SINGAPORE INSTITUTE OF TECHNOLOGY

Backtracking, Greedy, Dynamic Programming

Agenda

- Divide & Conquer
- Backtracking
- Greedy Algorithm
- Dynamic Programming

Recommended Readings

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

Agenda

- Divide & Conquer
- Backtracking
- Greedy Algorithm
- 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

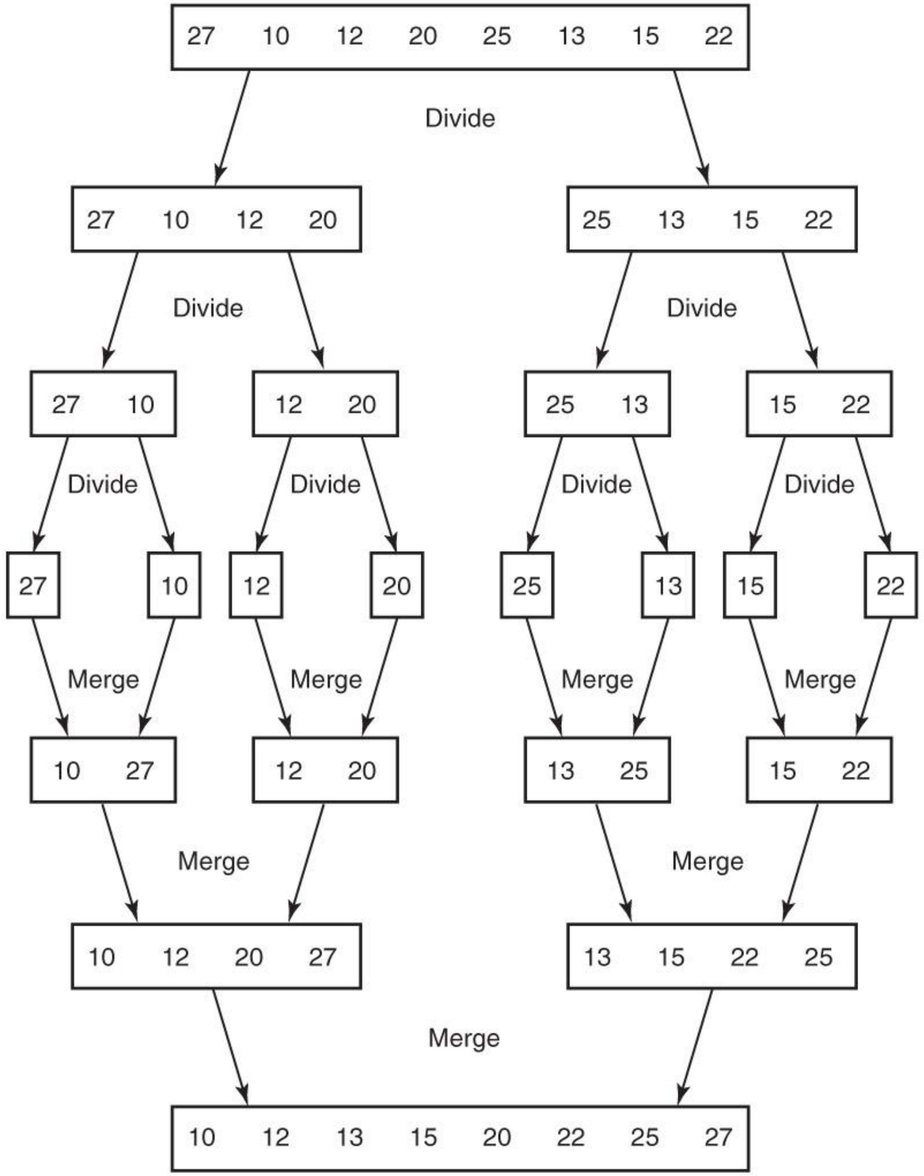
2. Conquer
the sub-problems by solving them recursively

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

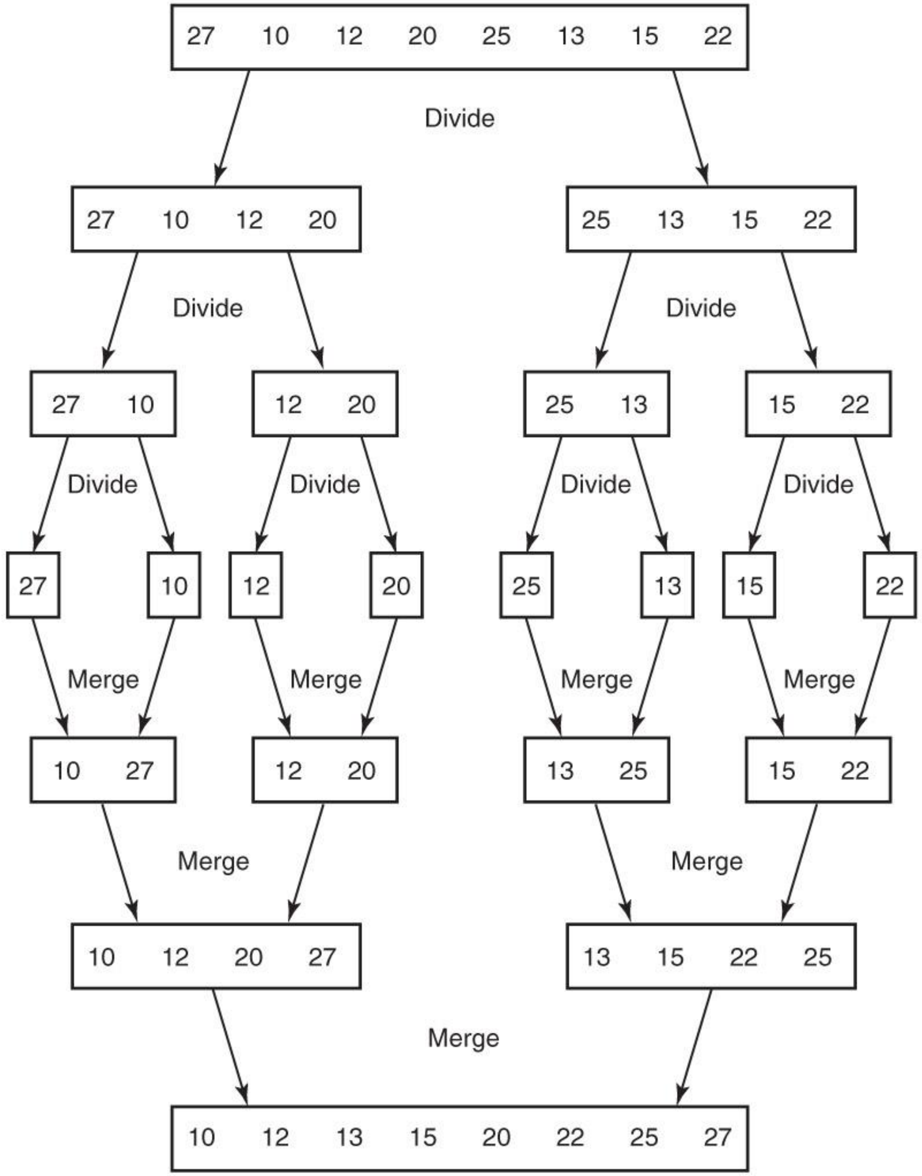
Divide & Conquer: Base case

- 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
  - Merge Sort
  - Binary Search
  - Powering a number
  - Fib Numbers

Example: Merge Sort



Example: Merge Sort



$$T(n) = 2T(n/2) + O(n)$$

Master Method Case 2: O(nlg(n))

SINGAPORE INSTITUTE OF TECHNOLOGY

$$T(n) = 2T\left(\frac{n}{2}\right) + n \qquad T(n) = O(nlgn)$$

$$T(n) \le 2\left(c\frac{n}{2}lg\frac{n}{2}\right) + n = cn(lgn - lg2) + n = cnlgn - cn + n = cnlgn - (c-1)n$$

$$T(n) \le cnlgn - (c-1)n \le cnlgn \quad \textit{via substitution method}$$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ via tree

n

$T\left(\frac{n}{2}\right) \quad T\left(\frac{n}{2}\right)$ $\qquad T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$

$n \qquad \frac{n}{2} \quad \frac{n}{2}$

$T\left(\frac{n}{2^2}\right) \quad T\left(\frac{n}{2^2}\right) \quad T\left(\frac{n}{2^2}\right) \quad T\left(\frac{n}{2^2}\right)$ $\qquad T\left(\frac{n}{2^2}\right) + 2T\left(\frac{n}{2^4}\right) + \frac{n}{2^2}$

$n \qquad \frac{n}{2^2} \quad \frac{n}{2^2} \quad \frac{n}{2^2} \quad \frac{n}{2^2}$

$n \times lg\, n$

master method

$T(n) = 2T\left(\frac{n}{2}\right) + n$

$a = 2, \; b = 2, \; f(n) = n$

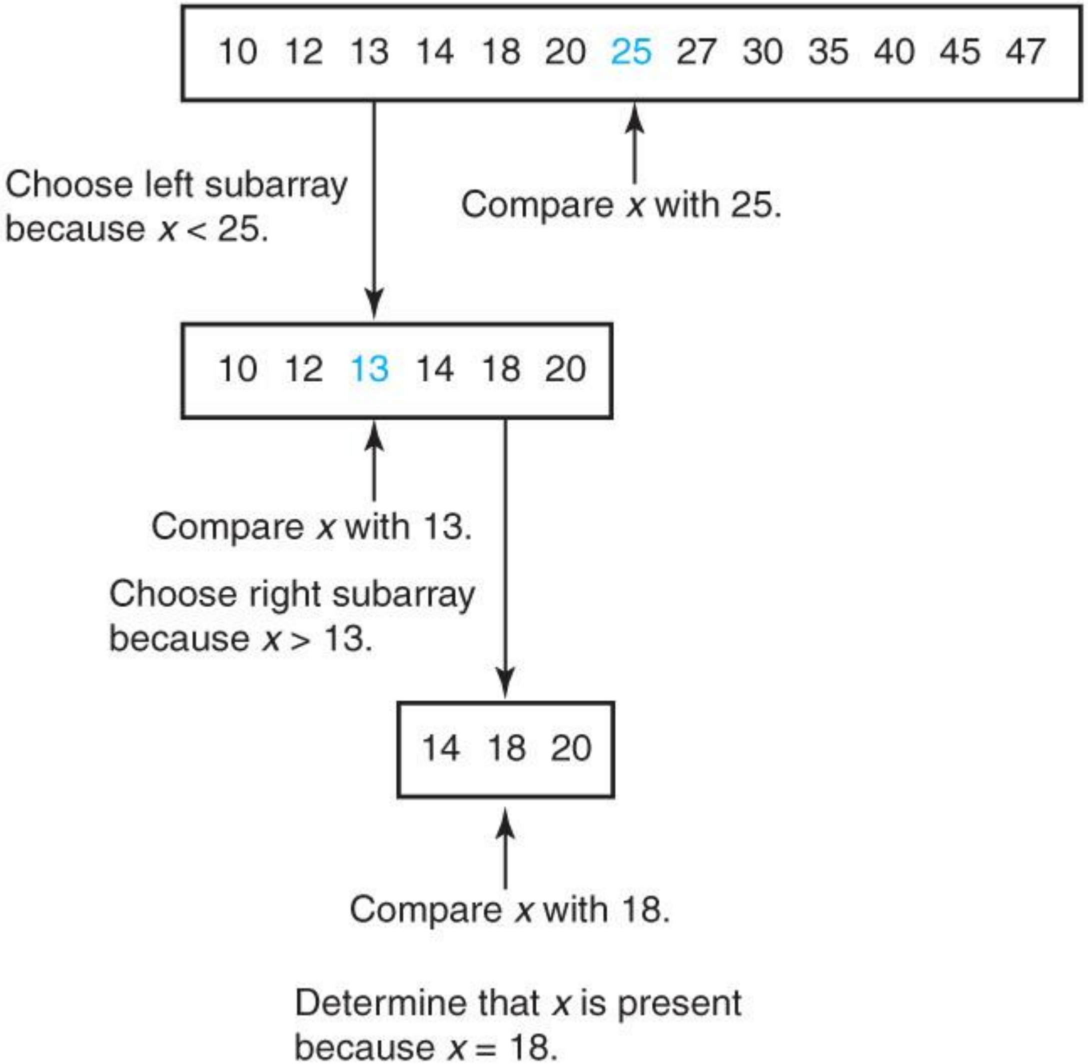
$g(n) = n^{\log_2 2}$

Since $f(n) = g(n) = n$

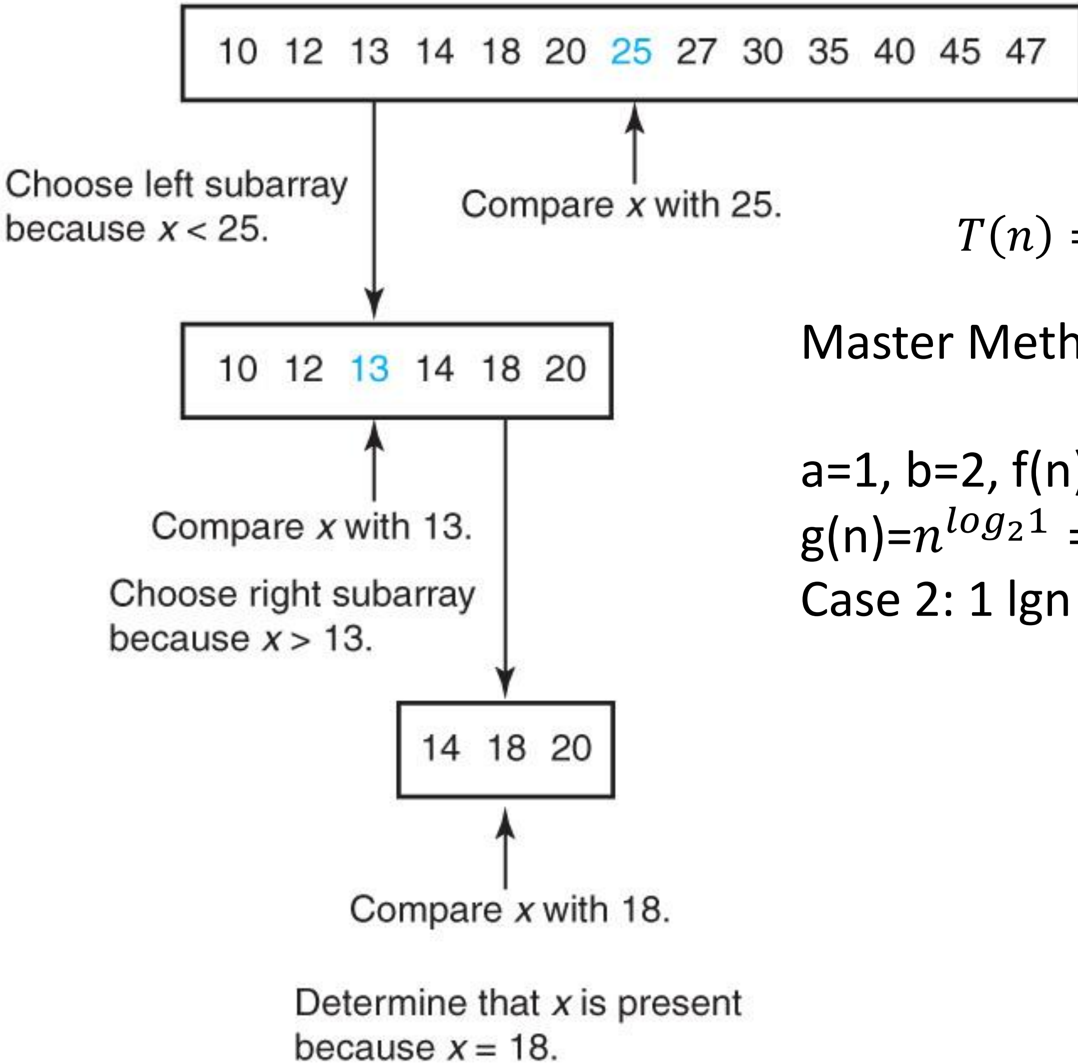
case 2

$n \times lg\, n$

Example: Binary Search for x=18



Example: Binary Search for x=18



$$T(n) = T(n/2) + O(1)$$

Master Method Case 2: O(lg(n))

a=1, b=2, f(n)=1

g(n)=$n^{log_2 1} = n^0 = 1$

Case 2: 1 lgn

Example: FAST Exponentiation

$$a^n = \begin{cases} a^{n/2}(a^{n/2}) & \text{if } n \text{ is even} \\ a^{n/2}(a^{n/2})(a) & \text{if } n \text{ is odd} \end{cases}$$

```
def power(a,n):
    if n==0: return 1
    answer = power(a,(int)(n/2))
    if n%2 == 0:
        return answer*answer
    else:
        return answer*answer*a
```

$f(n) = f(n/2) + O(1)$

Master Method Case 2: O(lg(n))

Note: It is important that we use the variable $answer$ twice instead of calling the function $power(a, n)$ twice.

Example: Fibonacci sequence

(Recursion) Recall that:

$$f_n = f_{n-1} + f_{n-2}, \qquad n \geq 2$$

$$f_0 = 0,\ f_1 = 1\ (initial\ condition)$$

```python
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

BASE CASE

$$T(n) = \Omega\left(\left(\frac{3}{2}\right)^n\right)$$

Example: Fibonacci sequence

(Iterative) Recall that:

$$f_n = f_{n-1} + f_{n-2}, \qquad n \geq 2$$

$$f_0 = 0, \; f_1 = 1 \; (initial\ condition)$$

```
fib_0 = 1
  fib_1 = 1

  fib_2 = 0
  for i in range(2,n):
     fib_2 = fib_0 + fib_1
     fib_0 = fib_1
     fib_1 = fib_2
  return fib_2
```

$T(n) = O(n)$

Agenda

- Divide & Conquer
- Backtracking
- Greedy Algorithm
- 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

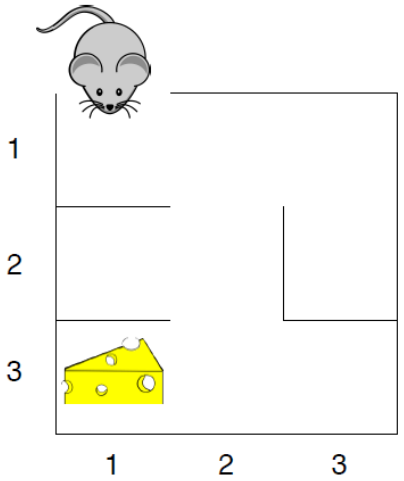
- Based on depth-first recursive search.
- Approach
  1. Tests whether solution has been found.
  2. If found solution, return it.
  3. Else, for each choice that can be made.
     a) Make that choice.
     b) Recur.
     c) If recursion gives a solution, return it.
  4. If no choices remain, return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

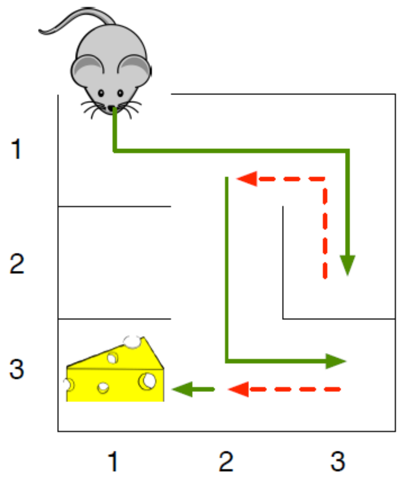
- Backtracking: systematic search technique to completely work through solution space.
- Prime example:
  How does the mouse find the cheese?



Backtracking

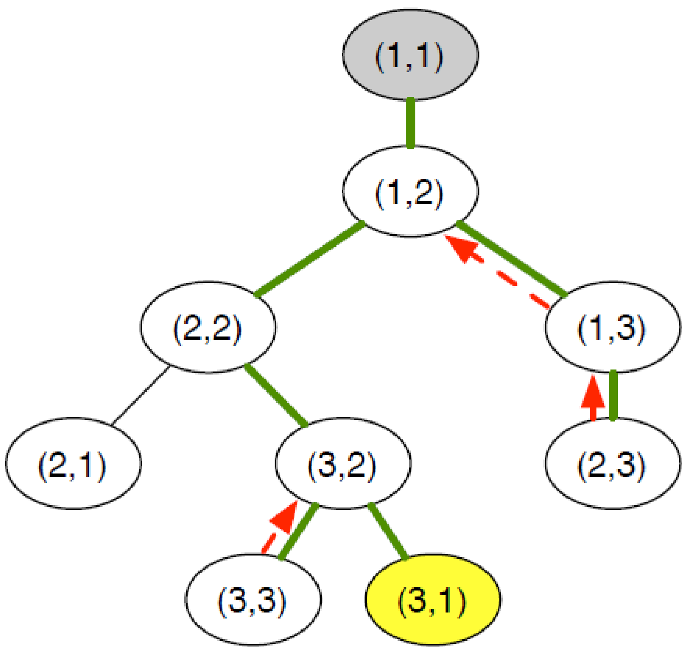
- Problem: How does the mouse find the cheese?
- Solution:
  - systematic exploration of the maze.
  - backtrack if meet deadend (hence backtracking).

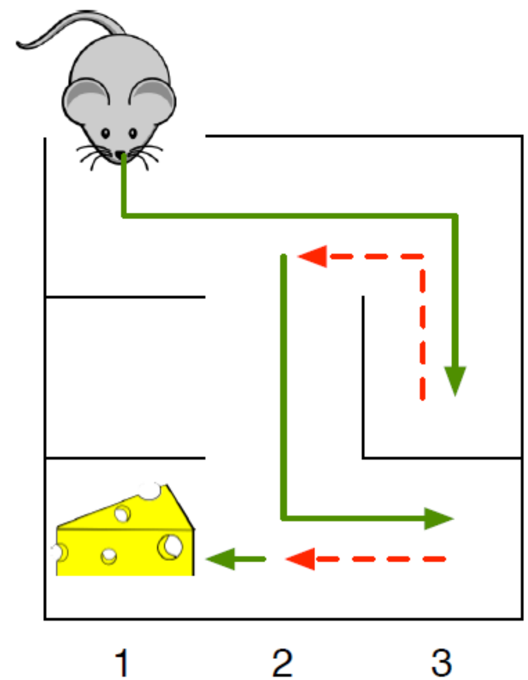
    → trial and error.



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

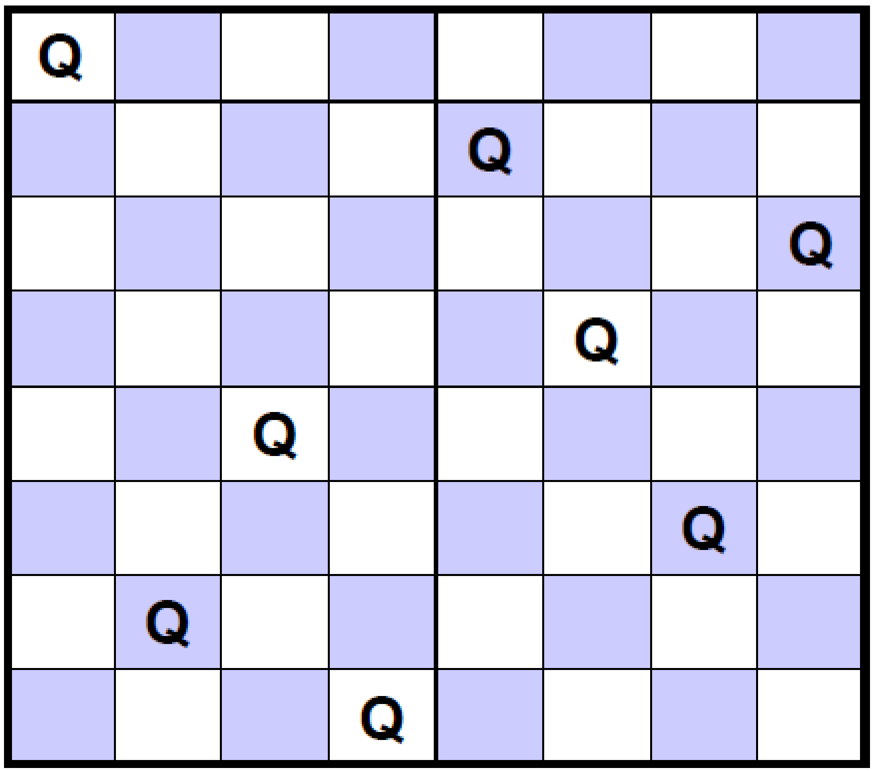
- Termination of backtracking:
  - only if solution space is finally exhausted.
  - only if it is ensured that no configurations remain to be tested.
- Complexity of backtracking:
  - directly dependent on the size of solution space.
  - usually exponential, thus $O(2^n)$ or worse!
  - can use for small 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

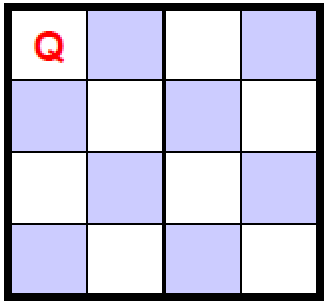
Sample solution for $n = 8$:



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

Recursive Strategy for n-Queens

row 0

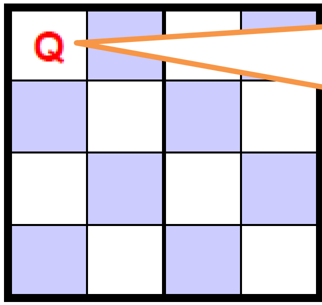


col 0: safe

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

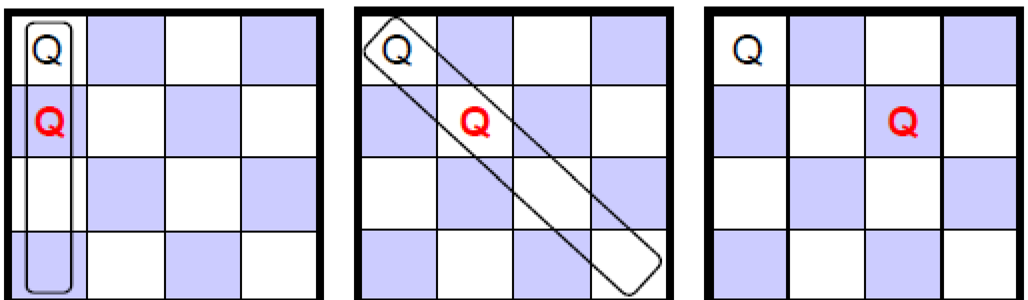
row 0



col 0: safe

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

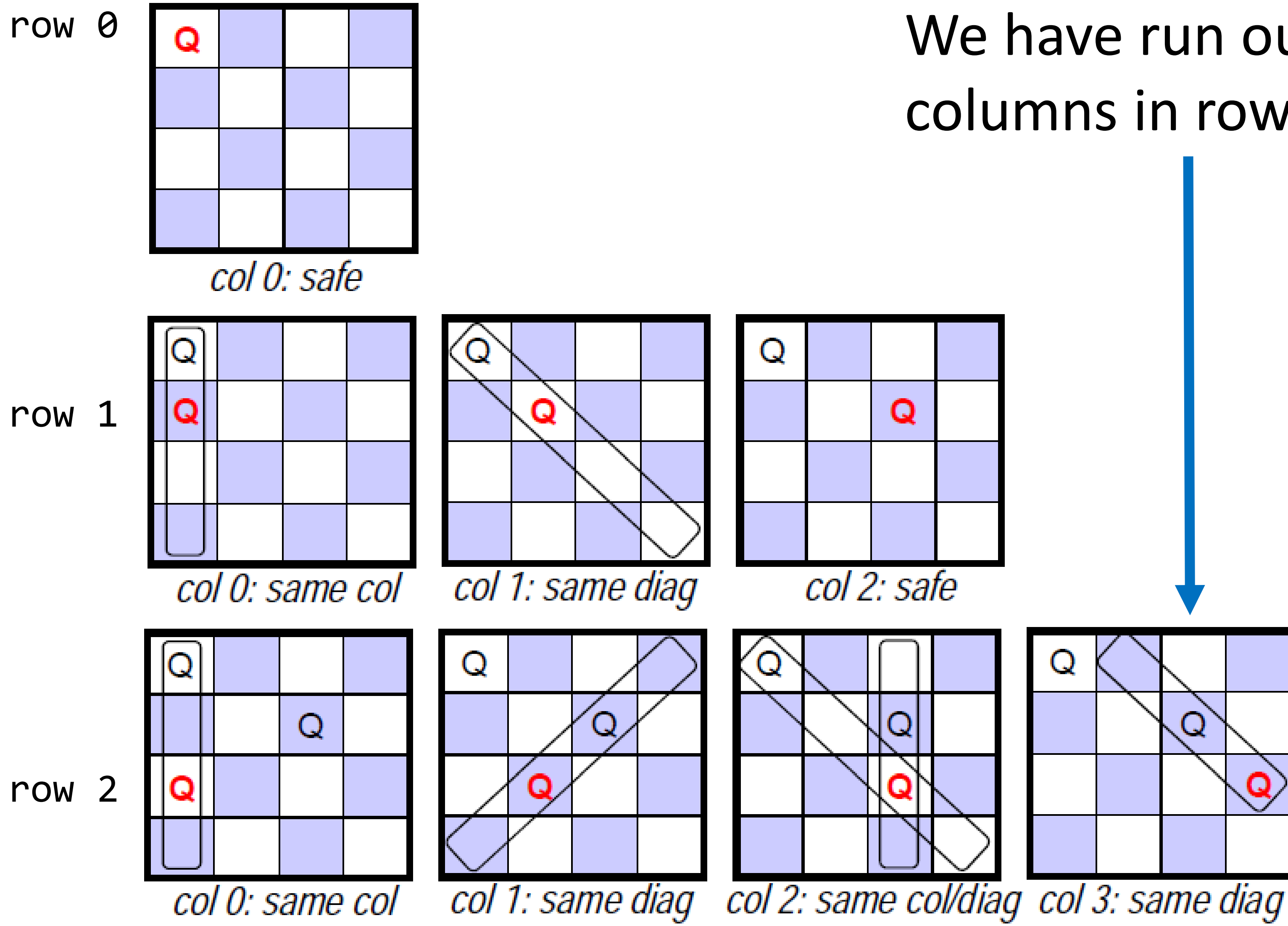
row 1



col 0: same col *col 1: same diag* *col 2: safe*

Recursive Strategy for n-Queens

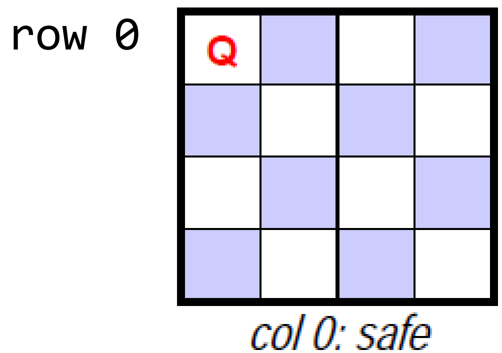
We have run out of columns in row 2!

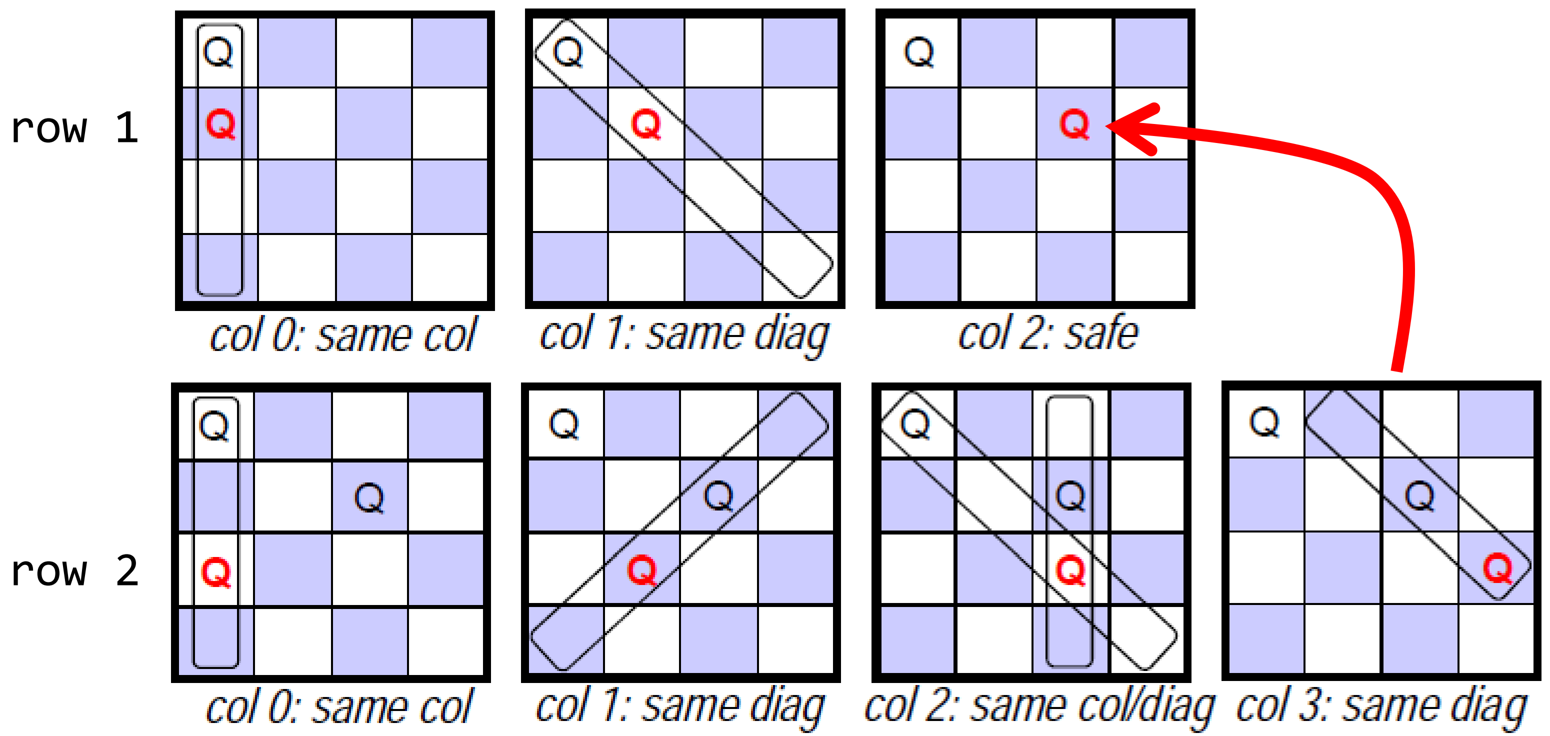


Recursive Strategy for n-Queens

`Backtrack` to row 1 by returning from the recursive call.

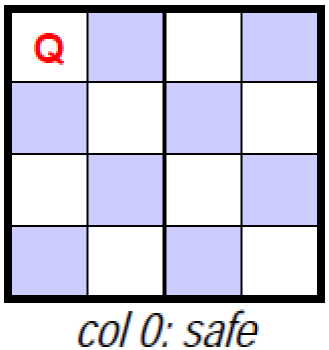
- pick up where we left off.
- we had already tried columns 0-2, so now we try column 3.





Recursive Strategy for n-Queens

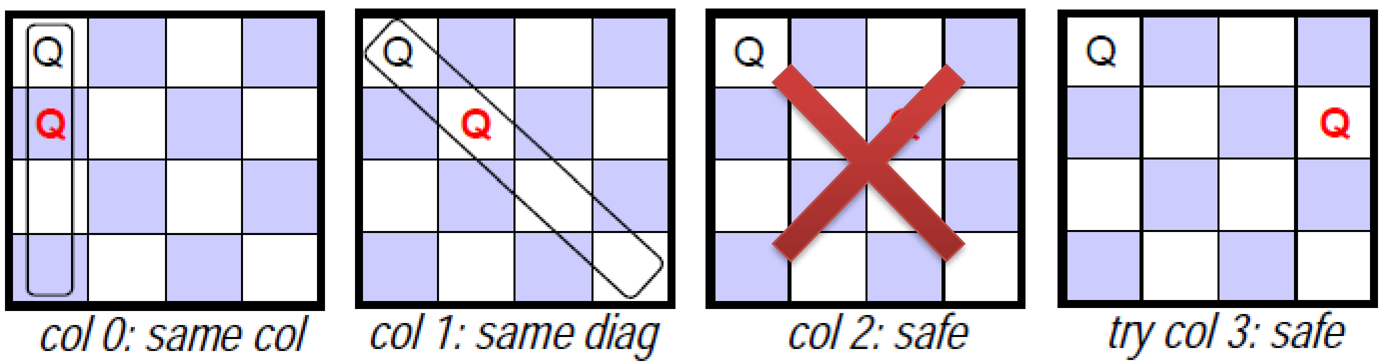
row 0



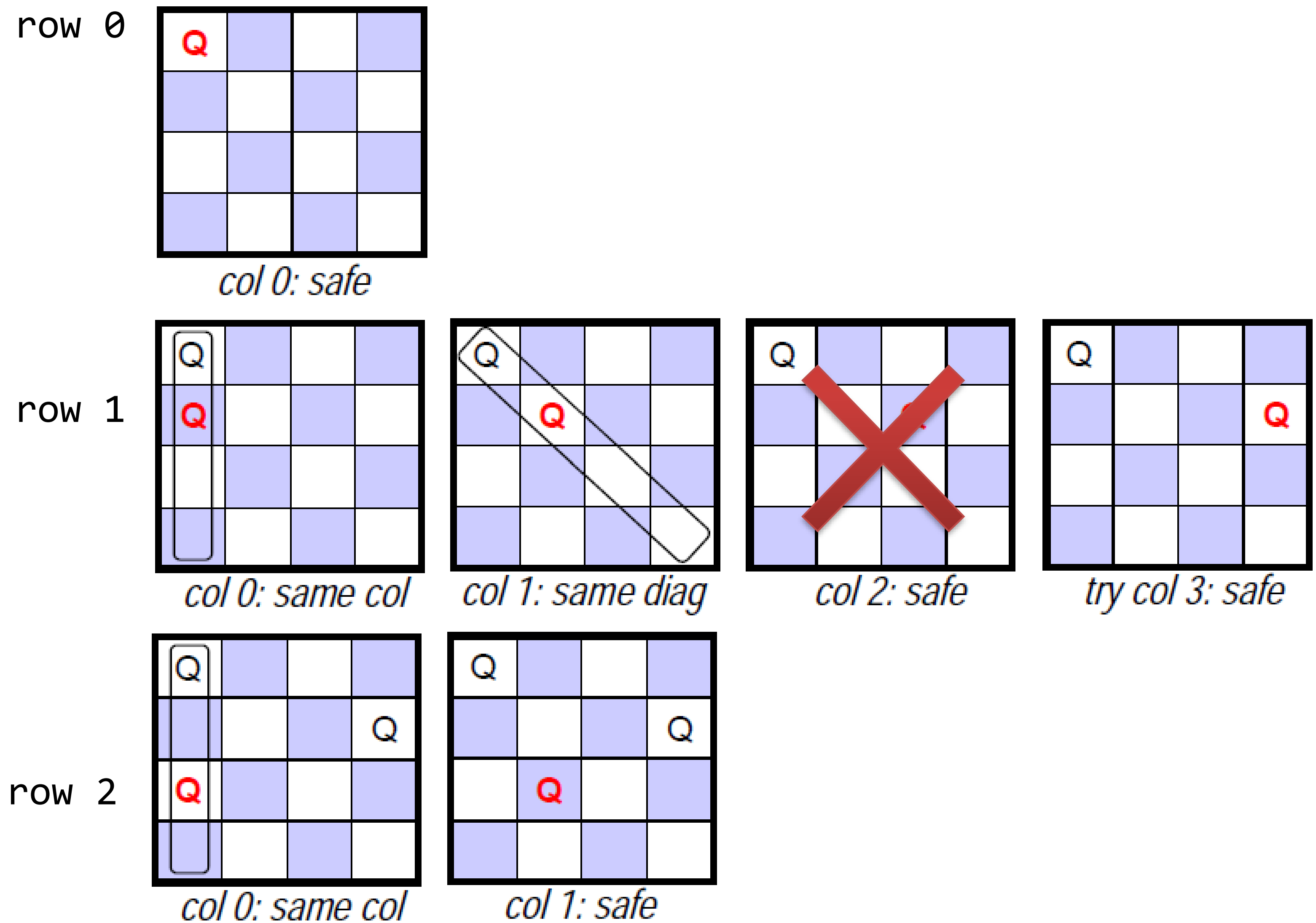
Backtrack to row 1 by returning from the recursive call.

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

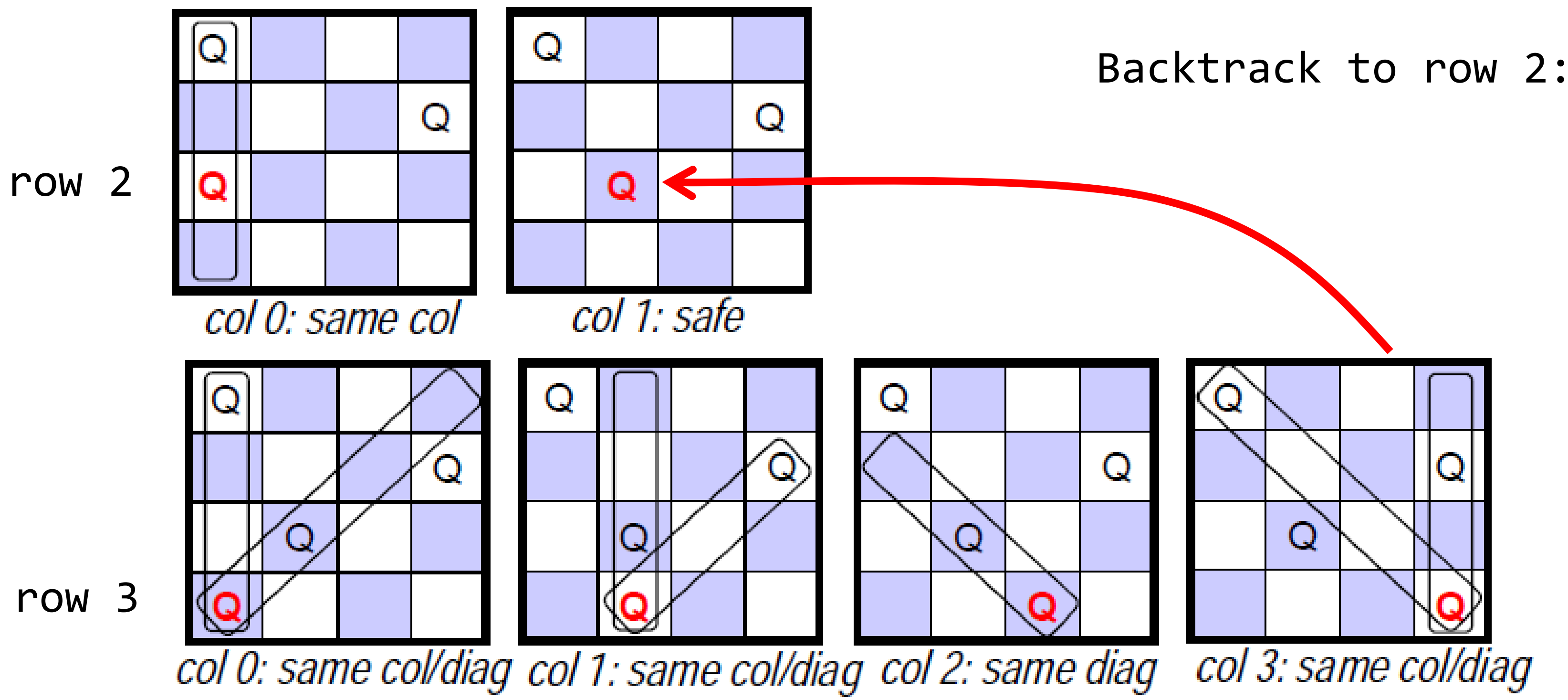
row 1



Recursive Strategy for n-Queens

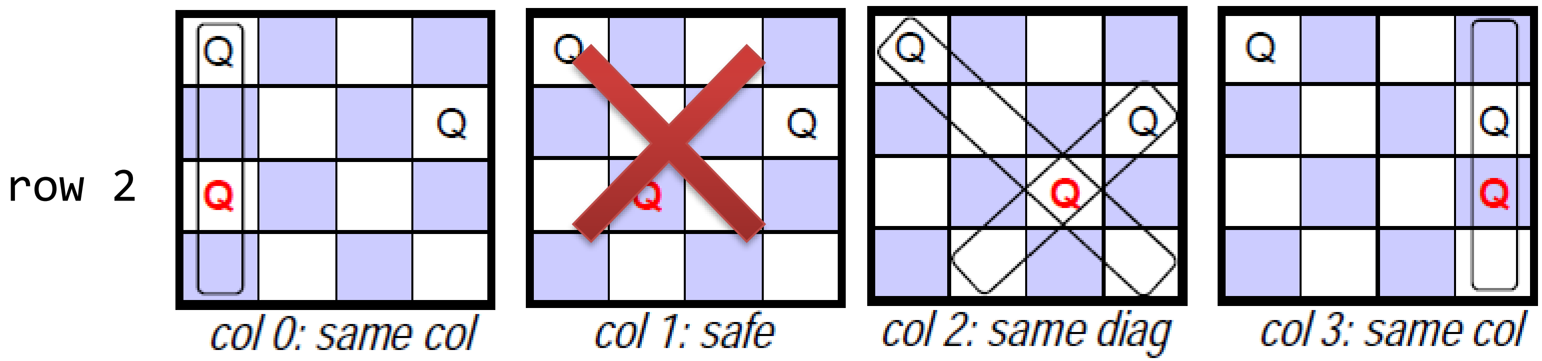


Recursive Strategy for n-Queens

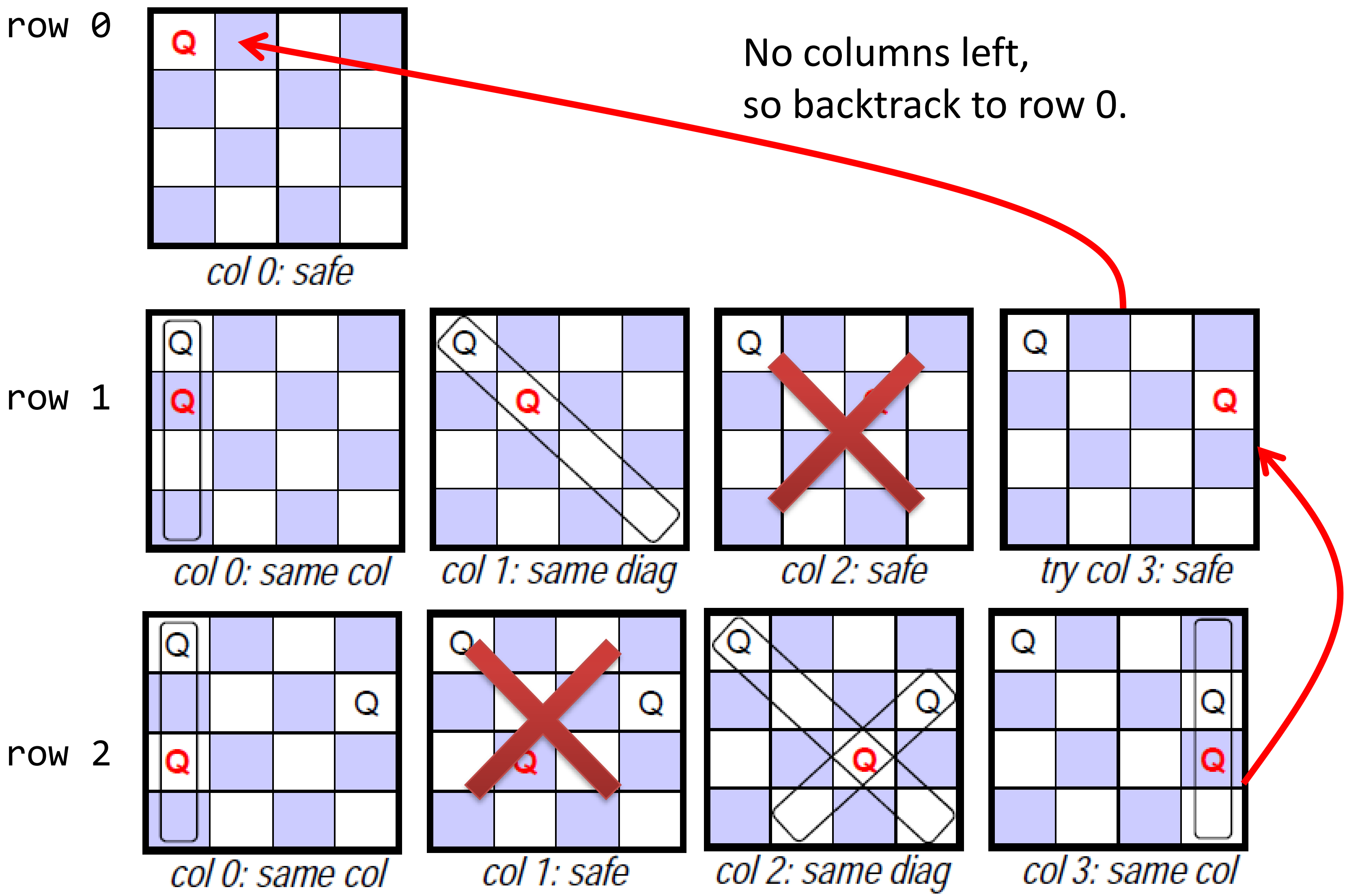


Recursive Strategy for n-Queens

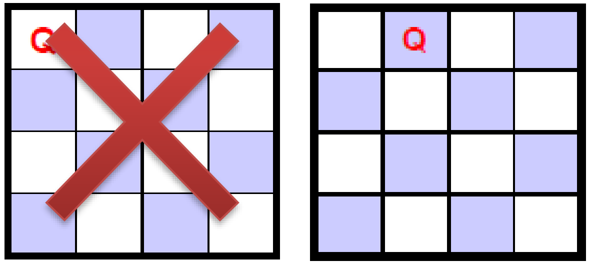
Backtrack to row 1.



Recursive Strategy for n-Queens

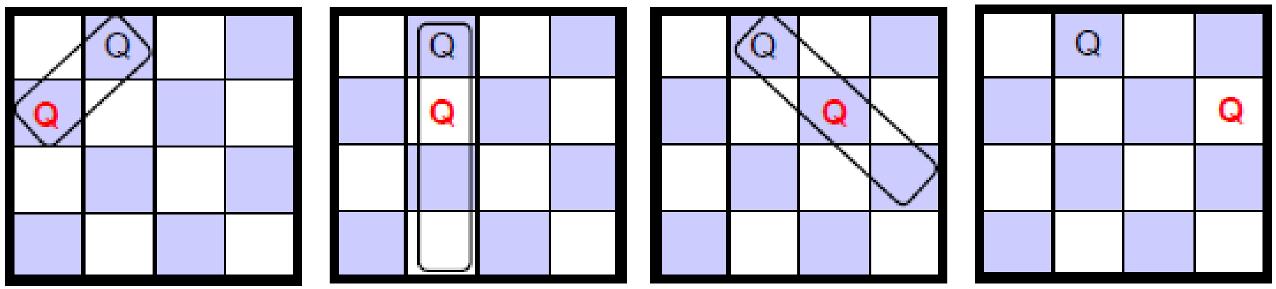


row 0

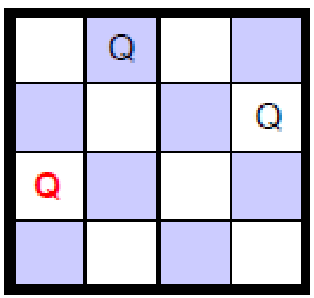


col 0: safe

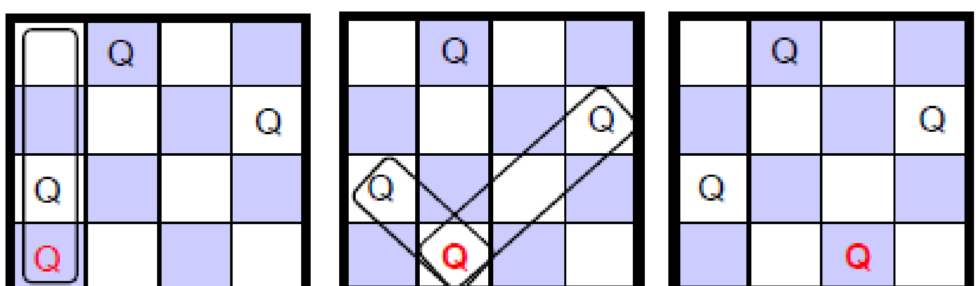
row 1



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

```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
```

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 the next row.

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
```

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

- Divide & Conquer
- Backtracking
- Greedy Algorithm
- Dynamic Programming

Optimization & Greedy Algorithms

- An optimization problem means to find *best* solution, not just *a* solution.
- A "greedy algorithm" sometimes works well for optimization problems.
- A greedy algorithm works in phases. At each phase:
  - take the best you can get right now, without regard for future consequences.
  - hope that choosing a *local* optimum at each step will end up at a *global* optimum.

Greedy = Optimal?

- Greedy algorithms
  do not always yield optimal solutions
  …although they do for many problems.
- Examples of Greedy Algorithms:
  – Dijkstra’s Shortest Path Algorithm.
  – Kruskal’s Minimum Spanning Tree Algorithm.
  – Prim’s Minimum Spanning Tree Algorithm.

Greedy Algorithm to Count Money

Suppose we want to gather an amount of money, using the fewest possible bills and coins.

- A greedy algorithm to do it:
  At each step, take the largest possible bill or coin that does not overshoot.
  eg. to form $6.39, we choose (for US$):
  - a $5 bill
  - a $1 bill, = $6
  - a 25¢ coin, = $6.25
  - a 10¢ coin, = $6.35
  - four 1¢ coins, = $6.39 ; total 8 pcs (bills & coins)
- For US money, the greedy algorithm always gives the optimal solution.

Failure of Greedy Algorithm

Suppose some foreign currency uses \$1, \$7, \$10 coins.

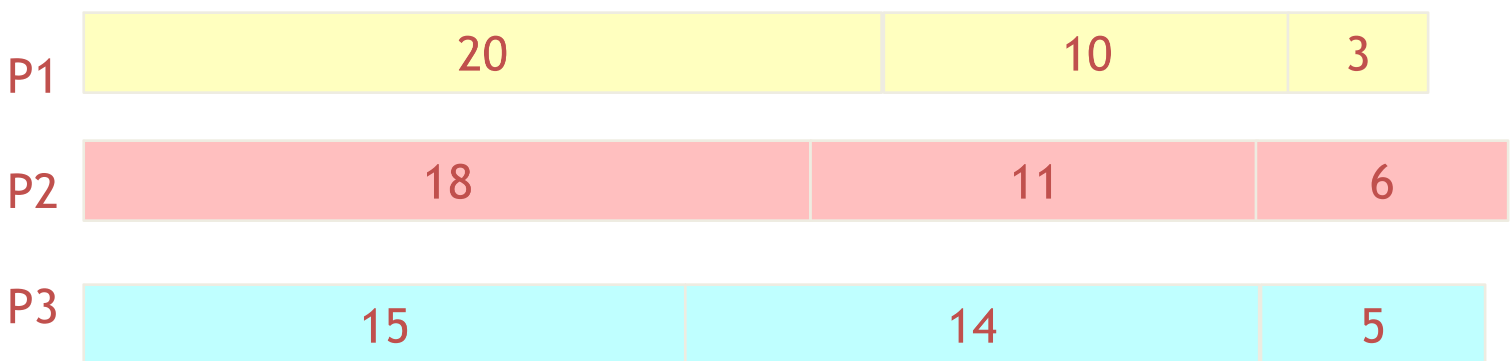
- A greedy algorithm to form \$15:
  one \$10 + five \$1 coins = 6 coins.
- A better solution:
  two \$7 + one \$1 = 3 coins.
- The greedy algorithm gives a solution,
  but not an optimal solution.

Greedy Algorithm for Scheduling Problem

Task: To execute nine jobs with these running times
3, 5, 6, 10, 11, 14, 15, 18, 20 minutes.

Resources: 3 processors to run the jobs.

- Approach 1: Do longest jobs first,
  on whatever processor is available.



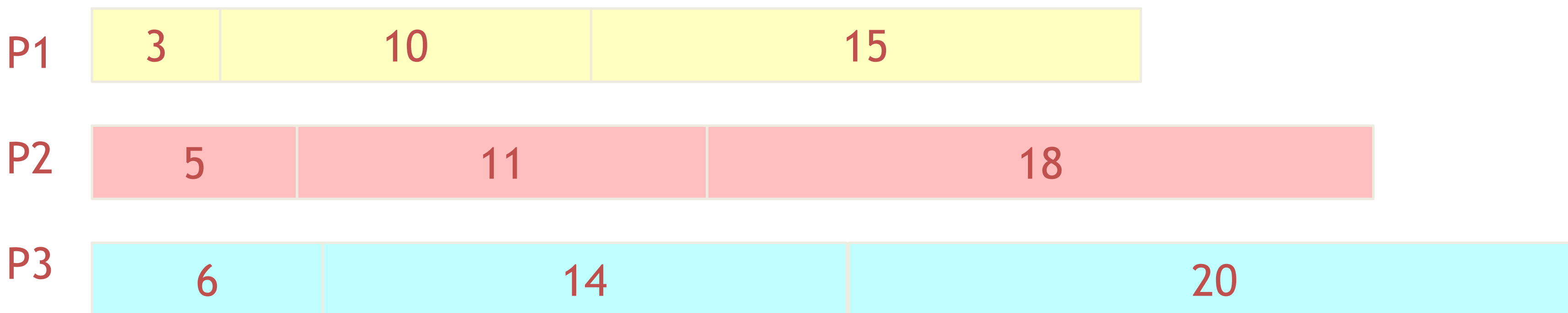
Time to completion: 18 + 11 + 6 = 35 minutes.

Is there a better solution?

Second Approach

- Approach 2: Do shortest jobs first.

(3, 5, 6, 10, 11, 14, 15, 18, 20 minutes)

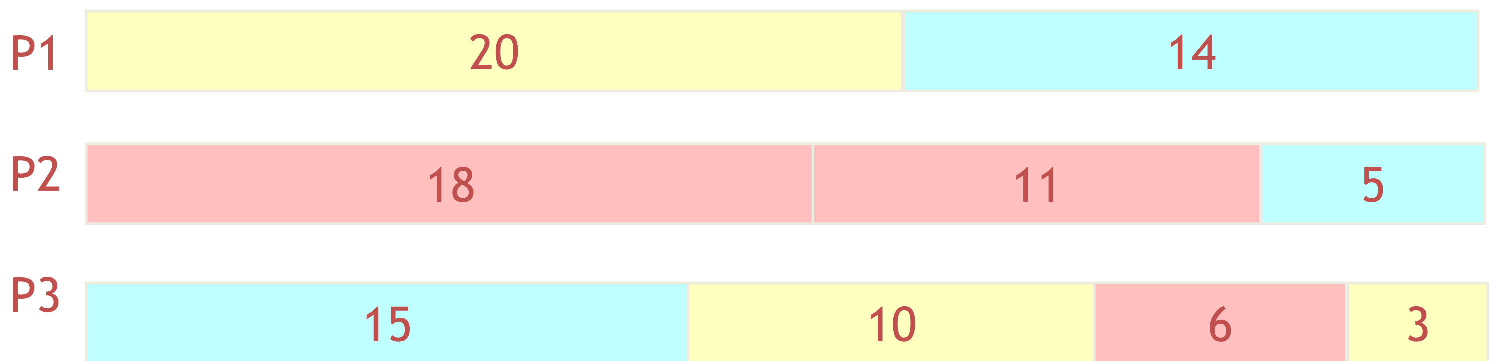


Not good; time needed is 6 + 14 + 20 = 40 minutes.

Note, however, that the greedy algorithm itself is fast; at each stage, just pick the minimum or maximum.

An Optimal Solution

- Better solutions do exist: (3, 5, 6, 10, 11, 14, 15, 18, 20 minutes)



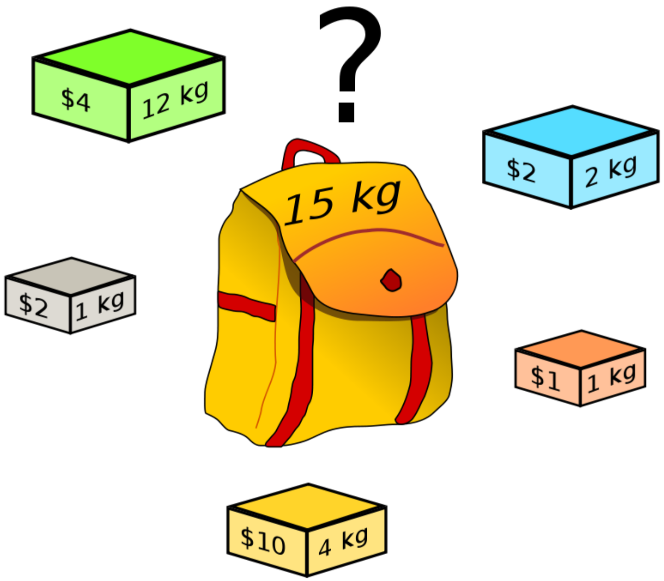
- This solution is clearly optimal. (34 mins)
- Clearly, there are other optimal solutions. max(18+15, 20+10+3, 14+11+6)=33 mins
- How do we find such a solution?
  - One way: Try all possible assignments of jobs to processors.
  - Unfortunately, this approach can take exponential time.

Knapsack Problem

- In the knapsack problem, imagine that you are the Professor in the Netflix series Bank Heist.
- After escaping the mint, you start loading bags of money into your getaway truck.
- The truck only can hold x kg of cash.
- Your bags of cash come in various weights and value.
- Your task is to maximize the value you can drive away with.

Knapsack Problem

Item	Value	Weight
0	4	5
1	3	3
2	10	5



Knapsack Problem

Item	Value	Weight
0	4	5
1	3	3
2	10	5

- There are two versions of the knapsack problem.
- The first version is a 0-1 version (take or don't take – e.g. if max 10kg takes item 2 and 0 to reach 10kg)
- The second is the fractional knapsack problem. (allow fraction of the item e.g. if max is 12kg takes item 2,0 and 2 kg of item 1)

Fractional Knapsack Problem

Item	Value	Weight	Value / Weight
0	4	5	4/5
1	3	3	1
2	10	5	2

- Eg if the truck can take 10kg.
- The greedy algorithm would give the optimal solution.
- Greedy solution is 5kg of item 2, 3kg of item 1 and 2kg of item 0 (allow fraction).

0-1 Knapsack Problem

Item	Value	Weight	Value / Weight
0	4	5	4/5
1	3	3	1
2	10	5	2

- Eg if the truck can take 10kg.
- The greedy algorithm would not give the optimal solution.
- The greedy solution will be item 2 and 1 (but can't reach 10kg)
- The optimal solution would be Item 2 and 0.

Agenda

- Divide & Conquer
- Backtracking
- Greedy Algorithm
- Dynamic Programming

Hallmarks of DP

- Hallmark 1:
  - Optimal substructure An optimal solution to a problem (instance) contains optimal solutions to subproblems.

- Hallmark 2:
  - Overlapping subproblems A recursive solution contains a “small” number of distinct subproblems repeated many times.

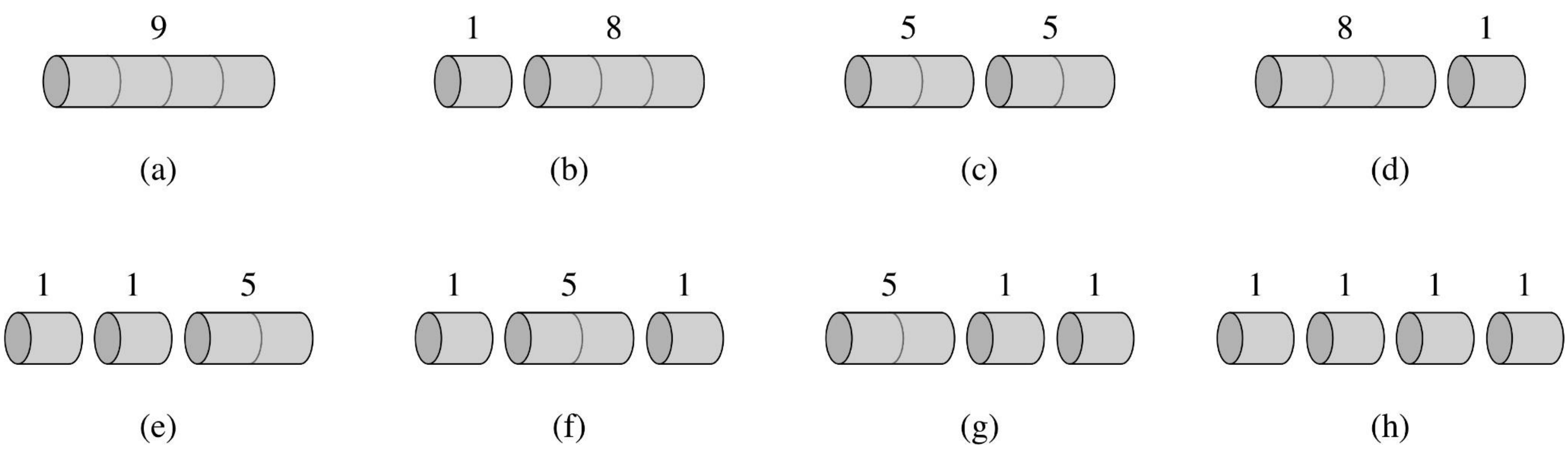
Dynamic Programming: Rod-cutting Problem

- 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

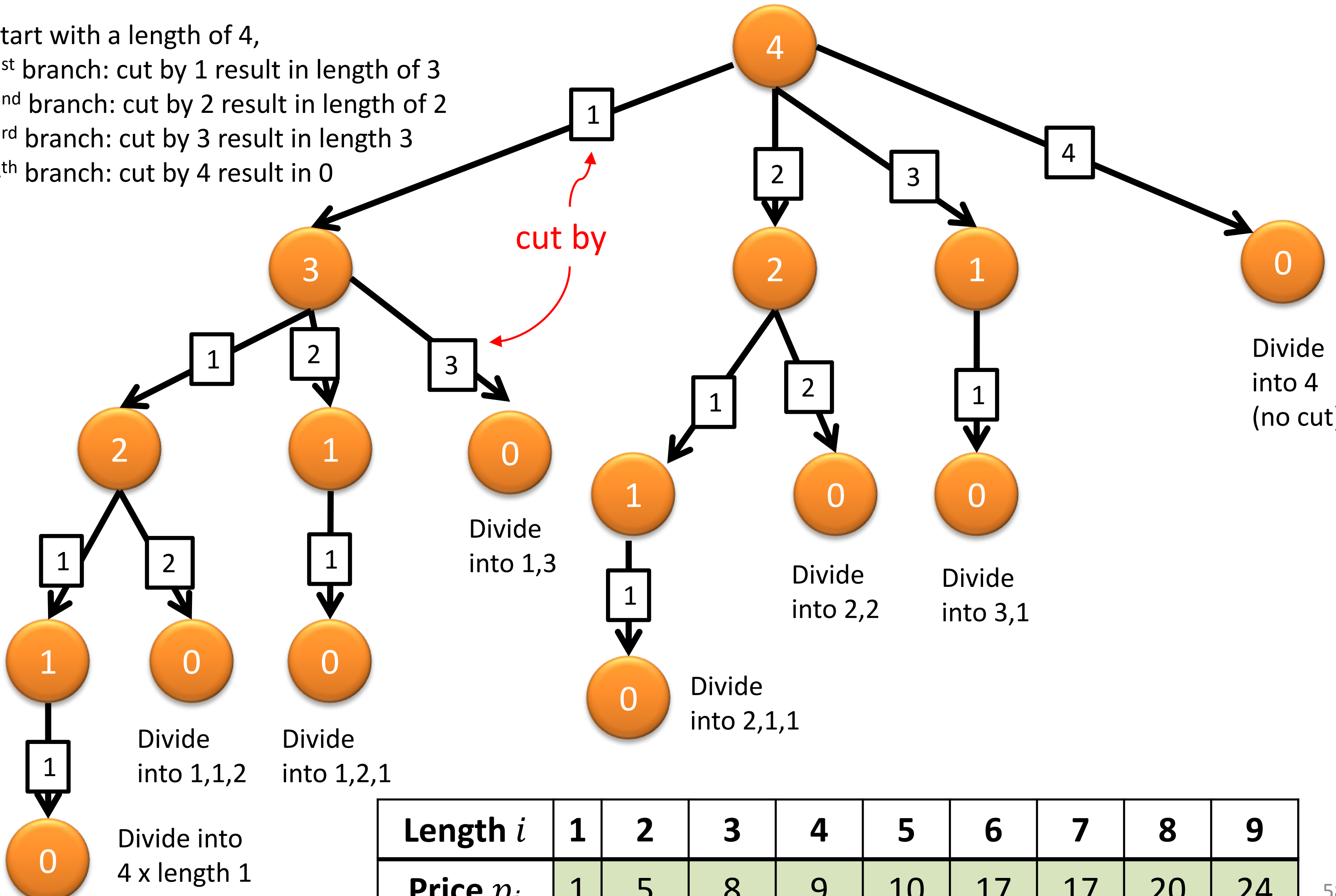
- For a rod of length n, there are 2^{n-1} ways to cut.
- Example, when $n = 4$ (rod is of length 4),
  there are 8 possible ways to cut the rod (no cut 4m=$9)



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

Start with a length of 4,
1st branch: cut by 1 result in length of 3
2nd branch: cut by 2 result in length of 2
3rd branch: cut by 3 result in length 3
4th branch: cut by 4 result in 0

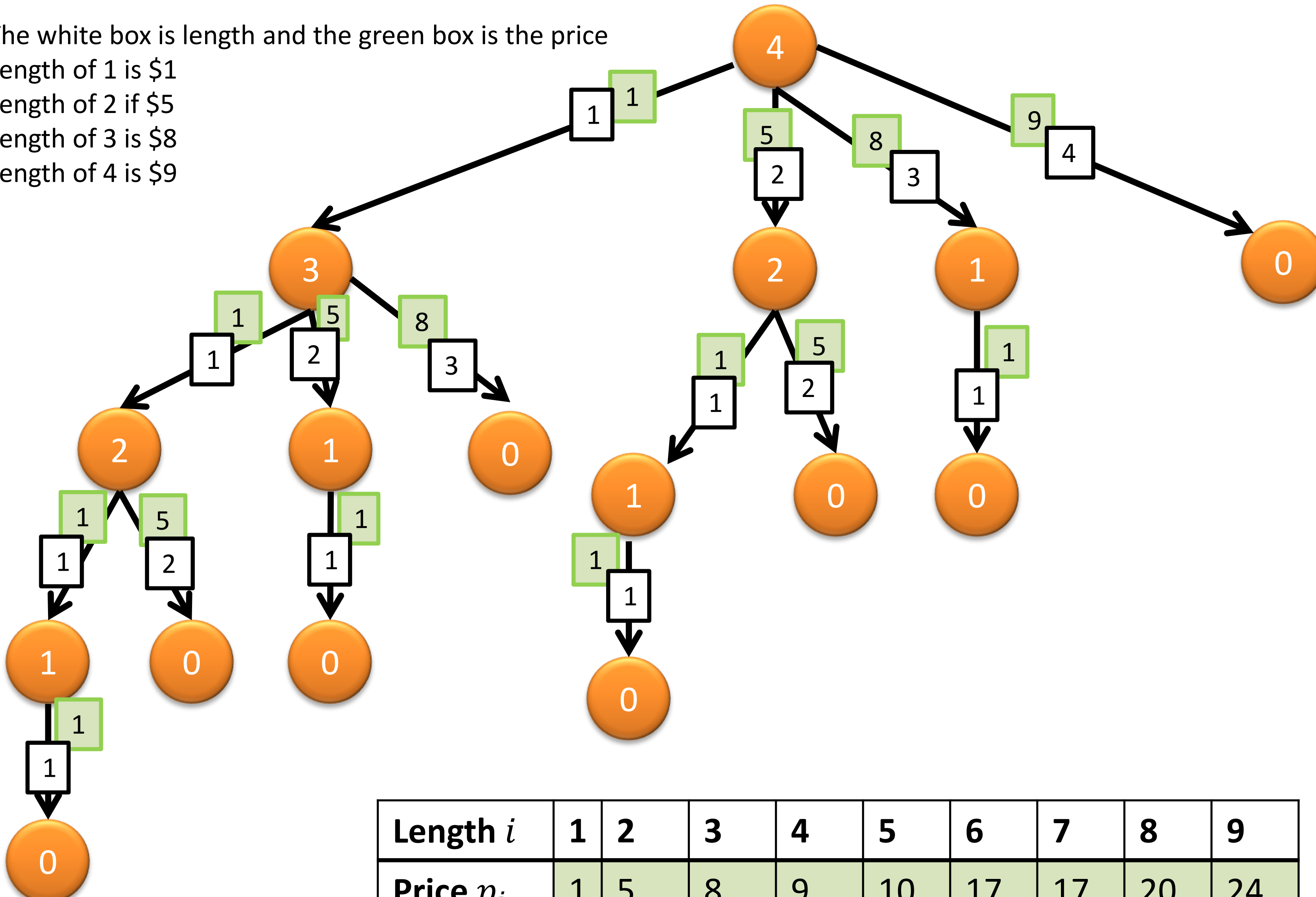


Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

Rod-cutting Problem: Divide and Conquer solution

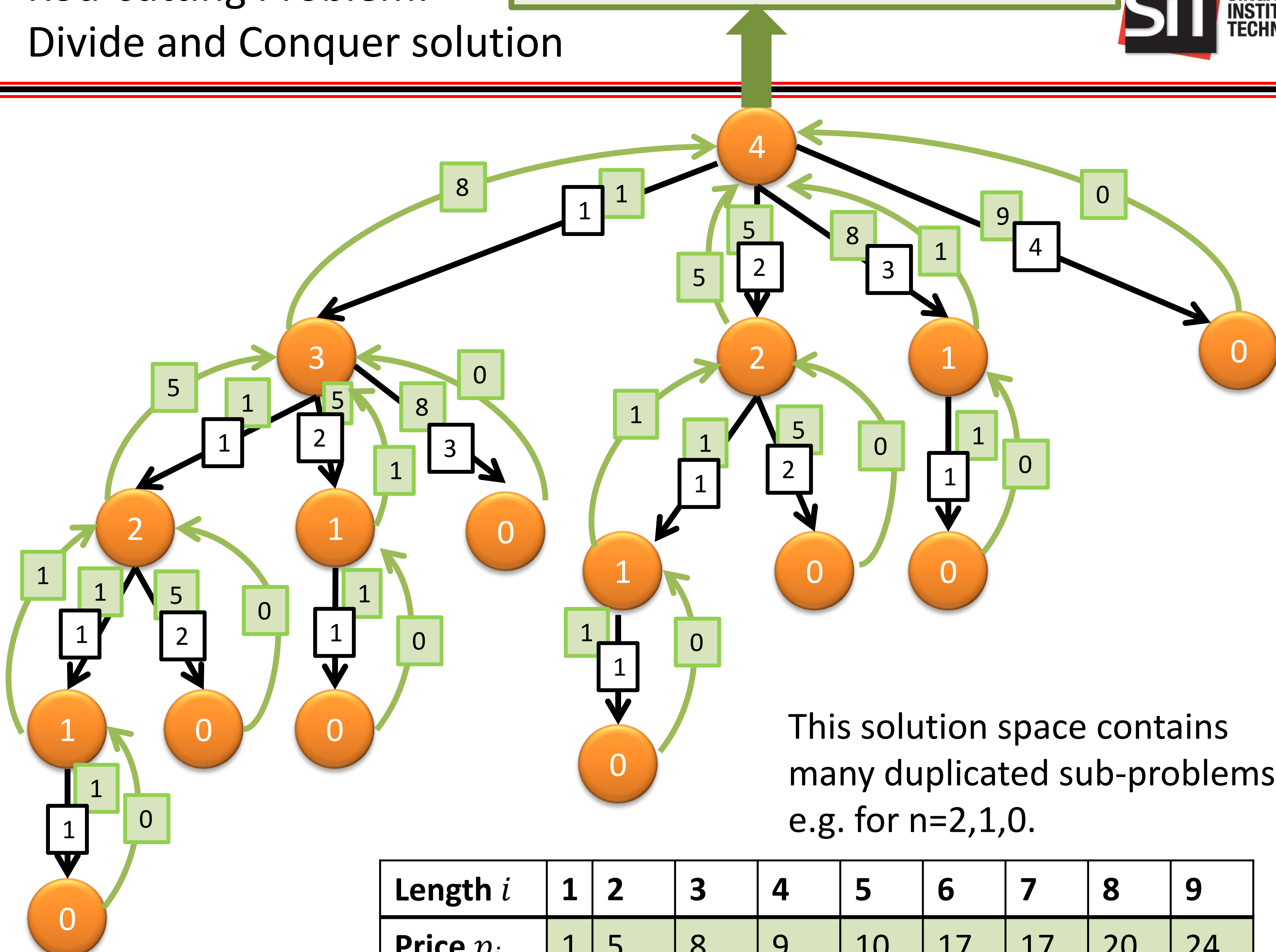
SINGAPORE INSTITUTE OF TECHNOLOGY

The white box is length and the green box is the price
Length of 1 is $1
Length of 2 if $5
Length of 3 is $8
Length of 4 is $9



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



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

```c
#include "stdio.h"
int price[] = { 0,1,5,8,9,10,17,17,20,24 };
int cnt = 1;

int rc(int n)
{   int i, max, temp;
    if (n == 0) return 0;
    max = 0;
    for (i = 1; i <= n; i++)
    {   temp = price[i] + rc(n - i);
        if (temp > max) max = temp;
    }
    printf("try=%d with n=%d and max=%d\n", cnt, n, max);
    cnt++;
    return max;
}

void main()
{   printf("start with n=4\n");
    printf("final %d\n", rc(4));
}
```

Microsoft Visual Studio Debug Con

```
start with n=4
try=1 with n=1 and max=1
try=2 with n=2 and max=5
try=3 with n=1 and max=1
try=4 with n=3 and max=8
try=5 with n=1 and max=1
try=6 with n=2 and max=5
try=7 with n=1 and max=1
try=8 with n=4 and max=10
final 10
```

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).

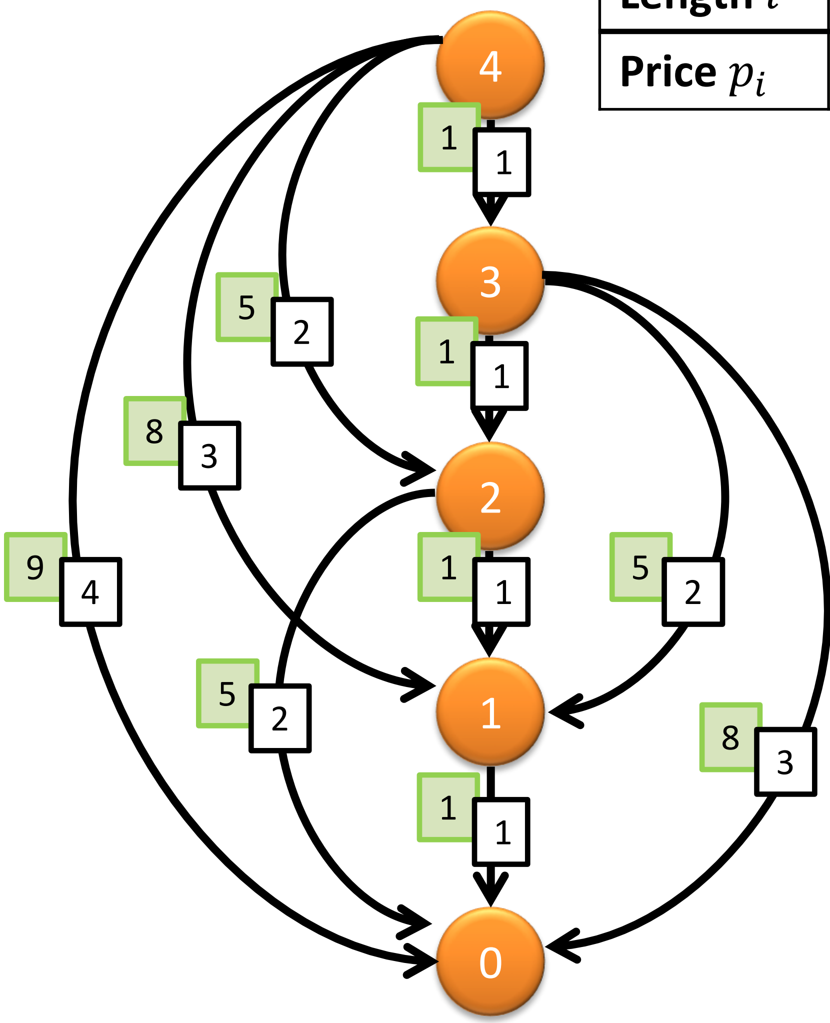
Dynamic Programming

- 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.

SIT Internal

Rod-cutting Problem: Dynamic Programming

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.

Rod-cutting Problem: Dynamic Programming

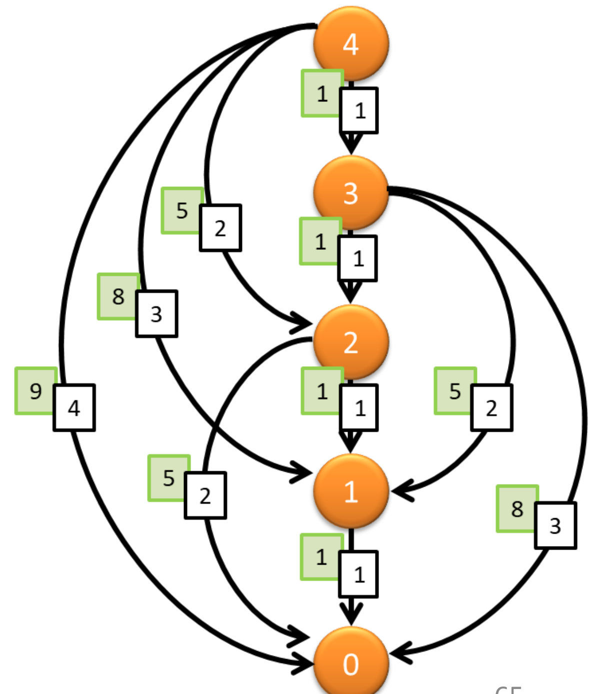
Len	1	2	3	4
Optimal Price	1	5	8	10

Len = 1
Optimal Price = 1

Len = 2
Optimal Price = max (1 + 1, 5)

Len = 3
Optimal Price = max (1 + 5, 5 + 1, 8)

Len = 4
Optimal Price = max (1 + 8, 5 + 5, 8 + 1, 9)



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: Dynamic Programming

```c
#include "stdio.h"
int price[] = { 0,1,5,8,9,10,17,17,20,24 };
int maxcost[] = { 0,-1,-1,-1,-1,-1,-1,-1,-1,-1 };
int cnt = 1;

int rc(int n)
{   int i, temp;

    if (maxcost[n] < 0)
    {   for (i = 1; i <= n; i++)
        {   temp = price[i] + rc(n - i);
            if (temp > maxcost[n]) maxcost[n] = temp;
        }
        printf("try=%d with n=%d and max=%d\n", cnt, n, maxcost[n]);
        cnt++;
    }
    return maxcost[n];
}
```

Microsoft Visual Studio Debug Cor

```
start with n=4
try=1 with n=1 and max=1
try=2 with n=2 and max=5
try=3 with n=3 and max=8
try=4 with n=4 and max=10
final 10
maxcost[0]=0
maxcost[1]=1
maxcost[2]=5
maxcost[3]=8
maxcost[4]=10
maxcost[5]=-1
maxcost[6]=-1
maxcost[7]=-1
maxcost[8]=-1
maxcost[9]=-1
```

```c
void main()
{   int i,n;
    printf("start with n=");
    scanf("%d", &n);
    printf("final %d\n", rc(n));
    for (i = 0; i < 10; i++)
        printf("maxcost[%d]=%d\n", i,maxcost[i]);
}
```