SINGAPORE INSTITUTE OF TECHNOLOGY

ICT1008 Data Structures and Algorithms

Lecture 4: Algorithm Design

Recommended Readings

1. Runestone Interactive book:
   "Problem Solving with Algorithms and Data Structures Using Python"
   – Section "Recursion"

Agenda

- Recursion Trees
- Divide & Conquer
- Backtracking
- Dynamic Programming

Recap of Recursion

- Recursion is usually less efficient than its iterative equivalent.
- Recursion algorithms are often simple, clear and easy to understand.
- Although every recursive procedure can be converted into an iterative version, the conversion is not always trivial.

Recursion Trees

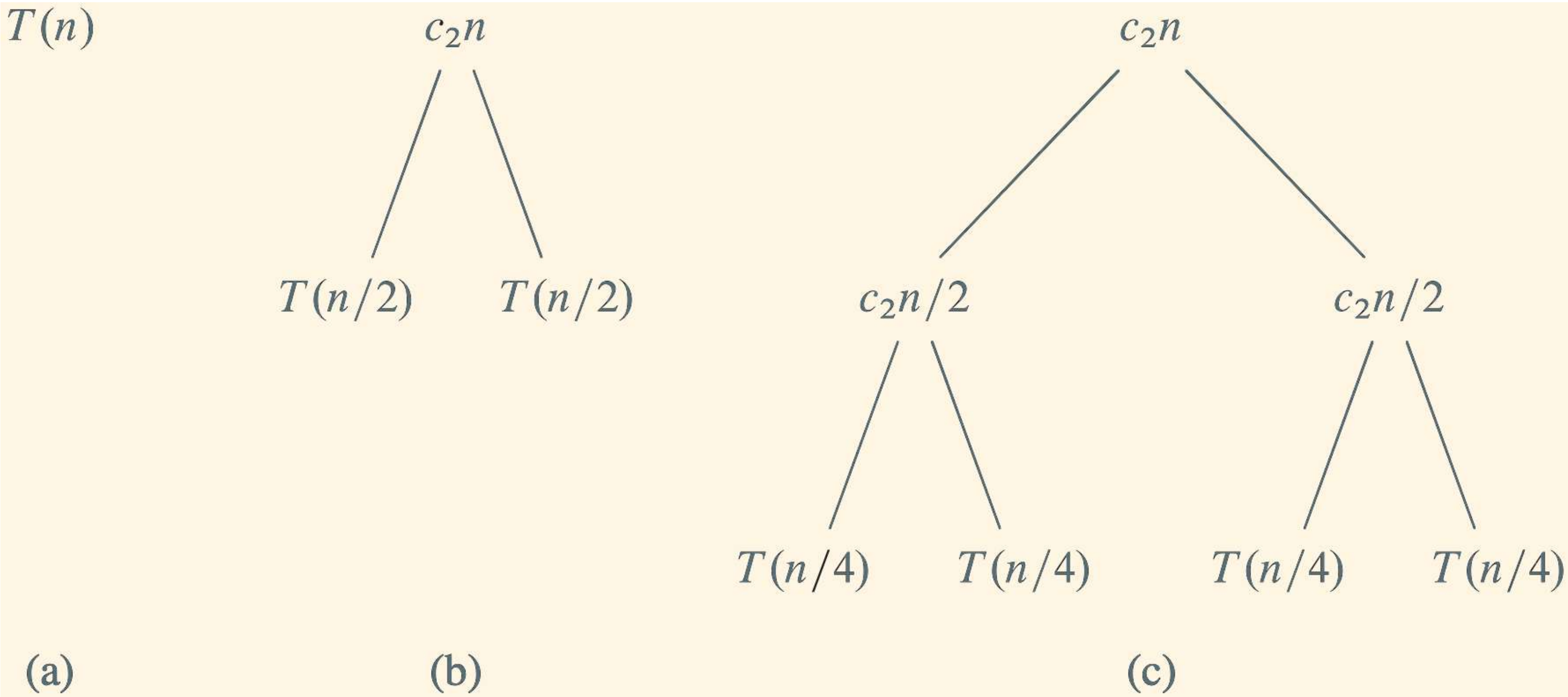
- One limitation of the substitution method is its reliance on guessing a solution. Recursion trees provide a graphical approach that often eliminates the need for guessing.
- Consider again the recursion relation for merge sort from Equation (1.4), which can be expressed as

$$T(n) \leq \begin{cases} C_1 & if\ n = 1, \\ 2T\left(\frac{n}{2}\right) + c_2 n & if\ \ n > 1 \end{cases}$$

- for sufficiently large c_1.

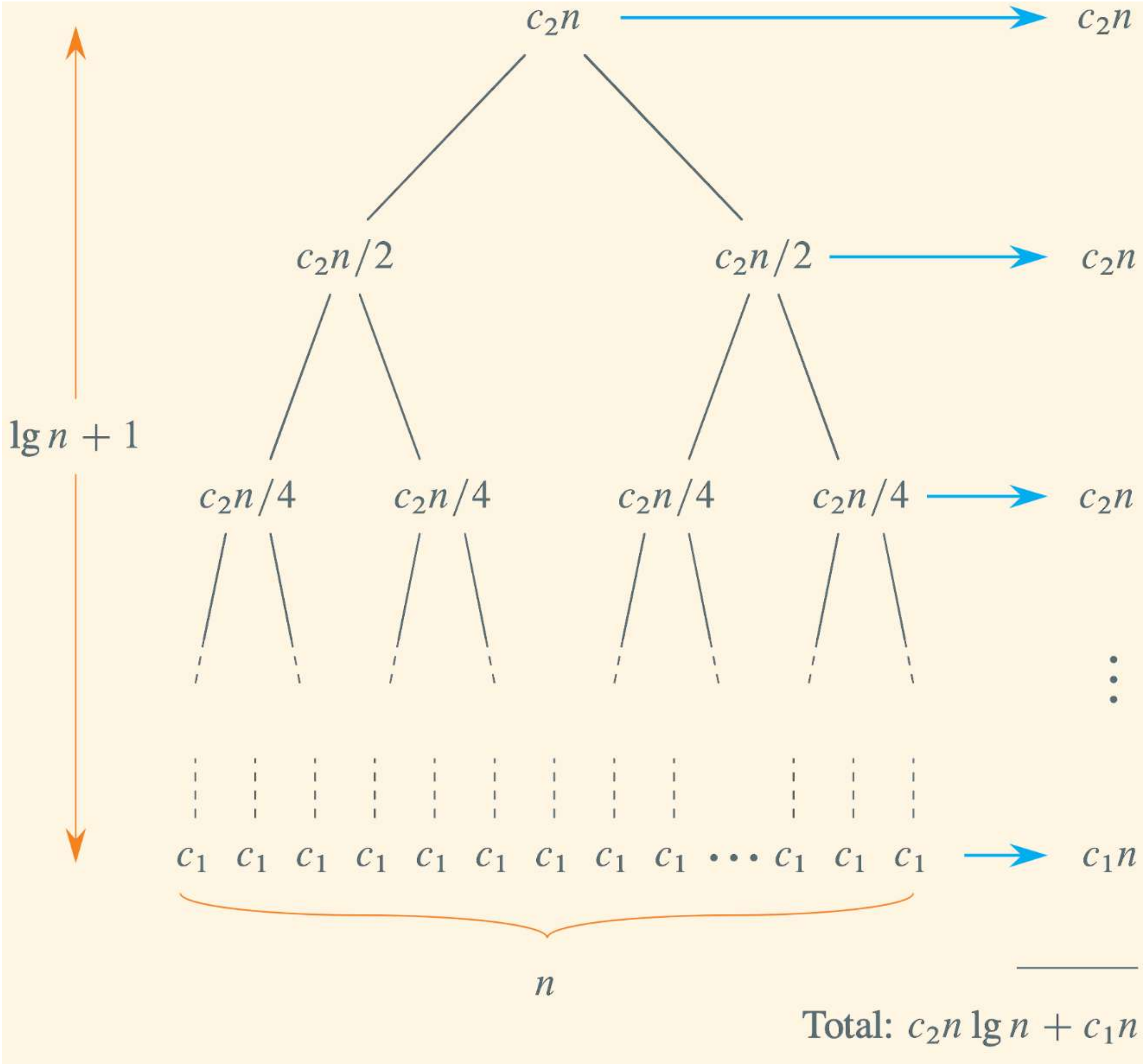
Recursion Tree for Merge Sort

- Figures 2.5(a)–(c) from Cormen *et al.* (2022) illustrate the initial steps in constructing the recursion tree for merge sort. The tree is built by recursively evaluating Equation (1.7) at each level.



Final Recursion Tree for Merge Sort

- Figure 2.5(d) from Cormen *et al.* (2022) depicts the complete recursion tree for merge sort. The tree consists of lgn+1 levels, where the root level represents the original problem of size n and the leaves correspond to subproblems of size 1.



(d)

Agenda

- Recursion Trees
- Divide & Conquer
- Backtracking
- Dynamic Programming

Divide & Conquer Principle

We can solve the problem recursively, applying the following three steps at each level of recursion:


"Really? — my people always say *multiply* and conquer."

1. Divide
the problem into a number of smaller sub-problems

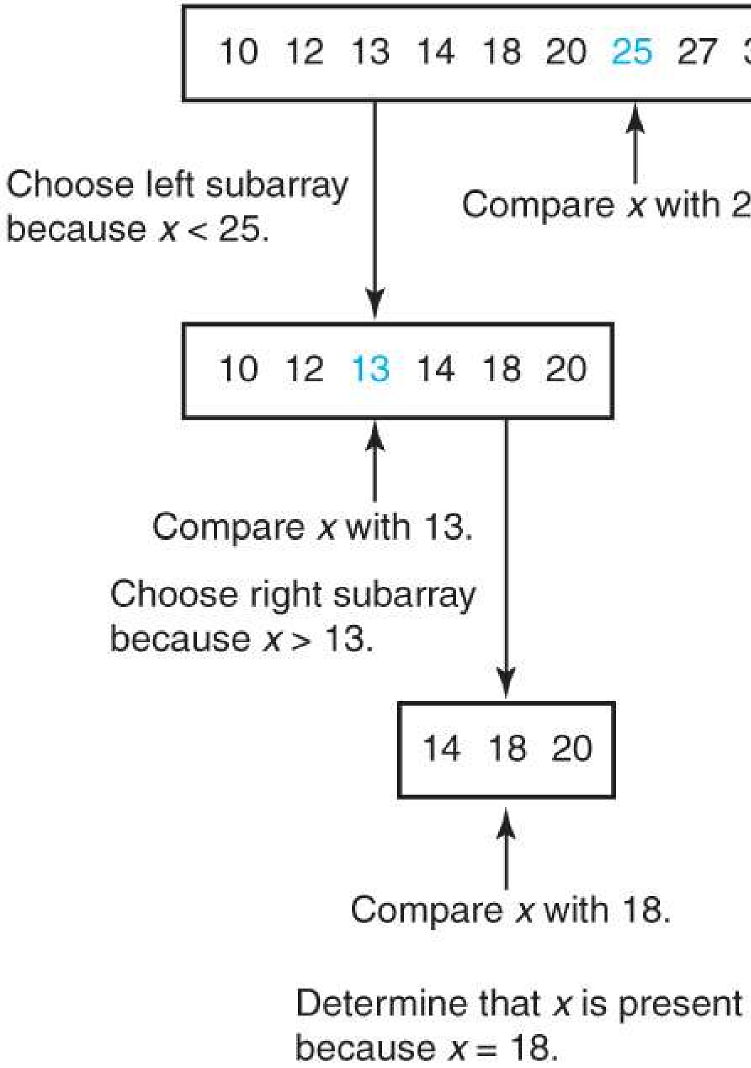
2. Conquer
the sub-problems by solving them recursively

3. Combine
the solutions to the sub-problems to form the solution. (optional)

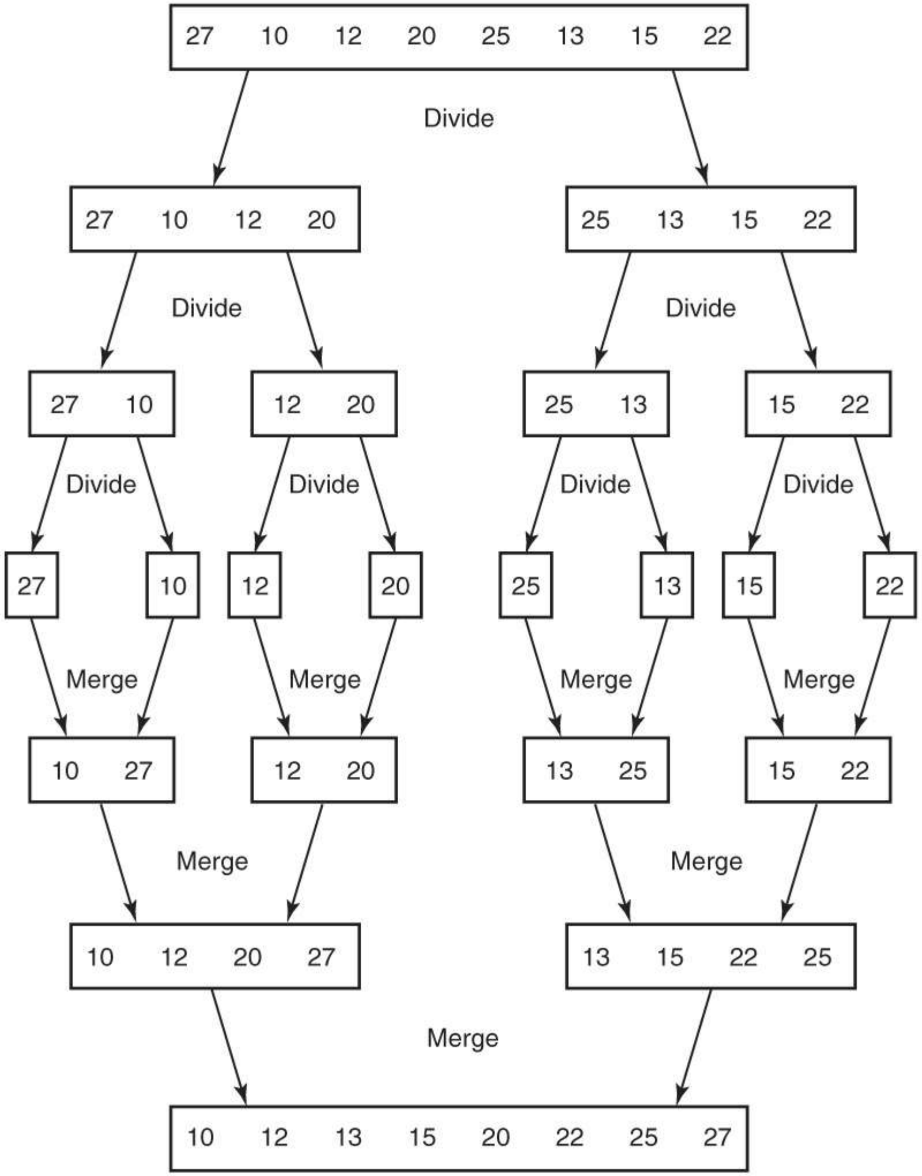
Divide & Conquer: Base case

- Once the sub-problem becomes small enough to solve easily, we stop the recurring divide.
- It means we have reached the base case.
- It is important that the divide process reaches the base case so that the algorithm does not recur infinitely.
- Examples
  - Binary Search
  - Merge Sort
  - Quick Sort

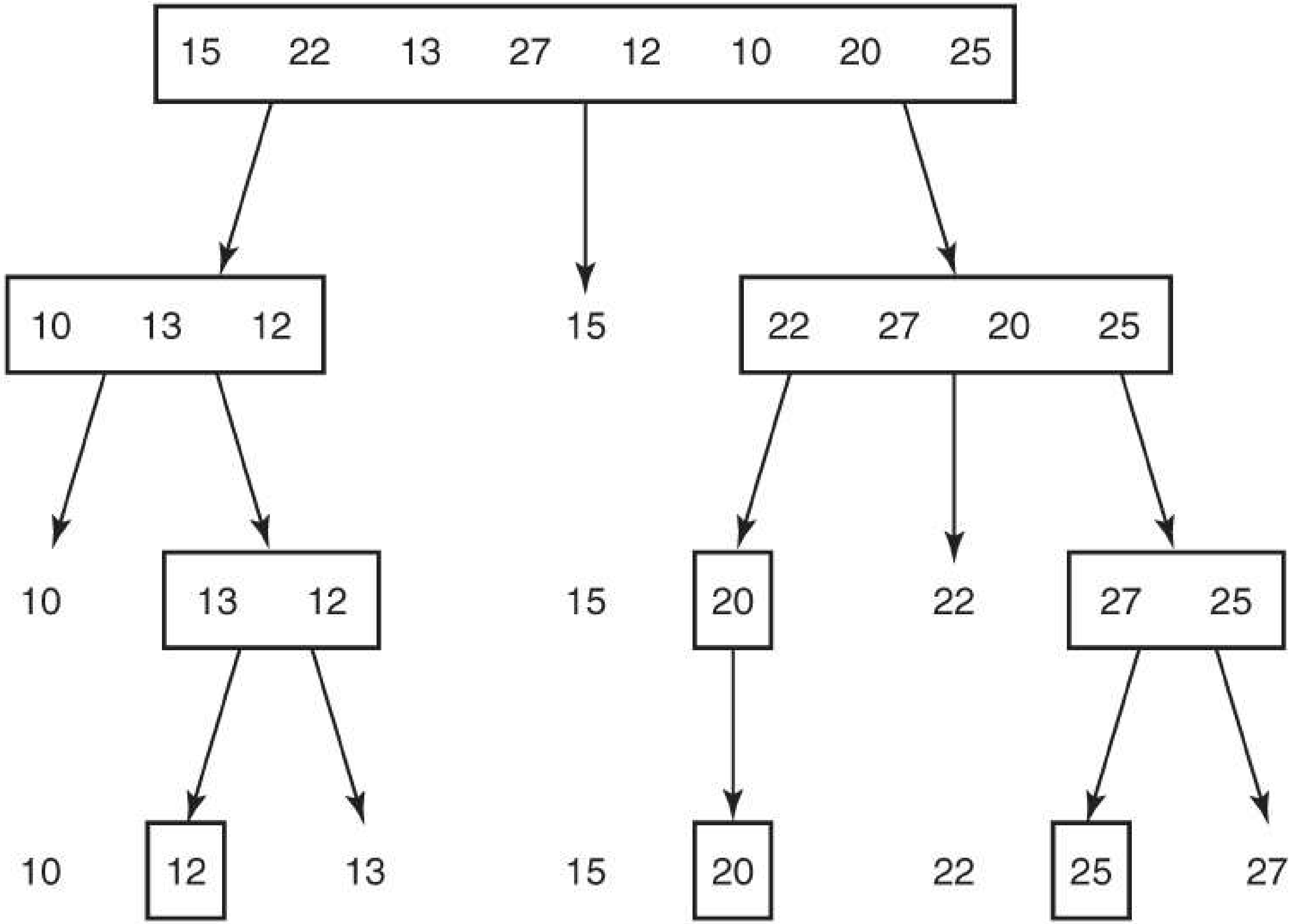
Example: Binary Search for x=18



Example: Merge Sort



Example: Quick Sort



Agenda

- Recursion Trees
- Divide & Conquer
- Backtracking
- Dynamic Programming

Backtracking Algorithms

- Sometimes, we have to make a series of *decisions,* among various *choices,* where
  - We don't have enough information to know what to choose.
  - Each decision leads to a new set of choices.
  - Some sequence of choices (possibly more than one) may be a solution to our problem.
- Backtracking is a methodical way of trying out various sequences of decisions, until we find one that works.

Backtracking Algorithms

SINGAPORE INSTITUTE OF TECHNOLOGY

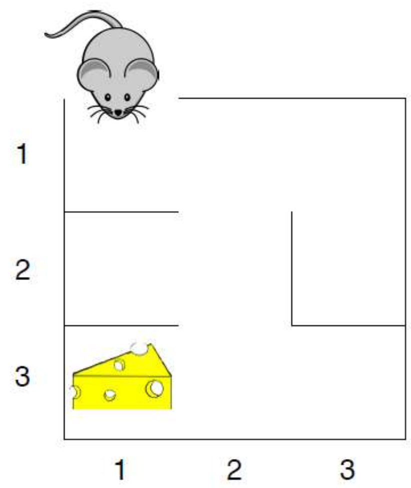
- Based on depth-first recursive search.
- Approach
  1. Tests whether solution has been found.
  2. If found solution, return it.
  3. Else, for each choice that can be made.
     a) Make that choice.
     b) Recur.
     c) If recursion gives a solution, return it.
  4. If no choices remain, return failure.
- Sometimes called a "search tree".

Backtracking Algorithm – Example

- Find path through maze.
  - Start at beginning of maze.
  - If at exit, return true.
  - Else, for each step from current location.
    - Recursively find path.
    - Return with first successful step.
    - Return false if all steps fail.

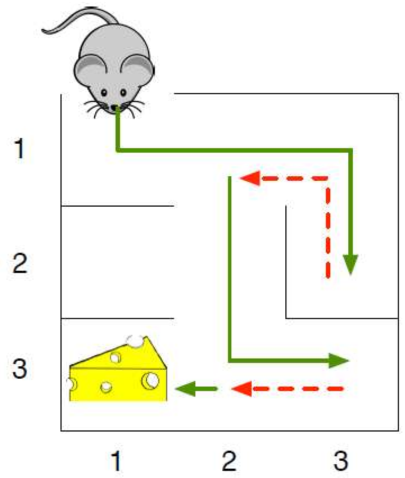
Backtracking Algorithm – Example

- Backtracking: systematic search technique to completely work through solution space.
- Prime example: labyrinth.
  How does the mouse find the cheese?



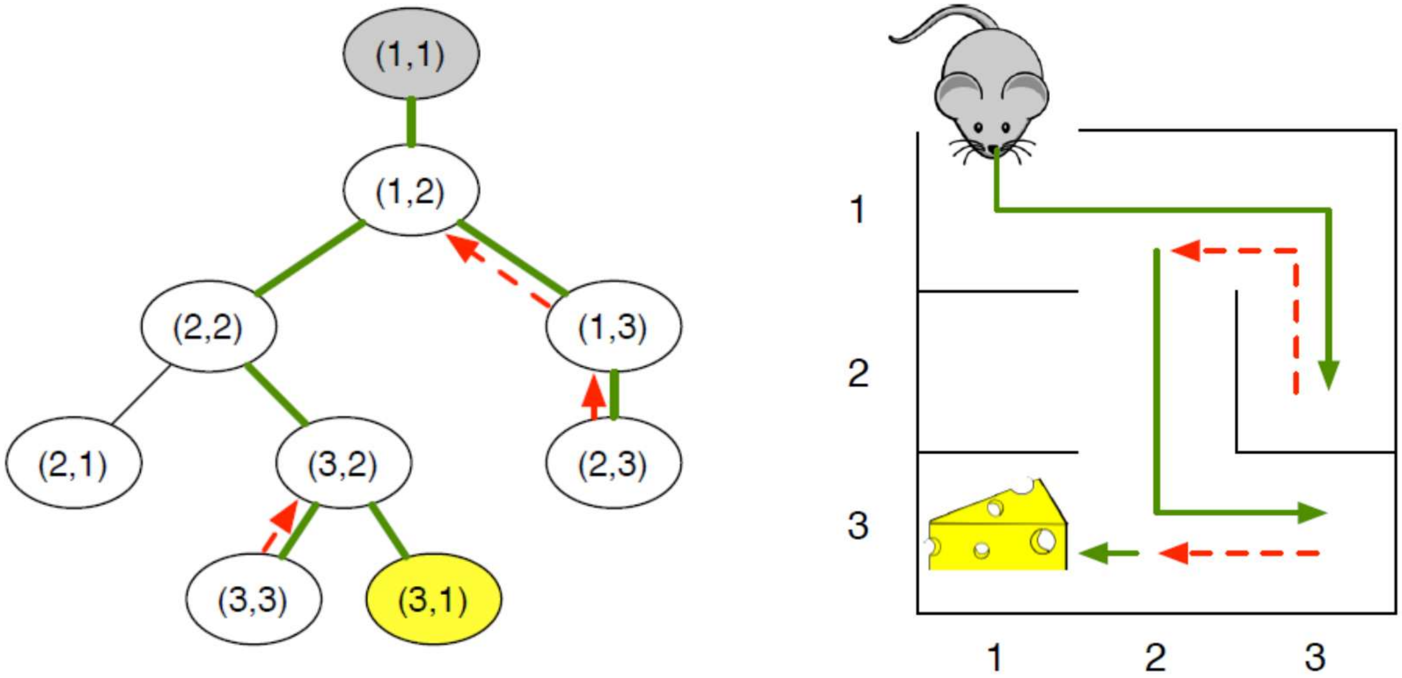
Backtracking

- Problem: How does the mouse find the cheese?
- Solution:
  – systematic exploration of the labyrinth.
  – backtrack if meet deadend (hence backtracking).
  → trial and error.



Backtracking

- Possible paths (use a tree to represent maze):



Backtracking – Pseudocode

Input: K configuration.

```
BackTrack (K):
    if  K is solution:
            output K;
    else:
            for each direct extension K' of K:
                    BackTrack (K')
```

Initial call using "BackTrack (K_0)".

Backtracking

- Termination of backtracking:
  - only if solution space is finally exhausted.
  - only if it is ensured that no configurations remain to be tested.
- Complexity of backtracking:
  - directly dependent on the size of solution space.
  - usually exponential, thus $O(2^n)$ or worse!
  - can use for small problems only.
- Alternative:
  - limit the depth of recursion.
  - then select the best solution so far,
    eg. chess programs.

The n-Queens Problem

Find all possible ways of placing n queens on an $n \times n$ chessboard so that no two queens occupy the same row, column, or diagonal.

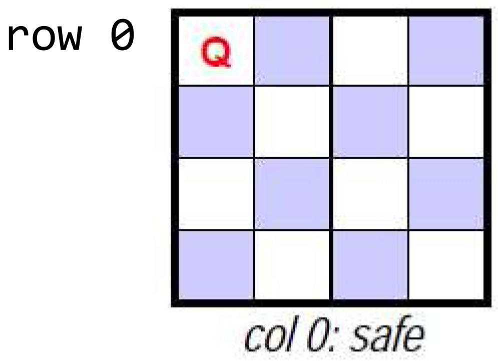
The n-Queens Problem

Sample solution for $n = 8$:

Q							
				Q			
							Q
					Q		
		Q					
						Q	
	Q						
			Q				

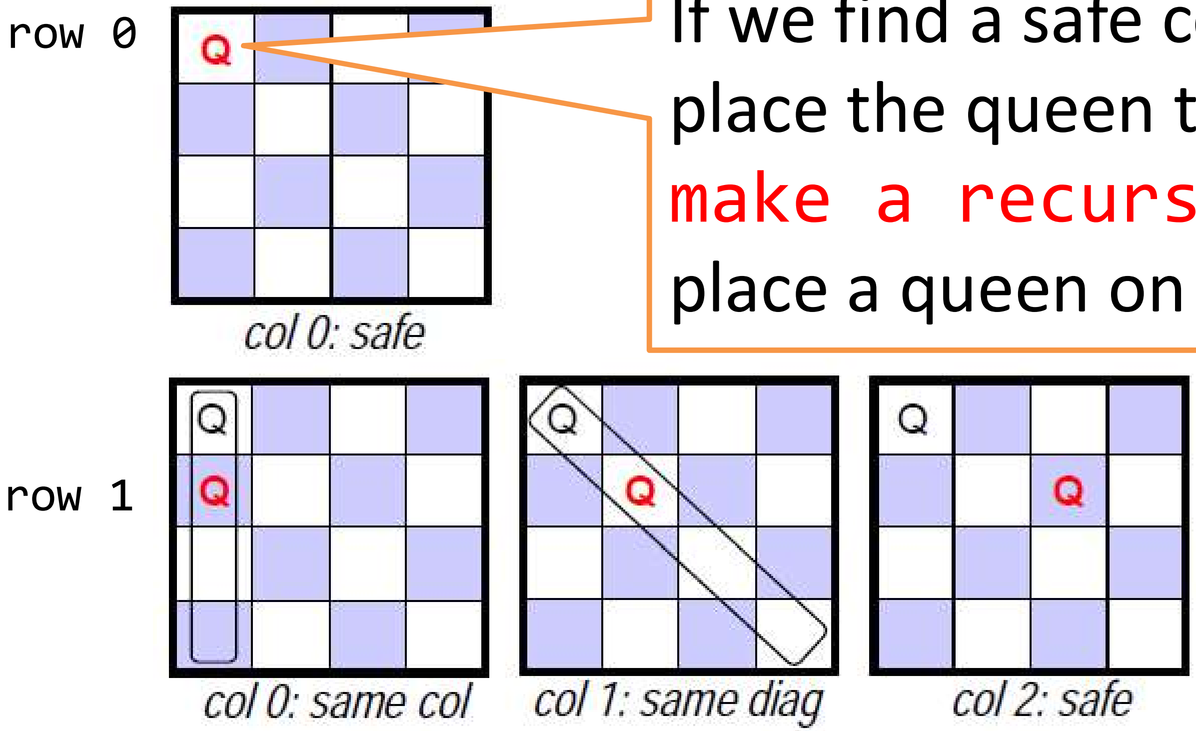
This is a classic example of a problem that can be solved using a technique called `recursive backtracking`.

Recursive Strategy for n-Queens



Consider one row at a time.
Within the row,
consider one column at a time.
Look for a "safe" column
to place a queen.

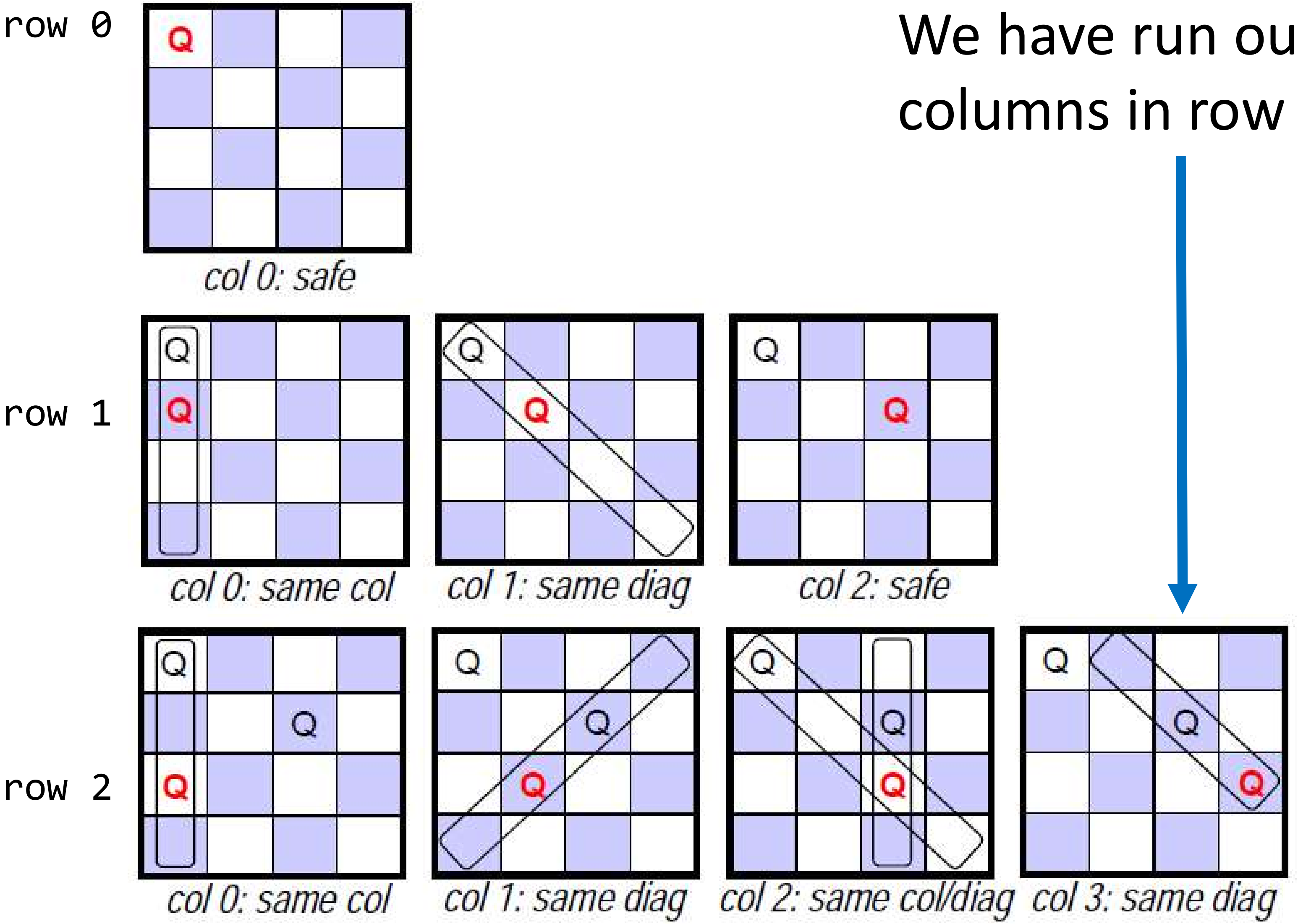
Recursive Strategy for n-Queens



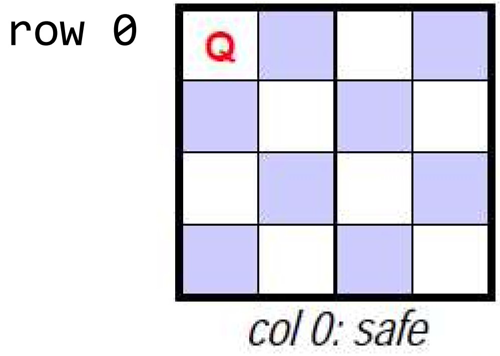
If we find a safe column, place the queen there, and `make a recursive call` to place a queen on the `next row`.

Recursive Strategy for n-Queens

We have run out of columns in row 2!

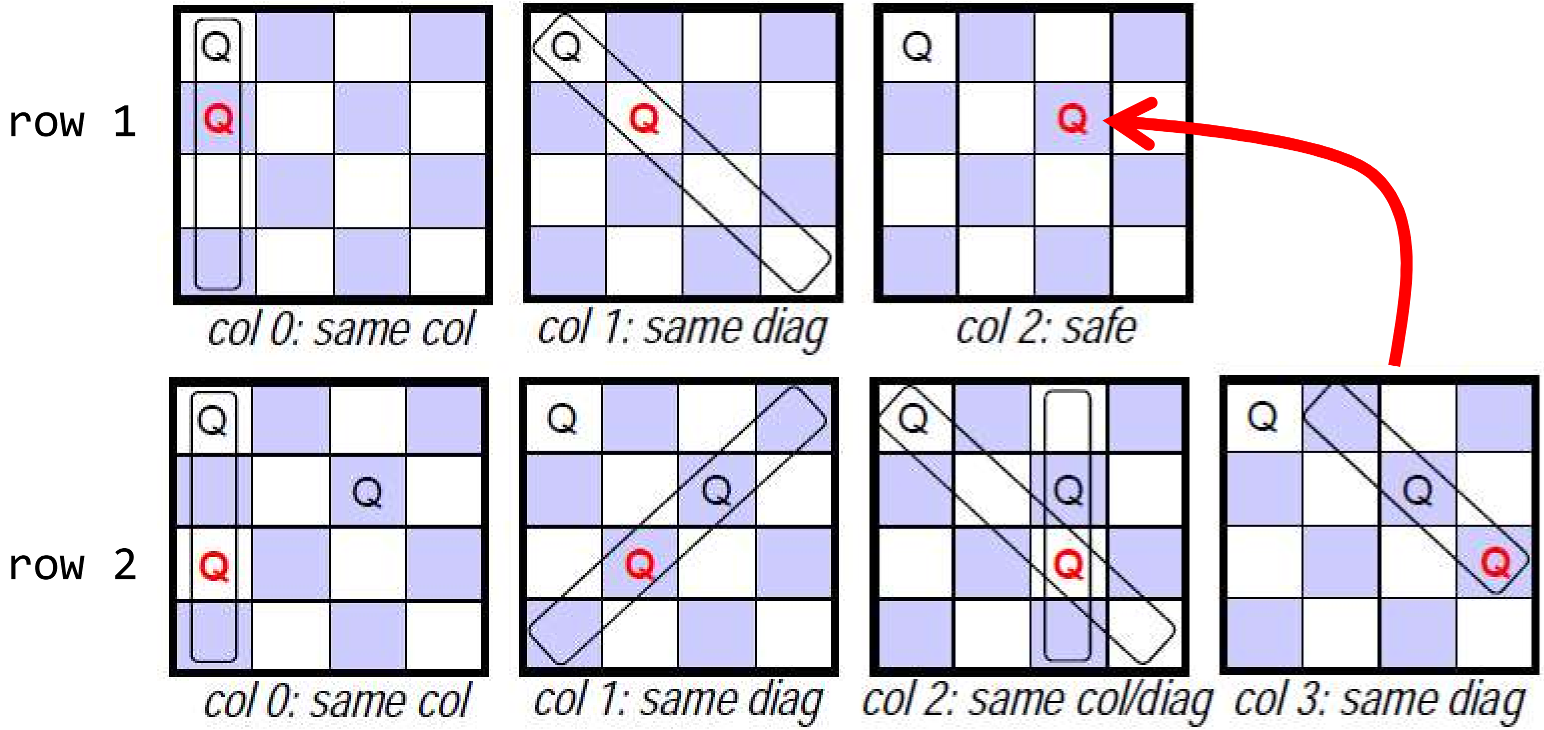


Recursive Strategy for n-Queens

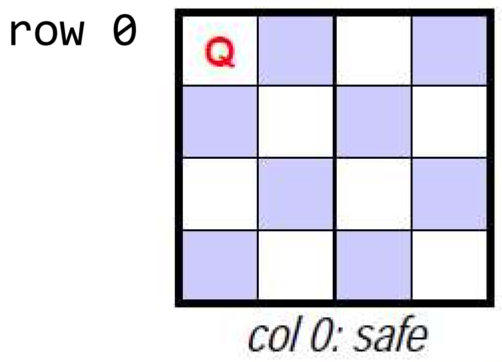


Backtrack to row 1 by returning from the recursive call.

- pick up where we left off.
- we had already tried columns 0-2, so now we try column 3.

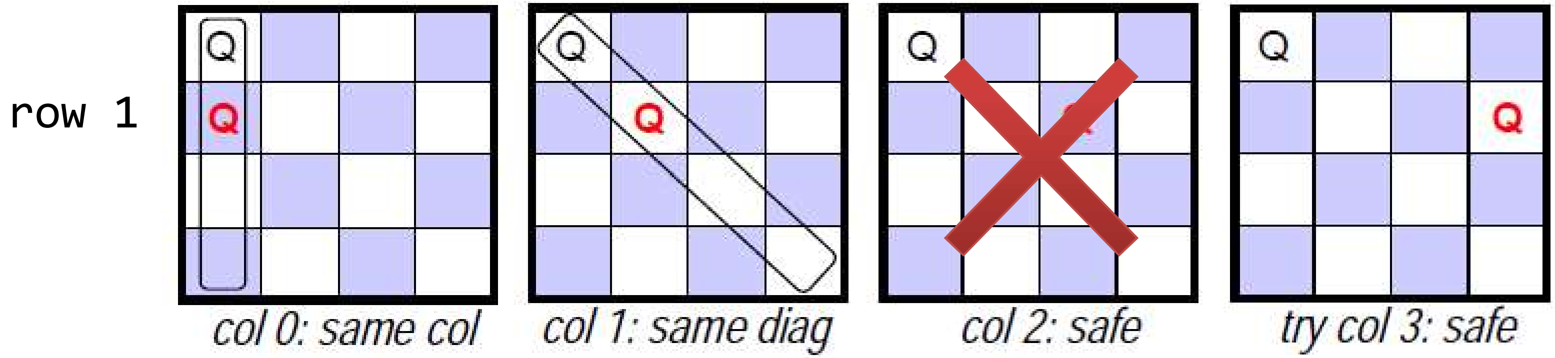


Recursive Strategy for n-Queens

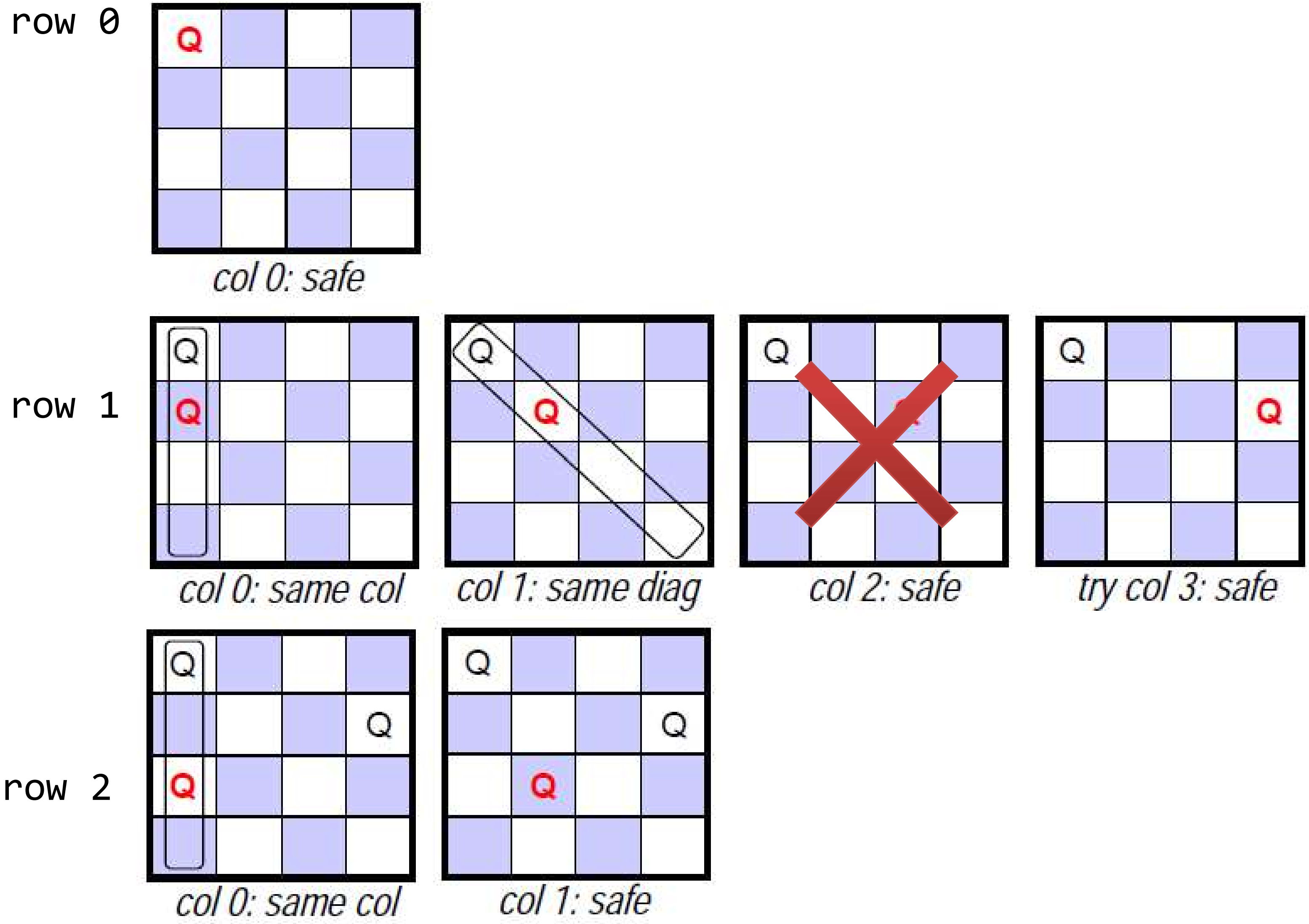


Backtrack to row 1 by returning from the recursive call.

- pick up where we left off.
- we had already tried columns 0-2, so now we try column 3.

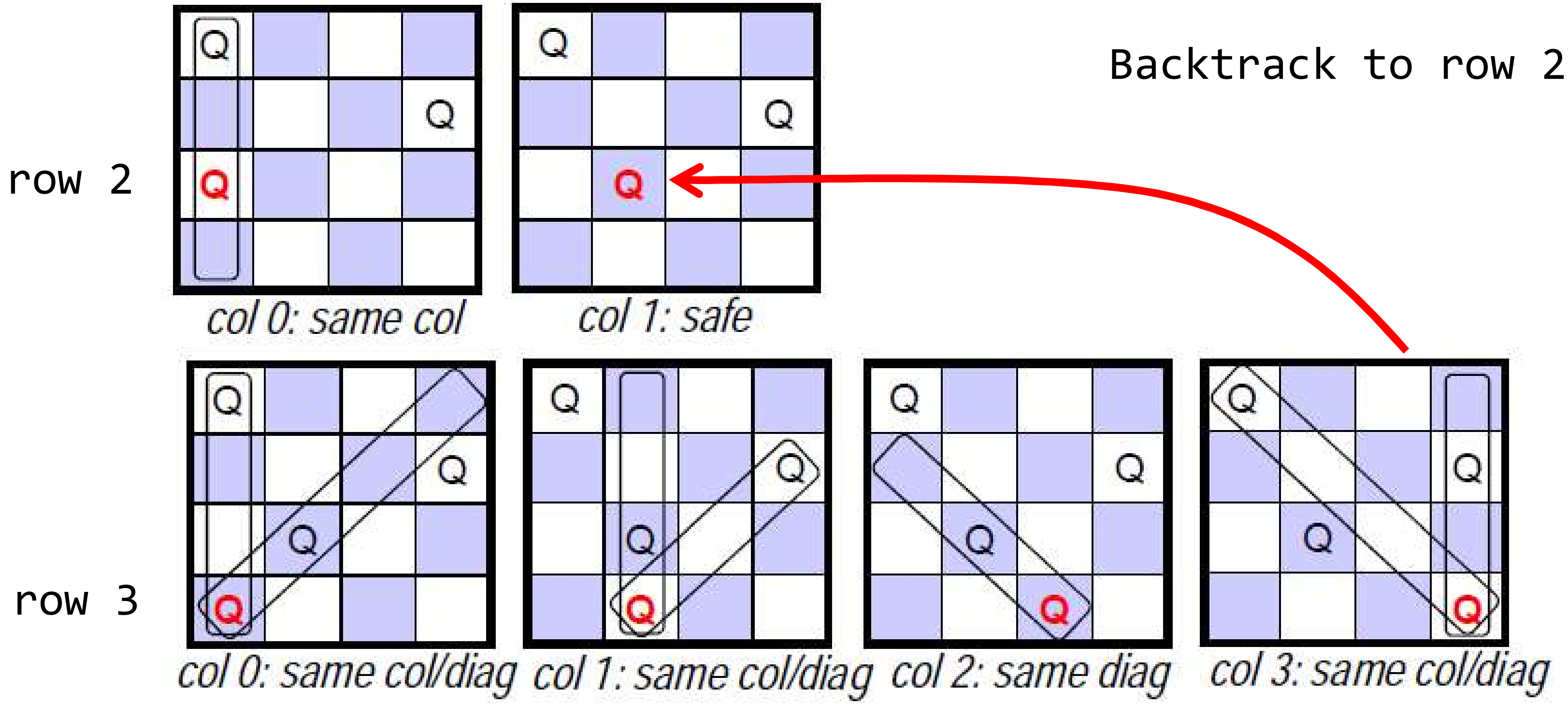


Recursive Strategy for n-Queens



Recursive Strategy for n-Queens

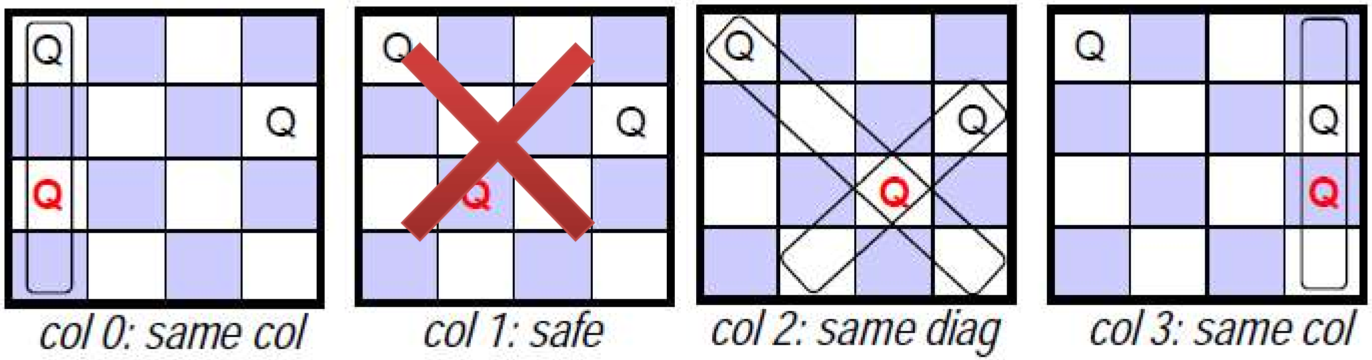
Backtrack to row 2:



Recursive Strategy for n-Queens

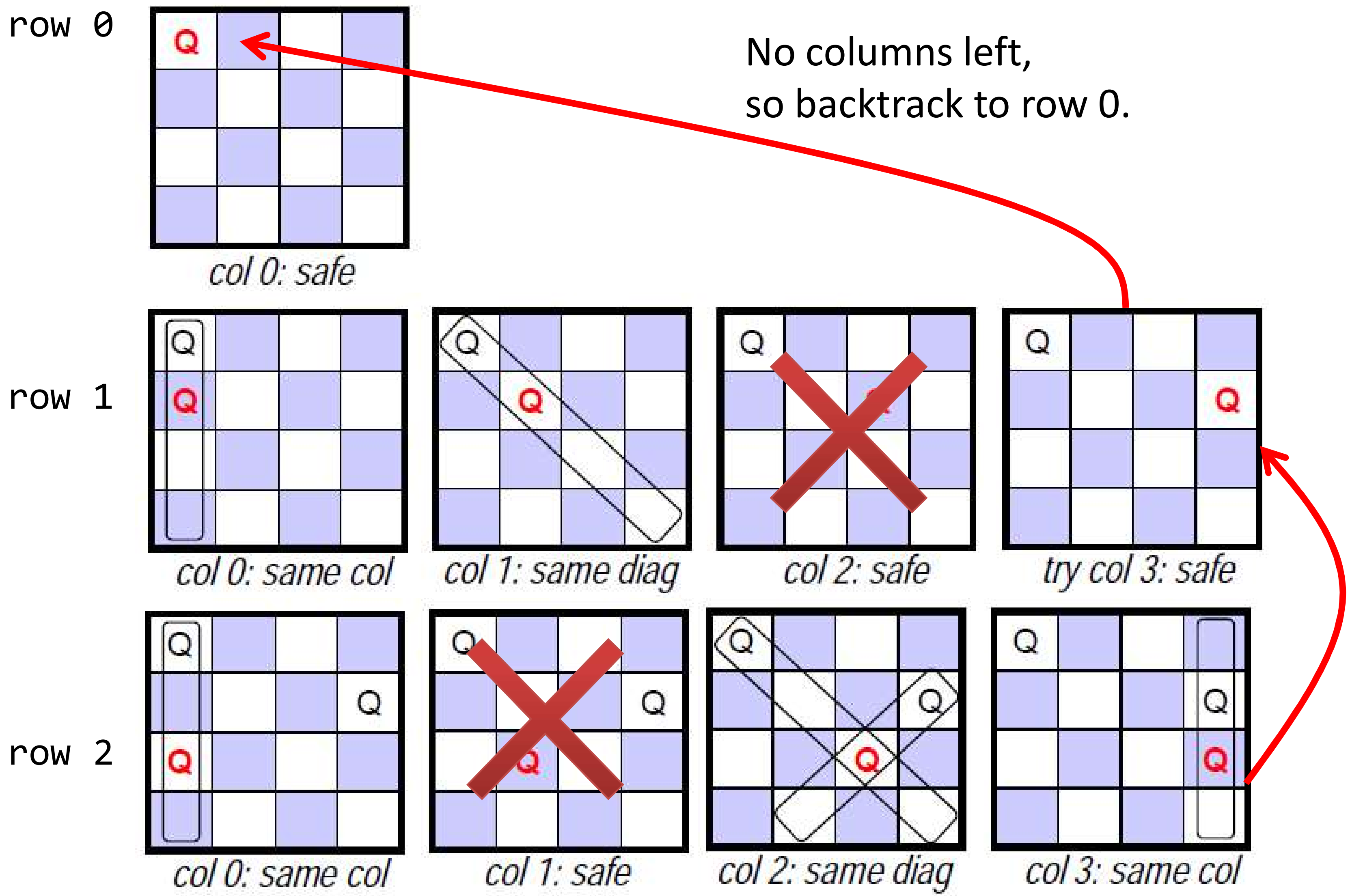
Backtrack to row 1.

row 2



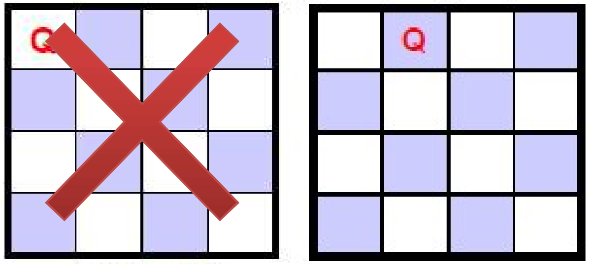
Recursive Strategy for n-Queens

No columns left,
so backtrack to row 0.



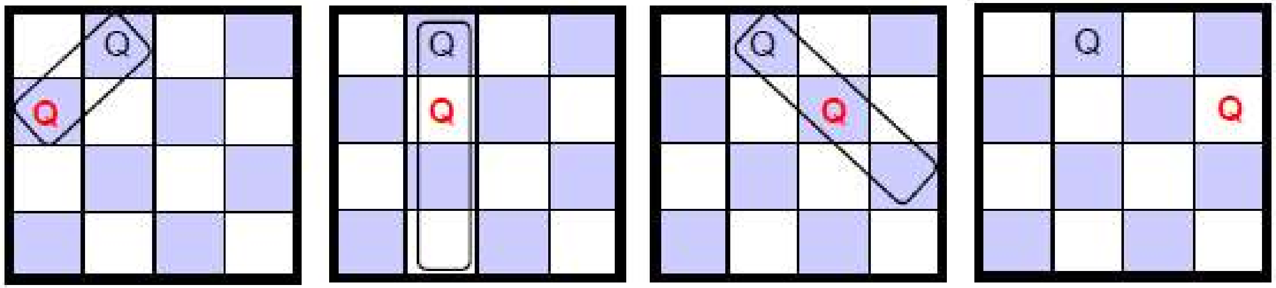
SINGAPORE INSTITUTE OF TECHNOLOGY

row 0

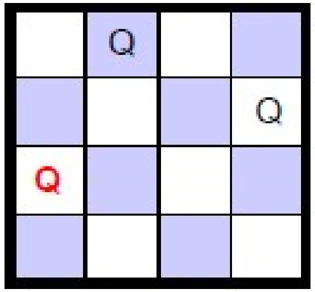


col 0: safe

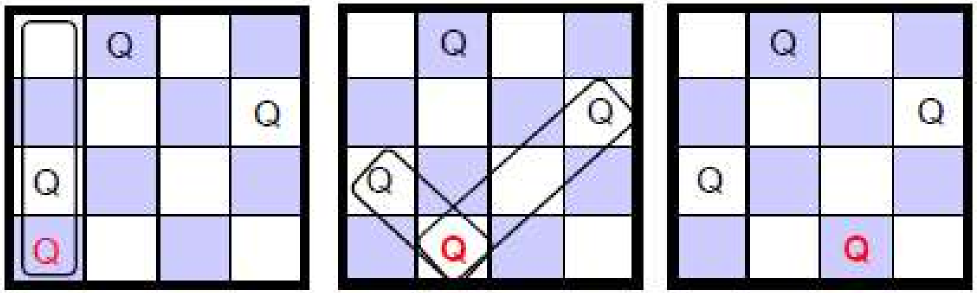
row 1



row 2



row 3



A solution!

Recursive Strategy for n-Queens

```python
def findValidCol(row,chessBoard):
    N=len(chessBoard)
    for col in range(N):
        if isValid(col,row,chessBoard):
            chessBoard[row][col]=1
            if (row < N-1):
                 findValidCol(row+1,chessBoard)
            else:
                printSolution(chessBoard)
                exit()
            chessBoard[row][col]=0
```

Recursive Strategy for n-Queens

```
def findValidCol(row,chessBoard):
    N=len(chessBoard)
    for col in range(N):
        if isValid(col,row,chessBoard):
            chessBoard[row][col]=1
            if (row < N-1):
                findValidCol( ow+1,chessBoard)
            else:
                printSolu     (chessBoard)
                exit()
            chessBoa         [col]=0
```

For the given row, if column col is valid
(ie. no queen in the same column and 2 diagonals),
put the queen at the col.

Recursive Strategy for n-Queens

```
def findValidCol(row,chessBoard):
    N=len(chessBoard)
    for col in range(N):
        if isValid(col,row,chessBoard):
            chessBoard[row][col]=1
            if (row < N-1):
                findValidCol(row+1,chessBoard)
            else:
                printSolution(chessBoard)
                exit()
            chessBoard[row][col]=0
```

If it is not the last row, `make a recursive call` to place a queen on the `next row`.

Recursive Strategy for n-Queens

```
def findValidCol(row,chessBoard):
    N=len(chessBoard)
    for col in range(N):
        if isValid(col,row,chessBoard):
            chessBoard[row][col]=1
            if (row < N-1):
                findValidCol(row+1,chessBoard)
            else:
                printSolution(chessBoard)
                exit()
            chessBoard[row][col]=0
```

If `row==(N-1)` (last row), it means a solution is found, then print the solution.

Recursive Strategy for n-Queens

```
def findValidCol(row,chessBoard):
    N=len(chessBoard)
    for col in range(N):
        if isValid(col,row,chessBoard):
            chessBoard[row][col]=1
            if (row < N-1):
                 findValidCol(row+1,chessBoard)
            else:
                printSolution(chessBoard)
                exit()
            chessBoard[row][col]=0
```

If the current valid col does not work, back track and try the next col, or back track to the previous row.

Agenda

- Recursion Trees
- Divide & Conquer
- Backtracking
- Dynamic Programming

Dynamic Programming

- Over-lapping sub-problem and
- Optimal Substructure
- Involves caching or memory of solved subproblems

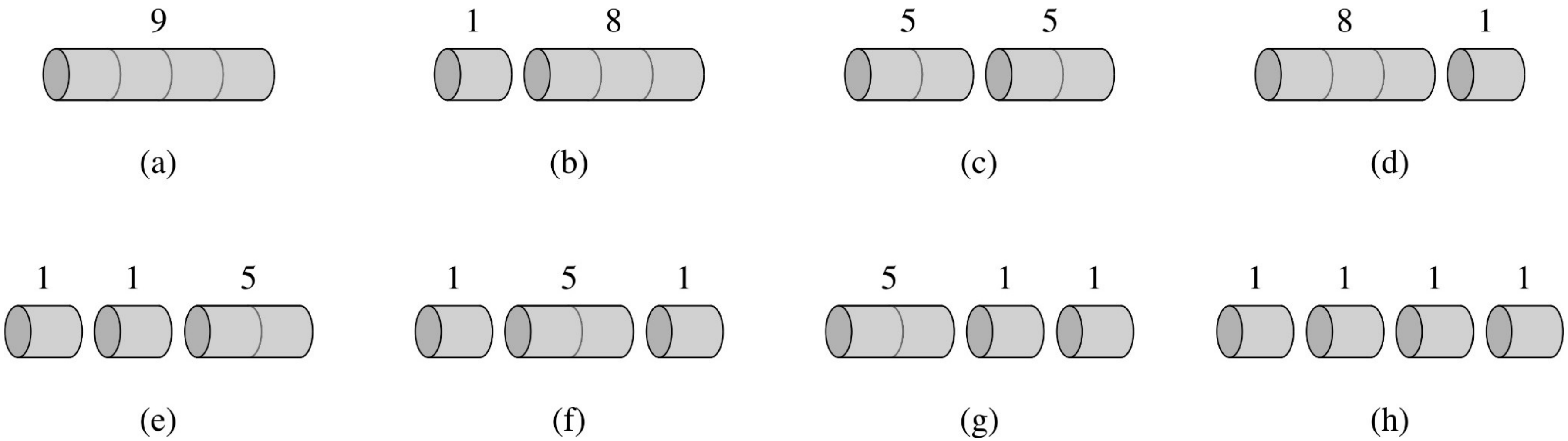
Dynamic Programming: Rod-cutting Problem

- Given a rod of length n metres and a table of prices p_i for length $i = 1, 2, \dots, n$. Determine the maximum revenue r_n for cutting up the rod and selling the pieces.
- Divide & conquer vs Dynamic programming.
- Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

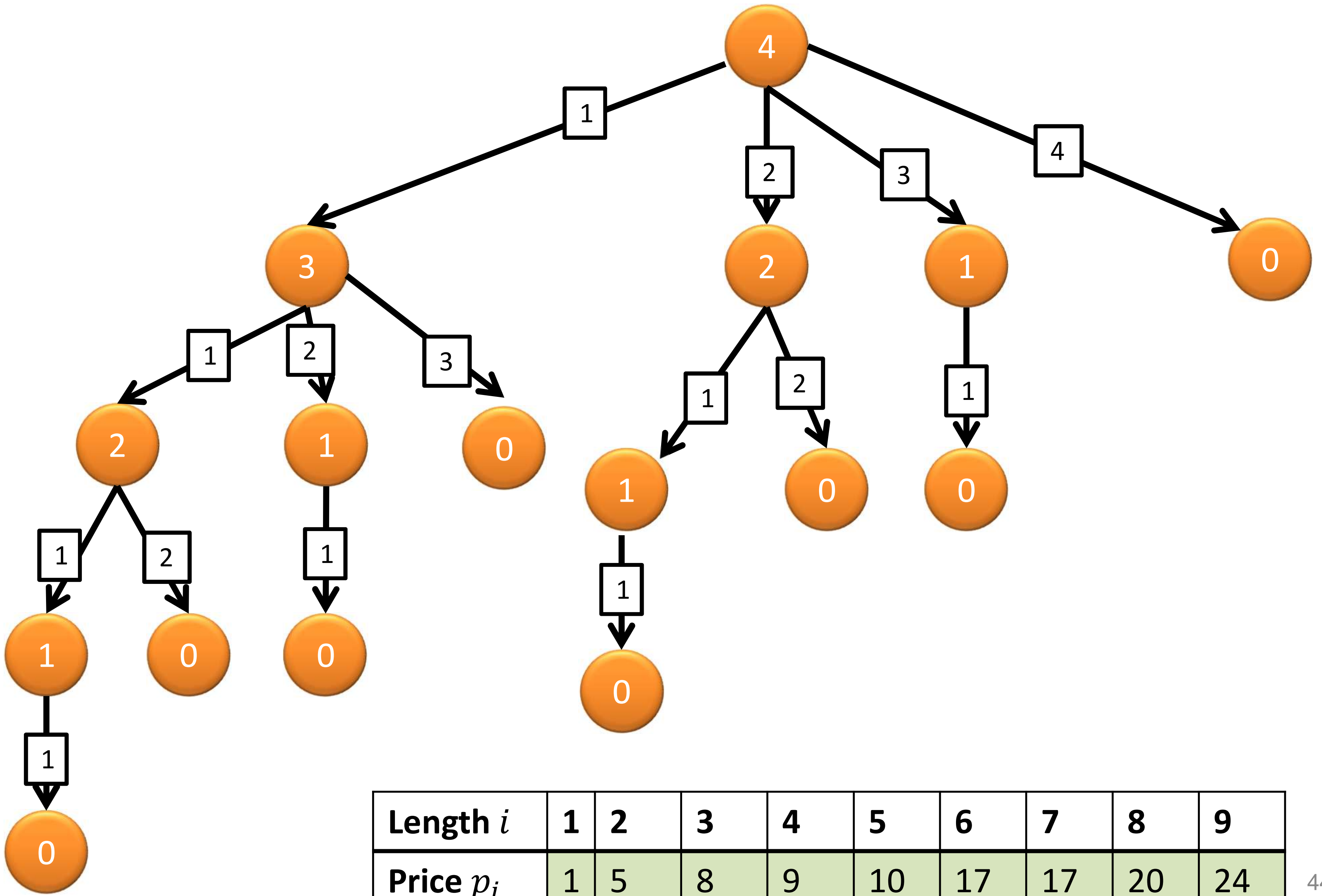
Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

Rod-cutting Problem

- For a rod of length n, there are 2^{n-1} ways to cut.
- Example, when $n = 4$,
  there are 8 possible ways to cut the rod.

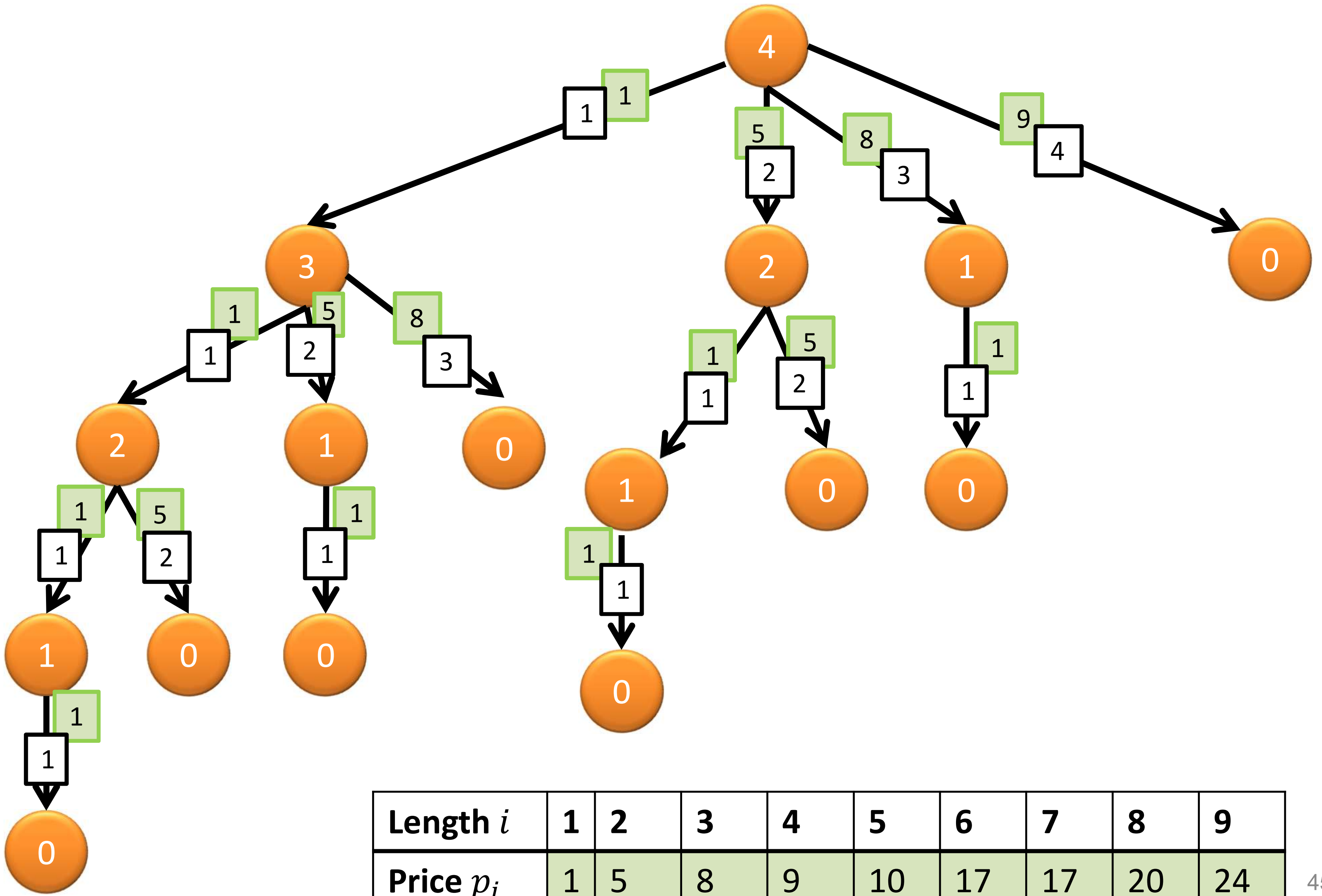


Rod-cutting Problem: Divide and Conquer solution



Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

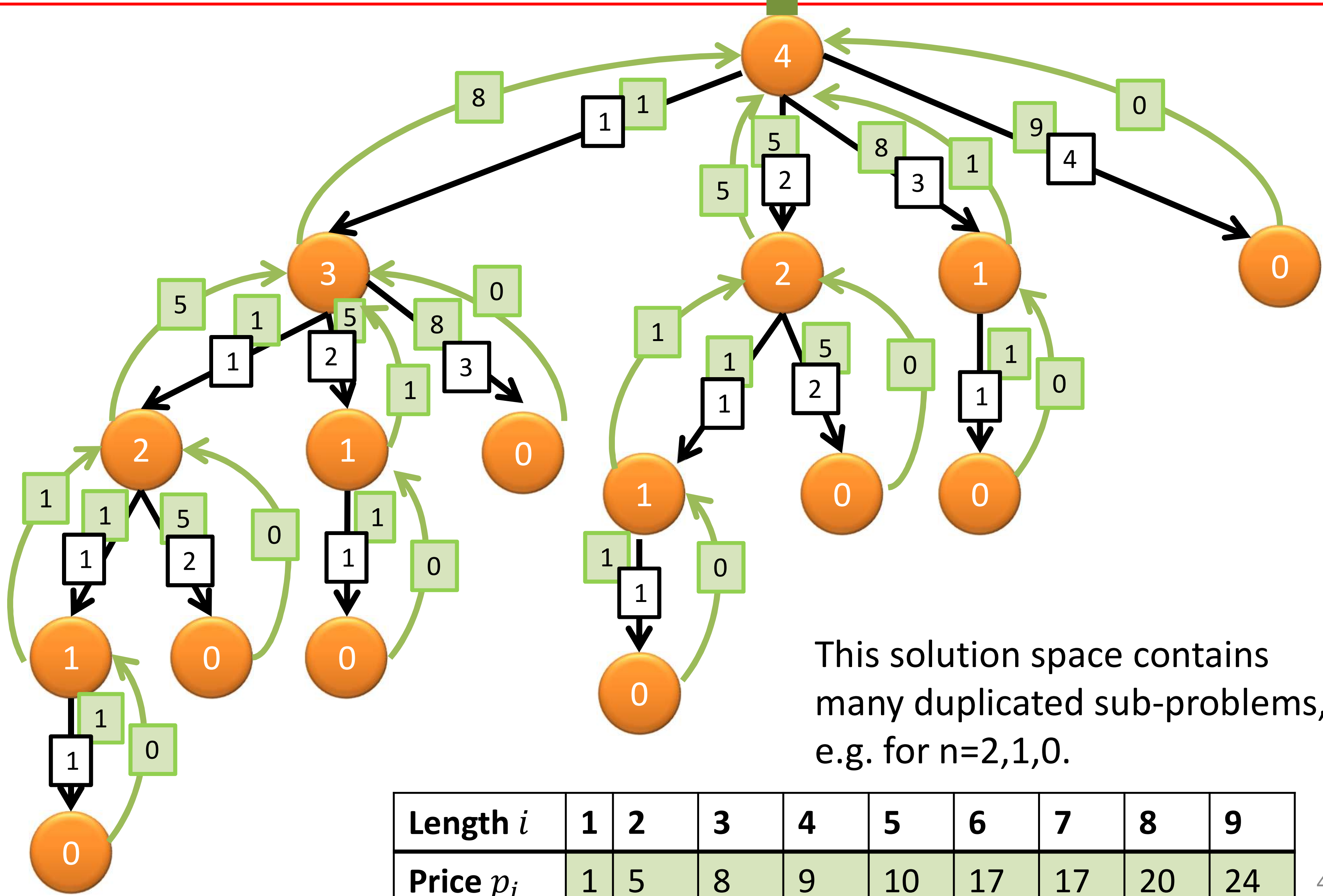
Rod-cutting Problem: Divide and Conquer solution



Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

Rod-cutting Problem: Divide and Conquer solution

Return max(1+8, 5+5, 8+1, 9+0) = 10



This solution space contains many duplicated sub-problems, e.g. for n=2,1,0.

Length i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Price p_i	1	5	8	9	10	17	17	20	24

Observations from Divide-Conquer Solution

- The sub-problems (with n=2,1,0) are solved repeatedly.
- Better to solve each sub-problem only once, and save each solution.
- If we encounter same sub-problem again, just look it up (don't recompute it).

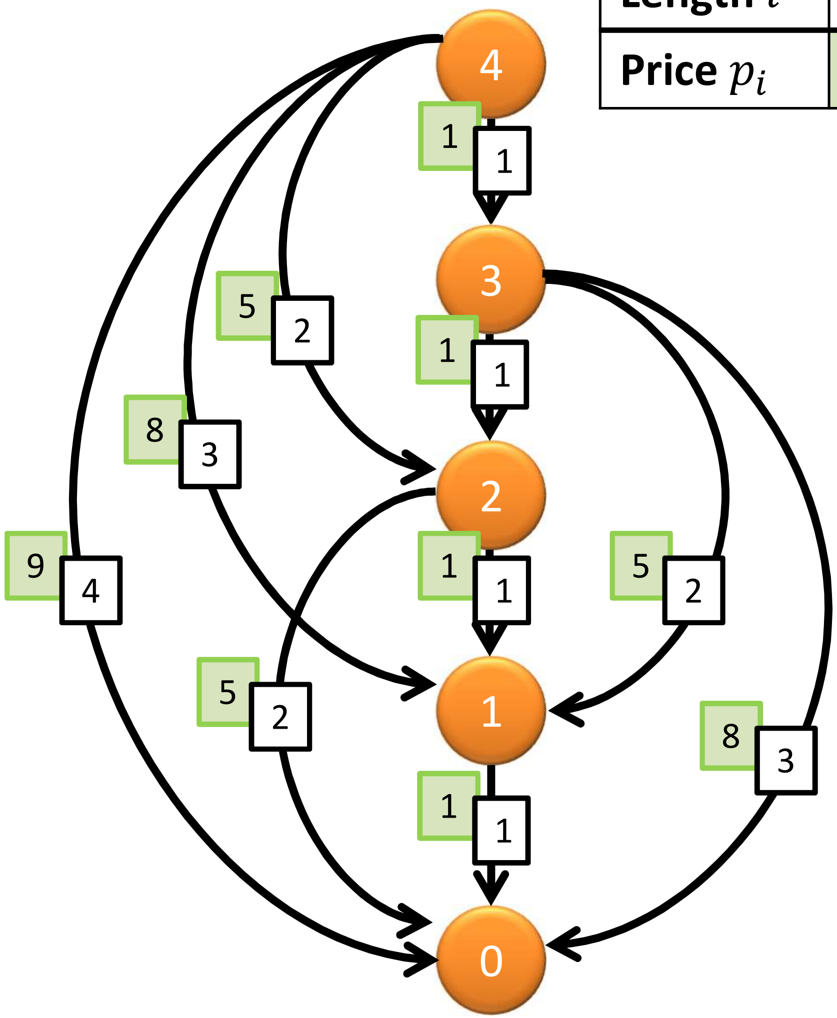
Dynamic Programming

- Dynamic programming stores the solutions to each sub-problem in case they are needed again.
- Uses additional memory to cut computation time.
- Time-memory trade-off.
- Dynamic programming can transform many exponential-time algorithms into polynomial-time.

Rod-cutting Problem: Dynamic Programming

SIT Internal

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24



In this solution, if the answer to a sub-problem has been stored, there will be no further recursive calls made.

A Final Note

- Dynamic programming is typically applied to optimization problems.
- Such problems can have many possible solutions.
- Each solution has a value and we wish to find a solution with the optimal value.