Recursive algorithms: optimizations

Lecture 21

Two optimization techniques:

- Recursion with memoization
- Recursion with backtracking

Memoization

Recall: When recursion feels natural

- Many problems are already defined as recursive problems
- Recursion is a natural solution to these problems

Recursive algorithms are particularly appropriate when the underlying problem or the data to be treated are defined in recursive terms

$$F_n = n * F_{n-1}$$

Recurrence relations

Note how the function F(n) is defined through F(n-1)

$$F_n = \begin{cases} 1, & \text{if } n = 0 \text{ or } n=1 \\ n * F_{n-1}, & \text{if } n > 1 \end{cases}$$

1, 1, 2, 6, 24, 120…

Factorial

$$F_n = \begin{cases} 0, & \text{if } n=0 \\ 1, & \text{if } n=1 \\ F_{n-1} + F_{n-2}, & \text{if } n > 1 \end{cases}$$

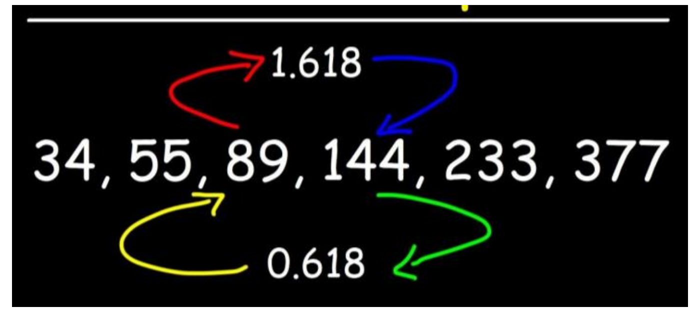
0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Fibonacci

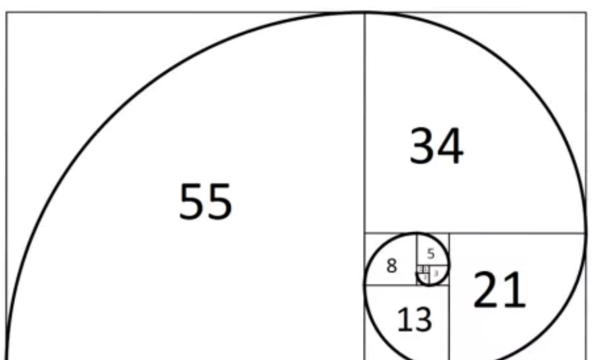
Fibonacci numbers

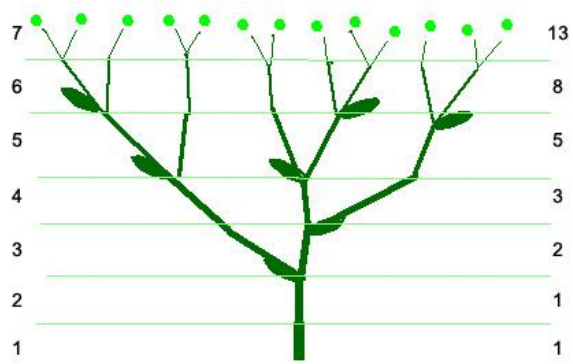
$$F_n = \begin{cases} 0, & \text{if } n=0 \\ 1, & \text{if } n=1 \\ F_{n-1} + F_{n-2}, & \text{if } n > 1 \end{cases}$$

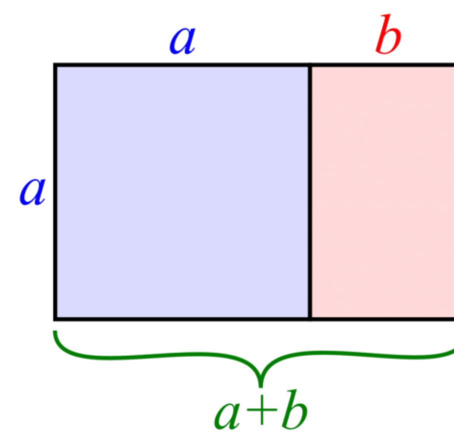
0, 1, 1, 2, 3, 5, 8, 13, 21, 34...



Golden ratio

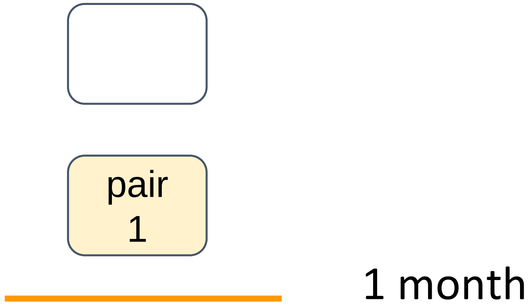




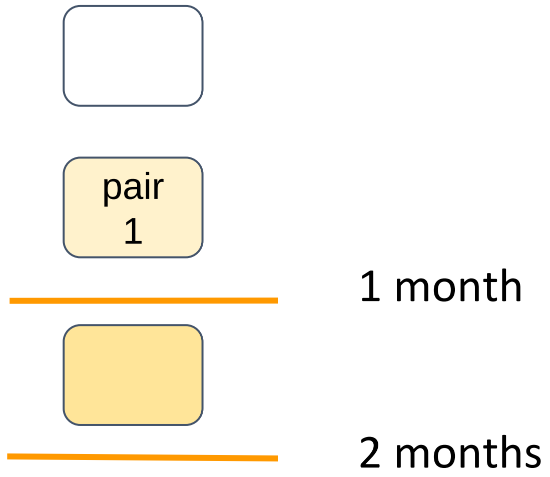


Fibonacci rabbits

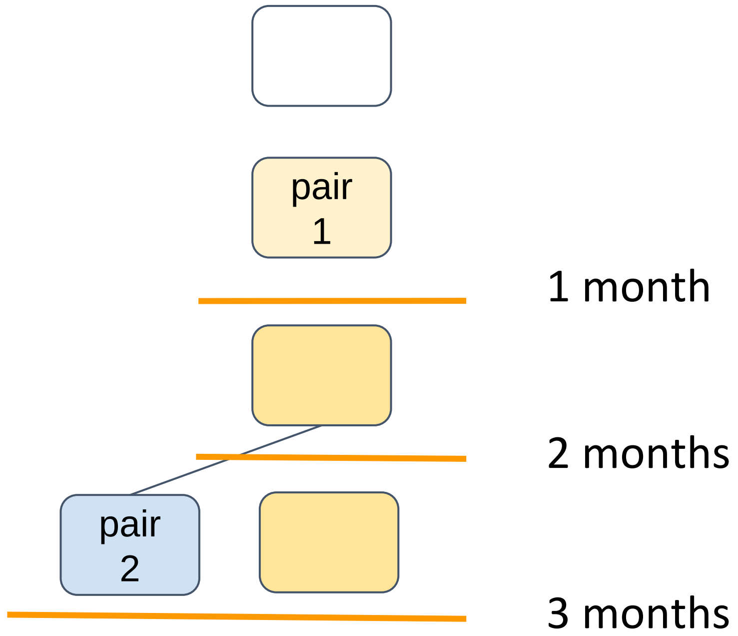
Leonardo
Fibonacci
c1175-1250



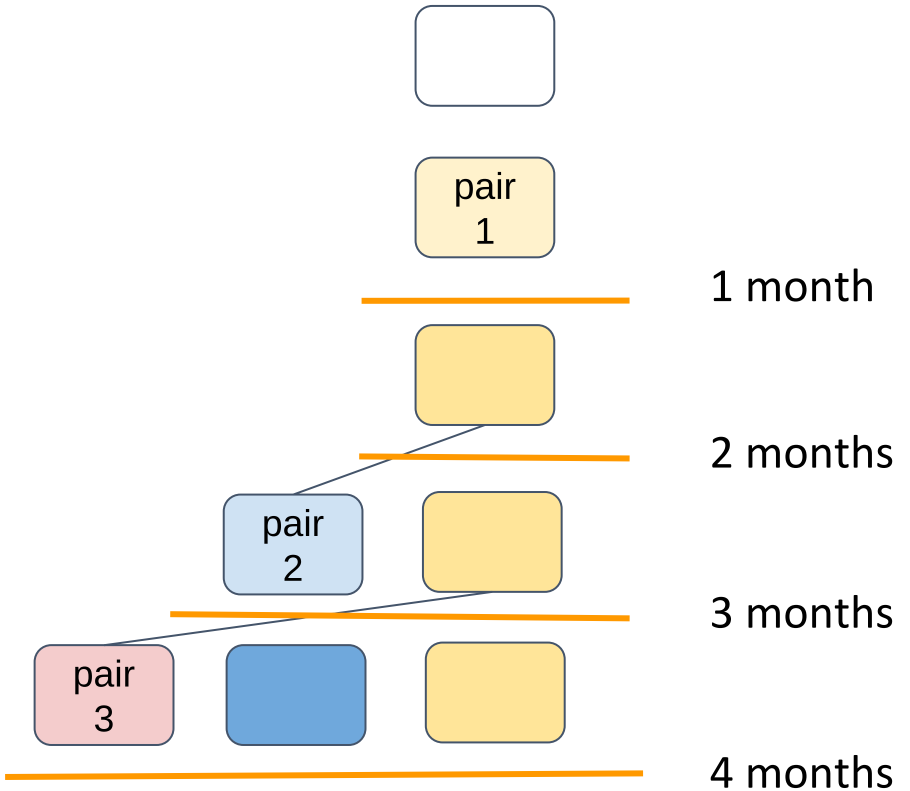
Fibonacci rabbits



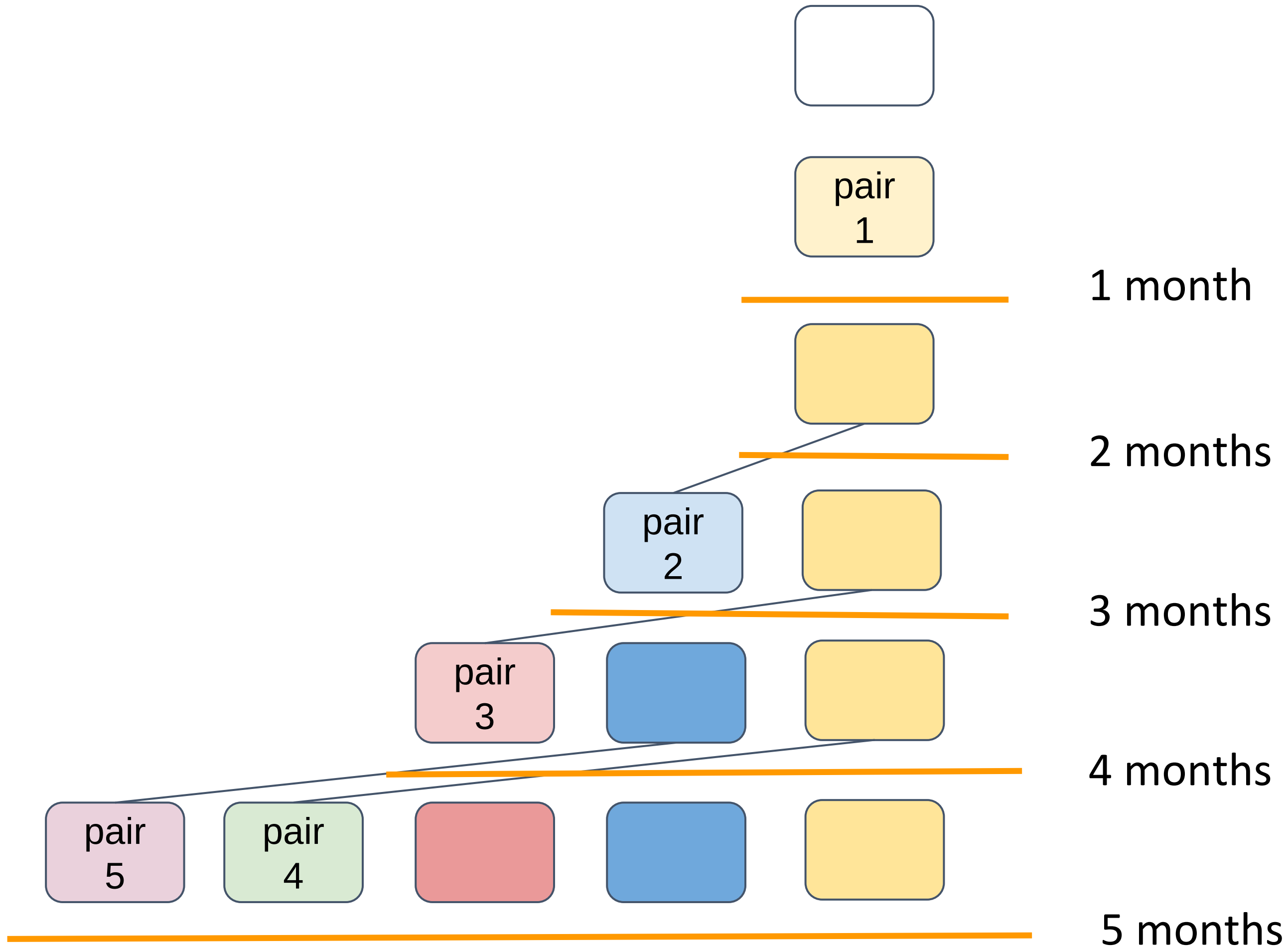
Fibonacci rabbits



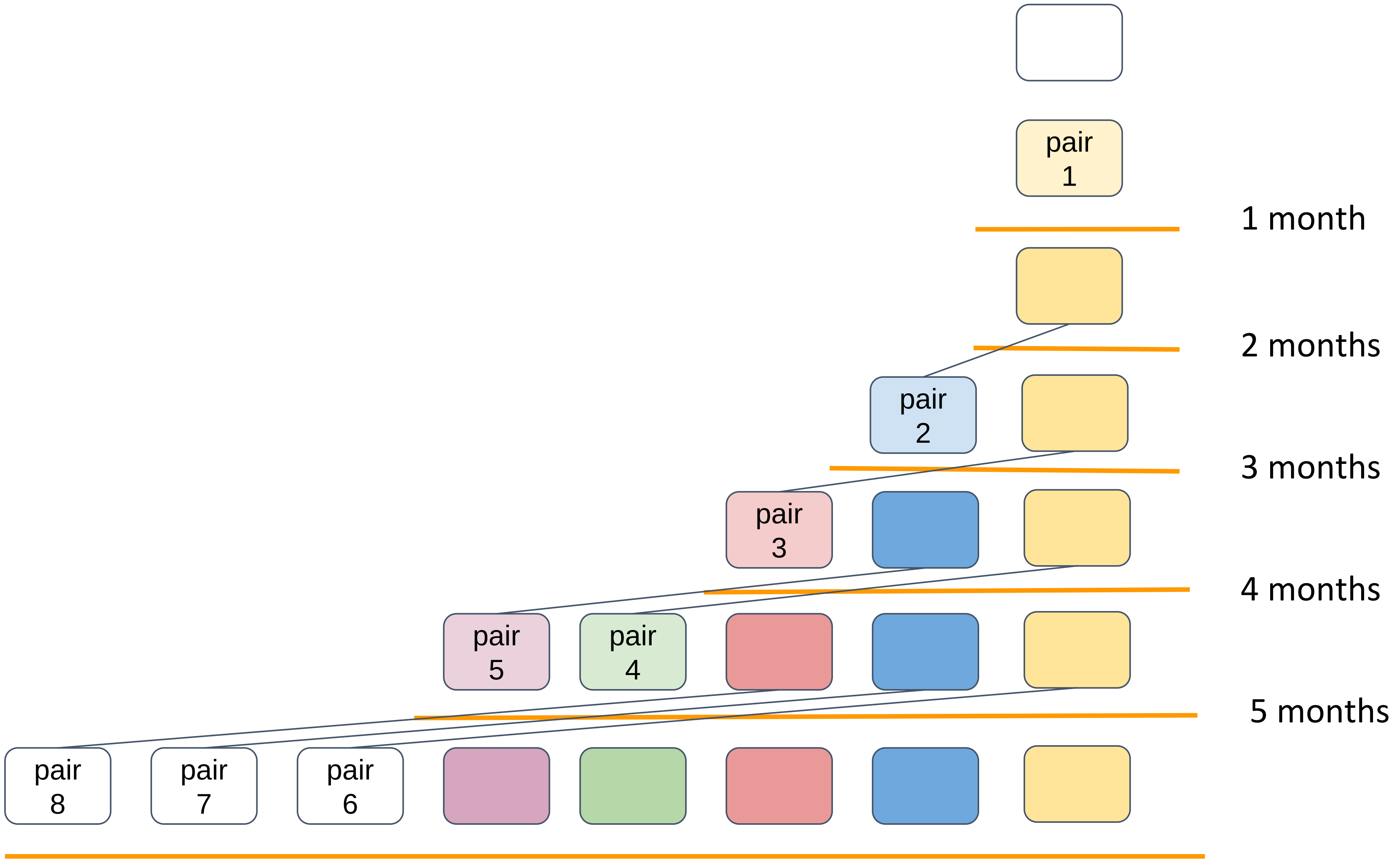
Fibonacci rabbits



Fibonacci rabbits



Fibonacci rabbits



Fibonacci numbers grow exponentially

Lemma

$F_n \geq 2^{n/2}$ for $n \geq 6$

Proof: By induction

Base case: n = 6, 7 (by direct computation).

$F_6 = 8 >= 2^{6/2} = 8$
$F_7 = 13 > 2^{7/2} = 8$

Inductive step:

Assume that it is true for F_{n-1}: $F_{n-1} >= 2^{(n-1)/2}$.

Let's show that it is true for F_n

$$F_n = F_{n-1} + F_{n-2}$$

$$\geq 2^{(n-1)/2} + 2^{(n-2)/2} \geq 2 \cdot 2^{(n-2)/2} = 2^{n/2} \quad \blacksquare$$

Theorem:

$$F_n = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$$

$$\phi = \frac{1+\sqrt{5}}{2}$$

ϕ = 1.618034...

F_{20} = 6765
F_{50} = 12586269025
F_{100} = 354224848179261915075
F_{500} = 139423224561697880139724382870407283950070256587697307264108962948325571622863290691557658876222521294125

Fibonacci numbers grow exponentially

$$F_n = F_{n-1} + F_{n-2}$$

- This recurrence relation defines an exponential growth

Recursive definition → recursive algorithm

$$F_n = \begin{cases} 0, & \text{if } n=0 \\ 1, & \text{if } n=1 \\ F_{n-1} + F_{n-2}, & \text{if } n > 1 \end{cases}$$

Problem: Compute F_n

Input: integer n >= 0
Output: F_n

Algorithm Fib_recurs(*n*)

if $n \leq 1$: return *n*

return Fib_recurs($n - 1$) + Fib_recurs($n - 2$)

What is the running time?

Recursive Fibonacci: running time

Algorithm Fib_recurs(n)

```
if n ≤ 1:
  return n
else:
   return Fib_recurs(n – 1) + Fib_recurs(n – 2)
```

Let $T(n)$ denote the count of **lines of code** executed by Fib_recurs(n).

if $n \leq 1$: $T(n) = 2$

if $n \geq 2$: $T(n) = 3 + T(n-1) + T(n-2)$

Number of operations

$$T(n) = \begin{cases} 2 \text{ if } n \lt= 1 \\ 3 + T(n-1) + T(n-2) \end{cases}$$

n-th Fibonacci number

$$F_n = \begin{cases} 0, \text{ if } n=0 \\ 1, \text{ if } n=1 \\ F_{n-1} + F_{n-2}, \text{ if } n > 1 \end{cases}$$

Therefore $T(n) \geq F_n$

Recursive Fibonacci: running time

Let $T(n)$ denote the count of **lines of code** executed by Fib_recurs(n).

Algorithm Fib_recurs(n)

```
if n ≤ 1:
  return n
else:
   return Fib_recurs(n – 1) + Fib_recurs(n – 2)
```

$$T(n) = \begin{cases} 2 \text{ if } n <= 1 \\ 3 + T(n-1) + T(n-2) \end{cases}$$

$T(n) \geq F_n$

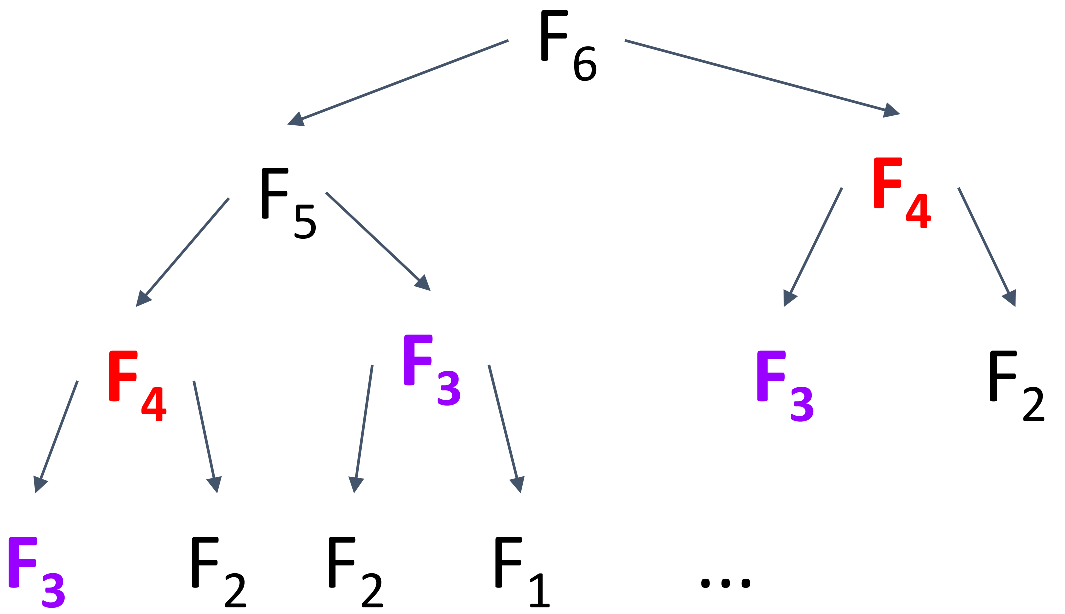
but $F_n \geq 2^{n/2}$ **for** $n \geq 6$ **!!!**

Running time $\Omega(2^n)$

$T(100) \approx 1.77 \cdot 10^{21}$
(1.77 sextillion)

Takes 56,000 years at 1GHz

Why so slow?



Recursion tree

Note the repeating calls with the same arguments

Recursion or not recursion?

Recursive algorithms are particularly appropriate when the underlying problem or the data to be treated are defined in recursive terms

- Such recursive definitions **do not guarantee** that a recursive algorithm is the best way to solve the problem
- This is especially true when the subproblems overlap and we need to call the algorithm with the same arguments multiple times.

What can we do to fix this recursive algorithm?

Idea: store computed values

- We can store the results of the previous computation of F_i at position *i* of the state array
- When the recursive call is issued to compute *fib*(*i*) we first check if the answer for this particular *i* already exists:
  - If it does not exist – we compute it and store for future use
  - If it does exist – we just use it – avoiding multiple recursion calls.

- This optimization technique is called *memoization*

Example: *memoization*

Algorithm *Fib_recurs_memo*(*n*, FibArray of size *n*)

```
if n ≤ 1:
   FibArray[n] = n
   return n
else:
    if FibArray[n - 1] is null
        FibArray[n - 1]  = Fib_recurs_memo (n - 1)
    if  FibArray[n -2] is null
        FibArray[n - 2]  = Fib_recurs_memo (n - 2)

    return FibArray[n - 1] + FibArray[n - 2]
```

If not yet computed – compute and remember

Efficient iterative algorithm

Algorithm Fib_list(n)

create an array $F[0 \ldots n]$

$F[0] \leftarrow 0$

$F[1] \leftarrow 1$

for i from 2 to n:

  $F[i] \leftarrow F[i-1] + F[i-2]$

return $F[n]$

Running time

$T(n) = 2n + 2$

So $T(100) = 202$

Backtracking

Exhaustive Search

- *Brute-force search* or exhaustive search is a very general problem-solving technique that systematically generates all possible candidates and for each candidate solution checks if it satisfies the problem's statement

Examples of exhaustive search

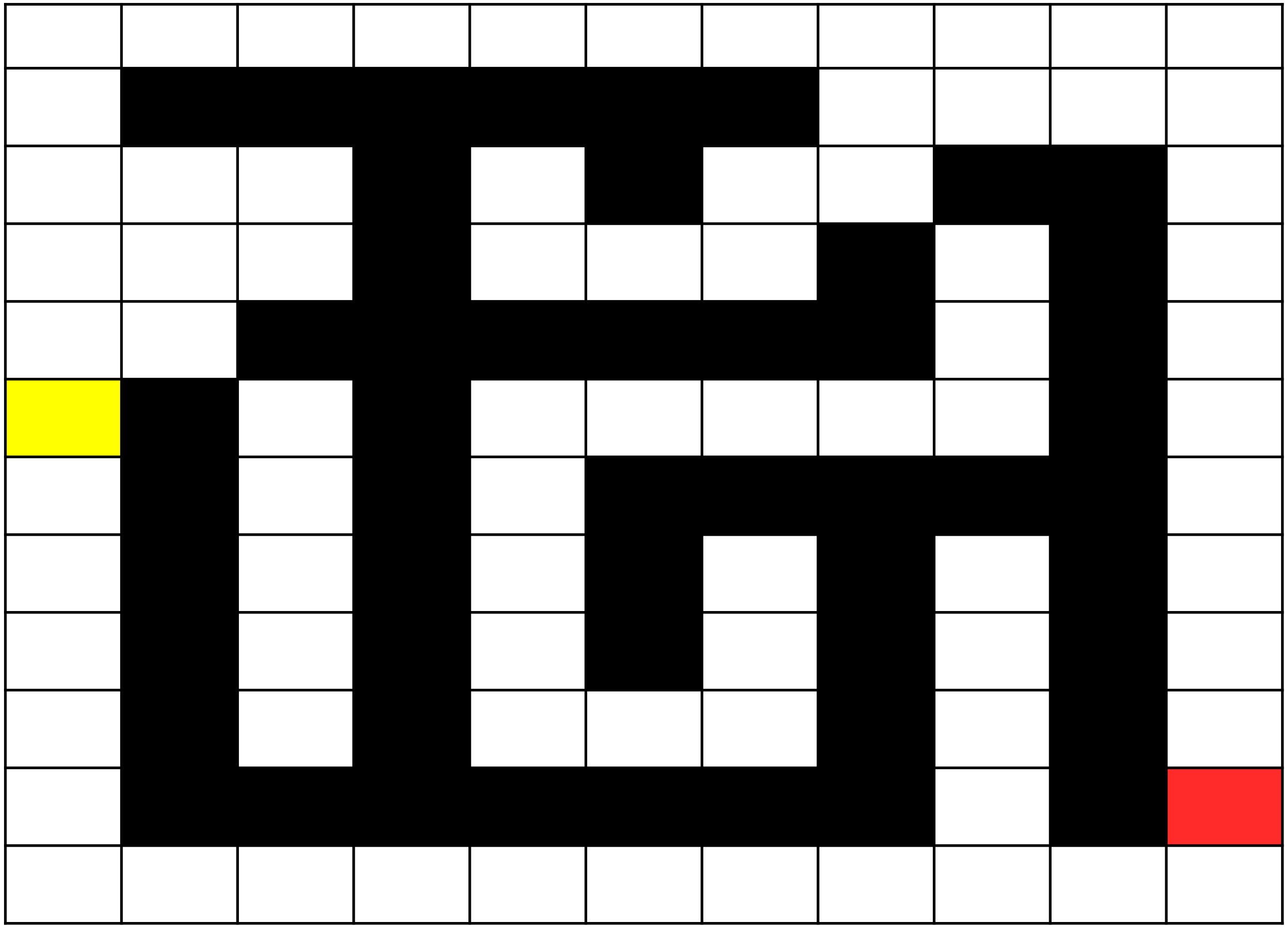
- Finding all divisors of a natural number *n*
- Generating all possible paths in a maze to find that one path that leads from start to exit

Optimization of exhaustive search: backtracking

- For some problems we do not actually need to explore **all** possible solutions
- While we are exploring one of the possible solutions we might see that this solution is not promising
- For example when exploring the path in the maze we can hit the dead end: So we undo the progress that we made and return to a point in the maze from where we can try an alternative path

- This return is called a *backtracking*

Example 1: solving mazes



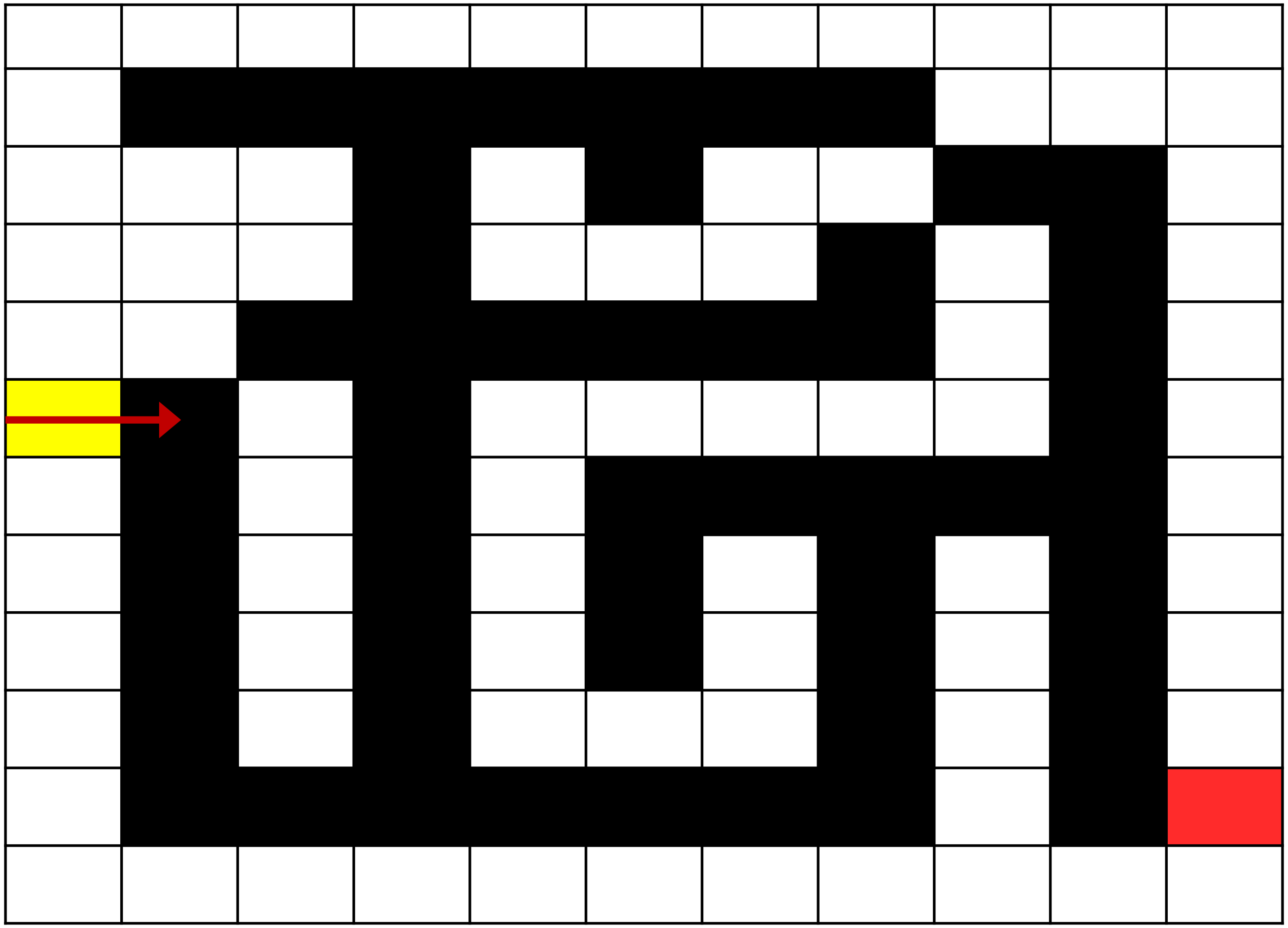
The goal is to find a path from the start square (yellow) to the exit square (red)

White cells represent the walls, black cells represent the passable cells

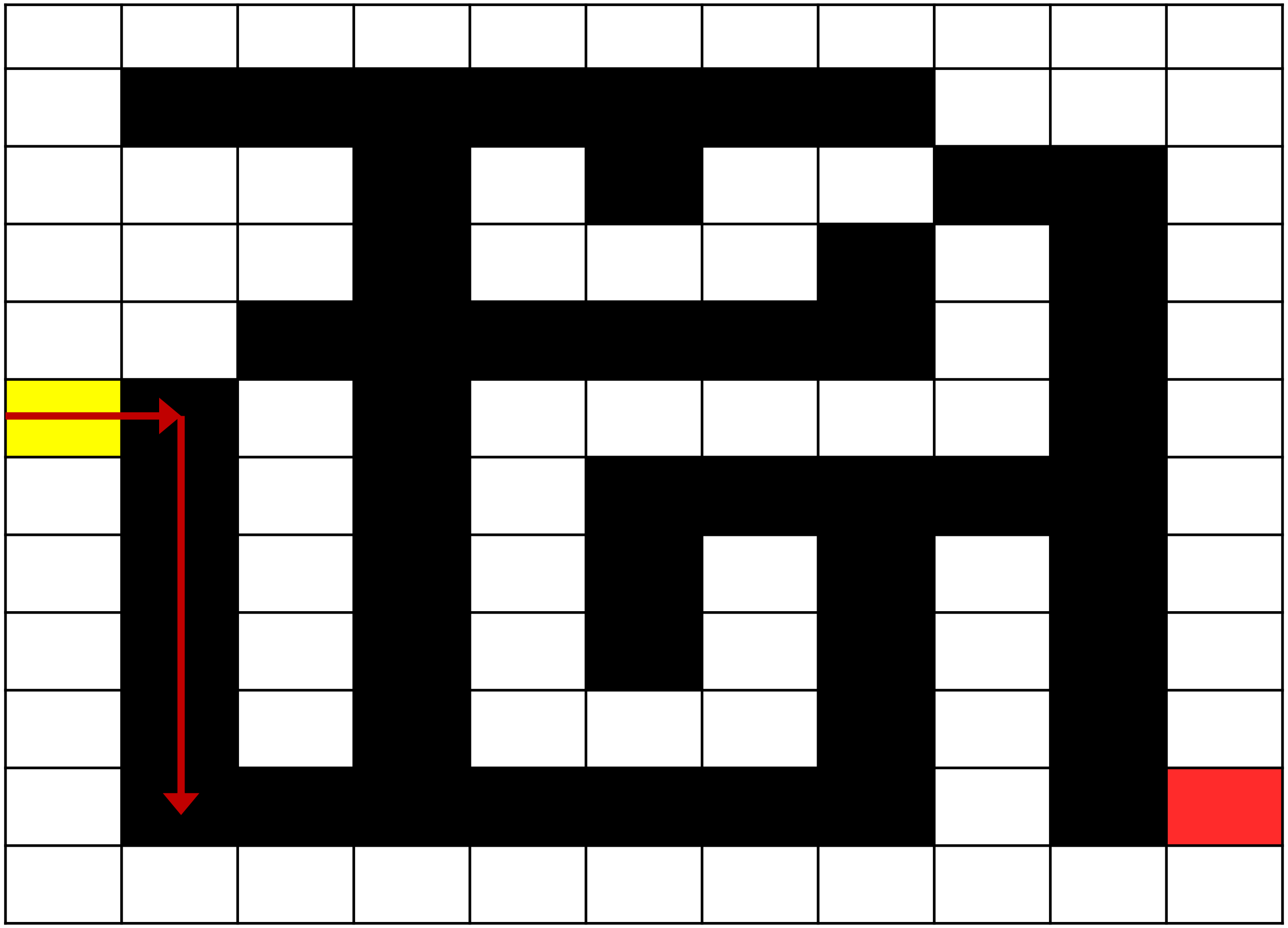
If we take a brute-force approach we will try to explore all possible paths from the current cell in hope that one of these paths will eventually hit the exit

The order of exploration is: Start at 12 o'clock and go clockwise: NESW

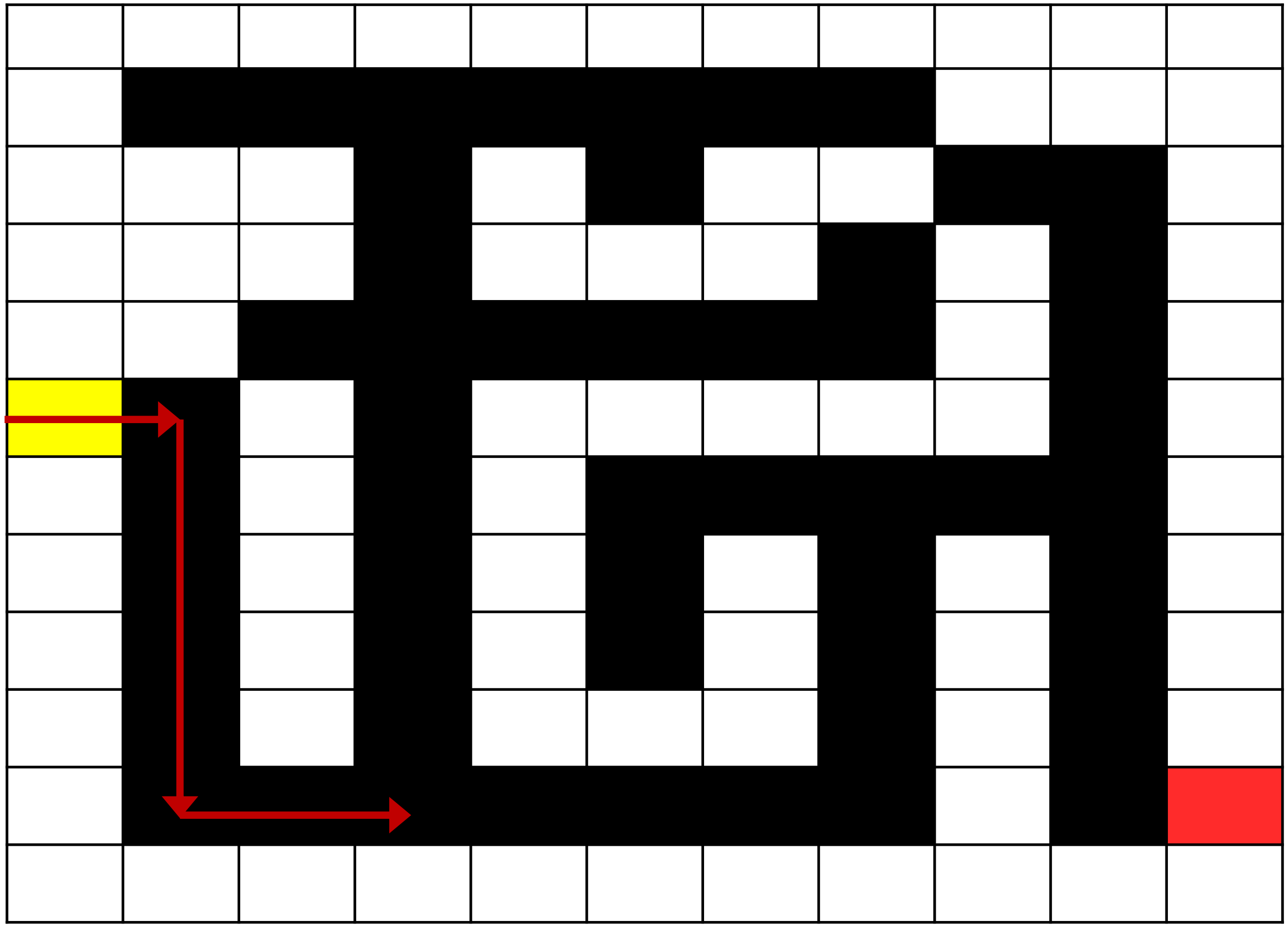
Maze solver example: Recursion with backtracking



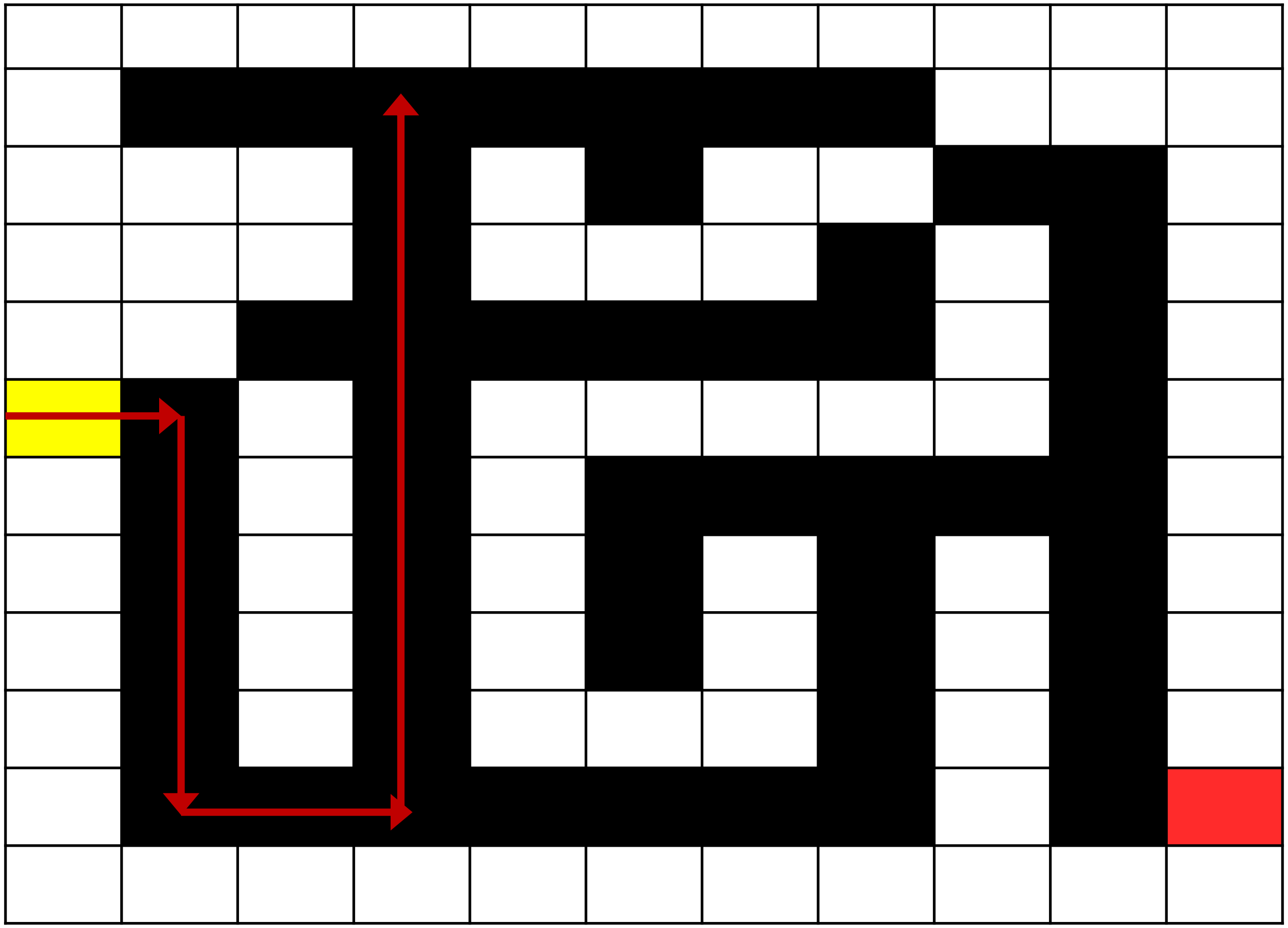
Maze solver example: Recursion with backtracking



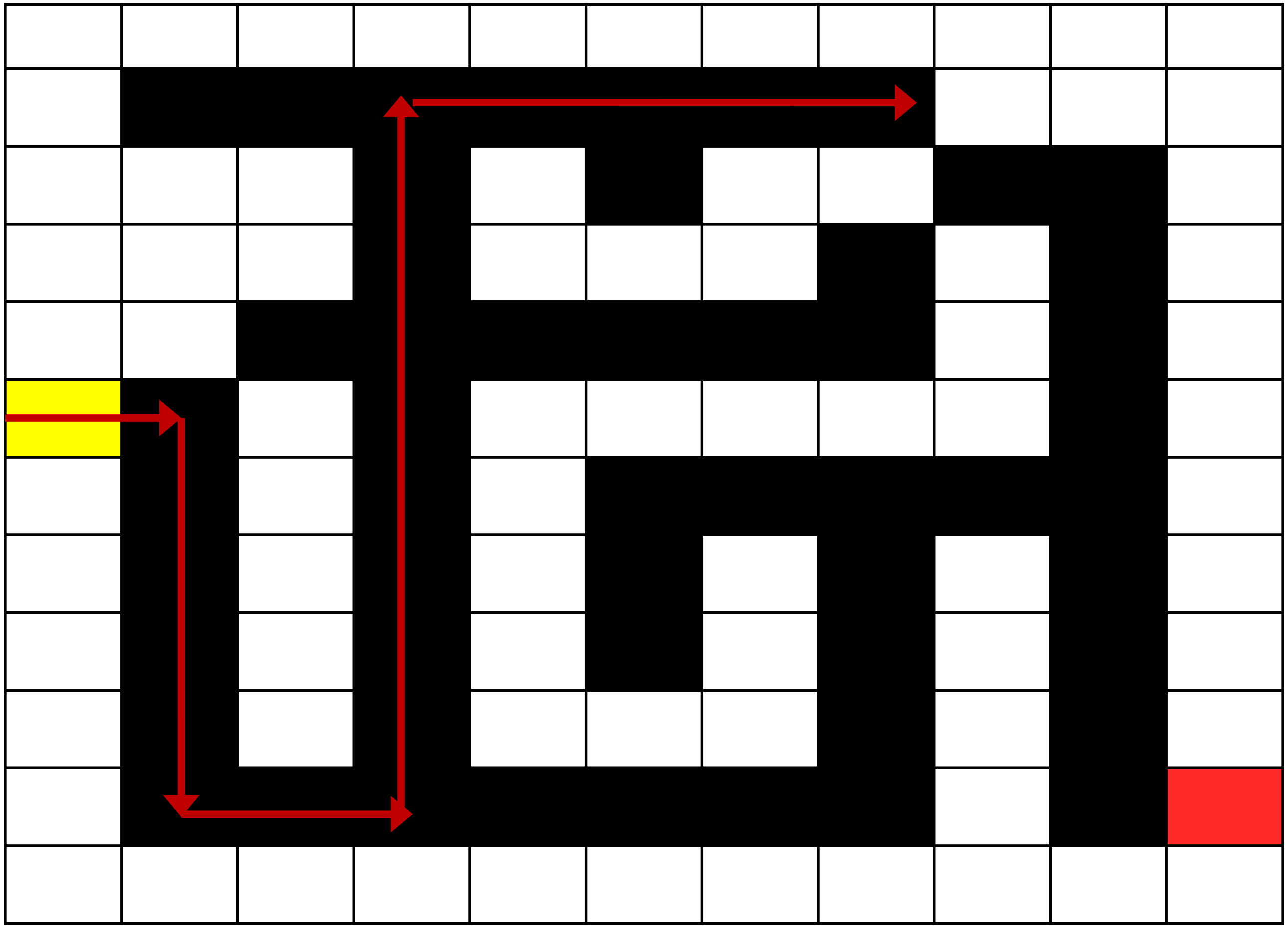
Maze solver example: Recursion with backtracking



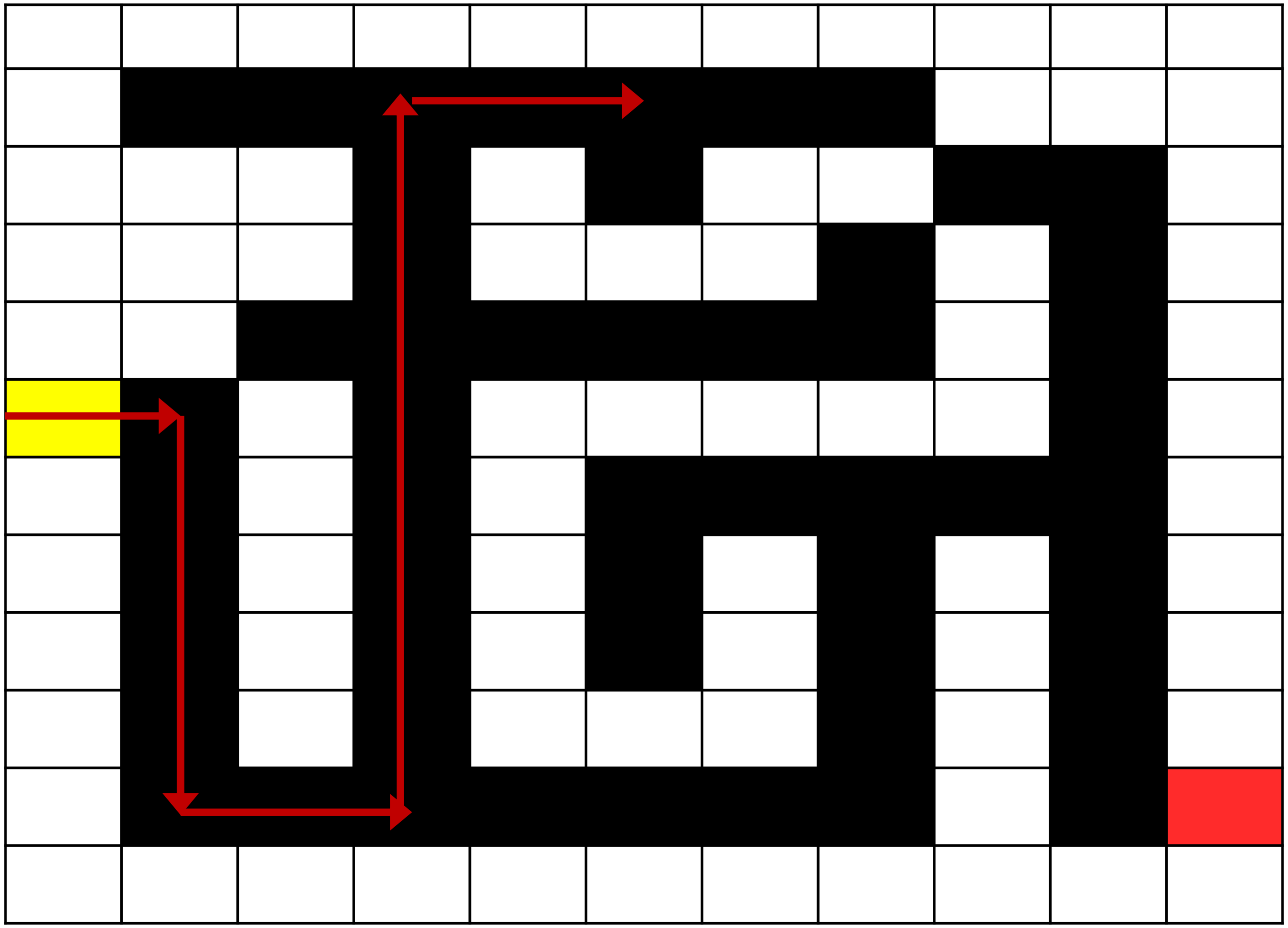
Maze solver example: Recursion with backtracking



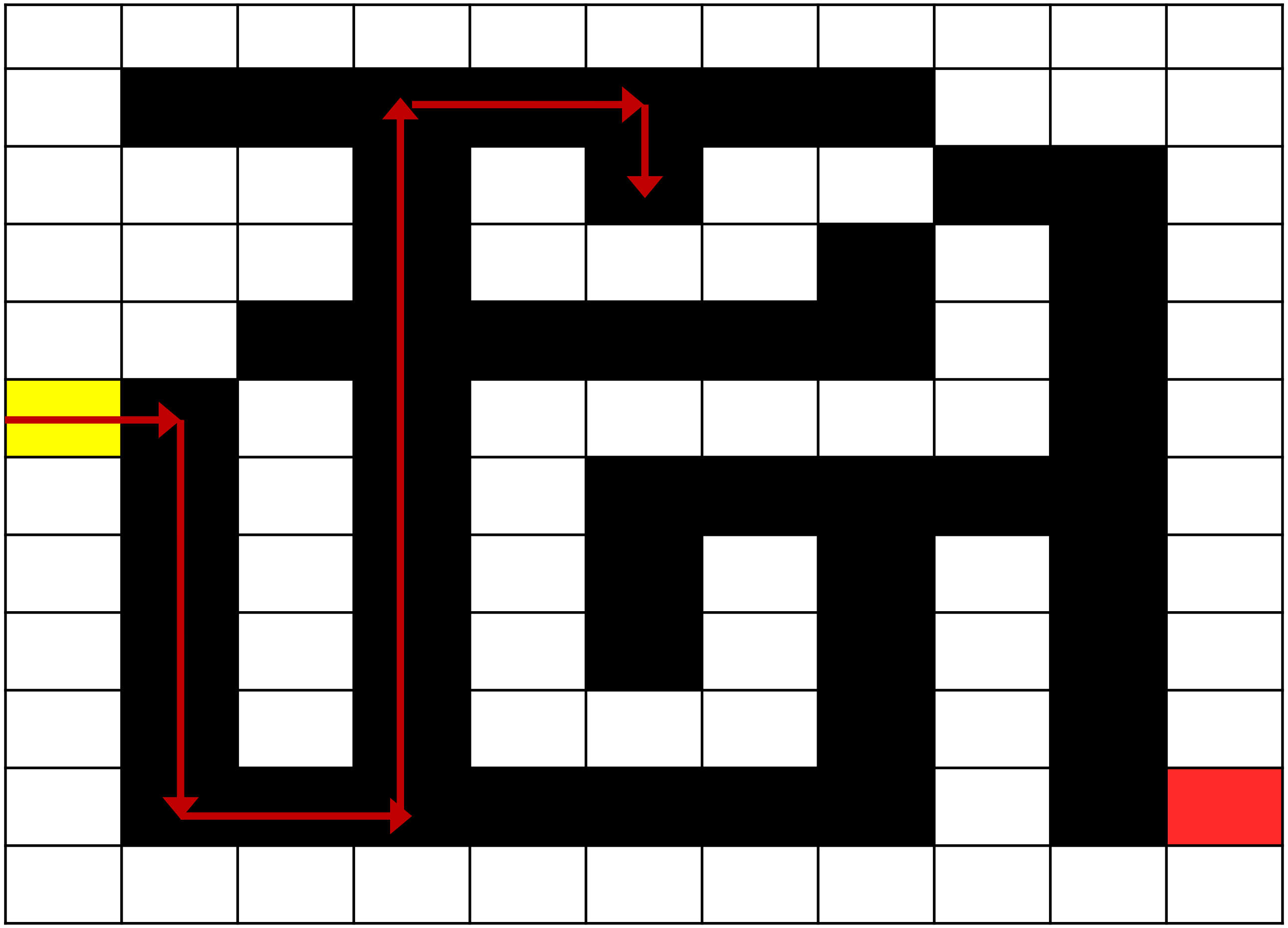
Maze solver example: Recursion with backtracking



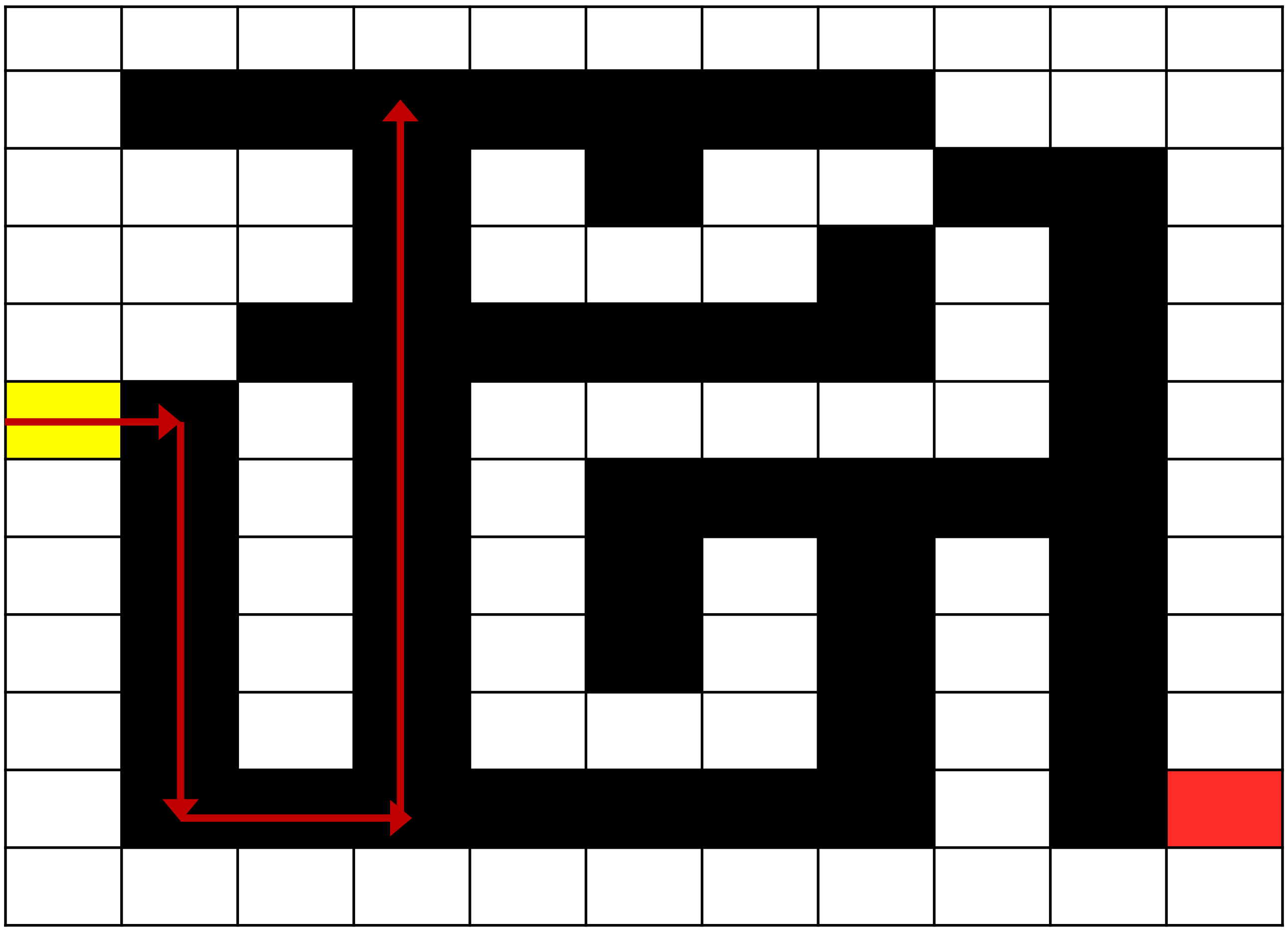
Maze solver example: Recursion with backtracking



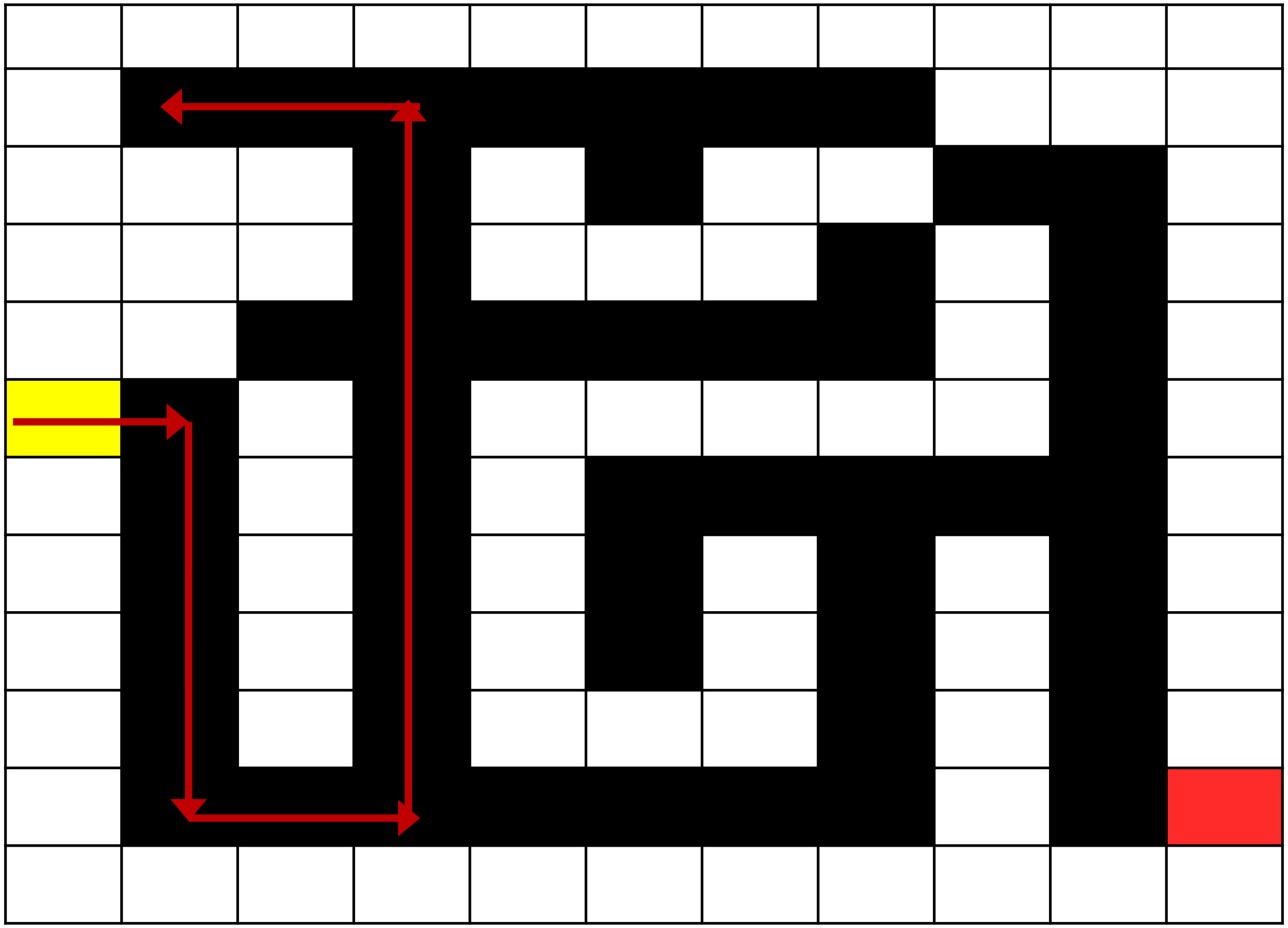
Maze solver example: Recursion with backtracking



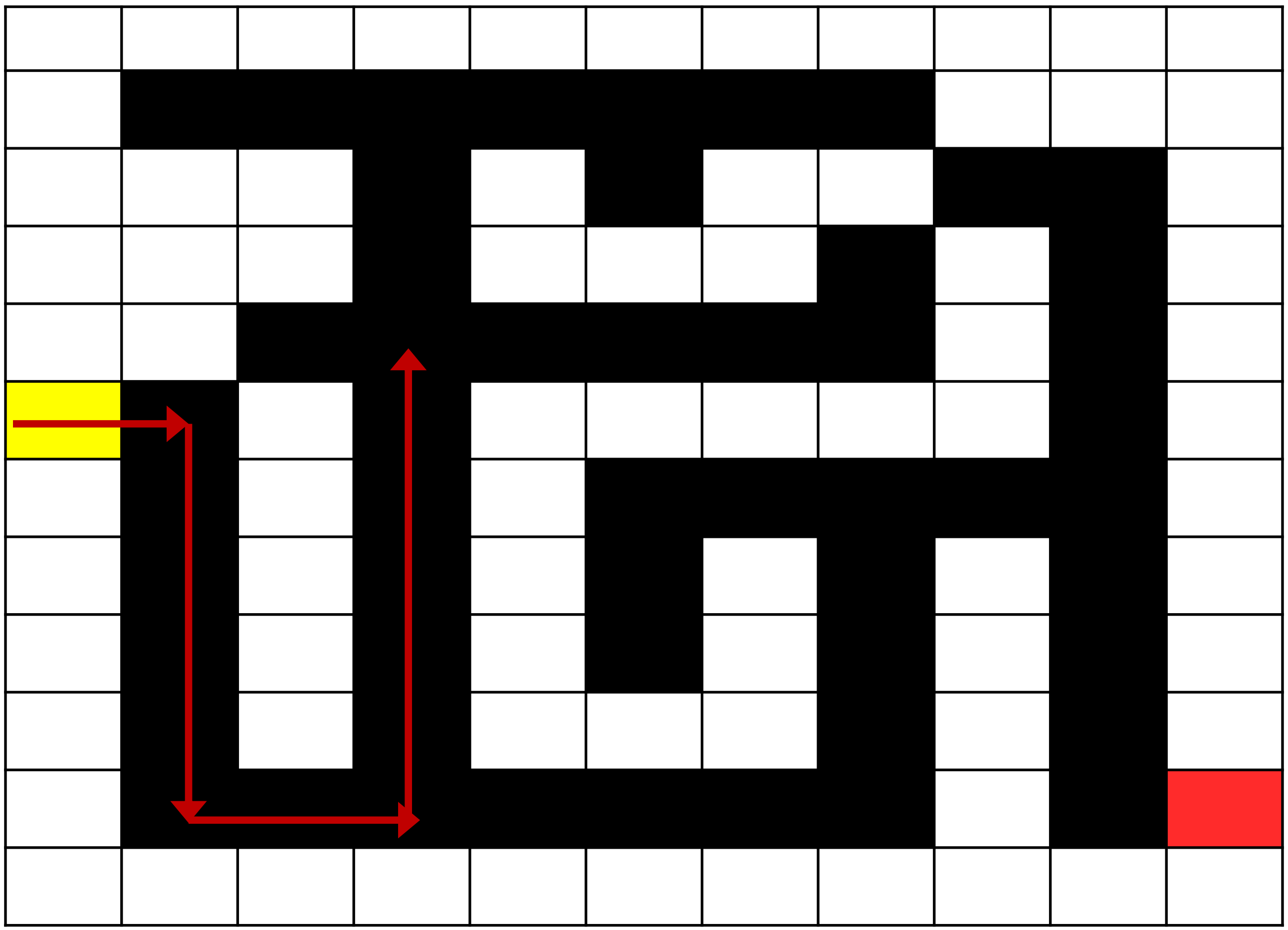
Maze solver example: Recursion with backtracking



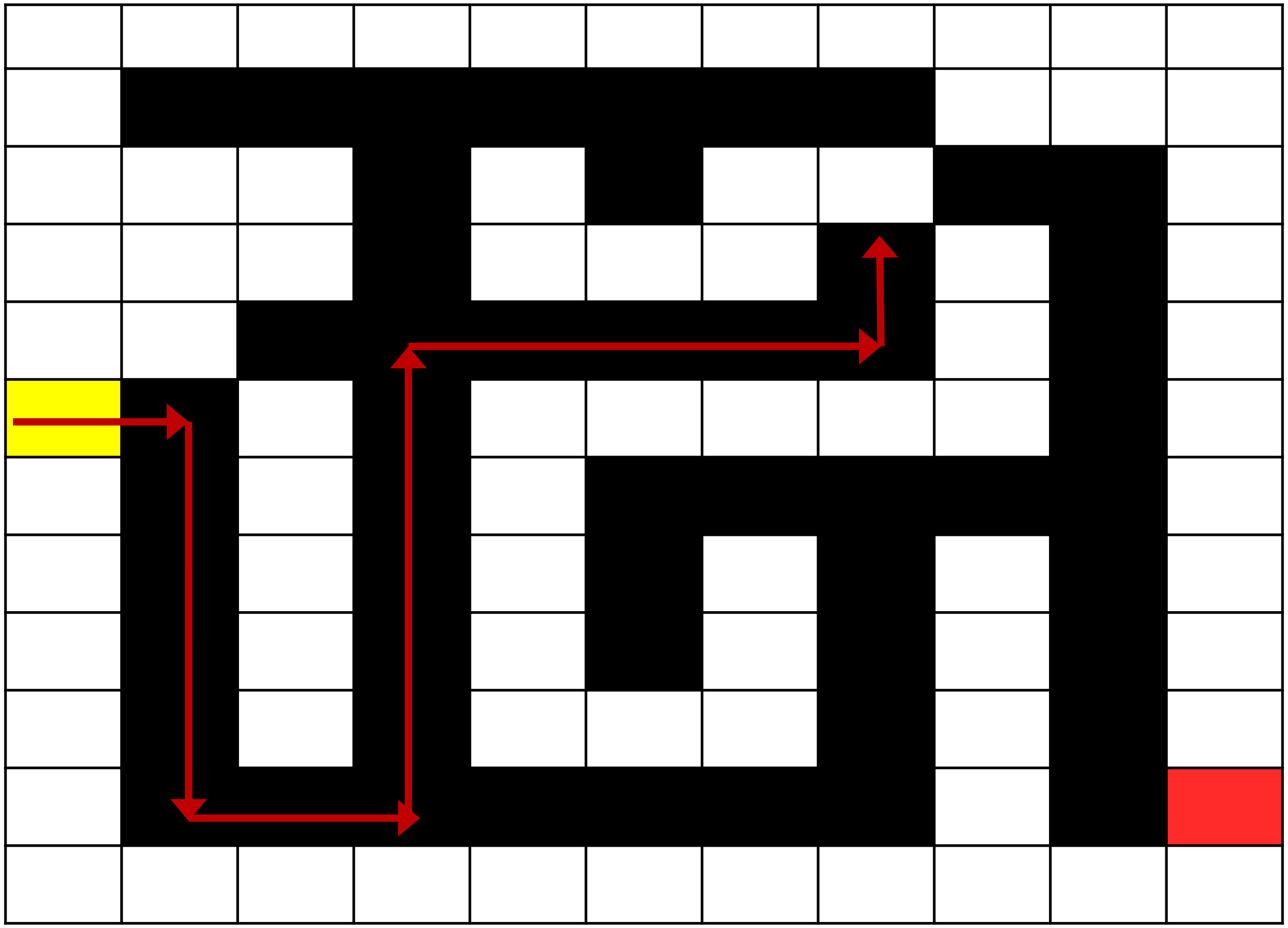
Maze solver example: Recursion with backtracking



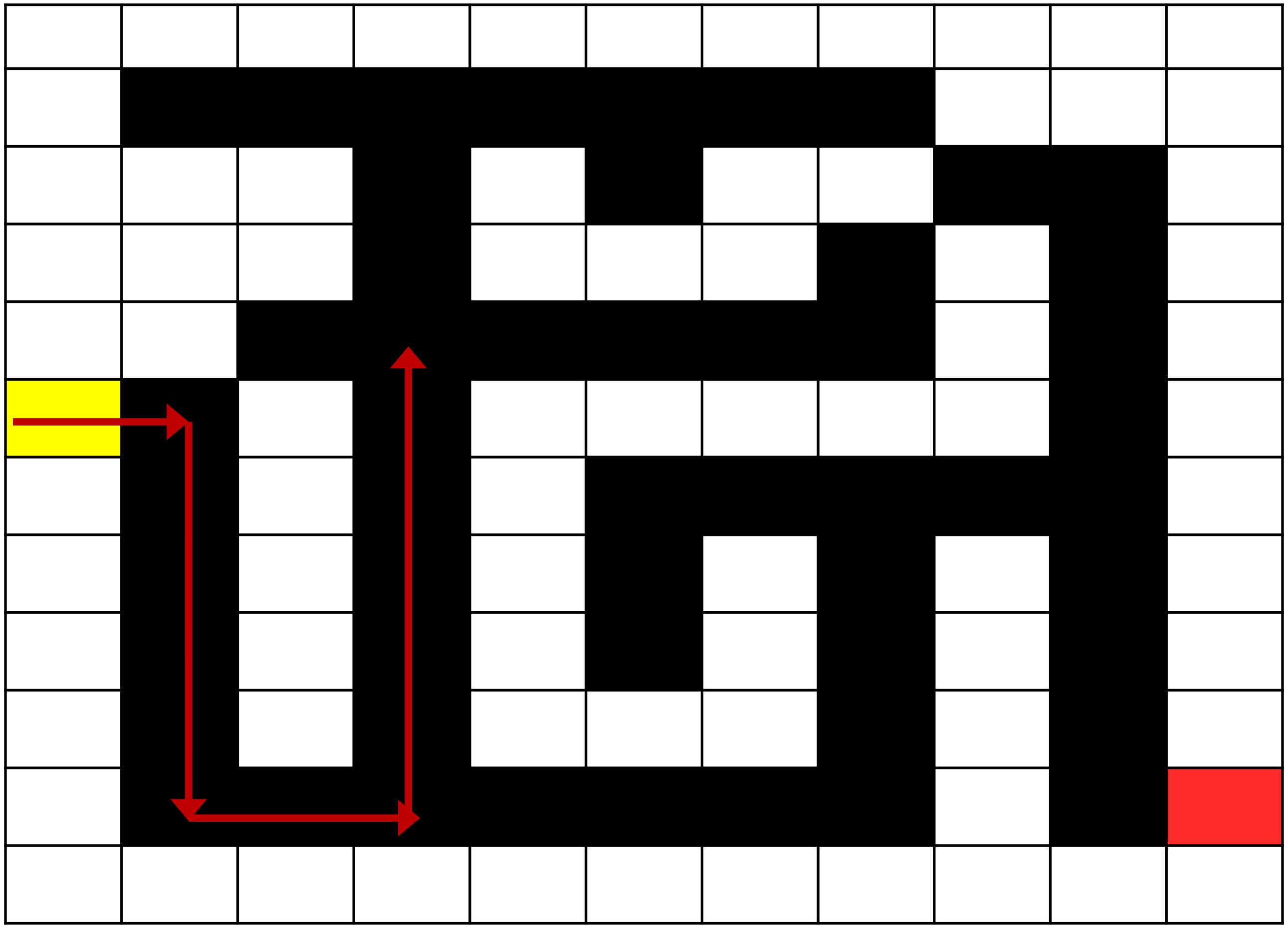
Maze solver example: Recursion with backtracking



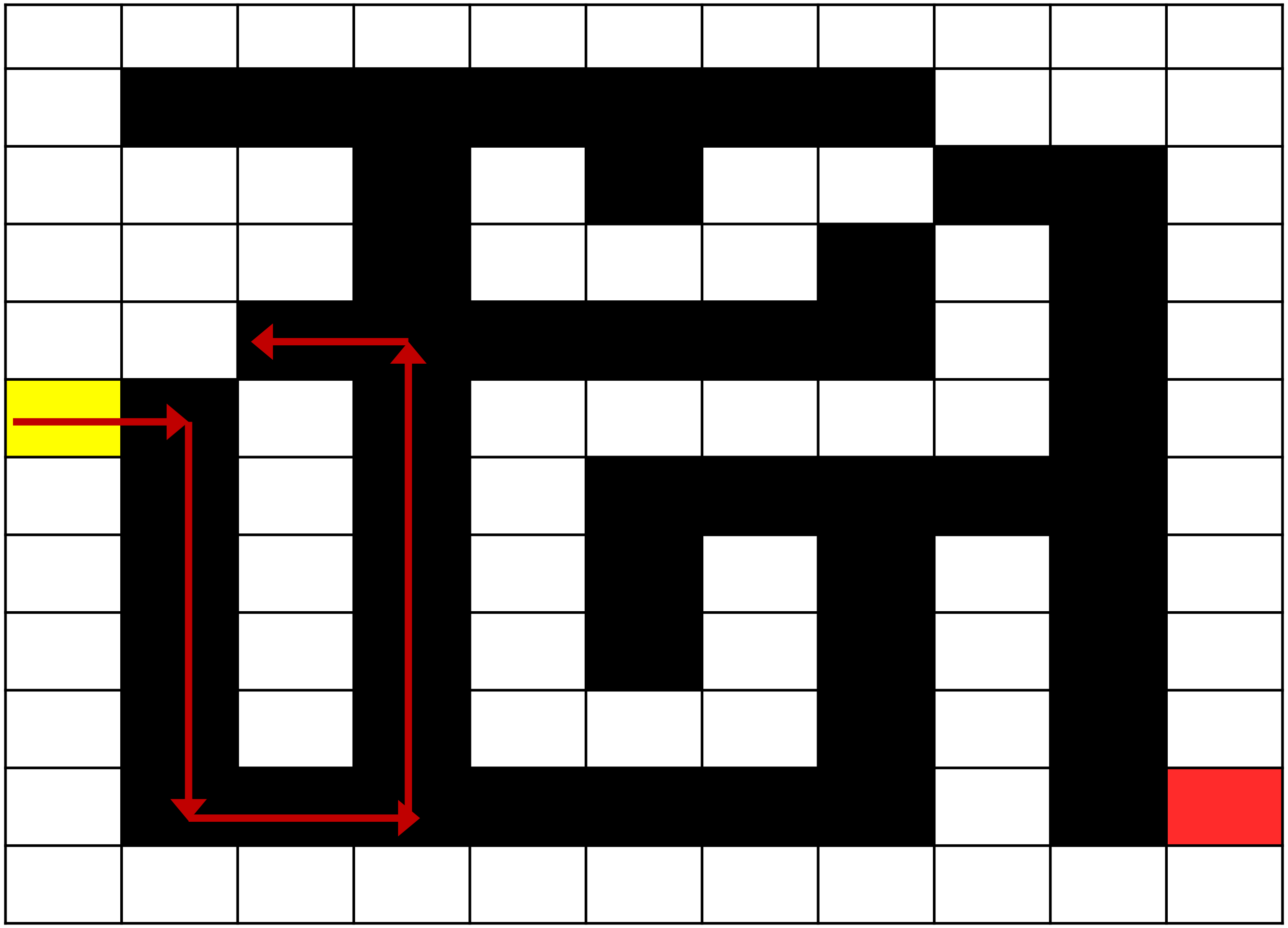
Maze solver example: Recursion with backtracking



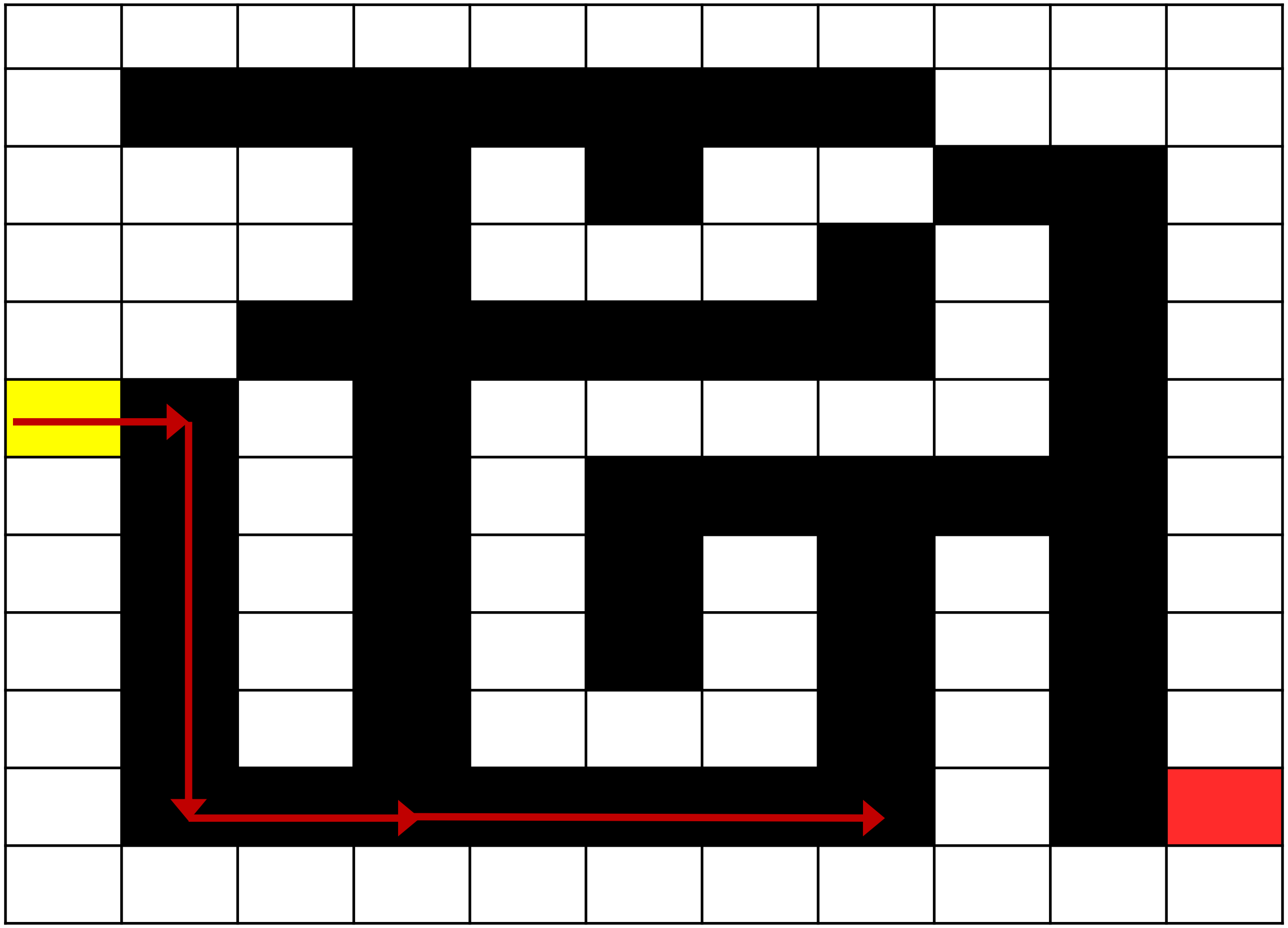
Maze solver example: Recursion with backtracking



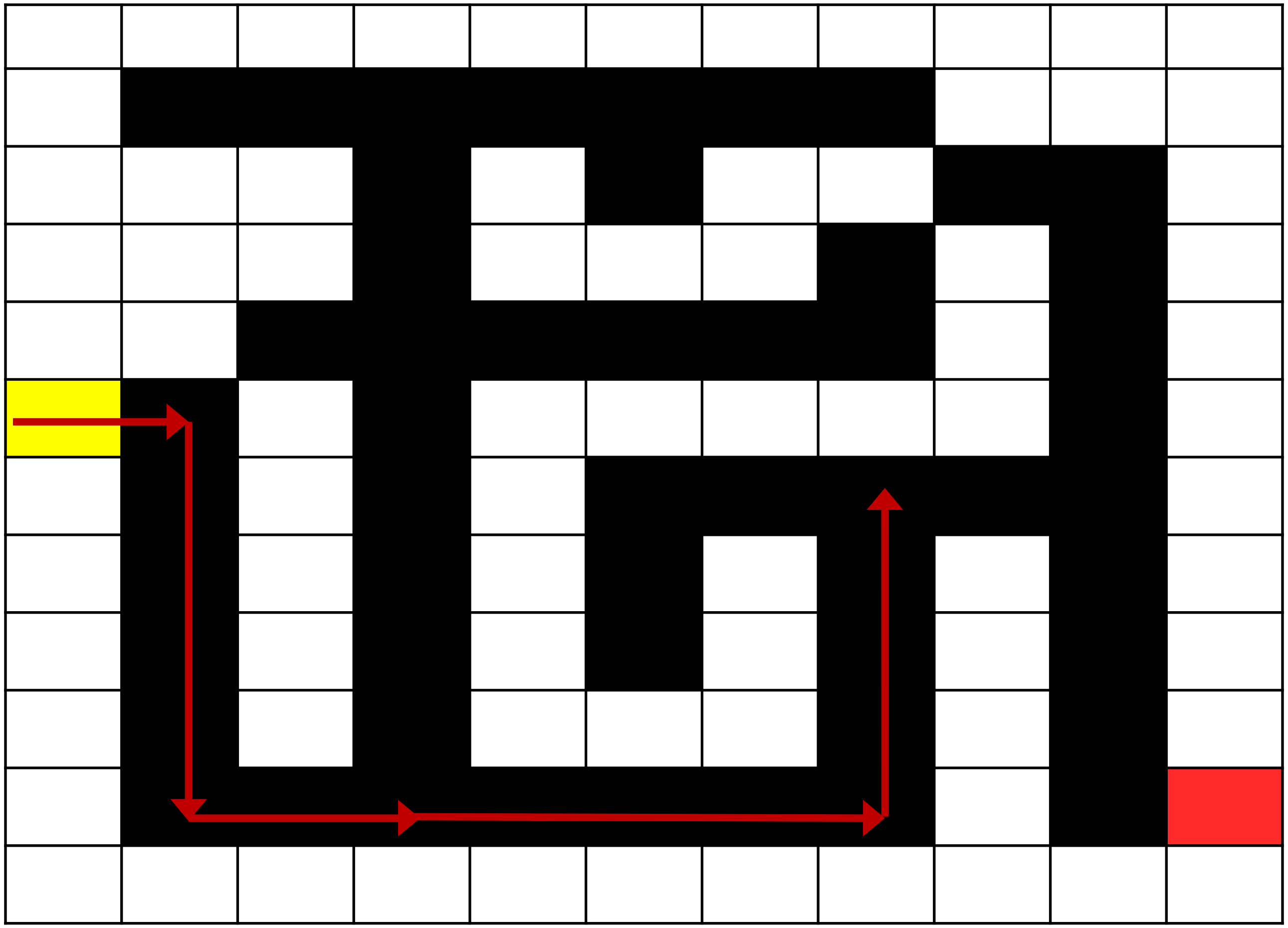
Maze solver example: Recursion with backtracking



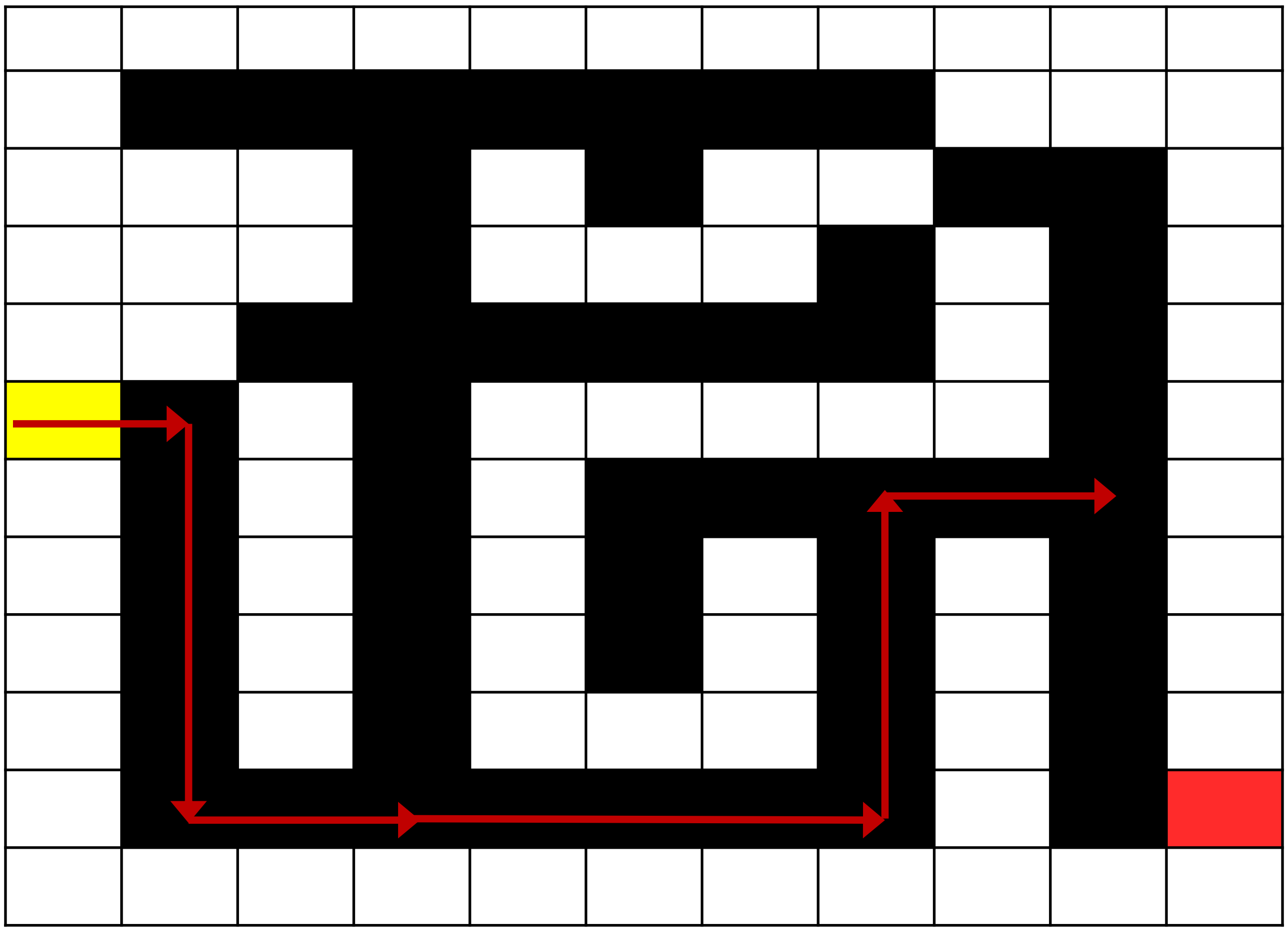
Maze solver example: Recursion with backtracking



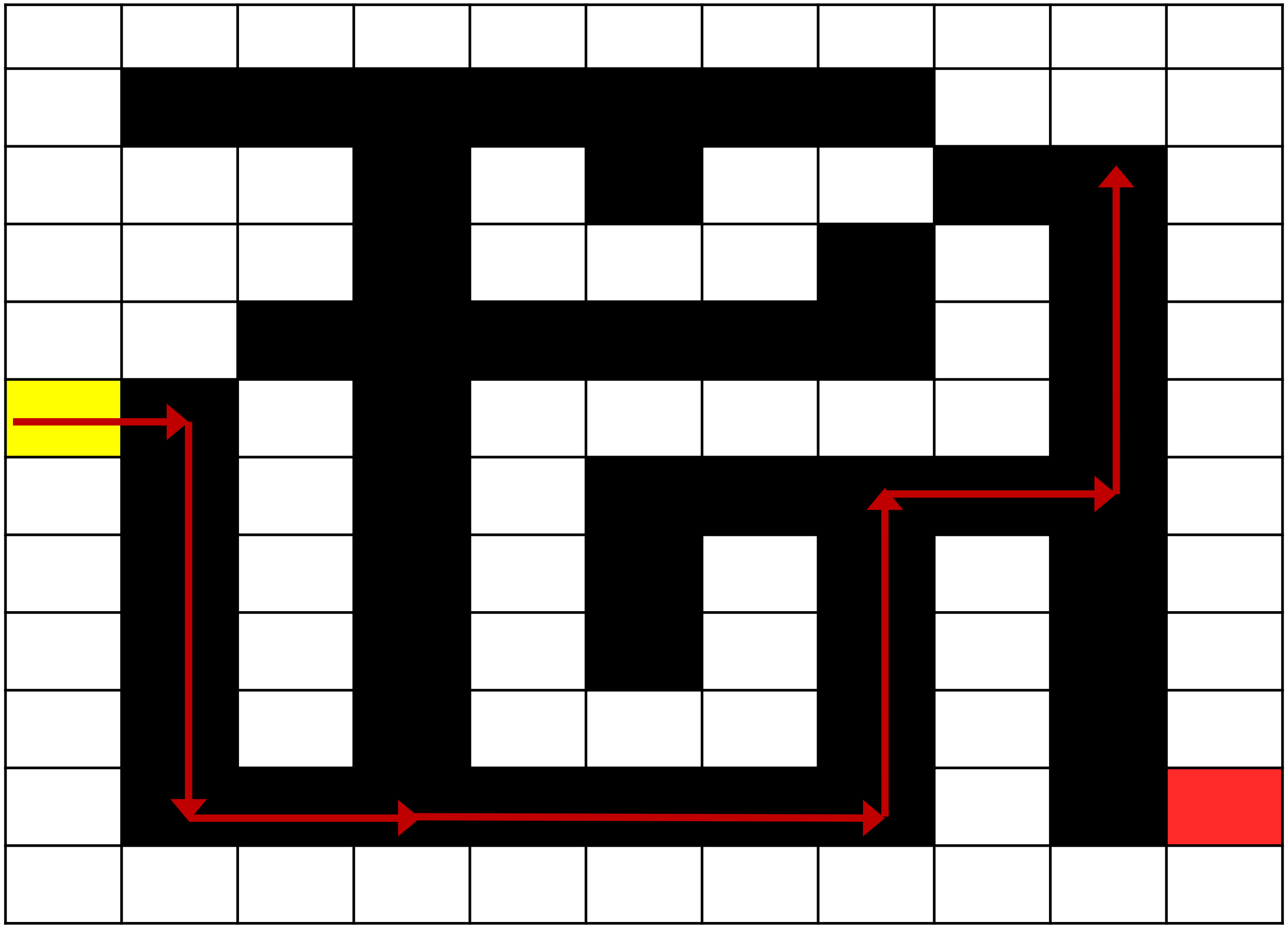
Maze solver example: Recursion with backtracking



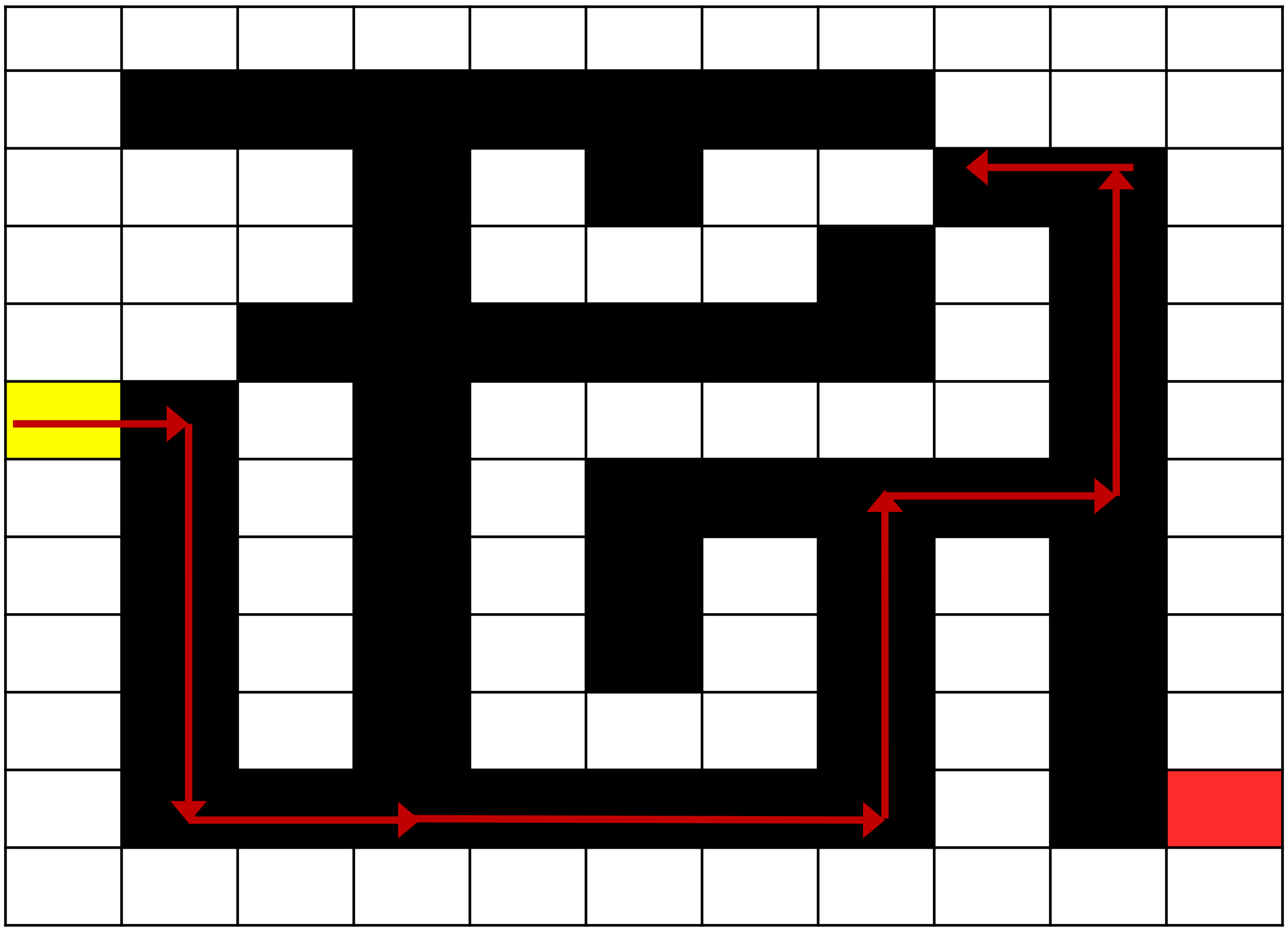
Maze solver example: Recursion with backtracking



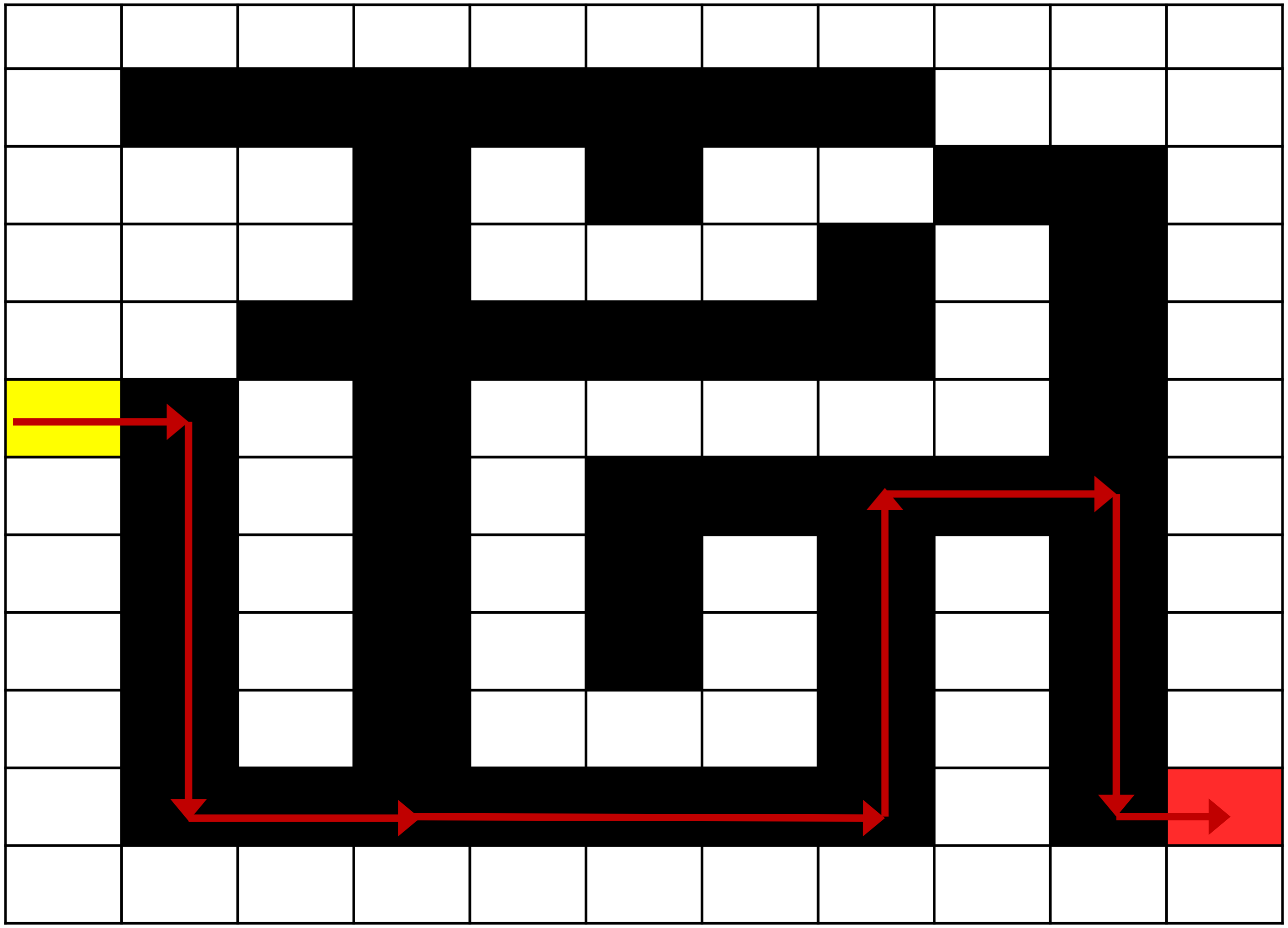
Maze solver example: Recursion with backtracking



Maze solver example: Recursion with backtracking



Maze solver example: Recursion with backtracking



Algorithm findExit(2D array *maze, row, col*)

```
if maze[row][col] == EXIT  // found solution
       return true;

// first thing: mark as visited not to go here again
maze[row][col] = VISITED

// try each direction in turn
if findExit (maze, row-1, col) //try north (UP)
       return true

if findExit (maze, row, col+1) //try east (RIGHT)
       return true

if findExit (maze, row+1, col) //try south (DOWN)
       return true

if findExit (maze, row, col-1) //try west (LEFT)
       return true

// if here - there were no path from this cell
// backtrack to the previous recursion call
return false
```

We check all possible directions

Each recursive call propagates to the base case and returns a boolean

Why it is easier to do this recursively and not iteratively?

- We need to store information about every intersection we passed in order to be able to return to it and try an unexplored option
- Without recursion, we would need to store / update this information ourselves
- This could be done (using our own Stack), but since the mechanism is already built into recursive programming, why not utilize it?
- With recursion stack when the top frame unloads we backtrack precisely to the place from where we left and we can continue exploring the intersection

Example 2. The n-Queens Puzzle

- Place n queens on an $n \times n$ chessboard so that no two queens attack each other by being in the same column, row, or diagonal.
- Recall that chess queens can move horizontally, vertically or diagonally for multiple spaces

PLAY

Observation

	0	1	2	3	4	5	6	7
0	Q							
1				Q				
2		Q						
3					Q			
4			Q					
5								
6								
7								

- We note that all queens must be in different rows and different columns
- If we consider each queen to already be placed in one of the columns from 0 to n-1, we only need to determine the row for each queen
- We will do it exhaustively trying each row in turn and checking the result

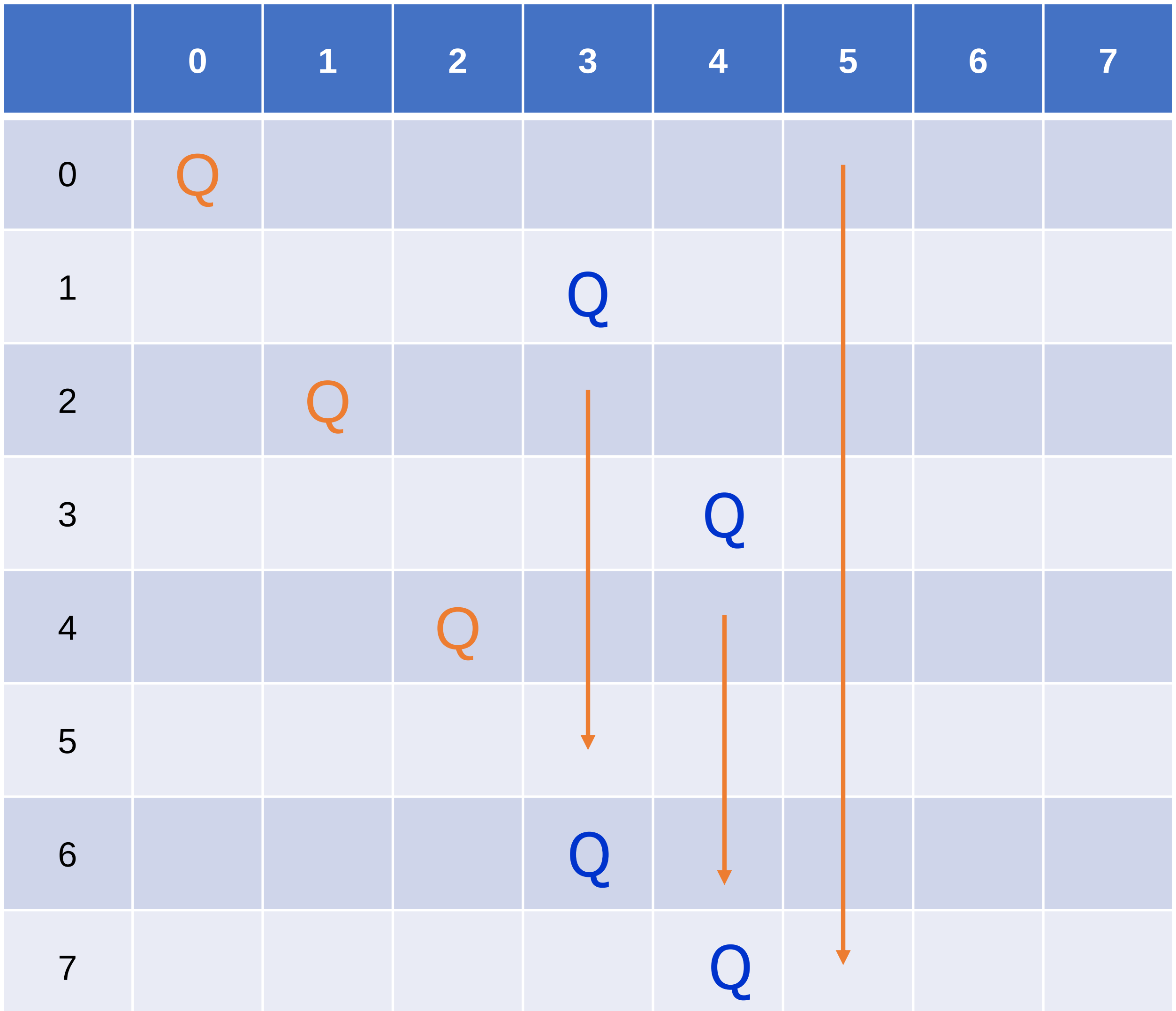
Algorithm findRow(2D array *board, row, col*)

```
// BASE CASE
// we are at the last column
if col == board[0].length - 1 // this is the last queen
      if underThreat (board, row, col) // we placed it!
            board[row][col] = 1
            return true

// try all different rows
// until we find compatible with previous placings
for i from 0 to board.length - 1
      if !underThreat (board, i, col)
            board[i][col] = 1  // place queen here temporarily
                  // and check next col until the end ...
            if !findRow (board, row, col+1);
                  board[i][col] = 0 //undo queen placement
            else
                  return true // queen stays in row i

// checked all the rows and cant place in any of them
// we need to undo the placement in the previous column
return false
```

We check all possible rows

Each recursive call propagates to the base case and returns a boolean

Visualization: 8-queens



- The call at column 5 would try all rows and fail, backtracking to column 4
- At column 4 we would move the queen down to the next legal row (7) and try again
- We try column 5 again but it fails again
- We backtrack to 4 and then to 3 (since 4 was at row 7)
- We move queen in col 3 to row 6 and move forward again

N-Queens optimization

READING LINK