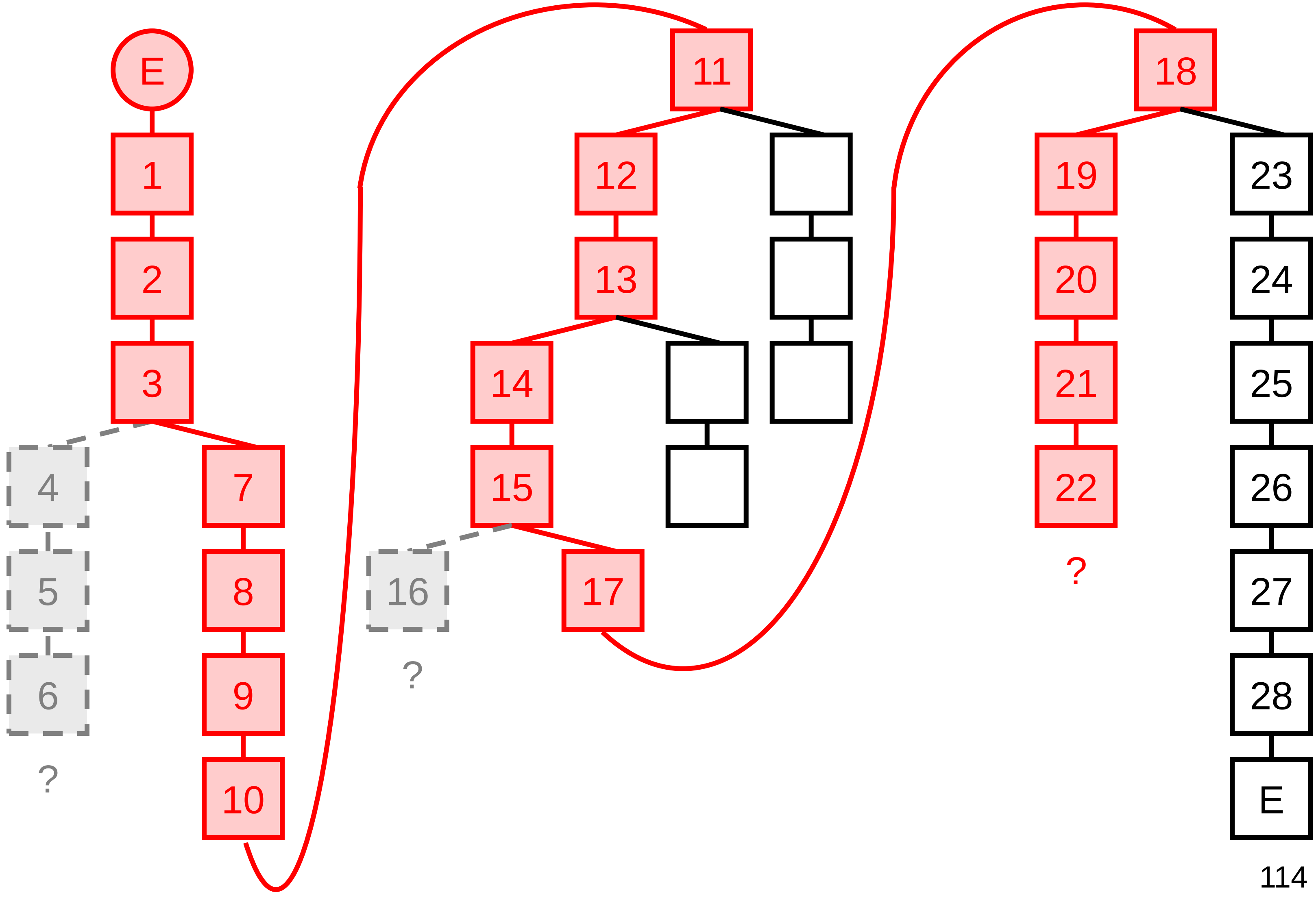
How Backtracking Works



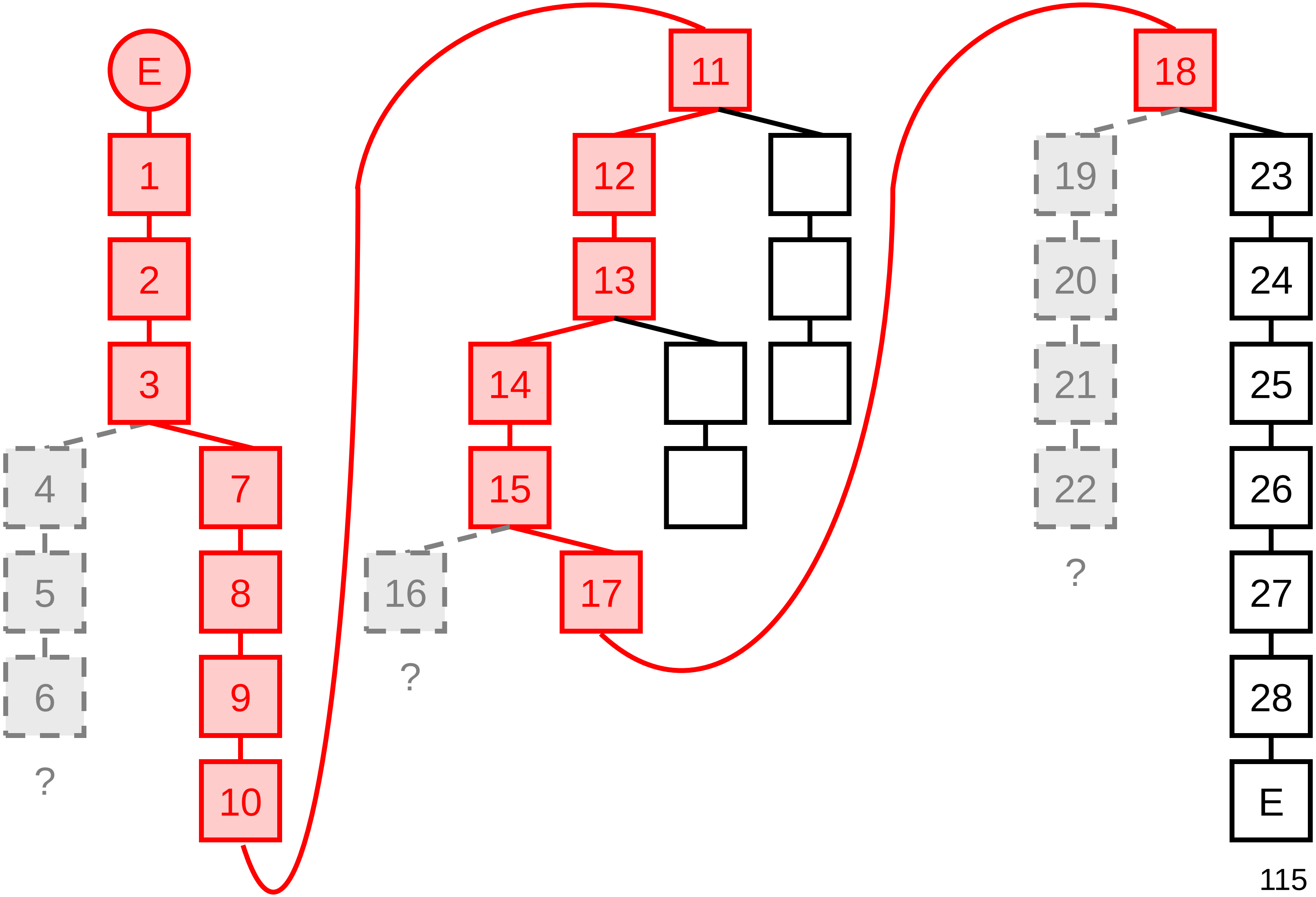
How Backtracking Works



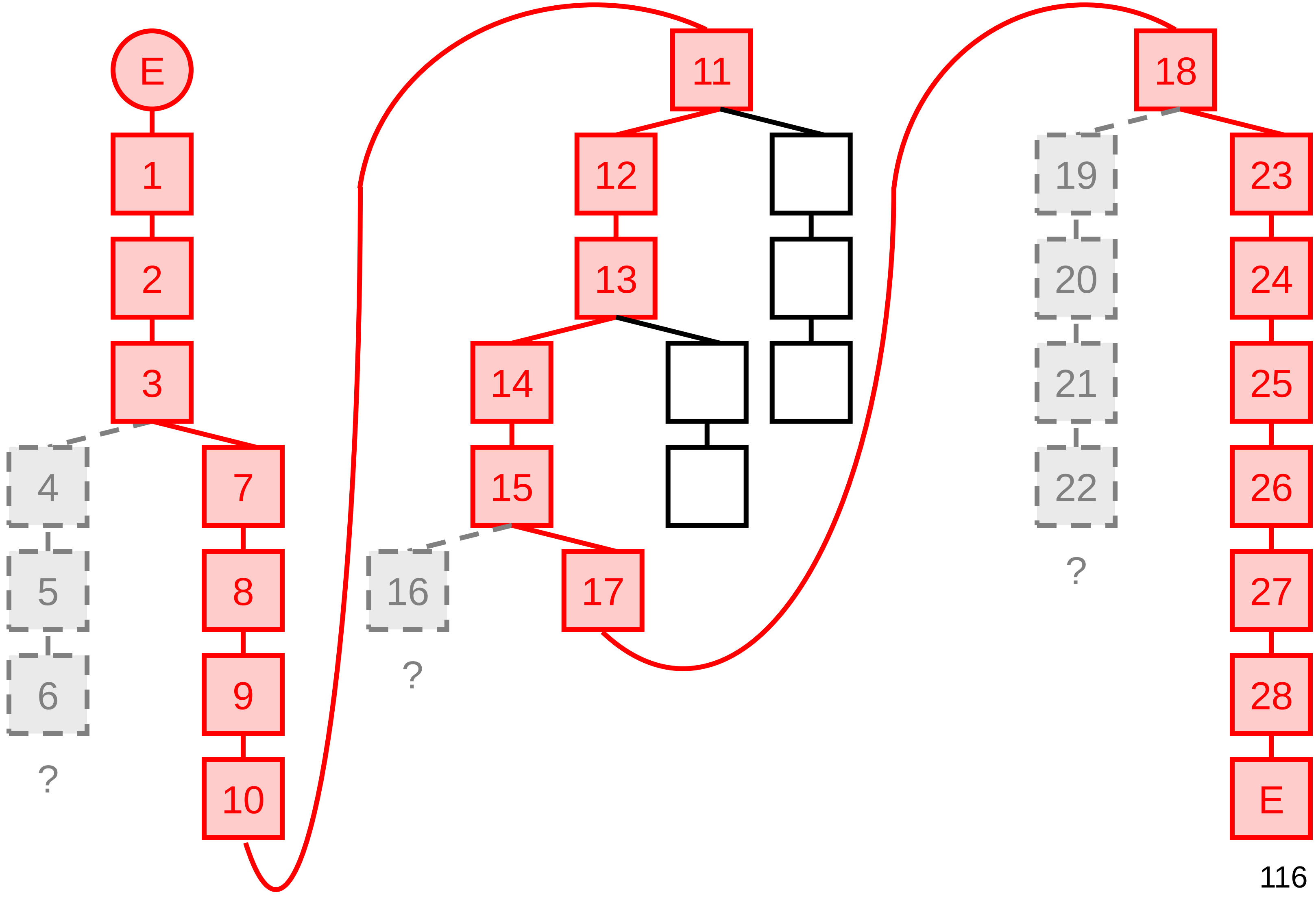
How Backtracking Works



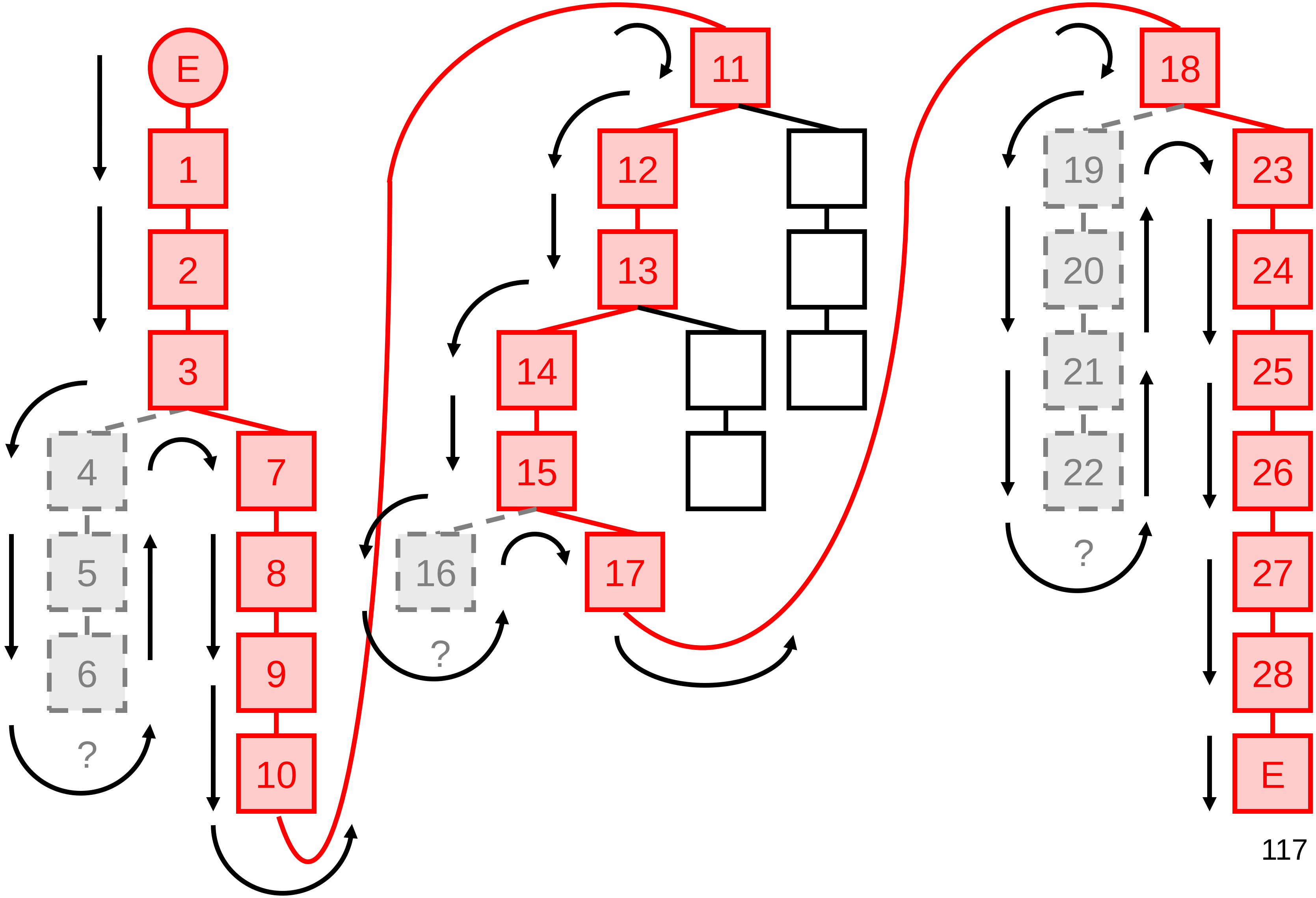
How Backtracking Works



How Backtracking Works



The Traverse Sequence



Solve Problem by Backtracking

- Trial-and-error
- Transform the search space into a tree diagram
- Try the next moves using recursion
- Whenever there is error (dead end) or solution, backtrack and return the result to previous point
- A depth-first recursive search (go down to the bottom of the tree first)

The Third Example

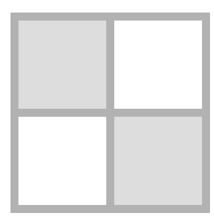
N Queens Problem

N Queens Problem

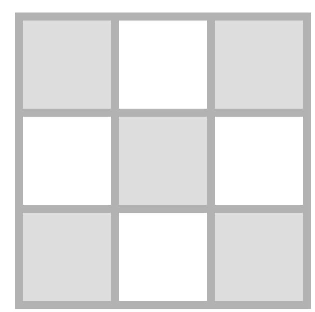
- A queen can capture a chess that is in the same row, column or diagonal
- To place *N* queens in a *N* x *N* chess board in the way such that the queens will not be captured by each other

N = 1

N = 2



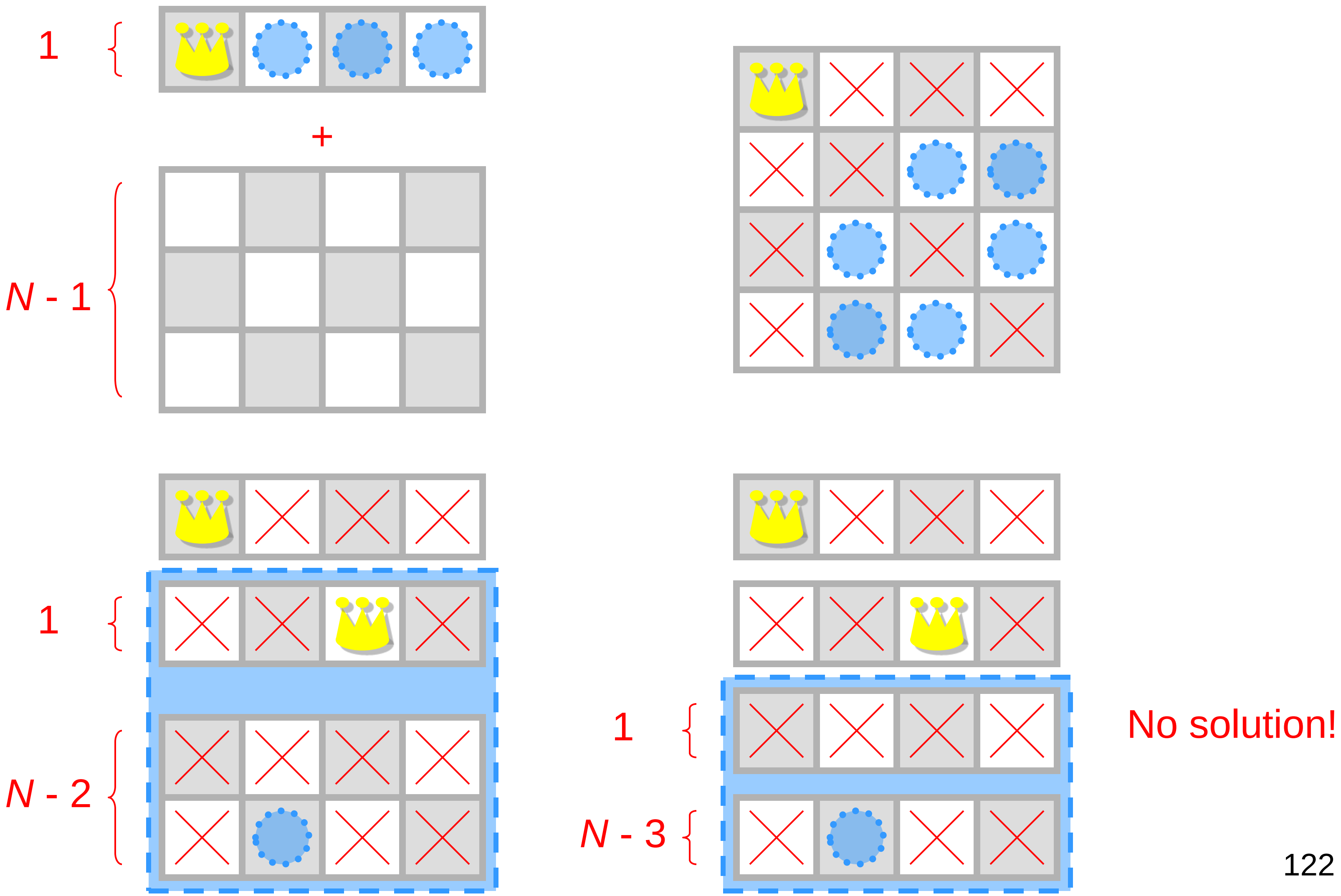
N = 3



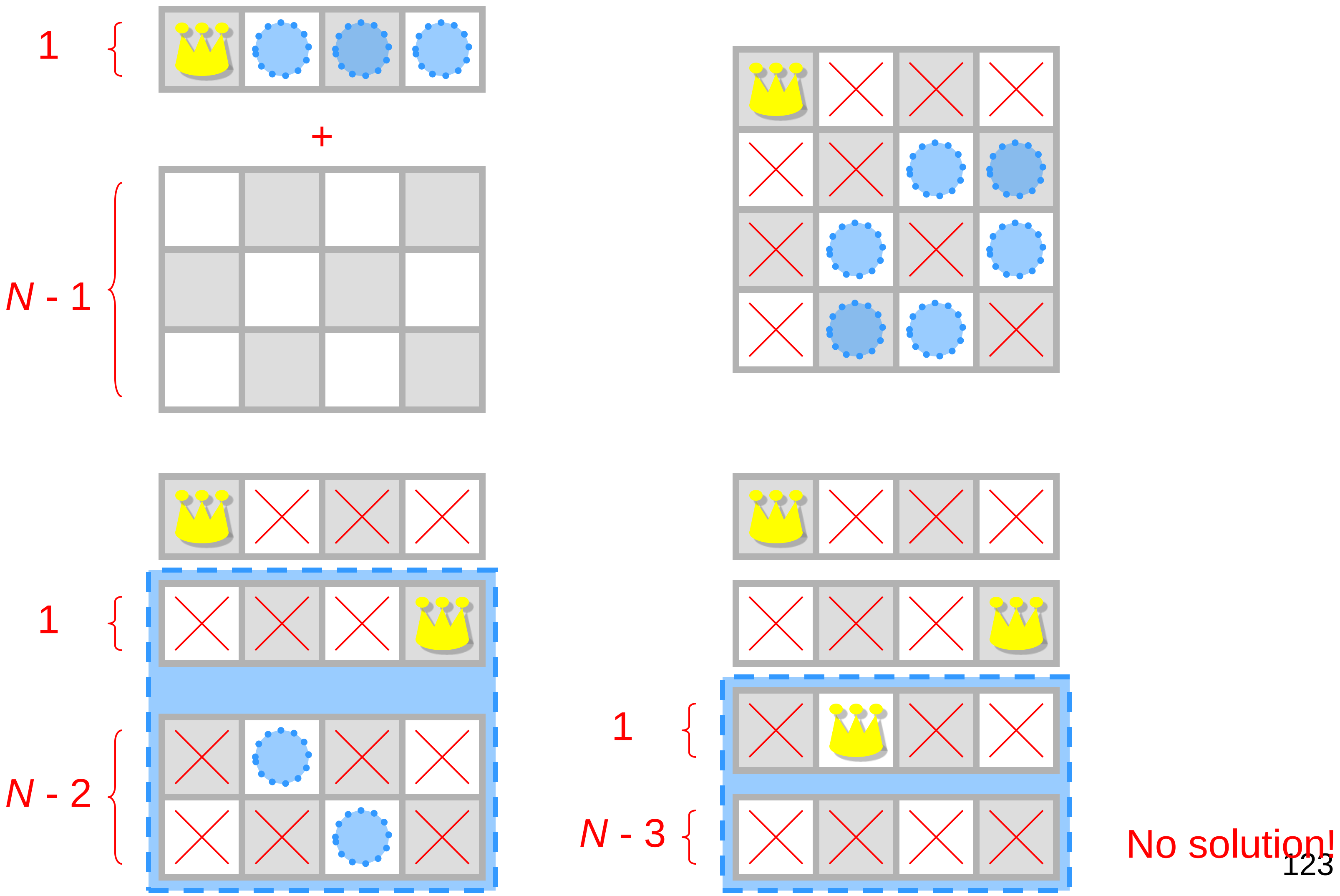
Outline of Solution

- Rather than viewing the chessboard as consisting of *NxN* squares, it can be seen as being comprised of *N* rows, each with *N* columns
- Each row/column/diagonal must have one Queen only
- Starting from the first row and first column, place a Queen and then take a step forward by making a recursive call
- This call will return having either **succeeded** in finding a solution, or having **failed**, meaning that there is no solution given the current placement of Queens.
- If the recursive call fails, we move the Queen to next column and try again.

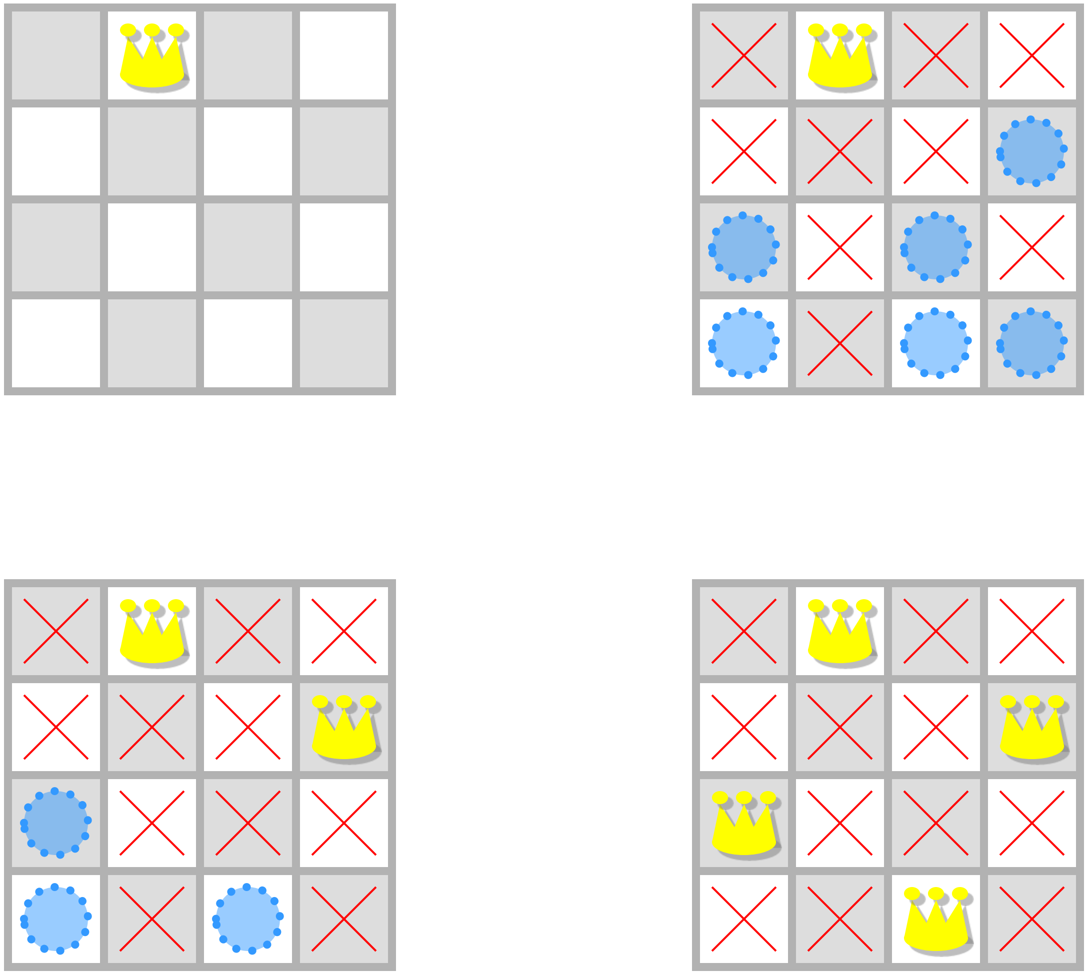
Trial-and-Error



Trial-and-Error

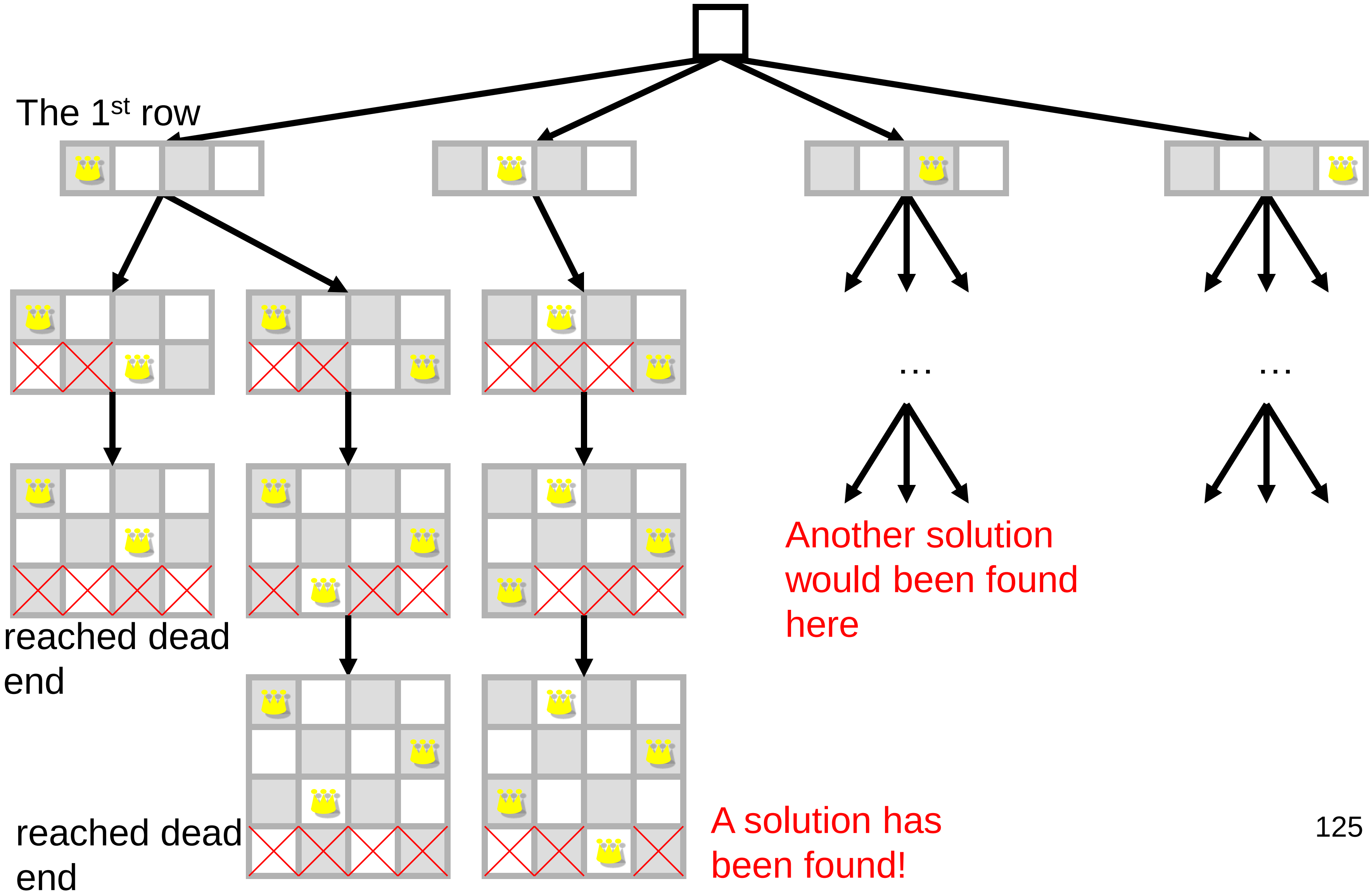


Trial-and-Error



How to translate into tree diagram?

Transform into Tree Diagram



N Queens's Pseudocode

```
function try (Row r)
    if r is beyond the last row, a solution has been found!
        return (success)

    for each coordinate (r, col_i) in row r {
        if coordinate (r, col_i) is safe {
            record the coordinate (r, col_i)
            result = try(r + 1)
            if result is fail, erase previous (r, col_i) record
            if result is success, return (success)
        }
    }
    return (fail)              // all positions not safe
end function
```

Base case

Recursive case