SAMSUNG

Samsung Innovation Campus

Coding and Programming

Together for Tomorrow!
Enabling People
Education for Future Generations

Chapter 6.

Algorithm 3 - Problem Solving with Algorithms

Coding and Programming

Chapter Description

Chapter objectives

- ✓ Learners will be able to solve various problems using algorithms. Learners will be able to understand algorithmic design techniques such as greedy approach, divide-and-conquer method, dynamic programming method, and backtracking, and will be able to apply them to actual problems in order to solve them.

Chapter contents

Unit 30.

Greedy Approach

Unit learning objective (1/3)

Learning objectives

- ✓ Learners can understand the greedy approach and solve given problems using the greedy approach.
- ✓ Learners can understand that problems such as coin exchange problems and activity selection problems can be solved by the greedy approach.
- ✓ Learners can design and implement greedy algorithms recursively.

Unit learning objective (2/3)

Learning overview

- ✓ Learn how to solve algorithmic problems using the greedy approach.
- ✓ Learn how to solve the coin exchange problem using the greedy approach.
- ✓ Learn how to solve the activity selection problem using the greedy approach.

Concepts you will need to know from previous units

- ✓ Be able to understand and implement the definition of a recursive function and its termination conditions.
- ✓ Be able to use a sorting algorithm to sort a given set of data sets.
- ✓ Be able to analyze the performance of recursive algorithms.

Unit learning objective (3/3)

Keywords

Greedy Algorithm

Activity Selection Problem

Coin Change Problem

Recursion

Iteration

Mission

1. Real world problem

1.1. How Many Meetings are Possible?

- Suppose there is a company with only one conference room.
- There are several teams in the company, and they are planning to hold several meetings in this conference room.
- Since the two teams cannot use the same conference room at the same time, the conference room manager tries to allocate the conference room so that a maximum number of meetings is held while keeping the conditions that the meetings do not overlap.
- Is it possible to know the maximum number of possible meetings when we know the start time and the end time for N number of conference room usage requests?
- This problem is called Activity Selection Problem.
- Let's create an algorithm to solve the problem of assigning conference rooms.

2. Mission

2.1. Activity Selection Problem

- Suppose that N=7 conference room usage requests were given as follows.
- Each meeting room request was given a start time and an end time.
- What is the maximum number of meetings that do not overlap here?

id	0	1	2	3	4	5	6
start	1	3	2	1	5	8	5
finish	2	4	5	6	6	9	9

3. Solution

3.1. How the Activity Selection works

- The final selectable meeting is [0, 1, 4, 5], and the maximum number of selectable meetings is 4.

```
start = [1, 3, 2, 0, 5, 8, 5]
finish = [2, 4, 5, 6, 6, 9, 9]
meetings = activity_selection1(start, finish)
maximum = len(meetings)
print(meetings, maximum)
```

```
[0, 1, 4, 5] 4
```

3. Solution

3.2. Activity Selection Final Code

```
def activity_selection1(start, finish):
    result = []
    i = 0
    result.append(i)
    for j in range(1, len(start)):
        if finish[i] <= start[j]:
            result.append(j)
            i = j
    return result
```

Key concept

1. Greedy Approach

1.1. What is the Greedy Approach?

- The greedy approach is an algorithm design technique that arrives a solution by selecting a “best” choice at the moment from a sequence of choices according to predefined criteria.
- It is a very efficient and simple algorithmic design techniques for solving optimization problems, such as coin change problem and activity selection problem.

Focus The greedy approach solves the problem in the following three steps.

- Selection Process: Choose the next element to add to the set.
- Feasibility Check: Check whether a new set is appropriate to be the answer.
- Solution Check: Determine if a new set is the answer to the problem.

1. Greedy Approach

1.2. Coin Change Problem

- Using the greedy method, let's solve a coin change problem as follows.
  - There are four types of coins in Korea, as shown below.
  - The amount of each coin is 500 won, 100 won, 50 won, and 10 won.
- When distributing changes to coins, we try to give the smallest number of coins.
  - You can suppose that the number of coins is sufficiently prepared so that there is no shortage.

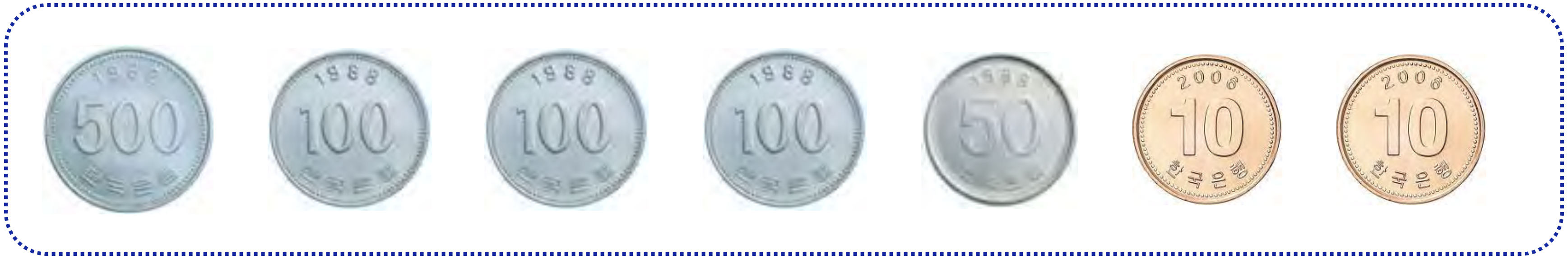
1. Greedy Approach

1.2. Coin Change Problem

- For example, suppose the change that needs to be returned is 870 won.
- You can make the change with 87 coins worth 10 won. However, in this case, there are too many number of coins for the change.
- How can one return 870 won by minimizing the number of coins as shown below?

₩870

change

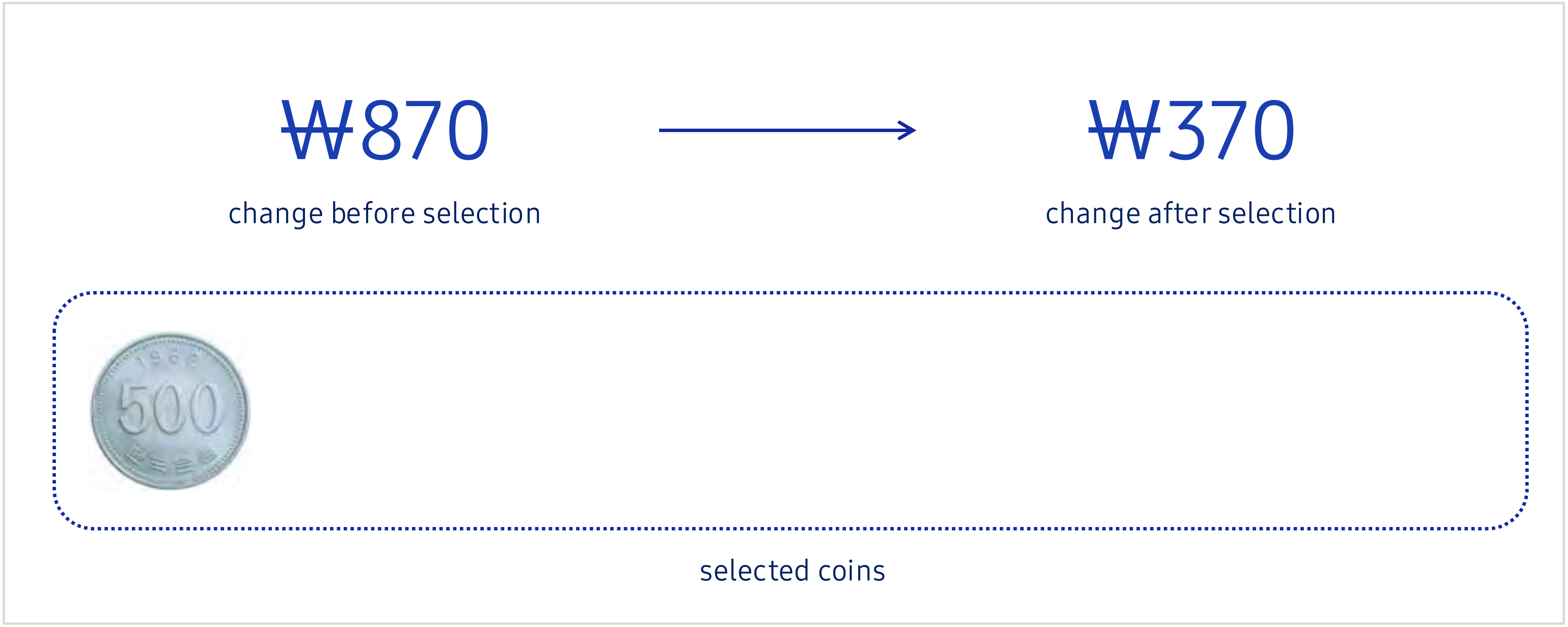


selected coins

1. Greedy Approach

1.2. Coin Change Problem

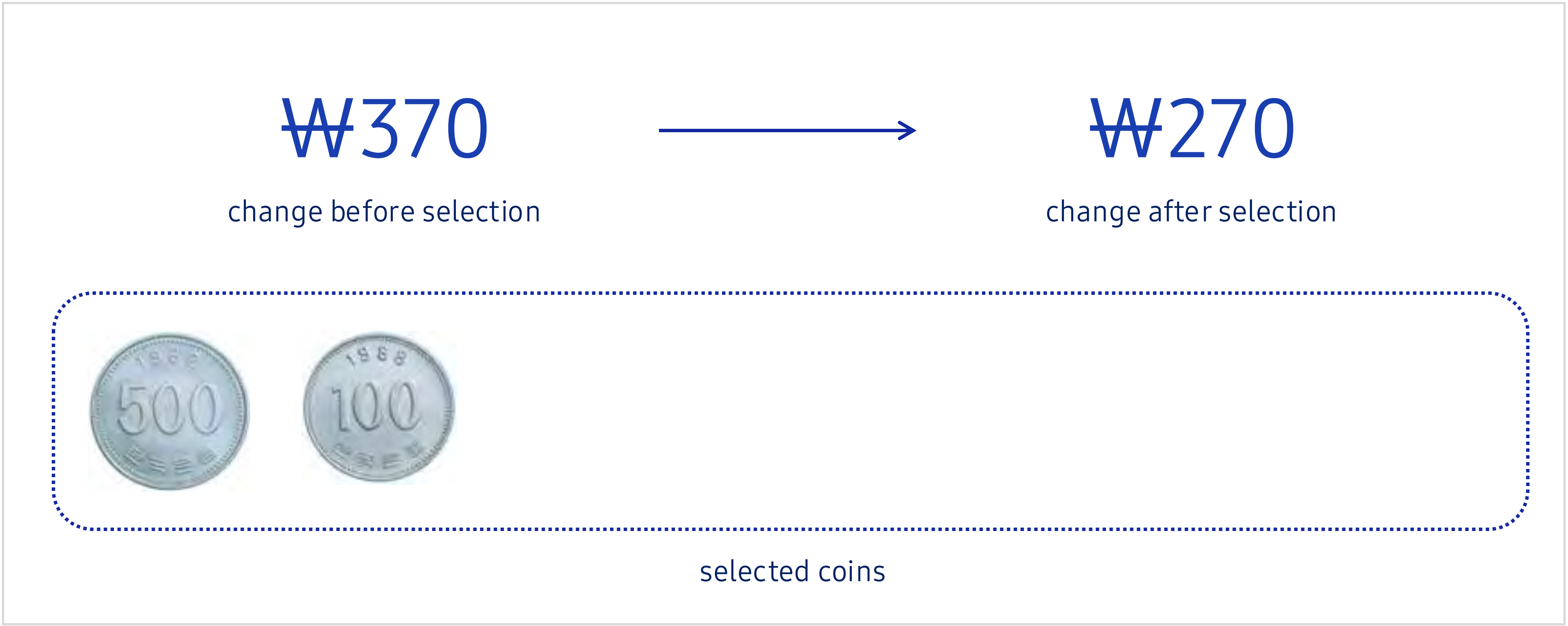
- First of all, let’s choose 500 won, the largest coin. There the remaining change is 370 won.
- Therefore, 500 won coins can no longer be chosen.



1. Greedy Approach

1.2. Coin Change Problem

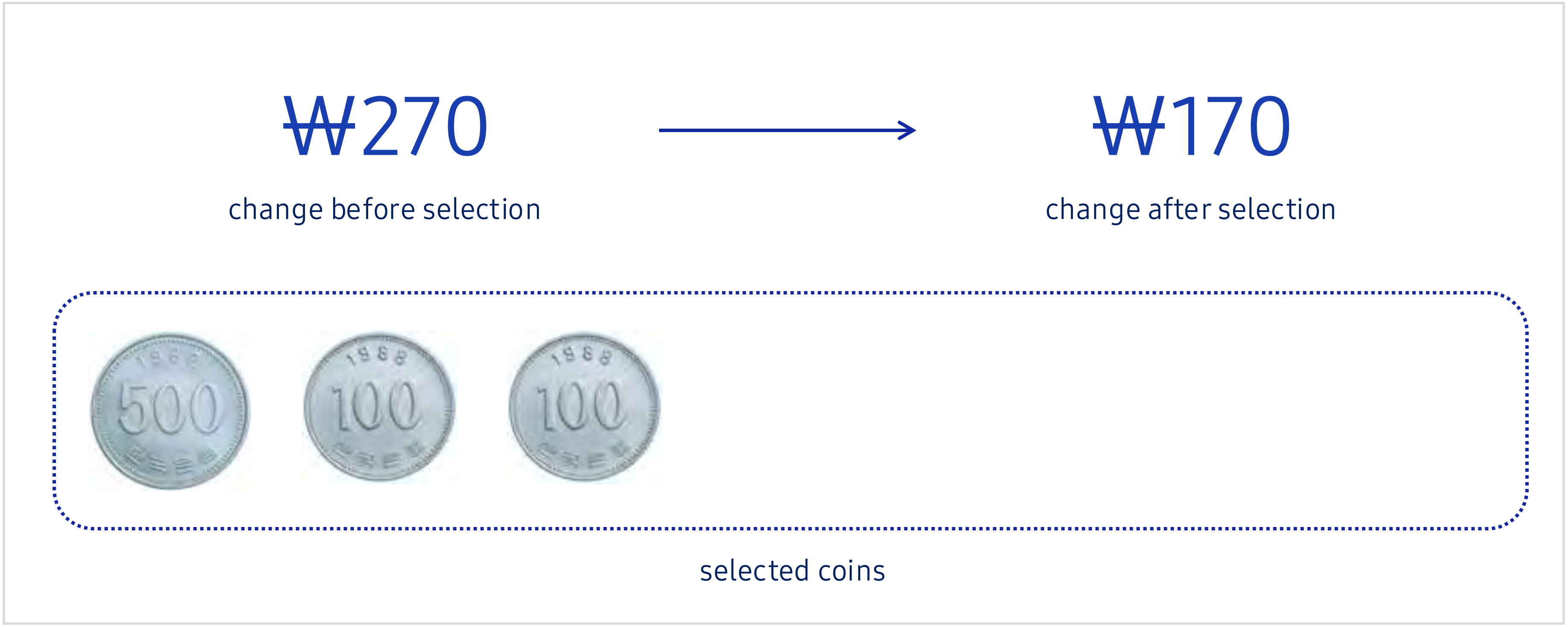
- Since 500 won coins can no longer be selected, the next largest coin, 100 won, is selected.
- The remaining change is 270 won, so you can still choose more 100 won coins.



1. Greedy Approach

1.2. Coin Change Problem

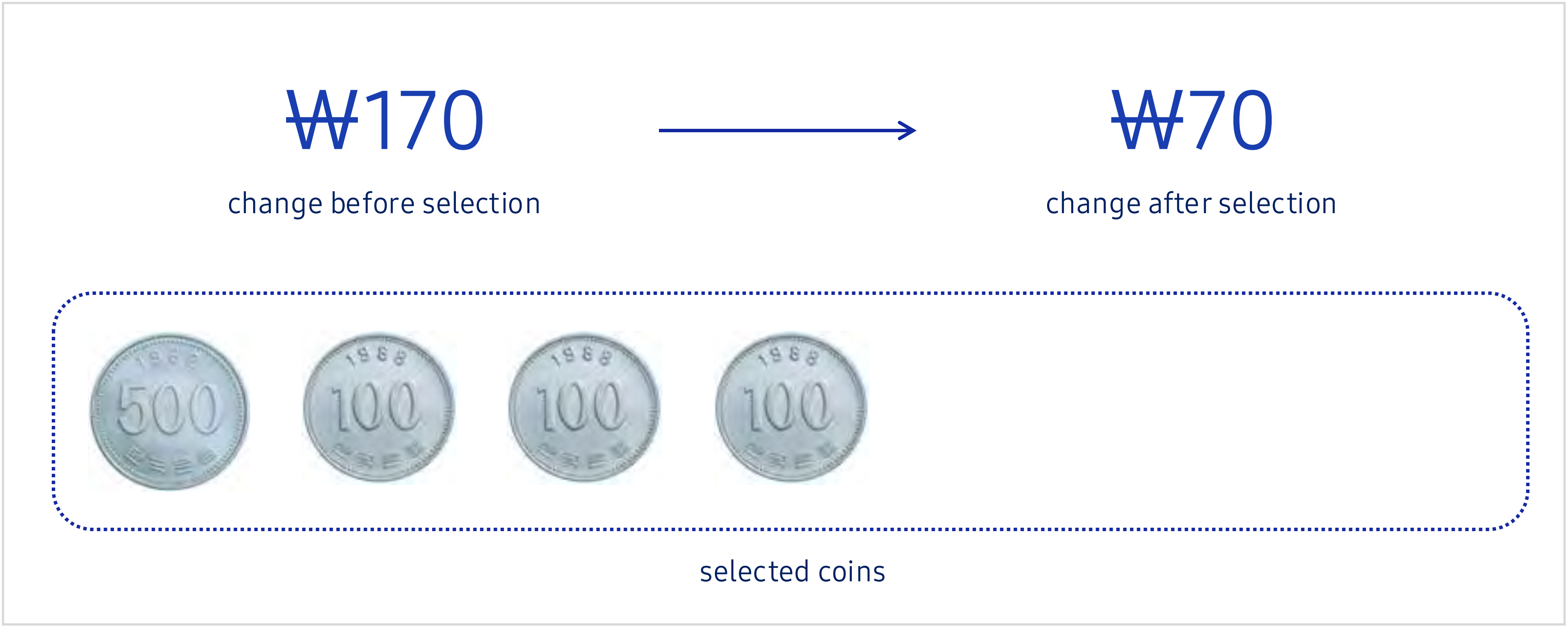
- If you choose a 100 won coin, the remaining change is 170 won, so you can still choose another 100 won coin.



1. Greedy Approach

1.2. Coin Change Problem

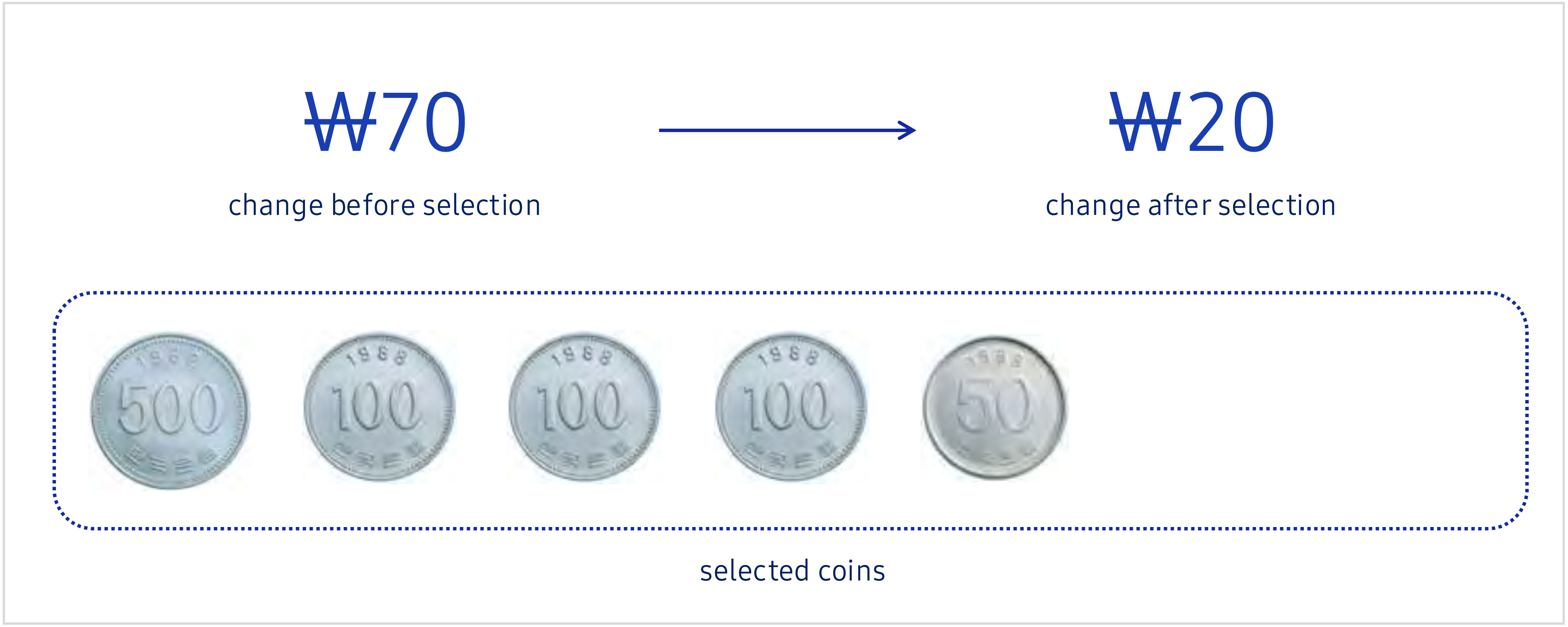
- If you choose a 100 won coin, the remaining change will be 70 won.
- Now, 100 won coins can no longer be chosen.



1. Greedy Approach

1.2. Coin Change Problem

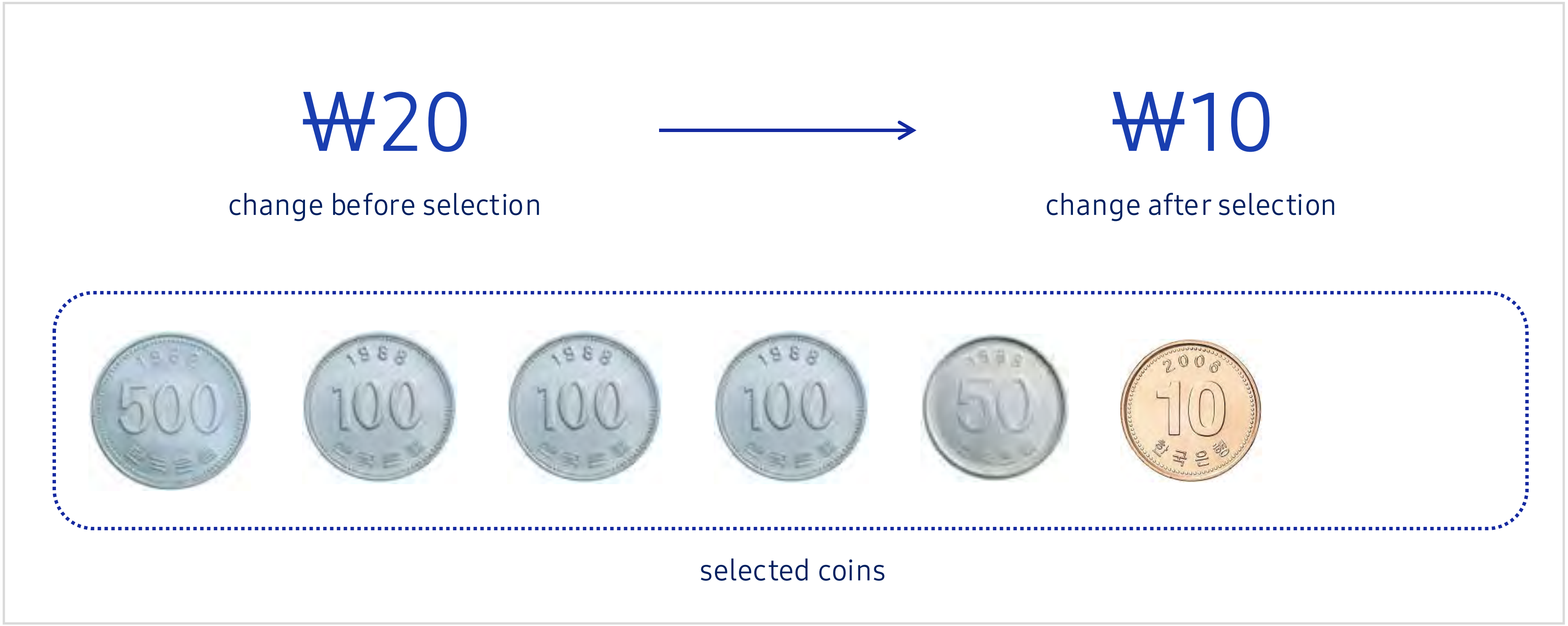
- Since you can't choose a 100 won coin, let's choose 50 won, which is the next largest coin.
- The remaining change was 20 won, and 50 won coins are no longer available.



1. Greedy Approach

1.2. Coin Change Problem

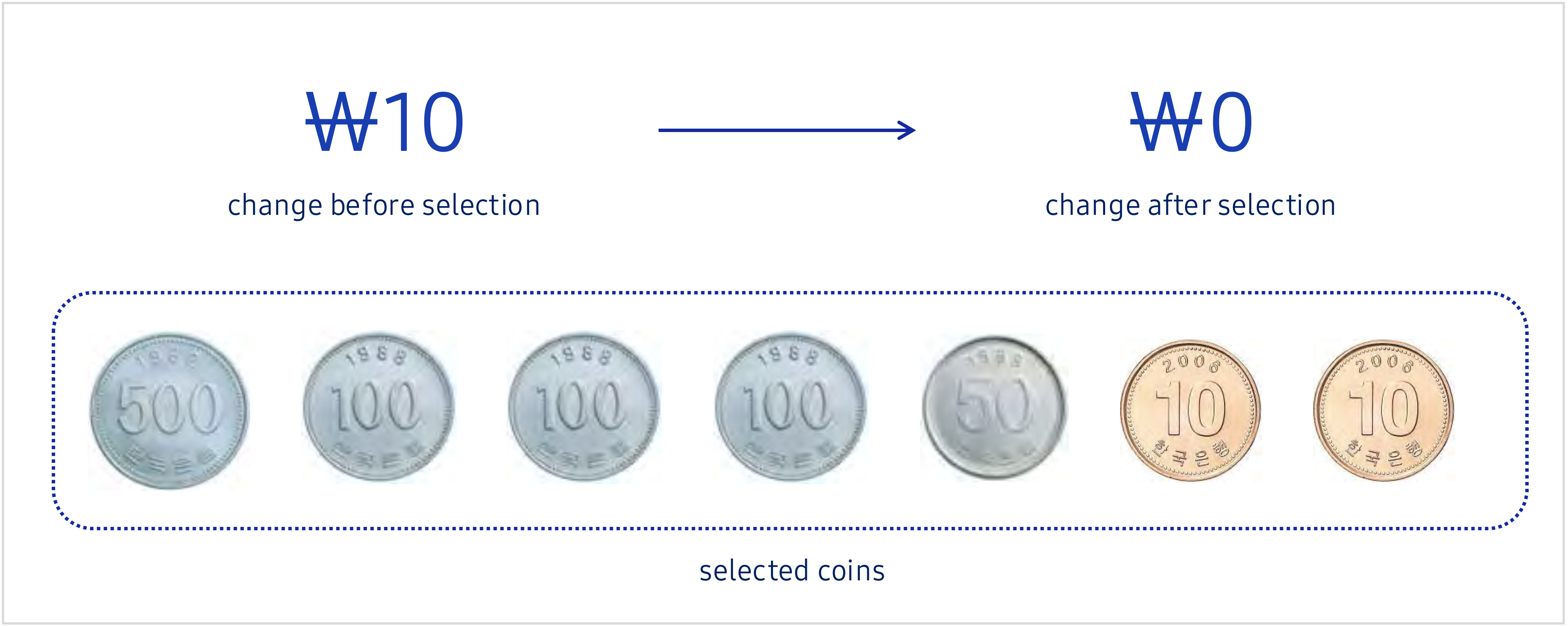
- The next largest coin after the 50 won coin is a 10 won coin, so let's choose 10 won.
- The remaining change is 10 won, so you can still choose more 10 won coins.



1. Greedy Approach

1.2. Coin Change Problem

- If you choose a 10 won coin, the remaining change will be 0 won.
- Therefore, all selected coins are 870 won, and the change can be returned with seven coins.



1. Greedy Approach

1.3. Designing the Greedy Algorithm

The above process is expressed in pseudocode as follows.

```
while (there are more coins and the problem is not solved) {
    grab the largest remaining coin;
    if (adding the coin doesn’t make the change exceed the amount owed)
        add the coin to the change;
    if (the total value of the change equals the amount owed)
        the problem is solved;
}
```

1. Greedy Approach

1.3. Designing the Greedy Algorithm

The greedy approach used in the coin exchange problem consists of three steps as follows.

- Selection Procedure:
  - Choose the coin with the highest face value.
  - That is, at this stage, the best element to be added to the solution set is selected.
- Feasibility Check:
  - Check whether the face value of the selected coin is greater than the total amount of change.
  - In other words, it is examined whether the best element selected at the current stage is appropriate to be the answer.
- Solution Check:
  - Check whether the amount of the selected coin is equal to the amount of change.
  - In other words, the addition of the best element selected at the current stage is examined to see if it is the answer.

1. Greedy Approach

1.3. Designing the Greedy Algorithm

- In the coin exchange problem, the process of selecting a coin corresponds to the selection procedure.

```
while (there are more coins and the problem is not solved) {
   grab the largest remaining coin;                      selection procedure
   if (adding the coin doesn't make the change exceed the amount owed)
        add the coin to the change;
   if (the total value of the change equals the amount owed)
        the problem is solved;
}
```

1. Greedy Approach

1.3. Designing the Greedy Algorithm

The process of checking whether the selected coin does not exceed the remaining amount of change belongs to the feasibility check stage.

```
while (there are more coins and the problem is not solved) {
    grab the largest remaining coin;
    if (adding the coin doesn't make the change exceed the amount owed)      feasibility check
        add the coin to the change;
    if (the total value of the change equals the amount owed)
        the problem is solved;
}
```

1. Greedy Approach

1.3. Designing the Greedy Algorithm

- The process of checking whether the currently selected set of coins is equal to the total amount of change belongs to the solution check process.

```
while (there are more coins and the problem is not solved) {
    grab the largest remaining coin;
    if (adding the coin doesn't make the change exceed the amount owed)
        add the coin to the change;
    if (the total value of the change equals the amount owed)
        the problem is solved;                    solution check
}
```

Let's code

1. Greedy Approach

1.1. Implementation and Coding

- Let's write an algorithm to solve the coin exchange problem.
- The input of the algorithm is the coin type and the amount of change, and the output is a list of coins.
- At this time, suppose that the amount of coins is given in descending order.

```
 1  def coin_change(coins, amount):
 2      changes = []
 3      largest = 0
 4      while amount > 0:
 5          if amount < coins[largest]:
 6              largest += 1
 7          else:
 8              changes.append(coins[largest])
 9              amount -= coins[largest]
10      return changes
```

Line 2-3

- The change list, which acts as a wallet for coins, is initialized as an empty list.
- Since coins are arranged in descending order, initialize the maximum amount of coin position 'largest' that can be placed in the current wallet as the first element of the coin list.

1. Greedy Approach

1.1. Implementation and Coding

Since the amount of coins is arranged in descending order, the greedy approach can be applied as follows.

```
 1  def coin_change(coins, amount):
 2      changes = []
 3      largest = 0
 4      while amount > 0:
 5          if amount < coins[largest]:
 6              largest += 1
 7          else:
 8              changes.append(coins[largest])
 9              amount -= coins[largest]
10      return changes
```

Line 4-9

- The while statement repeats until the change is still available.
- If the change is greater than or equal to the current 'largest' amount, subtract the amount from line 8-9.
- If the change is less than the current amount of 'largest', increase the largest and proceed with the next amount of coins.
- When you exit the door, you return the changes because the amount of the coin in the change is the amount of the change.

1. Greedy Approach

1.2. Running the Code

- The algorithm implemented earlier operates as follows.
- The list and number of coins are the output for the amount of change received from the user.
- At this time, it is assumed that the input value is an amount that can make change from a set of coins.

```python
coins = [500, 100, 50, 10]
amount = int(input("Input the amount: "))
changes = coin_change(coins, amount)
print(changes, len(changes))
```

```
Input the amount: 870
[500, 100, 100, 100, 50, 10, 10] 7
```

2. Greedy Approach to the Activity Selection Problem

2.1. Solving the Activity Selection Problem

- Let's apply the greedy approach to the activity selection problem presented above.
- Suppose that N=7 conference room usage requests were given as follows.
- Each meeting room request was given a start time and an end time. What is the maximum number of meetings that do not overlap here?

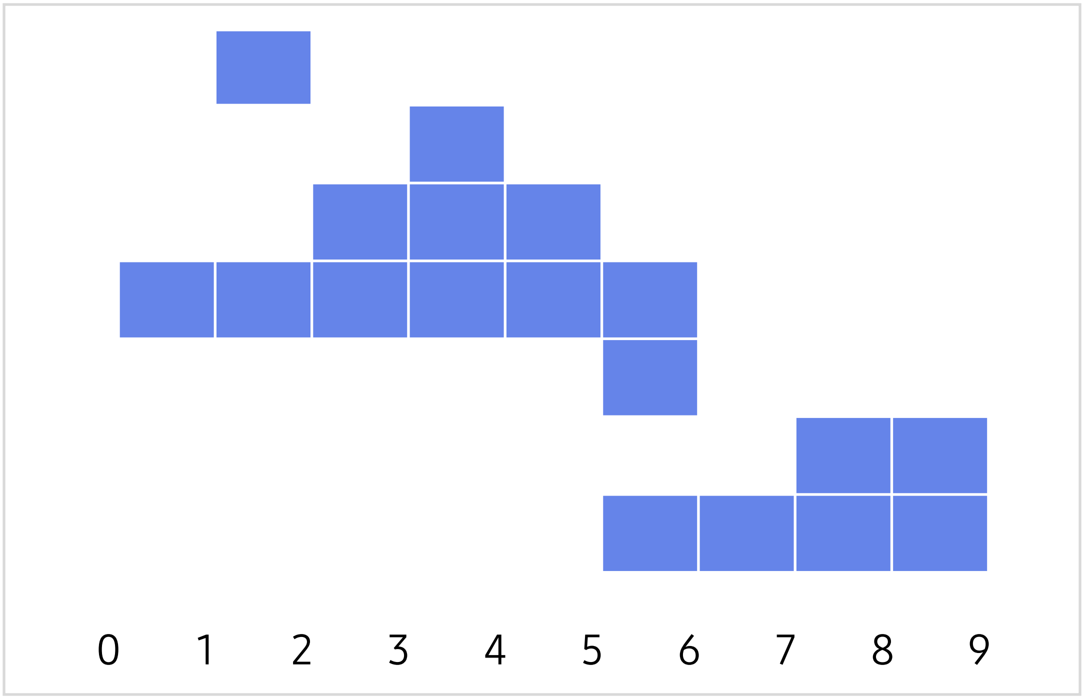
id	0	1	2	3	4	5	6
start	1	3	2	1	5	8	5
finish	2	4	5	6	6	9	9

2. Greedy Approach to the Activity Selection Problem

2.1. Solving the Activity Selection Problem

Drawing these conference room requests on the timetable is as follows.

id	start	finish
0	1	2
1	3	4
2	2	5
3	0	6
4	5	6
5	7	9
6	5	9

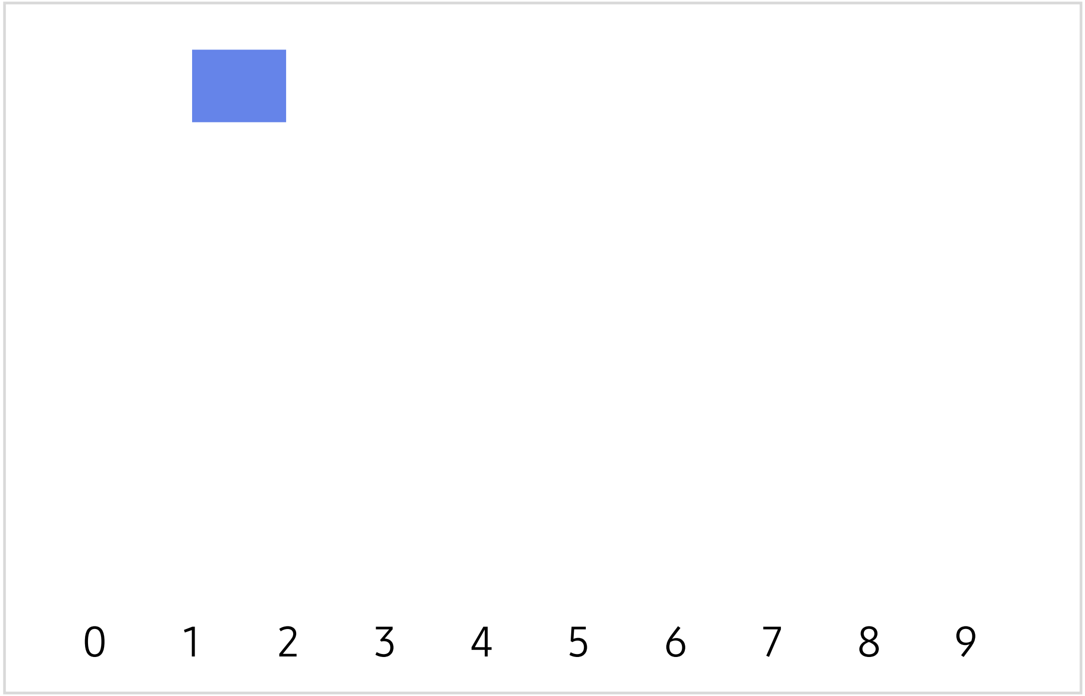


2. Greedy Approach to the Activity Selection Problem

2.1. Solving the Activity Selection Problem

- At first, all meetings can be held, so meeting 0 can be held.
- At this time, it should be noted that the meetings are arranged in order of end time. In other words, the 0th meeting is the one with the fastest end time.

id	start	finish
0	1	2

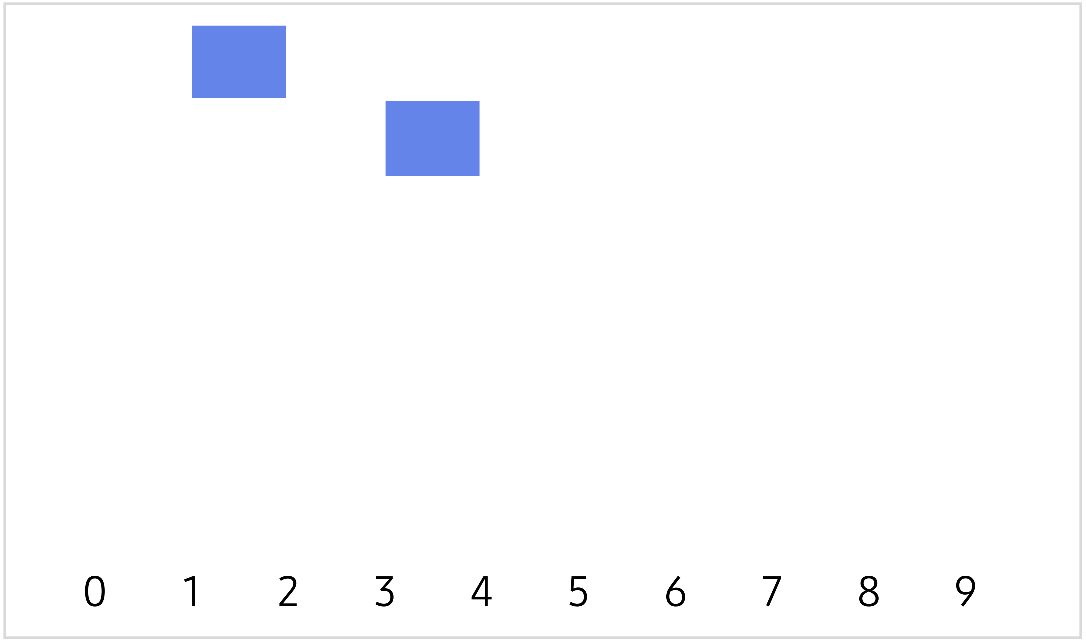


2. Greedy Approach to the Activity Selection Problem

2.1. Solving the Activity Selection Problem

- The end time of the meeting is later than the 0th meeting.
- Therefore, if the start time is behind the end time of the already selected meetings, it can be selected.
- It can be selected because the start time 3 of meeting 1 is greater than the end time 2 of meeting 0.

id	start	finish
0	1	2
1	3	4

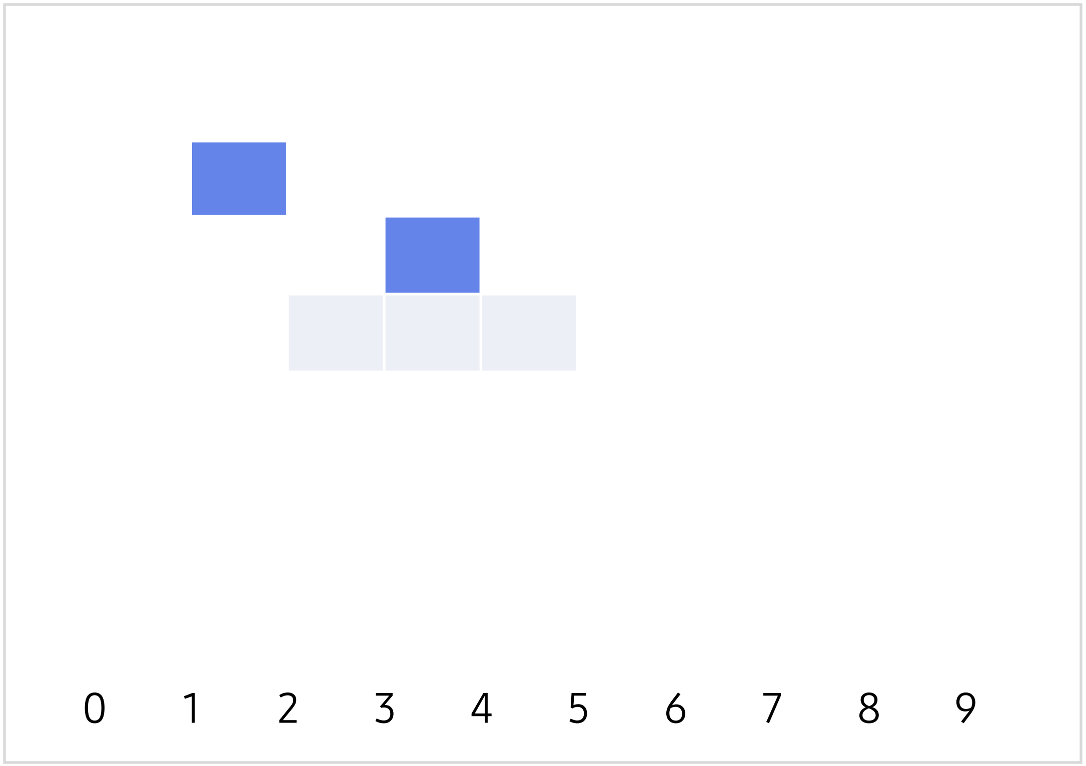


2. Greedy Approach to the Activity Selection Problem

2.1. Solving the Activity Selection Problem

- The start time of the second meeting is less than the end time of the first meeting, 4, so it not selectable.

id	start	finish
0	1	2
1	3	4
2	2	5

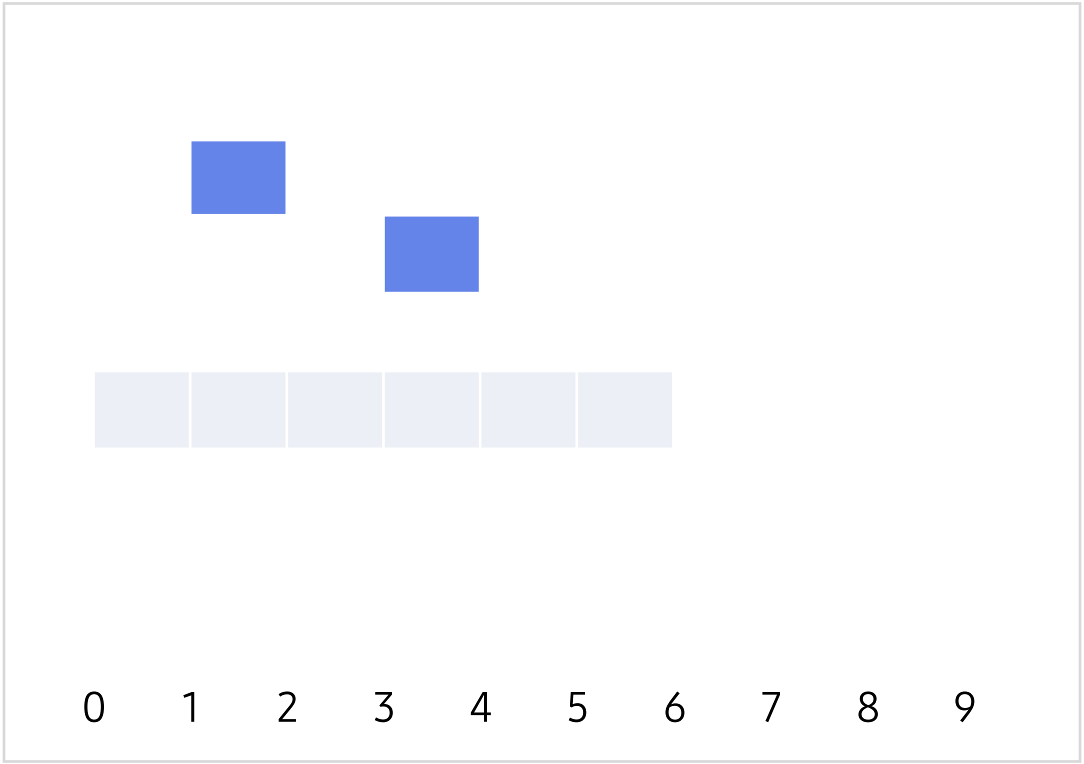


2. Greedy Approach to the Activity Selection Problem

2.1. Solving the Activity Selection Problem

- It is impossible to select because the start time of meeting 3 is less than the end time of meeting 1, which is 4.

id	start	finish
0	1	2
1	3	4
2	2	5
3	0	6

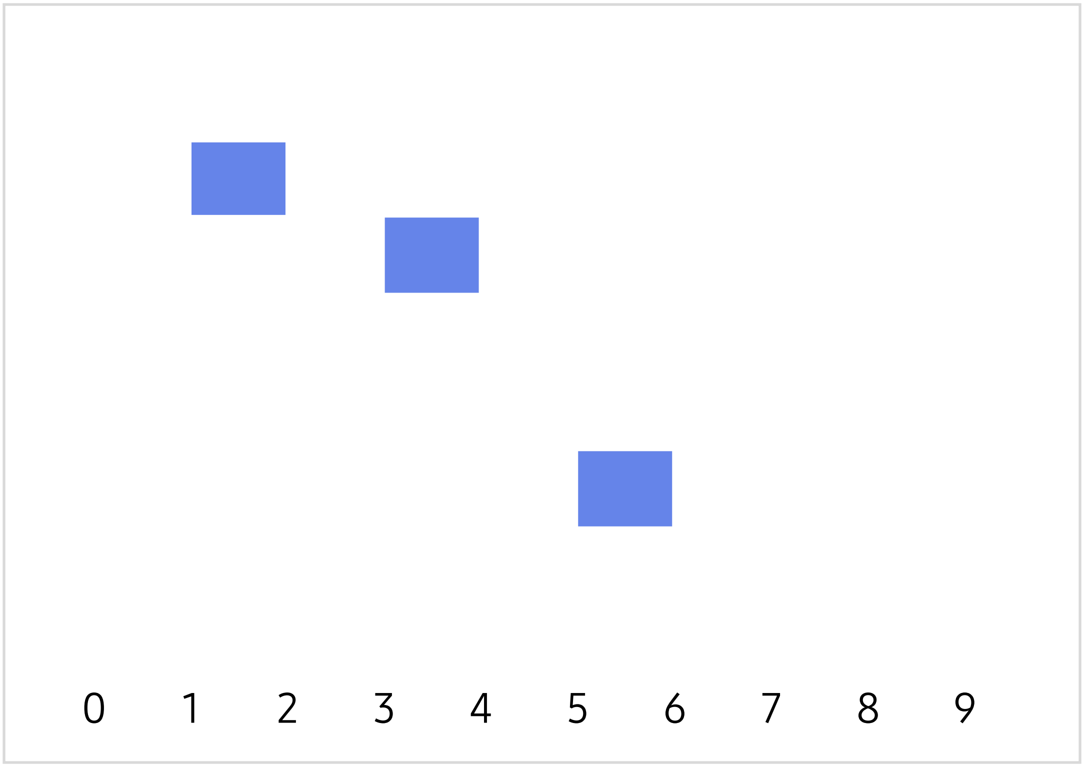


2. Greedy Approach to the Activity Selection Problem

2.1. Solving the Activity Selection Problem

- The start time of the 4th meeting, 5, is greater than the end time of the 1st meeting, so it is selectable.

id	start	finish
0	1	2
1	3	4
2	2	5
3	0	6
4	5	6

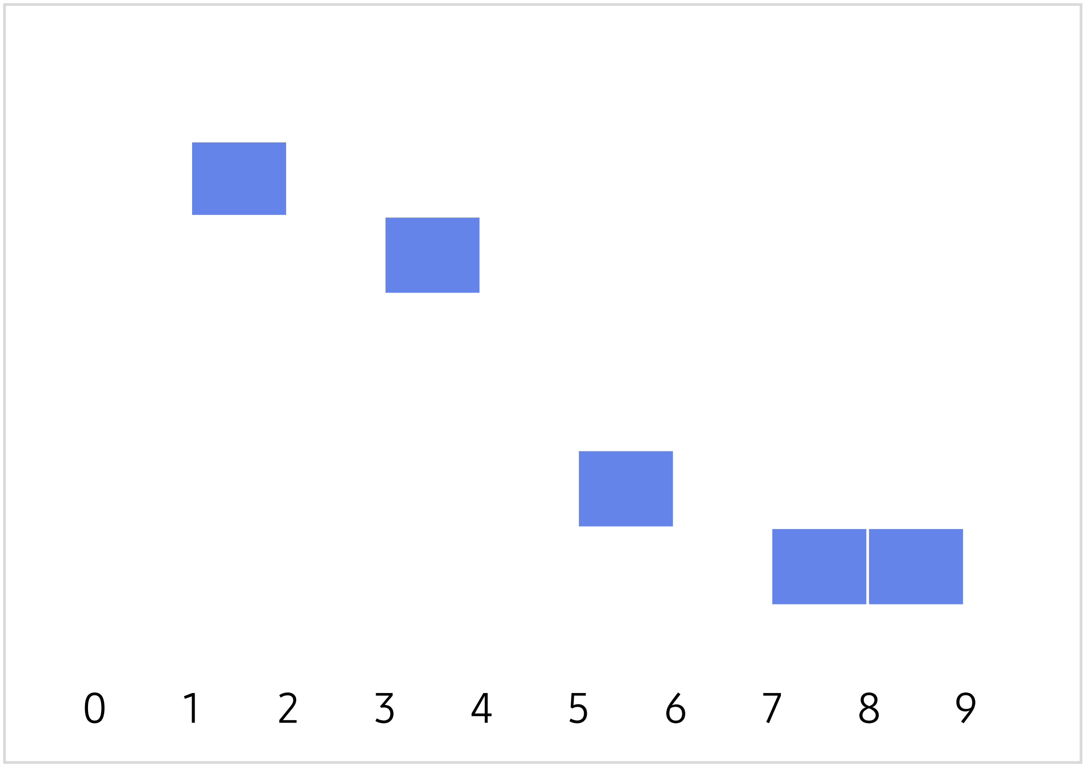


2. Greedy Approach to the Activity Selection Problem

2.1. Solving the Activity Selection Problem

- The start time of meeting 5, 7, is greater than the end time of meeting 4, which is 6, so it is also selectable.

id	start	finish
0	1	2
1	3	4
2	2	5
3	0	6
4	5	6
5	7	9

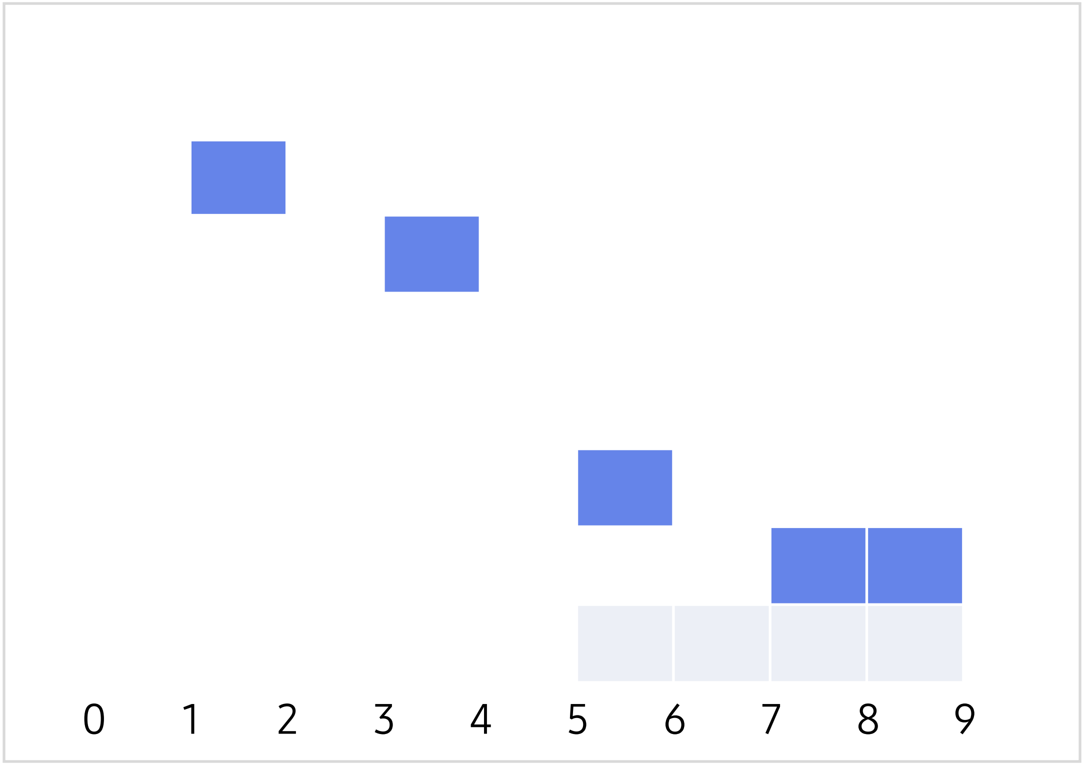


2. Greedy Approach to the Activity Selection Problem

2.1. Solving the Activity Selection Problem

- The start time of meeting 6, which is 5, is less than the start time of meeting 5, so it is not selectable.

id	start	finish
0	1	2
1	3	4
2	2	5
3	0	6
4	5	6
5	7	9
6	5	9

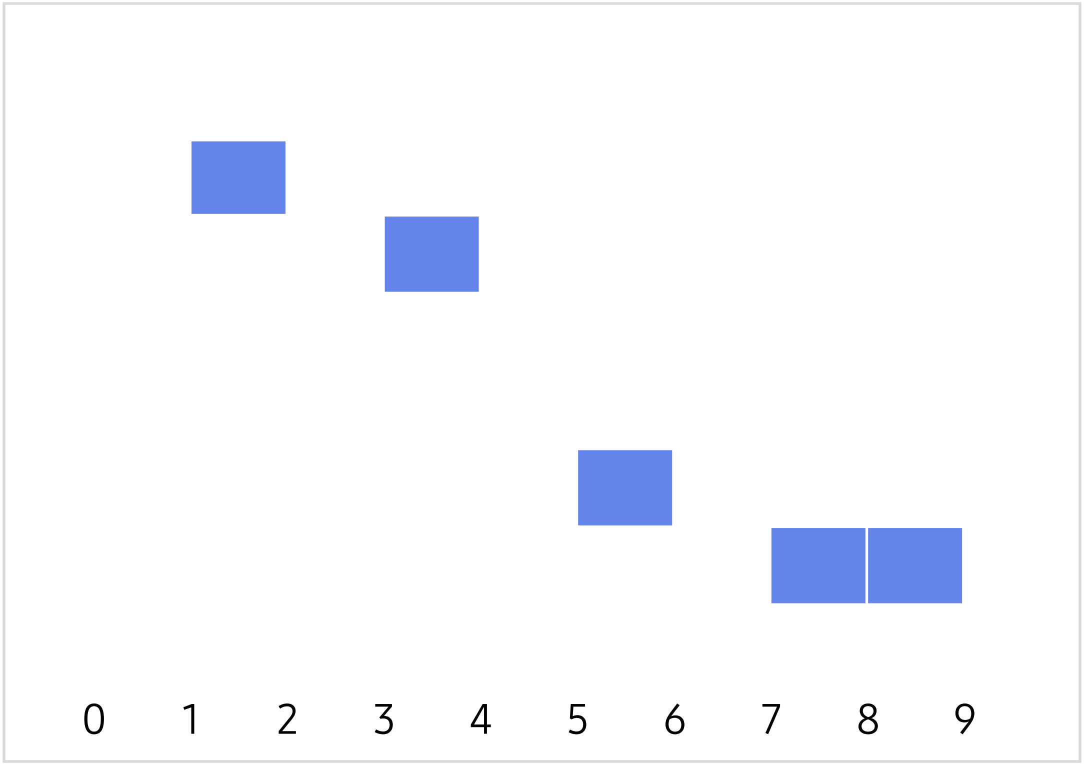


2. Greedy Approach to the Activity Selection Problem

2.1. Solving the Activity Selection Problem

- The final selectable meeting is [0, 1, 4, 5] and the maximum number of selectable meetings is 4.

id	start	finish
0	1	2
1	3	4
2	2	5
3	0	6
4	5	6
5	7	9
6	5	9



2. Greedy Approach to the Activity Selection Problem

2.2. Implementation and Coding

Let's solve the activity selection problem using the strategy introduced earlier.

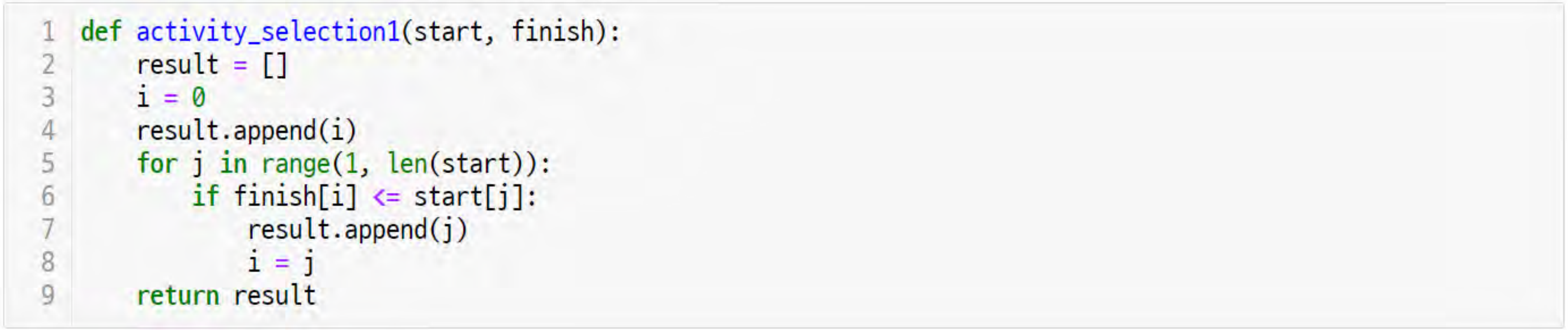
```python
1 def activity_selection1(start, finish):
2     result = []
3     i = 0
4     result.append(i)
5     for j in range(1, len(start)):
6         if finish[i] <= start[j]:
7             result.append(j)
8             i = j
9     return result
```

Line 1-4

- For the activity selection problem, the meeting start time 'start' and the meeting end time 'finish' are given as inputs.
- The 'result' list will include the meeting number.
- Therefore, the index I is initially stored as 0 and put in the result.

2. Greedy Approach to the Activity Selection Problem

2.2. Implementation and Coding



```python
1  def activity_selection1(start, finish):
2      result = []
3      i = 0
4      result.append(i)
5      for j in range(1, len(start)):
6          if finish[i] <= start[j]:
7              result.append(j)
8              i = j
9      return result
```

Line 5-9

- Since activity 0 has already been selected, attempts are made for meetings from 1 to len(start) -1.
- If finish[i] is less than or equal to start[j], it means that the j-th meeting does not overlap with the existing meeting.
- Therefore, j, which satisfies this condition, is added to the result.
- Now update I to the value of j to specify the last time of the meeting as the j-th meeting.

2. Greedy Approach to the Activity Selection Problem

2.3. Running the Code

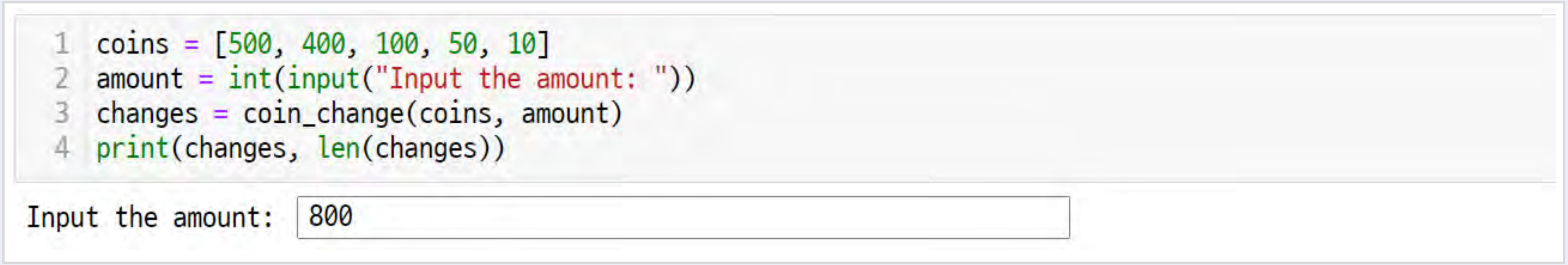
- The activity selection problem can be executed as follows.
- At this time, start and finish represent the meeting start time and the meeting end time, respectively, and the meetings output are the index number of the selected meeting, and maximum is the number of the maximum meeting.

```python
start = [1, 3, 2, 0, 5, 8, 5]
finish = [2, 4, 5, 6, 6, 9, 9]
meetings = activity_selection1(start, finish)
maximum = len(meetings)
print(meetings, maximum)
```

```
[0, 1, 4, 5] 4
```

Pop quiz

Q1. In the coin exchange problem, suppose there is a 400 won coin.
If so, write the output of how the algorithm coin_change() will determine the change for the 800 won coin.



```
coins = [500, 400, 100, 50, 10]
amount = int(input("Input the amount: "))
changes = coin_change(coins, amount)
print(changes, len(changes))
```

```
Input the amount:  800
```

Pair programming

Pair Programming Practice

Guideline, mechanisms & contingency plan

Preparing pair programming involves establishing guidelines and mechanisms to help students pair properly and to keep them paired. For example, students should take turns "driving the mouse." Effective preparation requires contingency plans in case one partner is absent or decides not to participate for one reason or another. In these cases, it is important to make it clear that the active student will not be punished because the pairing did not work well.

Pairing similar, not necessarily equal, abilities as partners

Pair programming can be effective when students of similar, though not necessarily equal, abilities are paired as partners. Pairing mismatched students often can lead to unbalanced participation. Teachers must emphasize that pair programming is not a "divide-and-conquer" strategy, but rather a true collaborative effort in every endeavor for the entire project. Teachers should avoid pairing very weak students with very strong students.

Motivate students by offering extra incentives

Offering extra incentives can help motivate students to pair, especially with advanced students. Some teachers have found it helpful to require students to pair for only one or two assignments.

Pair Programming Practice

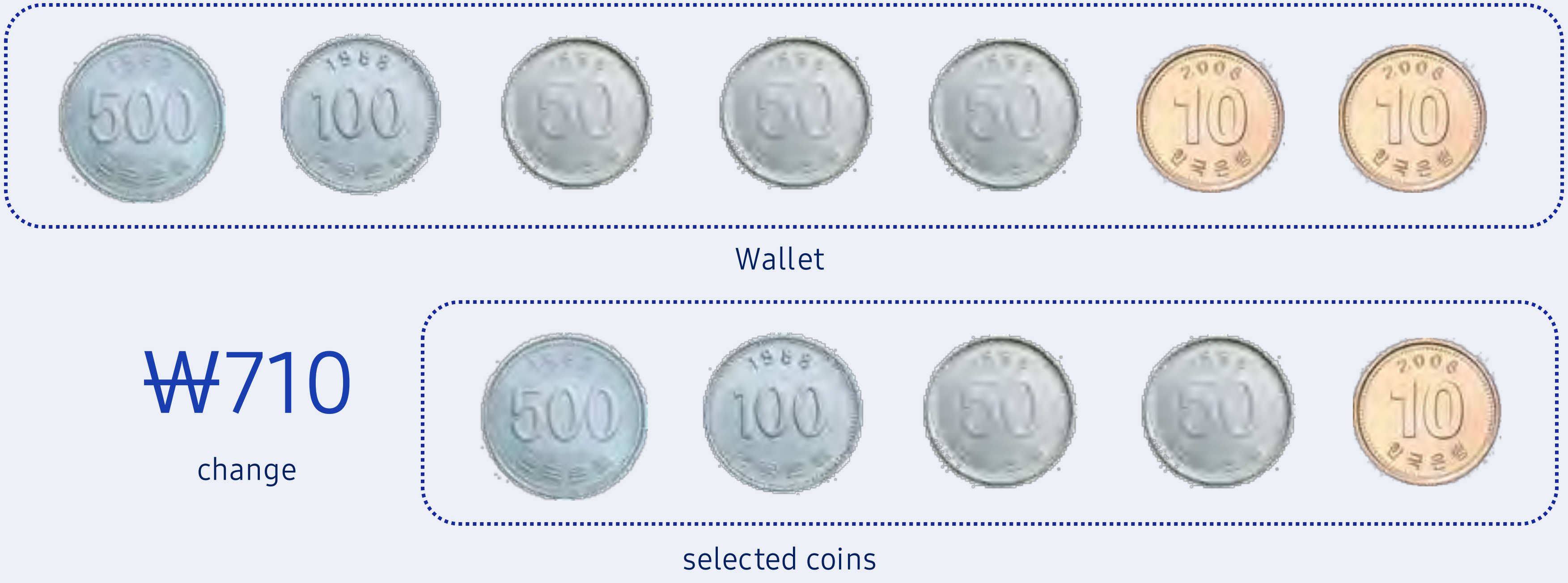
Prevent collaboration cheating

The challenge for the teacher is to find ways to assess individual outcomes, while leveraging the benefits of collaboration. How do you know whether a student learned or cheated? Experts recommend revisiting course design and assessment, as well as explicitly and concretely discussing with the students on behaviors that will be interpreted as cheating. Experts encourage teachers to make assignments meaningful to students and to explain the value of what students will learn by completing them.

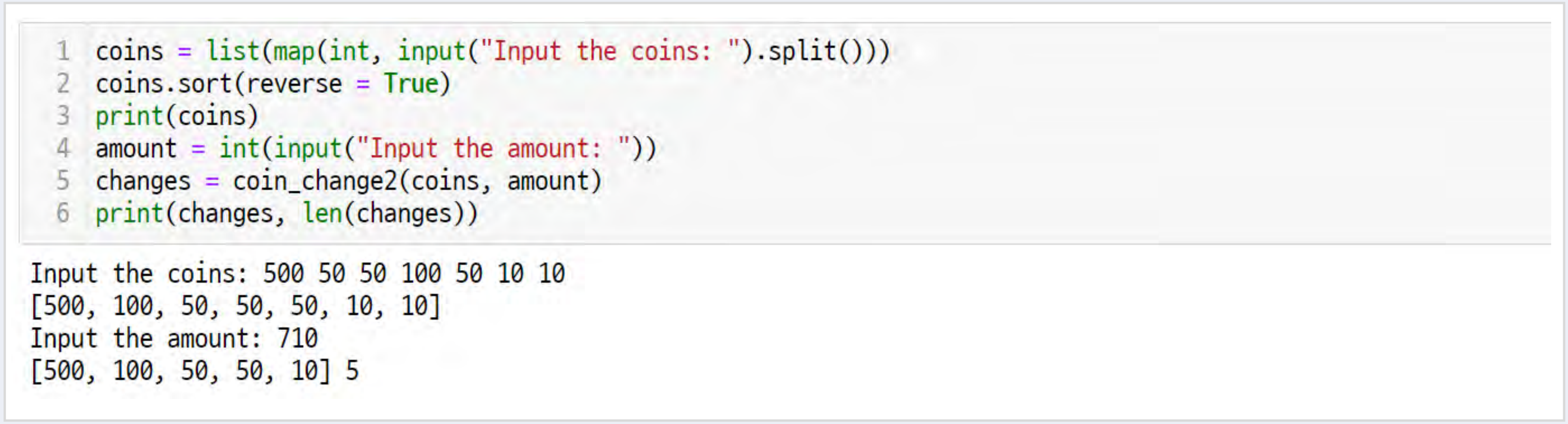
Collaborative learning environment

A collaborative learning environment occurs anytime an instructor requires students to work together on learning activities. Collaborative learning environments can involve both formal and informal activities and may or may not include direct assessment. For example, pairs of students work on programming assignments; small groups of students discuss possible answers to a professor's question during lecture; and students work together outside of class to learn new concepts. Collaborative learning is distinct from projects where students "divide-and-conquer." When students divide the work, each is responsible for only part of the problem solving and there are very limited opportunities for working through problems with others. In collaborative environments, students are engaged in intellectual talk with each other.

Q1. Suppose that the number of coins in the coin exchange problem is not infinite. For example, if you have the following coins in your wallet, how should you distribute 710 won as change?



Q2. Given the number of coins and the amount of change, find the minimum number of coins you can give back as change.



```
1 coins = list(map(int, input("Input the coins: ").split()))
2 coins.sort(reverse = True)
3 print(coins)
4 amount = int(input("Input the amount: "))
5 changes = coin_change2(coins, amount)
6 print(changes, len(changes))
```

```
Input the coins: 500 50 50 100 50 10 10
[500, 100, 50, 50, 50, 10, 10]
Input the amount: 710
[500, 100, 50, 50, 10] 5
```

Hint | Notice that you arranged the coins in descending order on line 2 with coins.sort(reverse = True).

Unit 31.

Divide-and-Conquer

Unit learning objective (1/3)

Learning objectives

- ✓ Learners will be able to understand the divide-and-conquer method and solve solve the triangle path problem using this method.
- ✓ Learners will be able to understand that binary search, merge sorting, and quick sorting are algorithms that apply the divide-and-conquer method.
- ✓ Learners will be able to recursively design and implement the divide-and-conquer algorithm.

Unit learning objective (2/3)

Learning overview

- ✓ Learn how to solve the Triangle Path problem using the divide-and-conquer method.
- ✓ Learn how to design the divide-and-conquer method through binary search, merge sorting, and quick sorting.
- ✓ Learn how to solve problems using the divide-and-conquer method.

Concepts you will need to know from previous units

- ✓ Be able to understand and implement the definition of the recursive function and their termination conditions.
- ✓ Be able to solve the binary search problem using loops and recursions.
- ✓ Be able to solve sorting problems using merge sorting and quick sorting.

Unit learning objective (3/3)

Keywords

Divide-and-Conquer

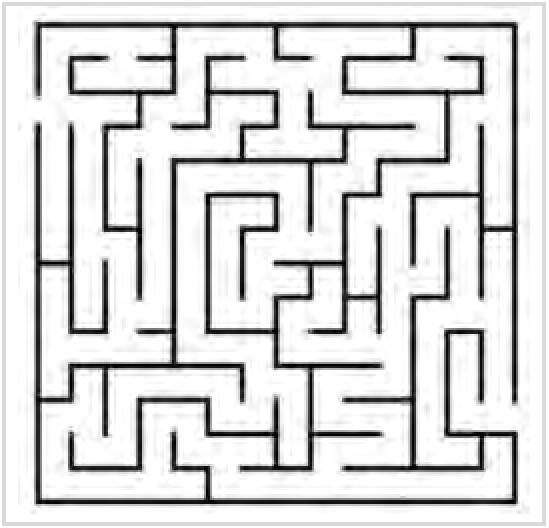
Recursion

Triangle Path Problem

Mission

1. Real world problem

1.1. Find a Path in the Maze

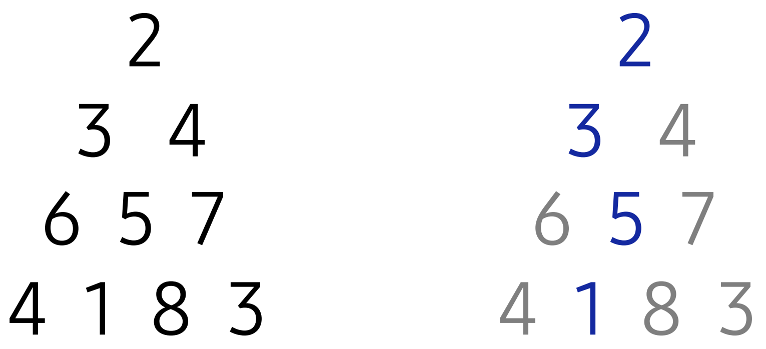


- Consider the problem of finding a way in a maze of several rooms.
- Suppose that a maze is given in a two-dimensional array, can only move in the right and down directions, and you must pay a certain amount of money when you pass through each room.
- How can we find the path that incurs the least cost?

2. Mission

2.1. Minimum Triangle Path

- Suppose that in a given triangle composed of positive intergers, it can start at the top of the triangle and go down one row at a time.
- If a move is possible only to adjacent left or right numbers, what is the sum of the smallest number of travel routes until it reaches the bottom?
- For example, in the triangle below, the minimum sum of the travel paths is 2 + 3 + 5 + 1 = 11.



3. Solution

3.1. How the Triangle Path Finder works

A program that finds the sum of shortest paths in a triangle is a program that finds and outputs the sum of shortest paths in this triangle with the given list of numbers, as shown below.

```python
triangle = [
    [2],
    [3, 4],
    [6, 5, 7],
    [4, 1, 8, 3]
]
minimum = find_minimum(0, 0, triangle)
print("The minimum cost is ", minimum)
```

```
The minimum cost is  11
```

3. Solution

3.2. Triangle Path Finder Final Code

```python
def find_minimum(row, col, triangle):
    if row == len(triangle):
        return 0
    else:
        minimum = min(find_minimum(row + 1, col, triangle),
                      find_minimum(row + 1, col + 1, triangle))
        return triangle[row][col] + minimum
```

| Key concept

1. Divide-and-Conquer

1.1. The Divide-and-Conquer Strategy

- The divide-and-conquer strategy divides the input instance of a given problem into two or more small input instances.
- If the answer to the divided input instance cannot be obtained immediately, divide it back into smaller input instances.
- If you have obtained the answer to the divided input instance, you can integrate this answer to find the answer to the original input instance.

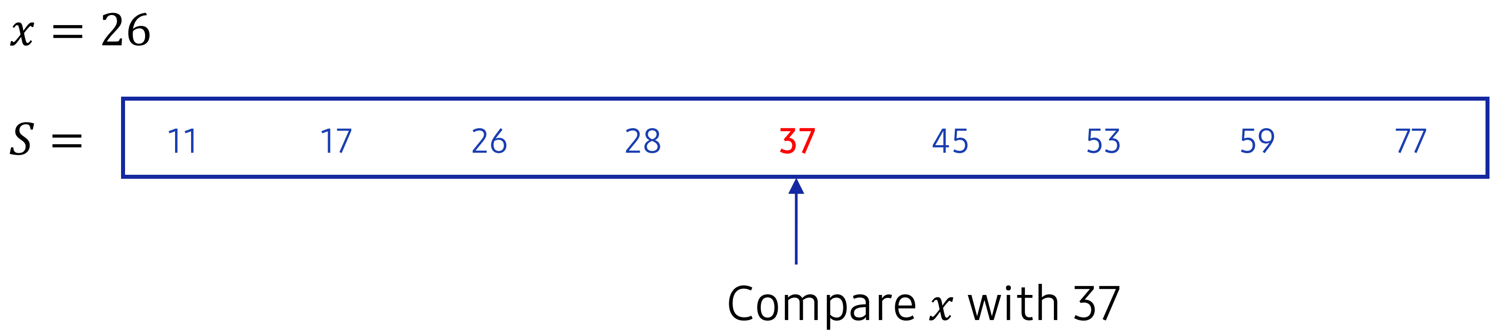
Focus The divide-and-conquer method is a strategy to solve the problems in the following steps.

- Divide: Divide the input instance of the problem into two or more small input instances.
- Conquer: Solve each of the divided input instances. If the divided instance is not small enough, use the recursion to solve it.
- Combine: If necessary, find the answer of the original input instance by combining the answers of the small input instance.

1. Divide-and-Conquer

1.2. Binary Search and the Divide-and-Conquer

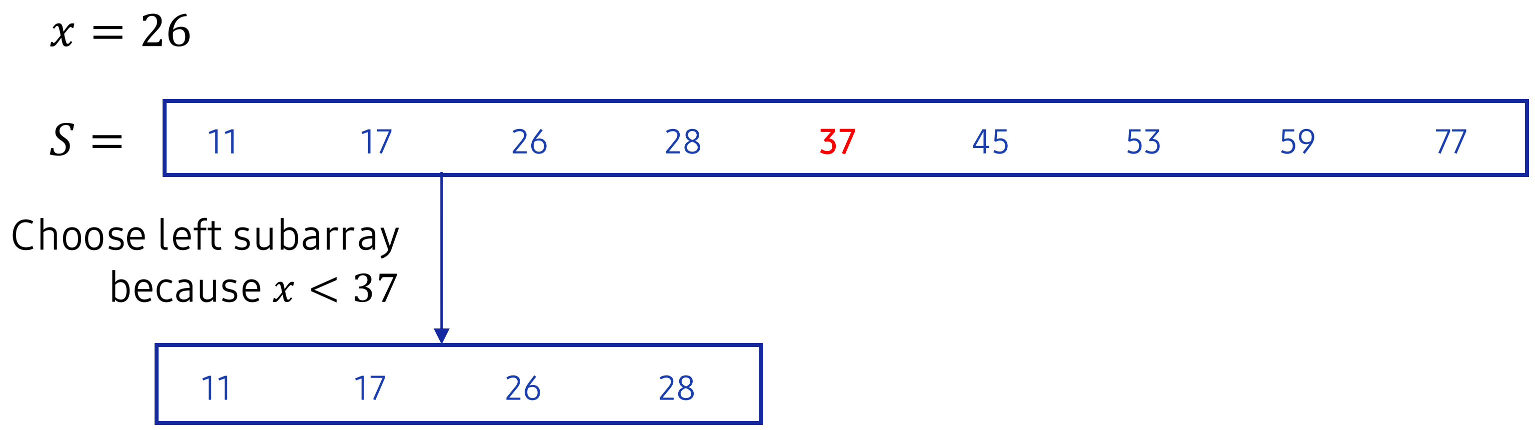
- The binary search studied in Unit 25 is a representative example of divide-and-conquer.
- Binary search uses a divide-and-conquer strategy to find a given element in a sorted list.
- For example, let's say we find x=26 as a binary search in an ordered list S with nine elements as shown below.
- First, we compare x with the central element 37 in this list.



1. Divide-and-Conquer

1.2. Binary Search and the Divide-and-Conquer

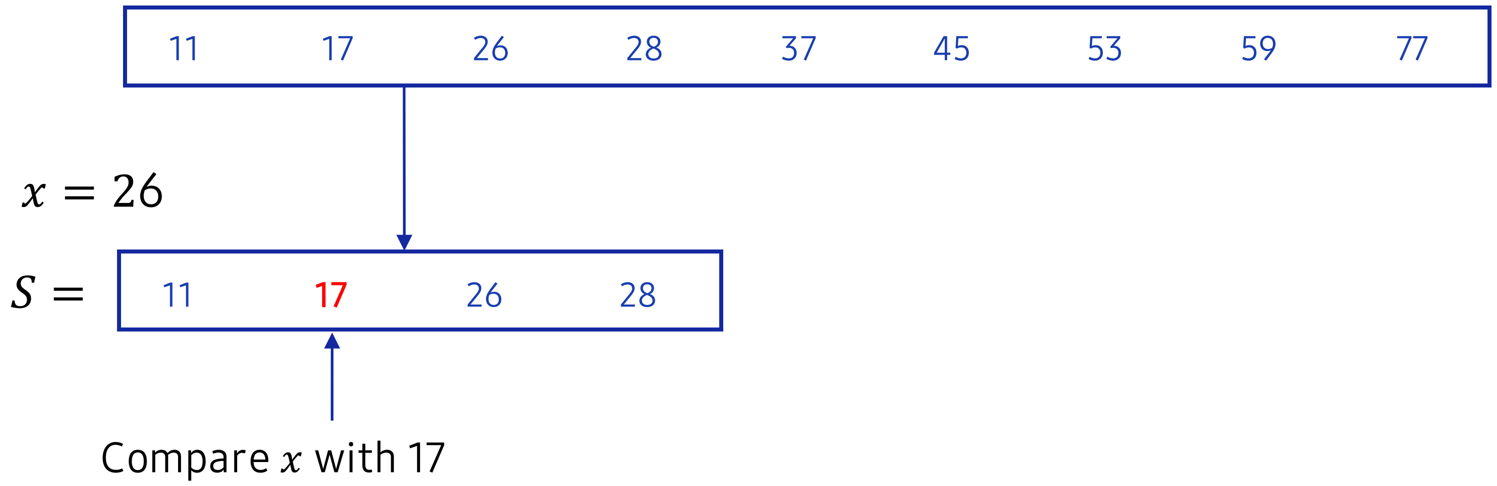
- Since x is less than 37, if x exists in S, it exists in the left part of this list, so the search space can be reduced by half.



1. Divide-and-Conquer

1.2. Binary Search and the Divide-and-Conquer

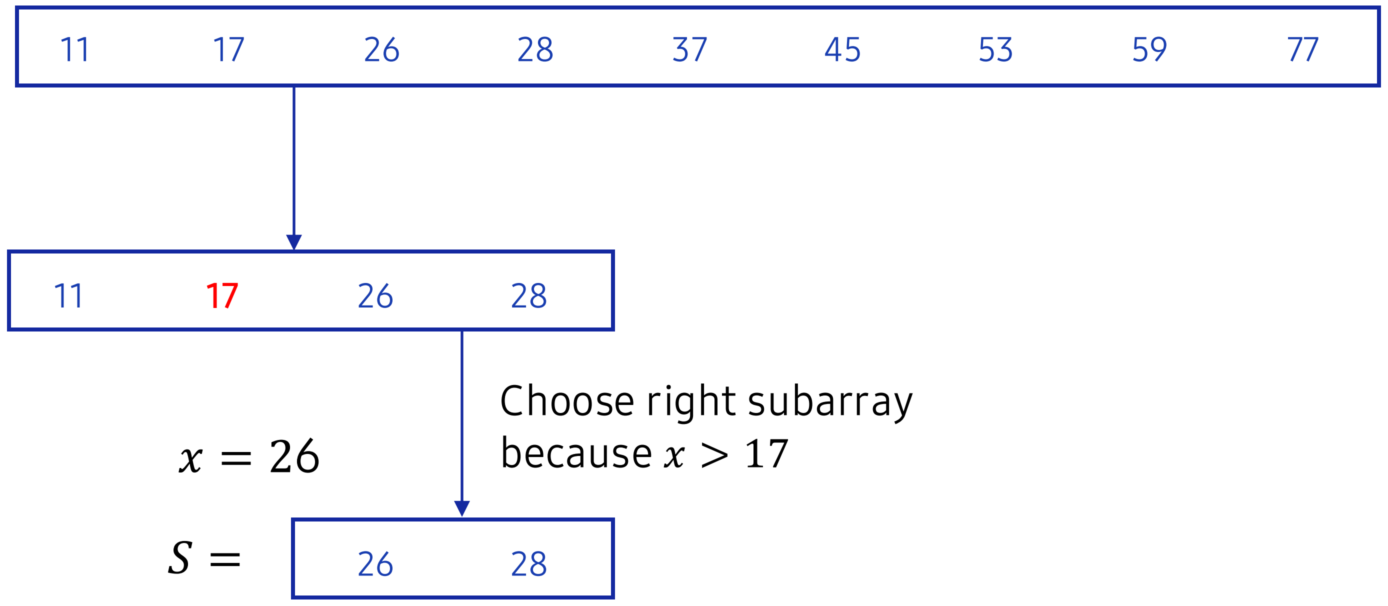
Compare the values of the middle element 17 and x again in the half-reduced search space S.



1. Divide-and-Conquer

1.2. Binary Search and the Divide-and-Conquer

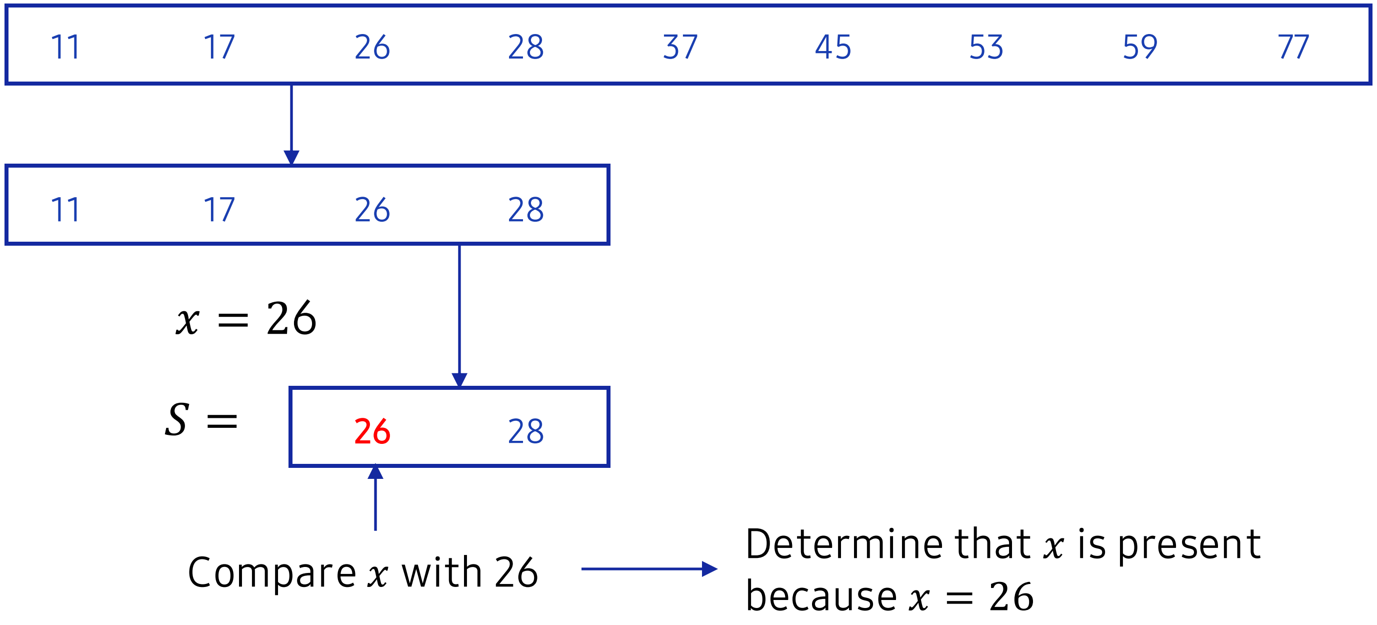
- Since x is greater than 17, if x exists in this list, it will exist in the right part of S, so we can reduce the search space again.



1. Divide-and-Conquer

1.2. Binary Search and the Divide-and-Conquer

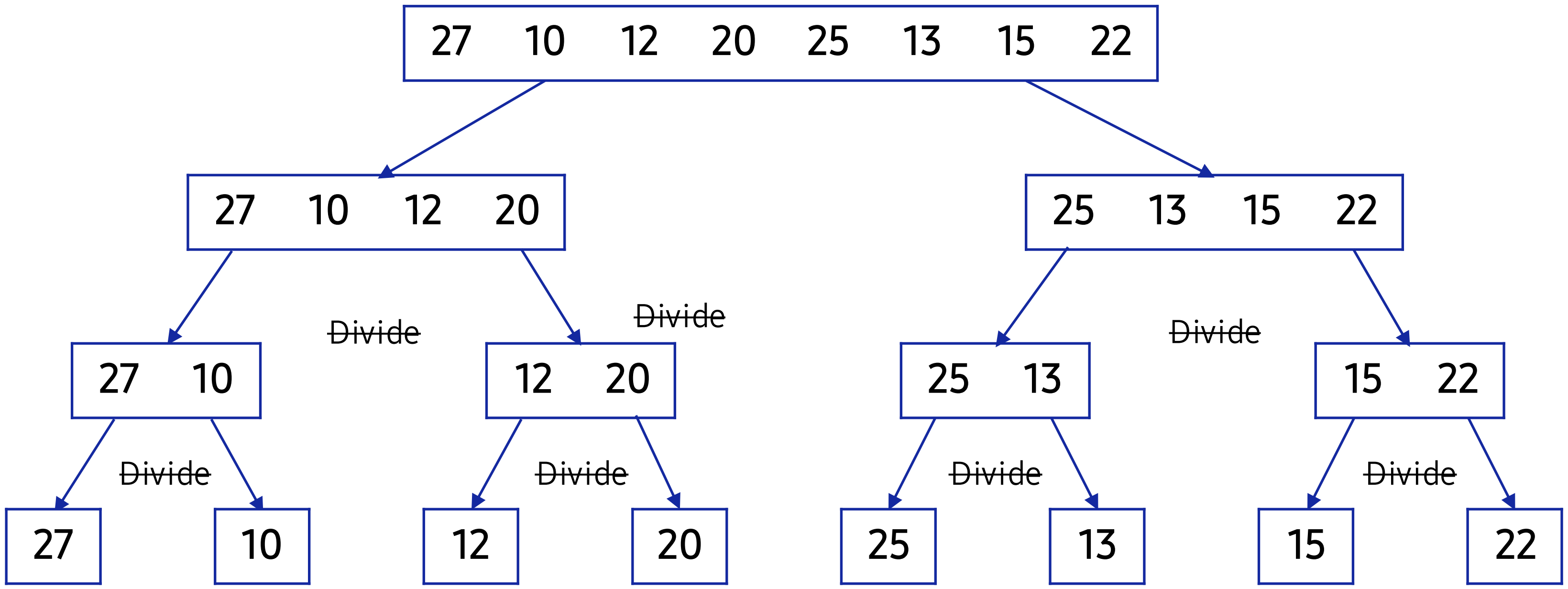
- If the middle element of S is equal to x, the search can be terminated.
- If S continues to search binary and becomes an empty set, we can see that x is an element that does not exist in the list S.



1. Divide-and-Conquer

1.3. Merge Sort and the Divide-and-Conquer

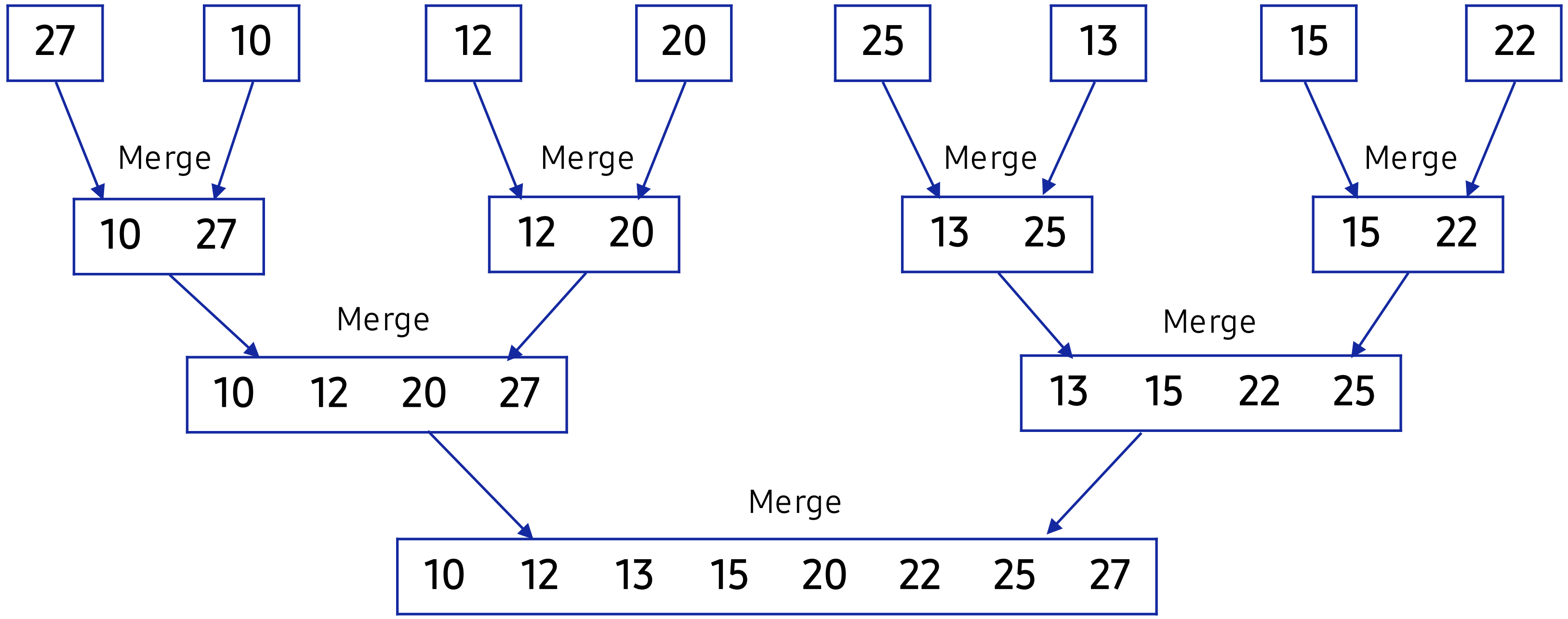
- The merge sorting studied in Unit 28 also uses divide-and-conquer.
- At this time, it should be noted that binary search conquers only one of the divided two-part problems, whereas merge sorting conquers each of the divided two-part problems.



1. Divide-and-Conquer

1.3. Merge Sort and the Divide-and-Conquer

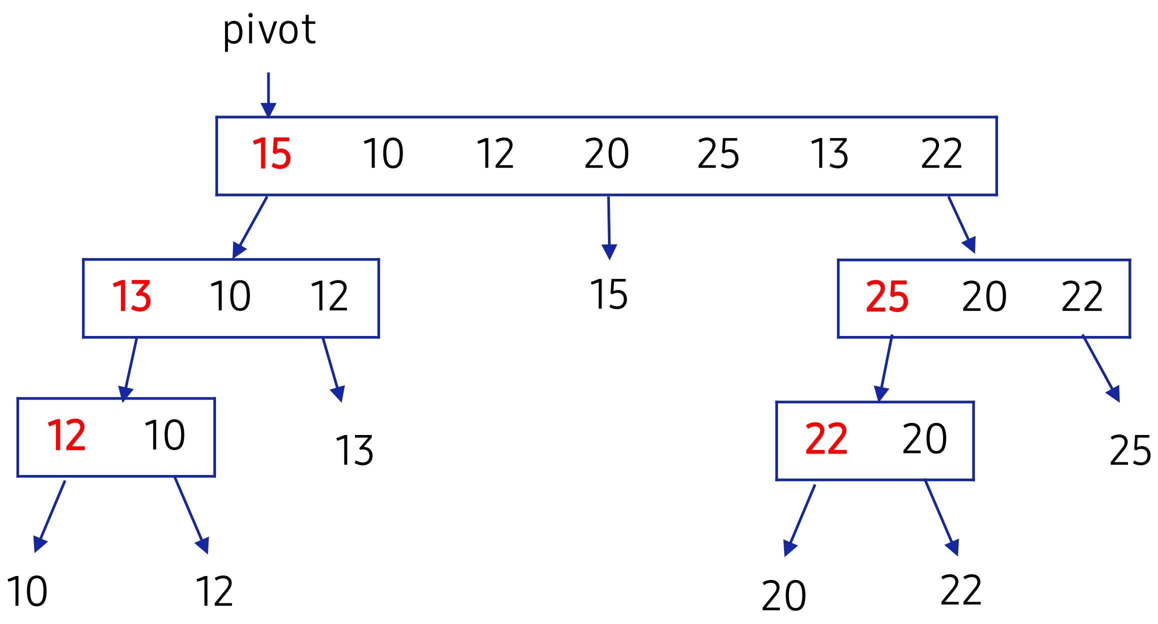
- In addition, it is worth noting that merge sorting required a separate procedure for merging the two sorted lists after dividing the problem into two partial problems of the same size and conquering them.



1. Divide-and-Conquer

1.4. Quick Sort and the Divide-and-Conquer

- Quick sorting, studied in Unit 29, also uses divide-and-conquer.
- At this time, quick sorting divides a given problem into two input cases, but the size of the two divided input cases is not equal.
- In addition, it should be noted that quick sorting does not have a process of integrating the two sorted lists.



Let’s code

1. Divide-and-Conquer

1.1. Binary Search with Iteration

- Let's look again at the implementation of the bin_search() function in Unit 25.
- This function used a strategy to reduce the scope of the search using low and high indices.

```
 1  def bin_search(nums, x):
 2      low, high = 0, len(nums) - 1
 3      while low <= high:
 4          mid = (low + high) // 2
 5          if nums[mid] == x:
 6              return mid
 7          elif nums[mid] > x:
 8              high = mid - 1
 9          else:
10              low = mid + 1
11      return -1
```

Line 1-11

- The binary search algorithm using the iterative statement reduces the scope of the search using the low and high indices.

1. Divide-and-Conquer

1.2. Binary Search with the Divide-and-Conquer

The binary search problem can be implemented recursively, as shown below, by applying the divide-and-conquer strategy.

```
 1  def bin_search2(nums, x, low, high):
 2      if low > high:
 3          return -1
 4      else:
 5          mid = (low + high) // 2
 6          if nums[mid] == x:
 7              return mid
 8          elif nums[mid] > x:
 9              return bin_search2(nums, x, low, mid - 1)
10          else:
11              return bin_search2(nums, x, mid + 1, high)
```

Line 2-3

- bin_search2() finds x in nums and returns the corresponding index.
- The termination condition of the recursive function is when the value of low is greater than the value of high.
- In this case, since x does not exist in S, return -1.

1. Divide-and-Conquer

1.2. Binary Search with the Divide-and-Conquer

```
 1  def bin_search2(nums, x, low, high):
 2      if low > high:
 3          return -1
 4      else:
 5          mid = (low + high) // 2
 6          if nums[mid] == x:
 7              return mid
 8          elif nums[mid] > x:
 9              return bin_search2(nums, x, low, mid - 1)
10          else:
11              return bin_search2(nums, x, mid + 1, high)
```

Line 2-3

- Mid is the middle element of the search range, and returns the mid index if nums[mid] is equal to the value of x.
- If x is less than nums[mid], search for the range [low, mid -1] by recursion.
- If x is greater than nums[mid], search for the range [mid + 1, high] by recursion.

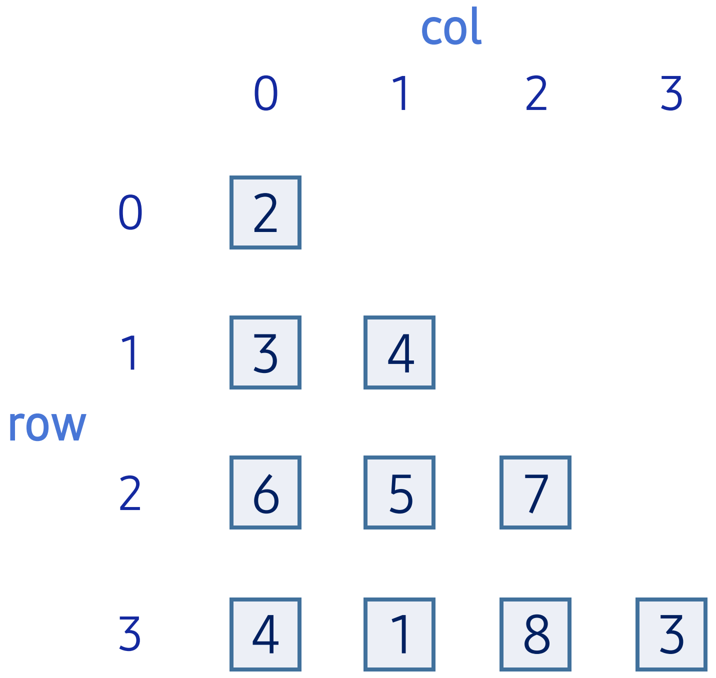
2. Triangle Path Problem

2.1. Finding Minimum Triangle Path with the Divide-and-Conquer

- Let's solve the problem of the minimum sum of triangular travel routes with the split conquest method.
- First of all, a given triangle can be expressed in the form of a list T as shown in the next right figure.

```
      2
     3 4
    6 5 7
   4 1 8 3
```

T=

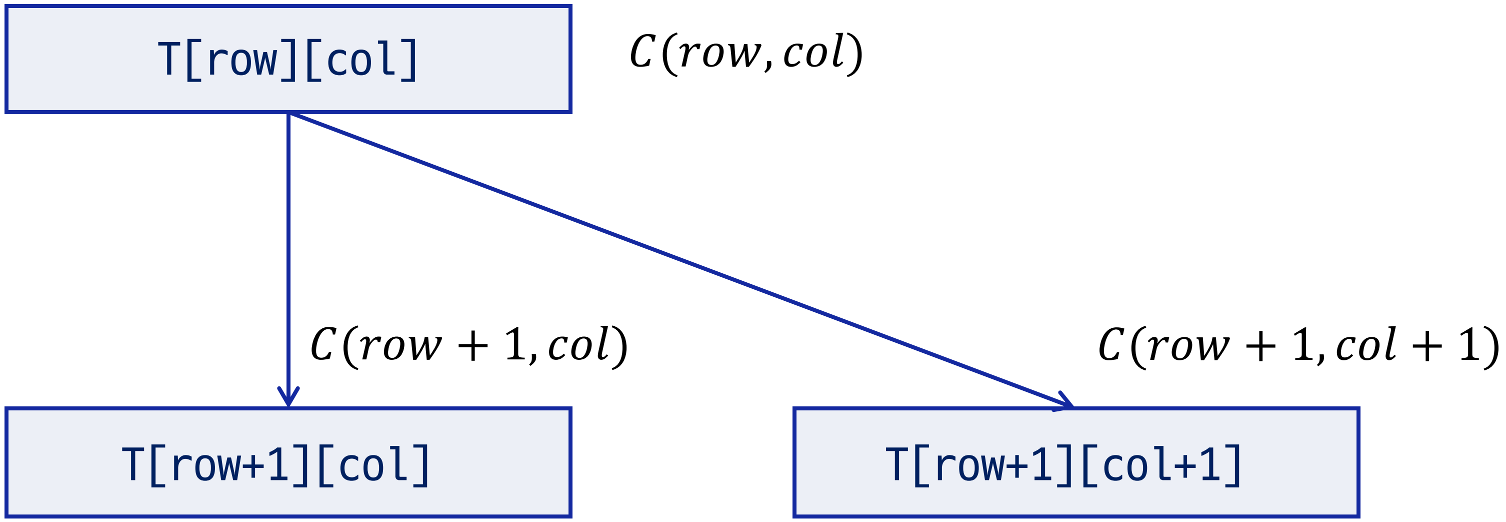


row \ col	0	1	2	3
0	2			
1	3	4		
2	6	5	7	
3	4	1	8	3

2. Triangle Path Problem

2.1. Finding Minimum Triangle Path with the Divide-and-Conquer

- Let C (row, col) be the sum of the minimum paths from the row and col positions in the list T of the list.



2. Triangle Path Problem

2.1. Finding Minimum Triangle Path with the Divide-and-Conquer

The sum of the minimum paths C (row, col) has the following recursive relationship.

- If the value of the row is equal to n, the sum of the paths is zero because it has arrived at the bottom of the triangle.
- If the value of the row is not n, the number T[row][col] of the current location becomes cost.
- Therefore, a path having a smaller cost may be selected from the sum of the minimum paths at the bottom and bottom right positions of the current position.

$$C(row, col) = T[row][col] + \min(C(row+1, col), C(row+1, col+1))$$

$$C(n, col) = 0$$

2. Triangle Path Problem

2.2. Implementation and Coding

Using the recursive relationship analyzed above, the find_minimum() function to which recursion is applied can be implemented as shown below.

```
1 def find_minimum(row, col, triangle):
2     if row == len(triangle):
3         return 0
4     else:
5         minimum = min(find_minimum(row + 1, col, triangle),
6                       find_minimum(row + 1, col + 1, triangle))
7         return triangle[row][col] + minimum
```

Line 1-3

- The input parameter of the find_minimum() function is the list 'triangle' indicating the current location of row, col, and list.
- Since the termination condition of the recursive function arrives at the bottom of the triangle, the values of row and len are the same.
- In this case, we return 0 to the sum of the paths.

2. Triangle Path Problem

2.2. Implementation and Coding

```python
1 def find_minimum(row, col, triangle):
2     if row == len(triangle):
3         return 0
4     else:
5         minimum = min(find_minimum(row + 1, col, triangle),
6                       find_minimum(row + 1, col + 1, triangle))
7         return triangle[row][col] + minimum
```

Line 4-6

- If the bottom of the triangle has not yet arrived, the sum of the two paths is calculated by recursion.
- At this time, a smaller value must be selected from the sum of the paths after moving to the bottom and bottom right.

2. Triangle Path Problem

2.2. Implementation and Coding

```
1 def find_minimum(row, col, triangle):
2     if row == len(triangle):
3         return 0
4     else:
5         minimum = min(find_minimum(row + 1, col, triangle),
6                       find_minimum(row + 1, col + 1, triangle))
7         return triangle[row][col] + minimum
```

Line 7

- Through recursive calls, a smaller value was selected as minimum among the bottom or bottom right paths.
- The sum of the minimum paths at the current location is the value that adds the cost triangle [row][col] of the current path to this minimal.

2. Triangle Path Problem

2.3. Running the Code

When the variable triangle is entered as a list of lists as the following, the top position becomes triangle[0][0], so that the values of row and col are 0, 0, respectively, and the find_minimum() function can be called.

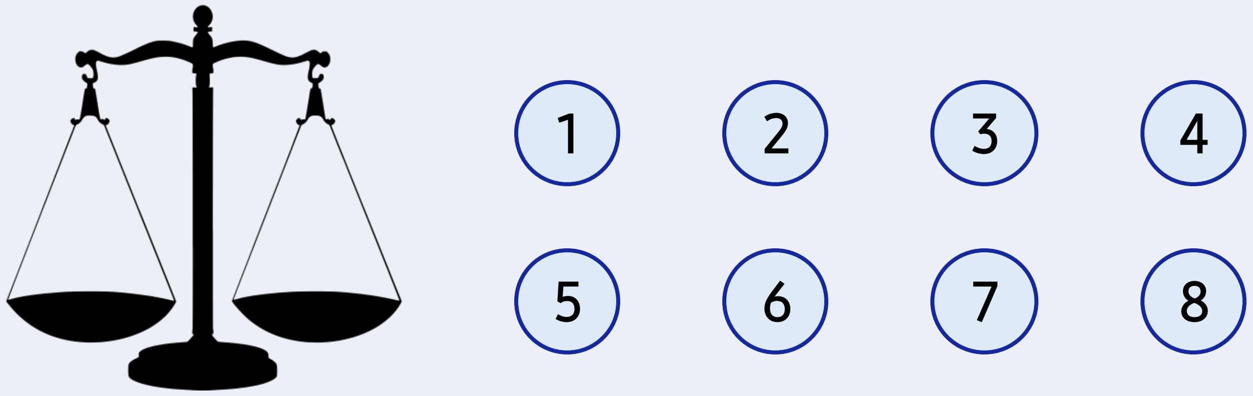
```
triangle = [
    [2],
    [3, 4],
    [6, 5, 7],
    [4, 1, 8, 3]
]
minimum = find_minimum(0, 0, triangle)
print("The minimum cost is ", minimum)
```

```
The minimum cost is  11
```

| Pop quiz

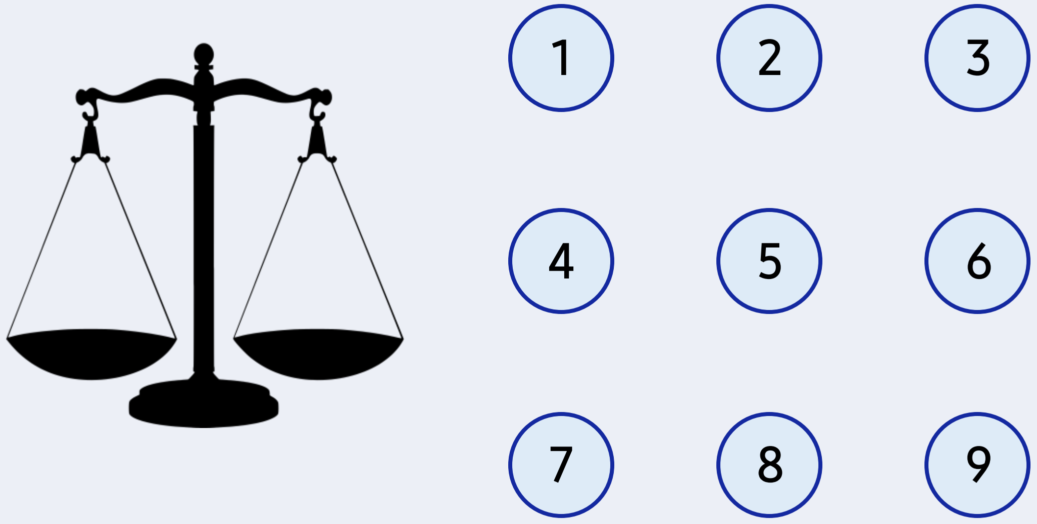
Pop quiz

Q1. Suppose there are eight identical-looking coins numbered from 1 to 8. Of these, only one coin is heavier than the other. If you can weight the cions on a two-arm scale, design an algorithm to pick out one heavy coin. At least how many times should both arm scales be used in order to pick out the coin?



Pop quiz

Q2. In the previous question, suppose there are nine identical-looking coins numbered from 1 to 9.
In this case, design an algorithm to pick out a heavy coin.
At least how many times should both arm scales be used in order to pick out the coin?



Pair programming

Pair Programming Practice

Guideline, mechanisms & contingency plan

Preparing pair programming involves establishing guidelines and mechanisms to help students pair properly and to keep them paired. For example, students should take turns "driving the mouse." Effective preparation requires contingency plans in case one partner is absent or decides not to participate for one reason or another. In these cases, it is important to make it clear that the active student will not be punished because the pairing did not work well.

Pairing similar, not necessarily equal, abilities as partners

Pair programming can be effective when students of similar, though not necessarily equal, abilities are paired as partners. Pairing mismatched students often can lead to unbalanced participation. Teachers must emphasize that pair programming is not a "divide-and-conquer" strategy, but rather a true collaborative effort in every endeavor for the entire project. Teachers should avoid pairing very weak students with very strong students.

Motivate students by offering extra incentives

Offering extra incentives can help motivate students to pair, especially with advanced students. Some teachers have found it helpful to require students to pair for only one or two assignments.

Pair Programming Practice

Prevent collaboration cheating

The challenge for the teacher is to find ways to assess individual outcomes, while leveraging the benefits of collaboration. How do you know whether a student learned or cheated? Experts recommend revisiting course design and assessment, as well as explicitly and concretely discussing with the students on behaviors that will be interpreted as cheating. Experts encourage teachers to make assignments meaningful to students and to explain the value of what students will learn by completing them.

Collaborative learning environment

A collaborative learning environment occurs anytime an instructor requires students to work together on learning activities. Collaborative learning environments can involve both formal and informal activities and may or may not include direct assessment. For example, pairs of students work on programming assignments; small groups of students discuss possible answers to a professor's question during lecture; and students work together outside of class to learn new concepts. Collaborative learning is distinct from projects where students "divide-and-conquer." When students divide the work, each is responsible for only part of the problem solving and there are very limited opportunities for working through problems with others. In collaborative environments, students are engaged in intellectual talk with each other.

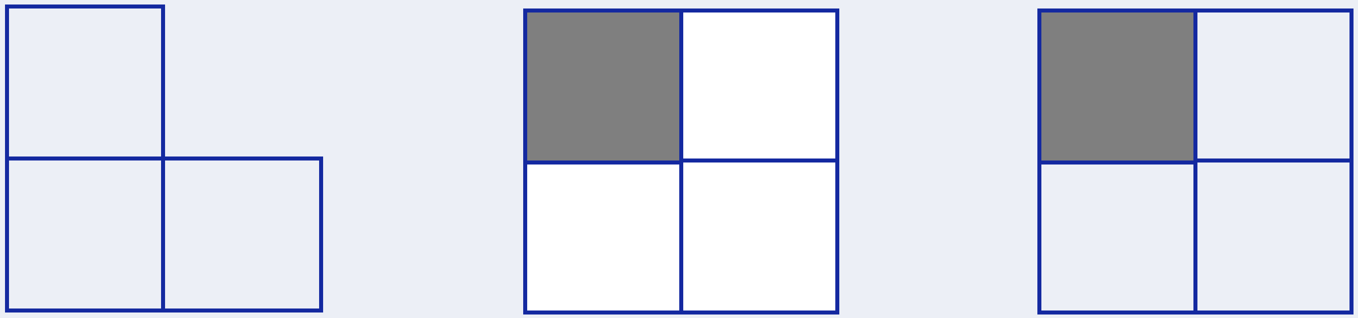
Q1. Design divide-and-conquer algorithm that solves the Tromino Problem.

- Three squares attached to each other is called a tromino.
- There is a checkerboard with m squares connected horizontally and vertically, and one compartment is marked with an X.
- Here, we assume that m is a power of 2.
- We want to fill the checkerboard with a tromino to meet the following conditions.
  - The entire checkerboard should be filled with a tromino.
  - A column marked with an X cannot be covered with a tromino.
  - Trominoes cannot be overlapped.
  - Tromino cannot stick out of the checkerboard.

Q1. Design divide-and-conquer algorithm that solves the Tromino Problem.

Hint

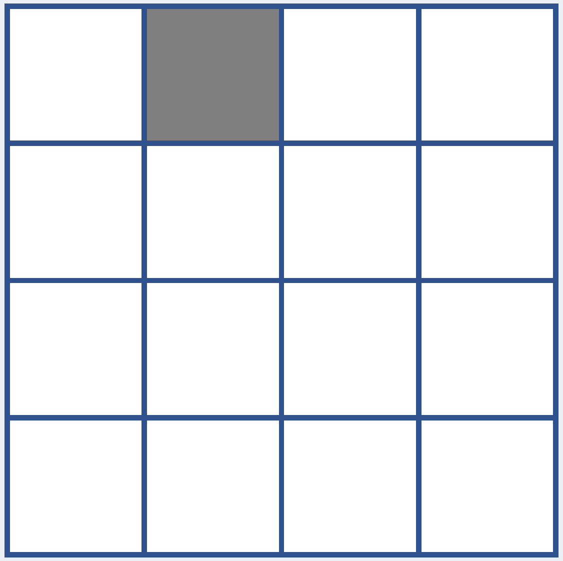
- The tromino consists of three squares. Given an N*N board, find a way to fill all the blanks with a tromino.
- In this case, a portion where X is displayed on the board cannot be covered with a tromino.

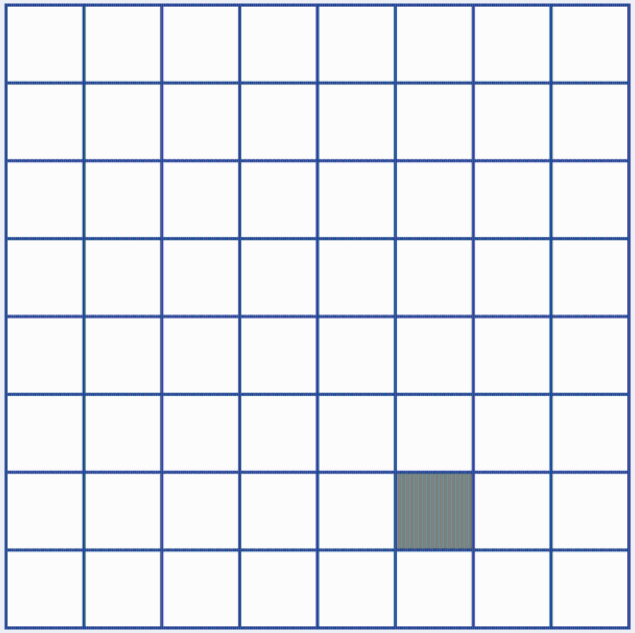


Q1. Design divide-and-conquer algorithm that solves the Tromino Problem.

Hint

- For example, consider a way to cover all compartments with a tromino on a 4*4 board and an 8*8 board.
- Think about dividing a given board into four.

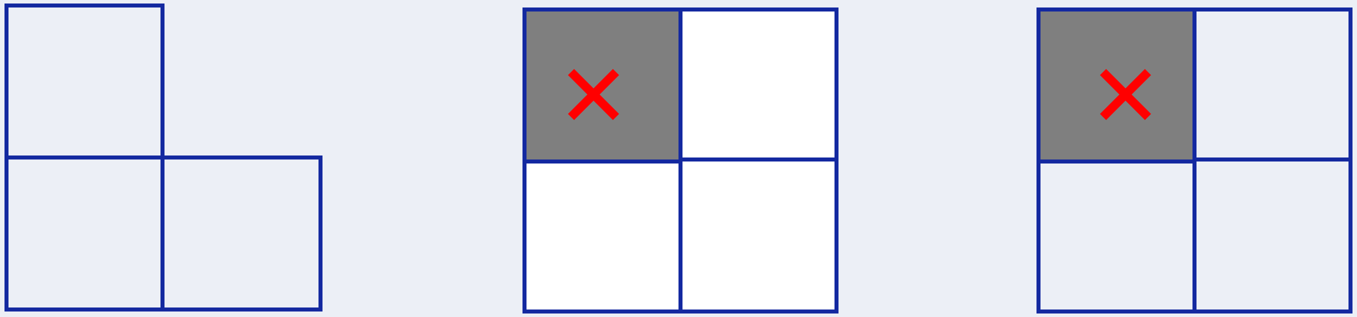




Q1. Design divide-and-conquer algorithm that solves the Tromino Problem.

Hint

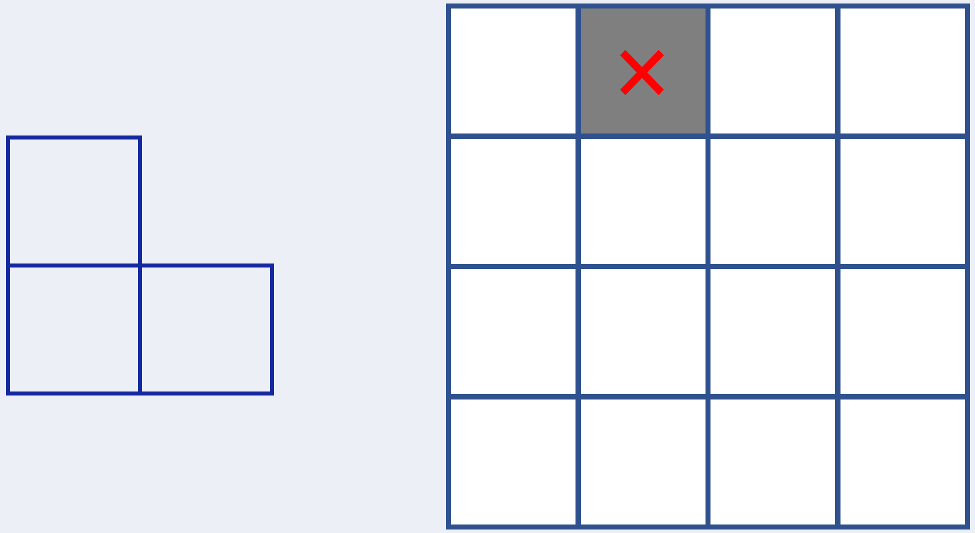
- The tromino consists of three squares. Given an N*N board, find a way to fill all blanks with a tromino.
- In this case, a portion where X is displayed on the board cannot be covered with a tromino.



Q1. Design divide-and-conquer algorithm that solves the Tromino Problem.

Hint

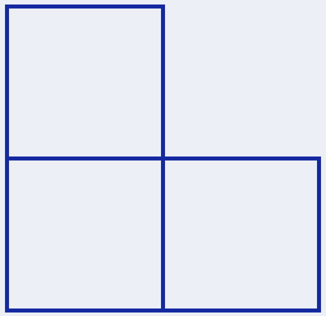
- For example, consider a way to cover all compartments with a tromino on a 4*4 board and an 8*8 board.
- Think about dividing a given board into four.

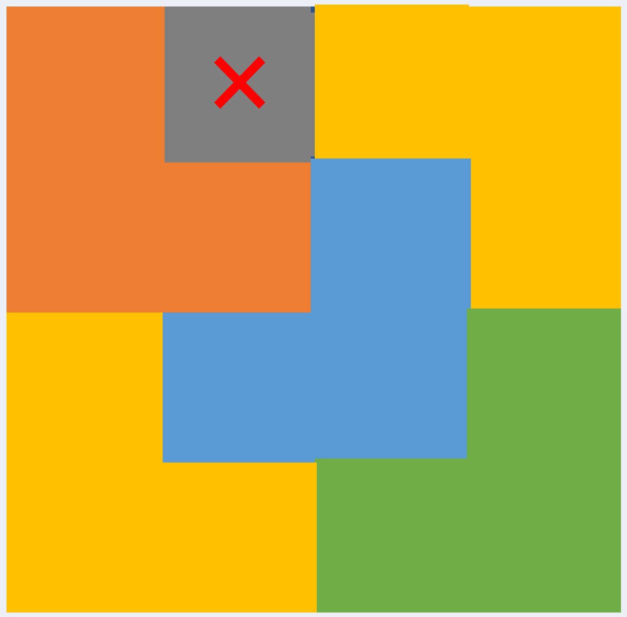


Q1. Design divide-and-conquer algorithm that solves the Tromino Problem.

Hint

The given board can be covered with a tromino as shown below.





Unit 32.

Dynamic Programming

Unit learning objective (1/3)

Learning objectives

- ✓ Learners will be able to understand dynamic programming and calculate the Fibonacci sequence using dynamic programming.
- ✓ Learners will be able to calculate the binomial coefficient using Pascal's triangle.
- ✓ Learners will be able to solve the optimization problem by using recursive relationship in the bottom-up fashion.

Unit learning objective (2/3)

Learning overview

- ✓ Learn how to recursively define the Fibonacci sequence and solve it with dynamic programming.
- ✓ Learn how to calculate Pascal's triangle using the recursive properties of binomial coefficients.
- ✓ Learn how to solve recursive relationships bottom-up using dynamic programming.

Concepts you will need to know from previous units

- ✓ Be able to define partial problems recursively using the divide-and-conquer method.
- ✓ Be able to implement recursive functions using recursive relationships.
- ✓ Be able to store values in a table using lists and dictionaries.

Unit learning objective (3/3)

Keywords

Dynamic Programming

Recursive Property

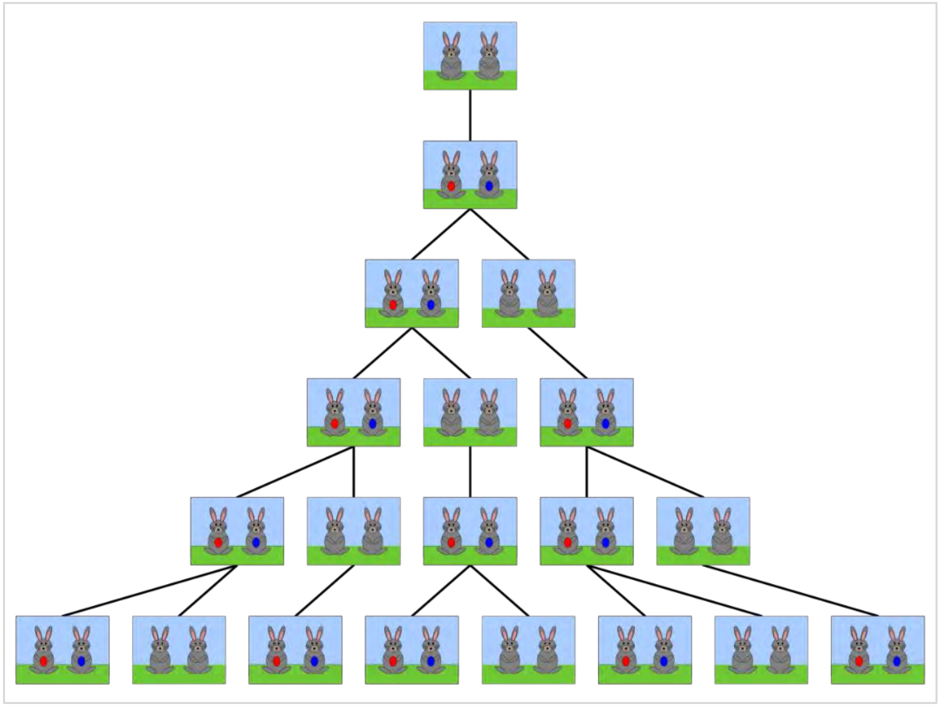
Bottom-Up Approach

Memoization

Mission

1. Real world problem

1.1. How Many Rabbits?

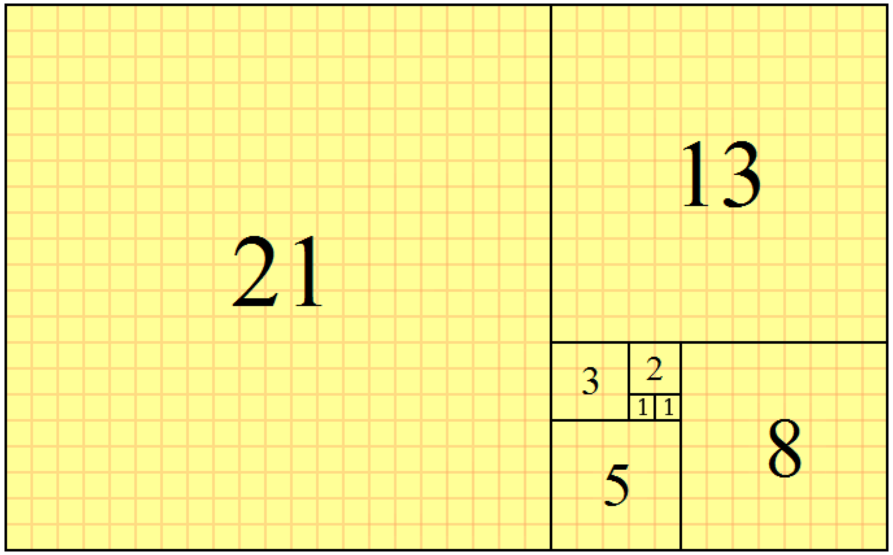


https://en.wikipedia.org/wiki/Fibonacci_number#/media/File:FibonacciRabbit.svg

- Leonardo of Pisa thought of this rule while pondering about the population of rabbits
  - In the first month, only a pair of rabbits exist.
  - In the second month, a pair of rabbits give birth to a pair of baby rabbits.
  - In the third month, two pairs of rabbits give birth to two pair of baby rabbits.
  - Suppose that a pair of reproducible rabbits give birth to a pair of baby rabbits every month, and that rabbits do not die.
- How many total pairs of rabbits will there be in the Nth month?

2. Mission

2.1. Fibonacci Sequence Problem



https://en.wikipedia.org/wiki/Fibonacci_number#/media/File:34*21-FibonacciBlocks.png

- Leonardo of Pisa is known by the name Fibonacci. This problem raised by Fibonacci is widely known as the Fibonacci sequence, a very significant problem in mathematical theory.
- The nth Fibonacci number F_n is a sequence defined by the following initial value and its recurrence as shown below.
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{(n-1)} + F_{(n-1)}$
- The Fibonacci sequence is as follows.
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...
- Let's write an algorithm to find the terms of the nth Fibonacci term.

3. Solution

3.1. Recursive n-th Fibonacci Term

- The problem of obtaining the nth Fibonacci term can be solved recursively by using recursive equations as follows.
- However, these recursive problem-solving methods have serious inefficiency.

```python
def fib1(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib1(n - 1) + fib1(n - 2)
```

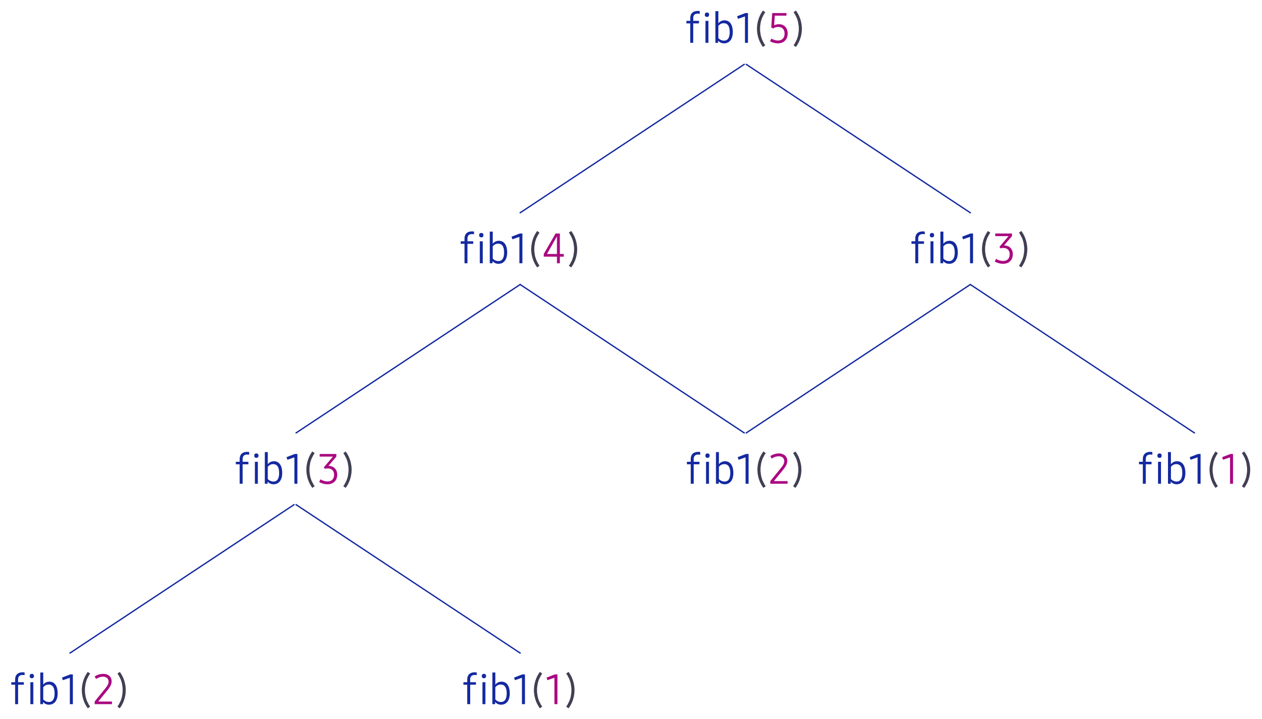
```python
N = int(input("Input a number: "))
print(fib1(N))
```

```
Input a number: 10
55
```

3. Solution

3.1. Recursive n-th Fibonacci Term

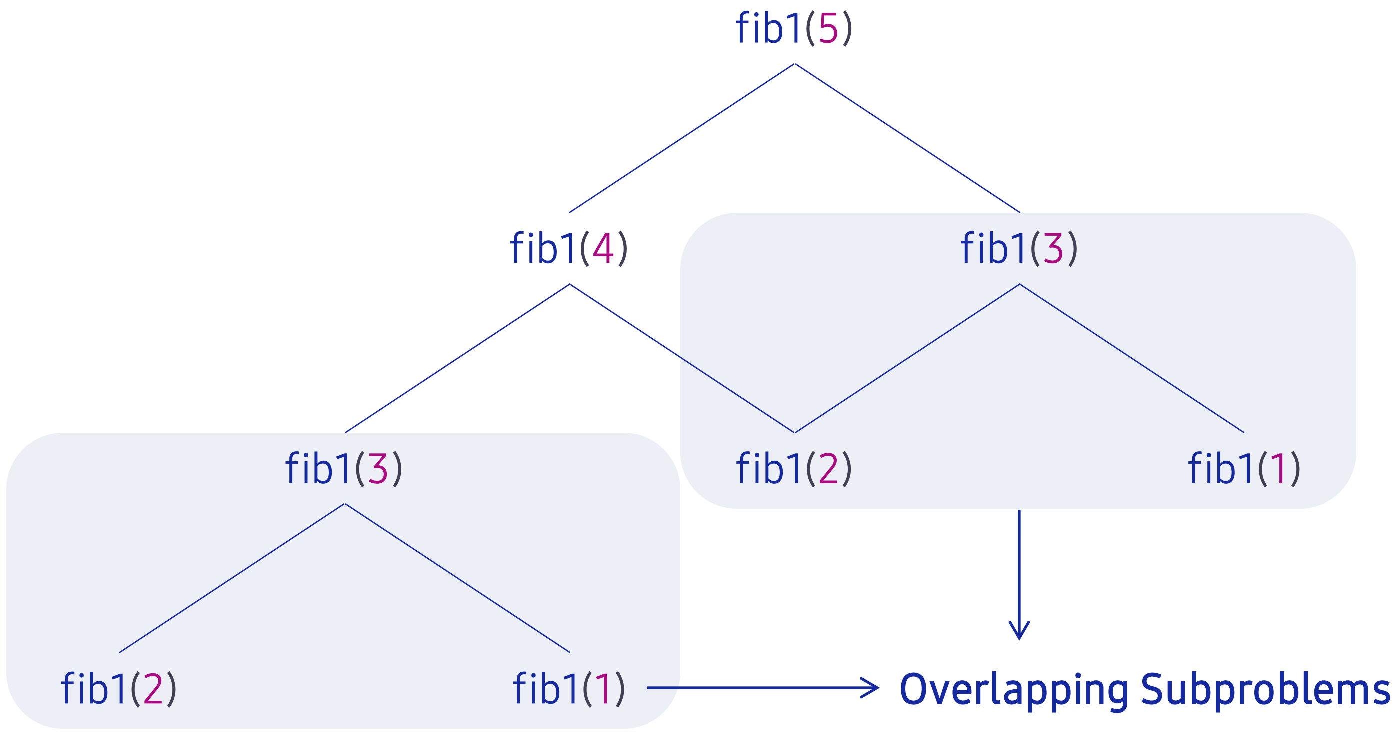
- For example, consider calling fib1(5).
- Recursive functions are called in the following form.



3. Solution

3.1. Recursive n-th Fibonacci Term

- Here, it should be noted that fib1(3), fib1(2), and fib1(1) are called twice, respectively.
- The problem of such duplicative recursive calls is called the overlapping subproblem.



3. Solution

3.2. Finding n-th Fibonacci Term with Dynamic Programming

In order to solve the overlapping subproblem, you can find a bottom-up solution from recursion as shown below and solve it with loops.

```python
F = {0: 0, 1: 1}

def fib2(n):
    if n in F:
        return F[n]
    else:
        F[n] = fib2(n - 1) + fib2(n - 2)
        return F[n]
```

```python
N = int(input("Input a number: "))
print(fib2(N))
```

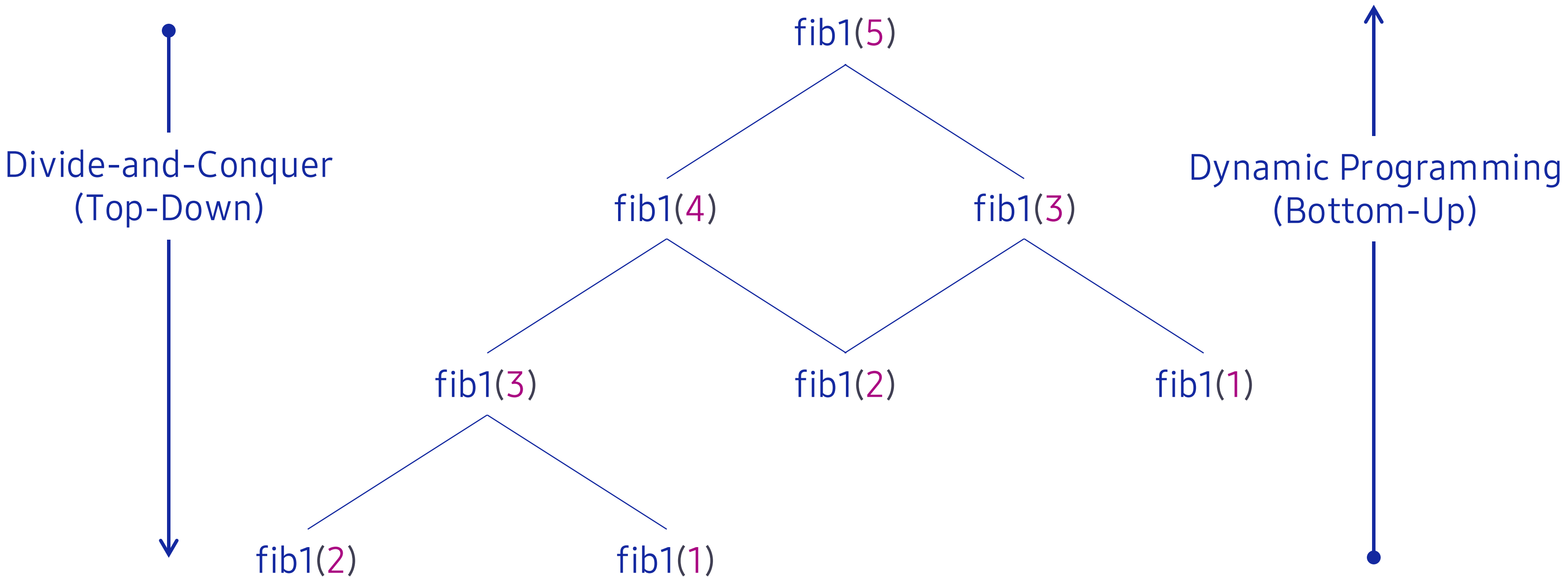
```
Input a number: 10
55
```

Key concept

1. Dynamic Programming

1.1. Dynamic Programming .vs. Divide-and-Conquer

- Dynamic programming is the same method as the divide-and-conquer method, covered in Unit 31, in that both are recursive solutions to a given problem.
- Since the divide-and-conquer method repeats inefficient calculations when the overlapping subproblem occurs, the dynamic programming method stores the subproblems in a variable or table so that the already-calculated subproblems are not recalculated.


fib1(5)
fib1(4)
fib1(3)
fib1(3)
fib1(2)
fib1(1)
fib1(2)
fib1(1)
Divide-and-Conquer (Top-Down)
Dynamic Programming (Bottom-Up)

1. Dynamic Programming

1.2. The Design of Dynamic Programming Algorithms

- Dynamic programming is an algorithmic design technique that analyzes the recursive characteristics of a given problem and then obtains an answer tin a solution in a bottom-up fashion.
- At this time, the technique of storing the answers to the overlapping subproblems in the table in order to solve the overlapping subproblem is called memoization

Focus The development procedure of the dynamic programming algorithm is as follows.

- Establish a recursive equation to calculate the answer to the input case of the problem
- The answer to the entire input case is obtained by a bottom-up method that solves the small input case first.

Focus Dynamic programming uses memoization for bottom-up calculations.

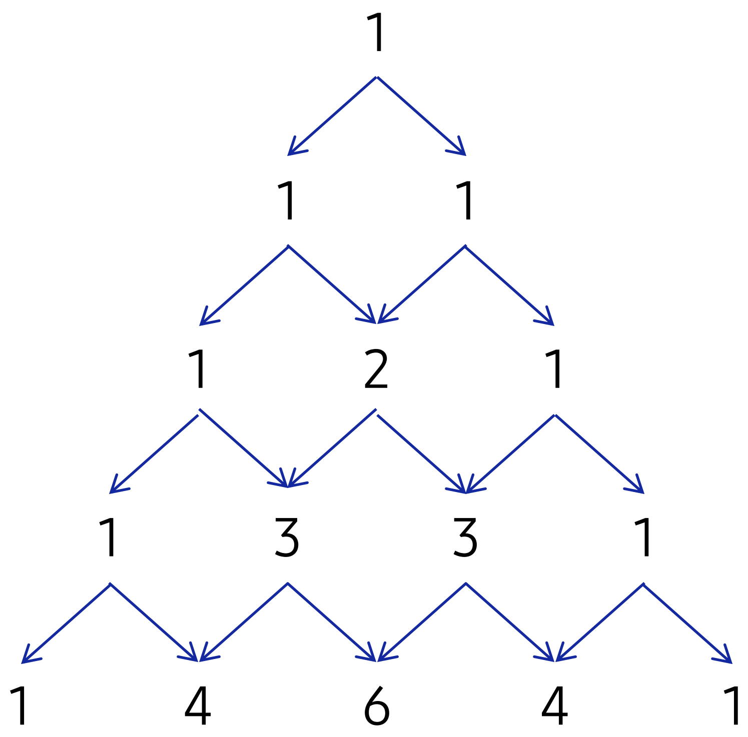
- The answers for the subproblems are stored in a table (or memory).
- When encountering the same subproblem, it is taken directly from the table (or memory) without recalculating.

1. Dynamic Programming

1.3. Pascal's Triangle

- Pascal's triangle is a famous algorithm that applies dynamic programming.
- Pascal's triangle is an array of binary coefficients in mathematics in a triangular shape.

Pascal's Triangle



1. Dynamic Programming

1.3. Pascal’s Triangle

- In Pascal’s triangle, the binomial coefficient has the following recursive properties depending on n and k.

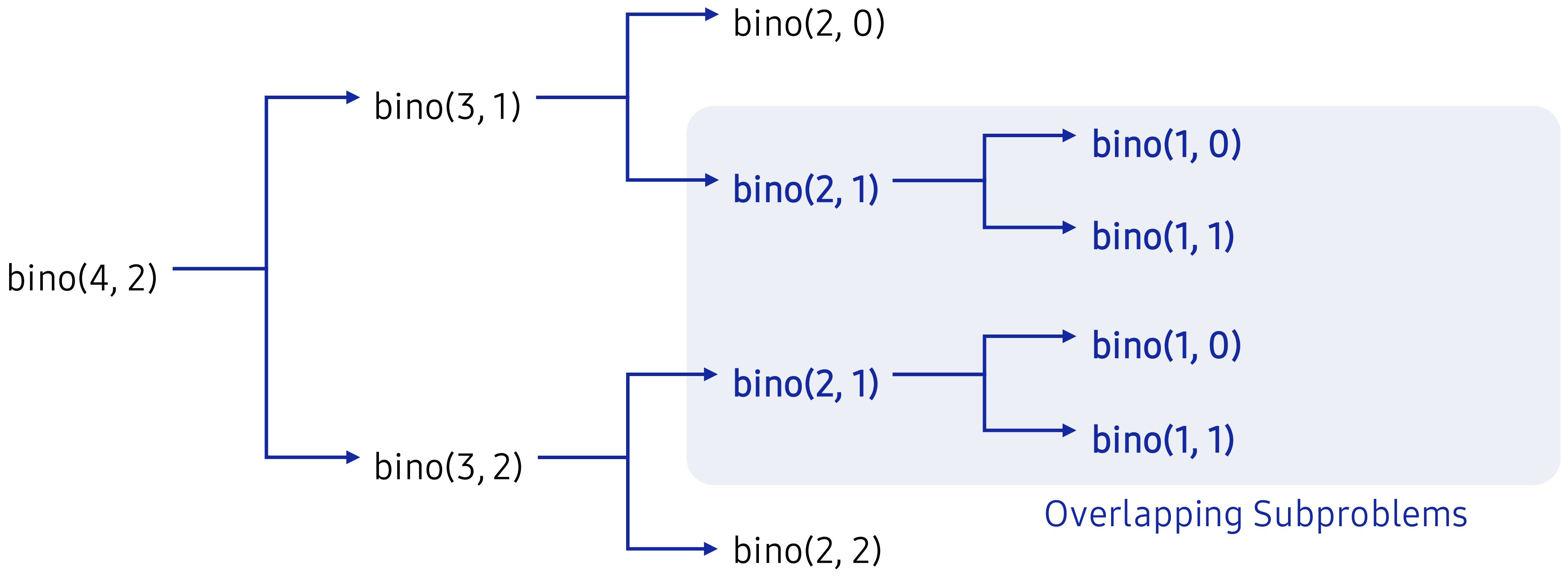
$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \text{ or } k = n \end{cases}$$

$$\begin{tikzcd}
bin(i-1, j-1) \arrow[dr] & & bin(i-1, j) \arrow[dl] \\
& bin(i, j) &
\end{tikzcd}$$

1. Dynamic Programming

1.4. Pascal's Triangle by the Divide-and-Conquer

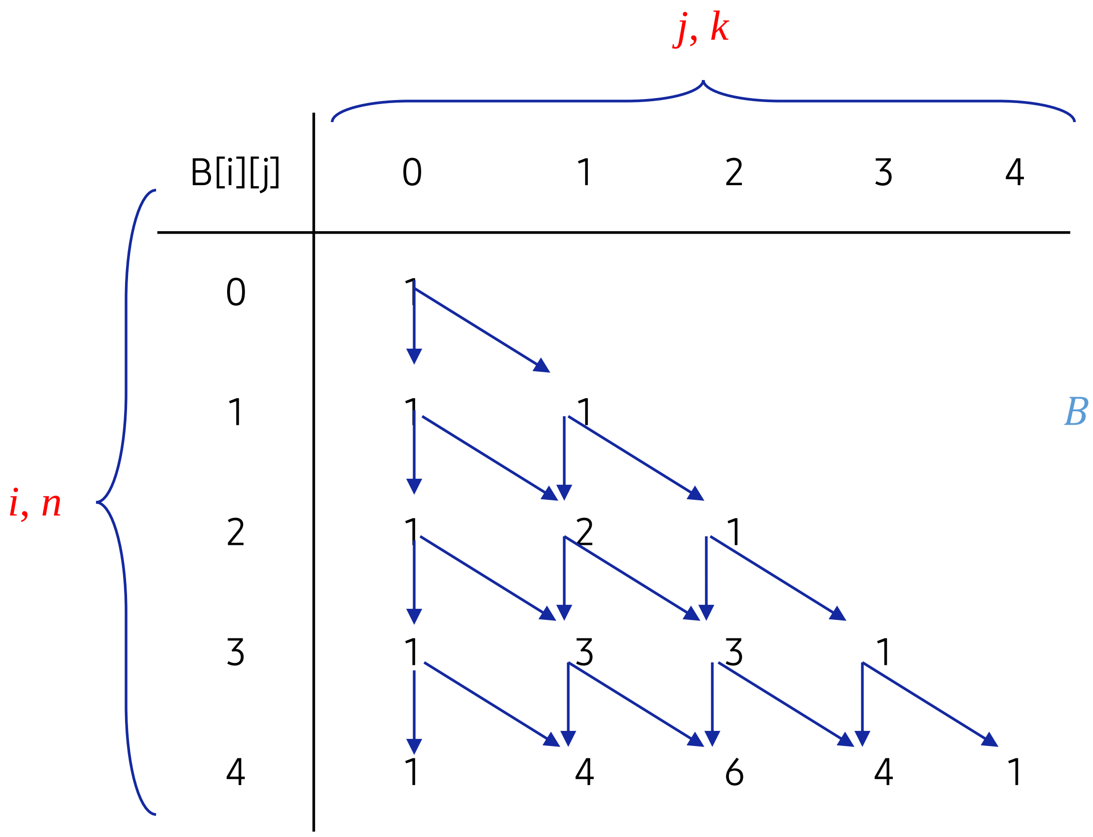
- However, such a recursive solution, like the Fibonacci sequence, causes the following overlapping subproblems.



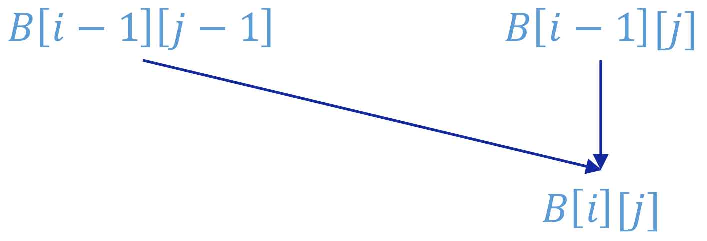
1. Dynamic Programming

1.4. Pascal's Triangle by the Divide-and-Conquer

- In order to apply dynamic programming to Pascal's triangle, a table can be used as shown below.



B[i][j]	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1



$$\begin{tikzcd}
B[i-1][j-1] \arrow[dr] & B[i-1][j] \arrow[d] \\
 & B[i][j]
\end{tikzcd}$$

Let's code

1. Dynamic Programming

1.1. n-th Fibonacci Term by Recursion

The fib1() function is an implementation of the recursive definition of the Fibonacci sequence through a recursive function.

```
1  def fib1(n):
2      if n == 0 or n == 1:
3          return n
4      else:
5          return fib1(n - 1) + fib1(n - 2)
```

Line 2-5

- Line 2-3: The termination condition of the recursive function returns a value of n when n is 0 or 1.
- Line 4-5: The recursive function returns the sum of n-1 and n-2 terms by the recursive definition of the Fibonacci sequence.

1. Dynamic Programming

1.2. n-th Fibonacci Term with Memoization

The fib2() function is implemented through a recursive function as a bottom-up solution by applying a dynamic programming method.

```
1  F = {0: 0, 1: 1}
2  
3  def fib2(n):
4      if n in F:
5          return F[n]
6      else:
7          F[n] = fib2(n - 1) + fib2(n - 2)
8          return F[n]
```

Line 1-5

- F is generated as a dictionary, and values F[0] and F[1] are initialized a 0 and 1, respectively.
- Line 4-5: If n exists in Dictionary F, it is a subproblem that has already been calculated and can return F[n] immediately.

1. Dynamic Programming

1.2. n-th Fibonacci Term with Memoization

```
1  F = {0: 0, 1: 1}
2
3  def fib2(n):
4      if n in F:
5          return F[n]
6      else:
7          F[n] = fib2(n - 1) + fib2(n - 2)
8          return F[n]
```

Line 6-8

- If the value of n does not exist in Dictionary F, it is a partial problem that has not yet been calculated, so a recursive call is made.
- Line 7-8: The value calculated through the recursive call is stored in F so as not to be recalculated and then returned as F[n].

1. Dynamic Programming

1.2. n-th Fibonacci Term with Memoization

- The fib3() function implements memoization using a list instead of a dictionary.
- In this case, the value of the element of the Fibonacci term, which is not calculated, may be initialized to -1 to determine whether there is a duplicate calculation.

```
1 def fib3(F, n):
2     if n <= 1:
3         return F[n]
4     else:
5         if F[n] < 0:
6             F[n] = fib3(F, n - 1) + fib3(F, n - 2)
7         return F[n]
```

Line 1-7

- Line 2-3: If n has a value less than 1, return the value of F[n].
- Line 5-7: When the value of F[n] is less than 0, a recursive call is made, or F[n] is returned immediately.

1. Dynamic Programming

1.2. n-th Fibonacci Term with Memoization

When calling the fib3() function, the list F, which stores the value of the uncalculated term as -1, must be given as the input.

```python
def fib3(F, n):
    if n <= 1:
        return F[n]
    else:
        if F[n] < 0:
            F[n] = fib3(F, n - 1) + fib3(F, n - 2)
        return F[n]
```

```python
N = int(input("Input a number: "))
F = [0, 1] + [-1] * (N - 1)
print(fib3(F, N))
```

```
Input a number: 10
55
```

Line 2-3

- The 0th and 1st elements of the list F are initialized to 0 and 1, respectively, and the remaining N – 1 element is initialized to -1.
- Pass F and N to the parameter values of fib3() function.

1. Dynamic Programming

1.3. Pascal's Triangle by Recursion

Using the recursive properties of the binomial coefficient, bin(n, k) can be calculated as follows.

```
1  def bin1(n, k):
2      if k == 0 or n == k:
3          return 1
4      else:
5          return bin1(n - 1, k - 1) + bin1(n - 1, k)
```

```
1  n = int(input("Input the value of n: "))
2  k = int(input("Input the value of k: "))
3  print(f"binomial({n}, {k}) is {bin1(n, k)}")
```

```
Input the value of n: 5
Input the value of k: 3
binomial(5, 3) is 10
```

Line 1-5

- Line 2-3: If k is 0 or n, return 1.
- Line 4-5: If k is not 0 or 1, a recursive call is made using the recursive nature of the binomial coefficient.

1. Dynamic Programming

1.4. Pascal's Triangle by Dynamic Programming

By applying dynamic programming, recursive expressions can be solved in a bottom-up manner using loops as shown below.

```
1 def bin2(n, k):
2     B = [[0] * (i + 1) for i in range(n + 1)]
3     for i in range(n + 1):
4         for j in range(i + 1):
5             if j == 0 or j == i:
6                 B[i][j] = 1
7             else:
8                 B[i][j] = B[i - 1][j - 1] + B[i - 1][j]
9     return B[n][k]
```

Line 1-2

- For the input parameters n and k, Pascal's triangle is configured in the form of a list.
- List B of the list has n + 1 rows, and row I is a list with a length of I + 1, in which all values are initialized to 0.

1. Dynamic Programming

1.4. Pascal's Triangle by Dynamic Programming

```
1  def bin2(n, k):
2      B = [[0] * (i + 1) for i in range(n + 1)]
3      for i in range(n + 1):
4          for j in range(i + 1):
5              if j == 0 or j == i:
6                  B[i][j] = 1
7              else:
8                  B[i][j] = B[i - 1][j - 1] + B[i - 1][j]
9      return B[n][k]
```

Line 3-9

- From the nth row of Pascal's triangle, we compute each binomial coefficient from bottom -up.
- Each binomial coefficient B[i][j]is B[i-1][j-1] and B[i-1[j] is already calculated, so they can be calculated immediately through addition.

1. Dynamic Programming

1.4. Pascal's Triangle by Dynamic Programming

By receiving values of n and k from the user, binomial coefficients bin(n, k) may be output as shown below.

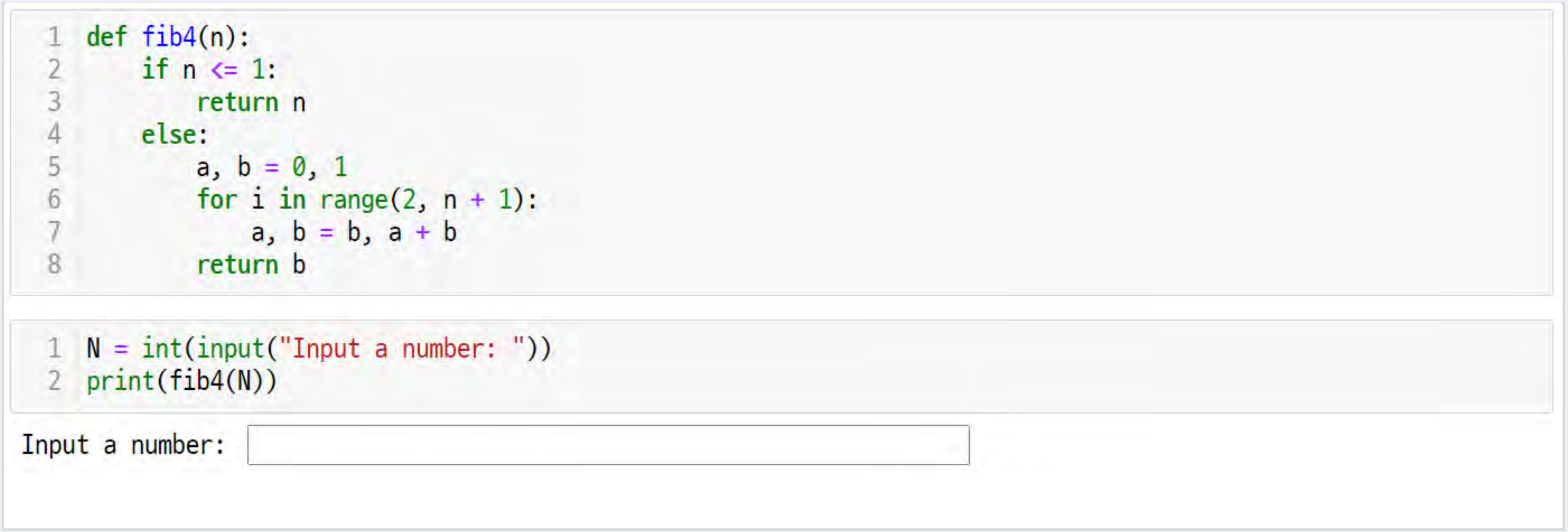
```python
def bin2(n, k):
    B = [[0] * (i + 1) for i in range(n + 1)]
    for i in range(n + 1):
        for j in range(i + 1):
            if j == 0 or j == i:
                B[i][j] = 1
            else:
                B[i][j] = B[i - 1][j - 1] + B[i - 1][j]
    return B[n][k]
```

```python
n = int(input("Input the value of n: "))
k = int(input("Input the value of k: "))
print(f"binomial({n}, {k}) is {bin2(n, k)}")
```

```
Input the value of n: 5
Input the value of k: 3
binomial(5, 3) is 10
```

Pop quiz

Q1. Analyze the execution result of the next code and compare it with the performance of the functions fib1(), fib2(), and fib3().



```python
def fib4(n):
    if n <= 1:
        return n
    else:
        a, b = 0, 1
        for i in range(2, n + 1):
            a, b = b, a + b
        return b
```

```python
N = int(input("Input a number: "))
print(fib4(N))
```

```
Input a number: 
```

Q2. Analyze the execution results of the following code and compare it with the performance of functions bin1() and bin2().

```python
def bin3(n, k):
    B = [0] * (n + 1)
    for i in range(n + 1):
        for j in range(n, -1, -1):
            if j == 0 or j == i:
                B[j] = 1
            else:
                B[j] = B[j] + B[j - 1]
    return B[k]
```

```python
for i in range(10):
    for j in range(i + 1):
        print(bin3(i, j), end=" ")
    print()
```

Pair programming

Pair Programming Practice

Guideline, mechanisms & contingency plan

Preparing pair programming involves establishing guidelines and mechanisms to help students pair properly and to keep them paired. For example, students should take turns "driving the mouse." Effective preparation requires contingency plans in case one partner is absent or decides not to participate for one reason or another. In these cases, it is important to make it clear that the active student will not be punished because the pairing did not work well.

Pairing similar, not necessarily equal, abilities as partners

Pair programming can be effective when students of similar, though not necessarily equal, abilities are paired as partners. Pairing mismatched students often can lead to unbalanced participation. Teachers must emphasize that pair programming is not a "divide-and-conquer" strategy, but rather a true collaborative effort in every endeavor for the entire project. Teachers should avoid pairing very weak students with very strong students.

Motivate students by offering extra incentives

Offering extra incentives can help motivate students to pair, especially with advanced students. Some teachers have found it helpful to require students to pair for only one or two assignments.

Pair Programming Practice

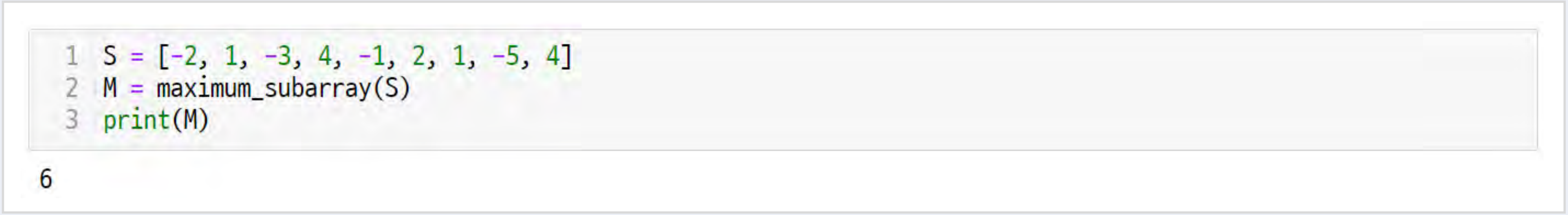
Prevent collaboration cheating

The challenge for the teacher is to find ways to assess individual outcomes, while leveraging the benefits of collaboration. How do you know whether a student learned or cheated? Experts recommend revisiting course design and assessment, as well as explicitly and concretely discussing with the students on behaviors that will be interpreted as cheating. Experts encourage teachers to make assignments meaningful to students and to explain the value of what students will learn by completing them.

Collaborative learning environment

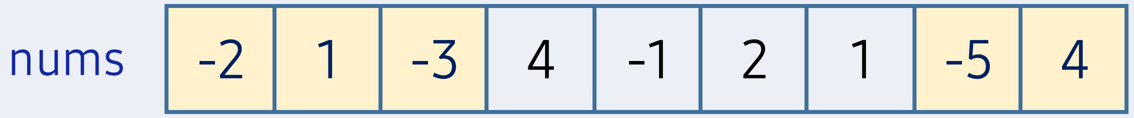
A collaborative learning environment occurs anytime an instructor requires students to work together on learning activities. Collaborative learning environments can involve both formal and informal activities and may or may not include direct assessment. For example, pairs of students work on programming assignments; small groups of students discuss possible answers to a professor's question during lecture; and students work together outside of class to learn new concepts. Collaborative learning is distinct from projects where students "divide-and-conquer." When students divide the work, each is responsible for only part of the problem solving and there are very limited opportunities for working through problems with others. In collaborative environments, students are engaged in intellectual talk with each other.

Q1. Given n integers in a successive one-dimensional array, write an algorithm that finds when the sum of successive vales in the array is maximized.



```
S = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
M = maximum_subarray(S)
print(M)
```

```
6
```



$$maximum = 6$$

Unit 33.

Backtracking

Unit learning objective (1/3)

Learning objectives

- ✓ Learners will be able to understand backtracking techniques and solve N-Queens problems with backtracking.
- ✓ Learners will be able to understand that the backtracking technique is a depth-first search using a functional call stack.
- ✓ Learners will be able to efficiently implement algorithms through pruning in the depth-first search process.

Unit learning objective (2/3)

Learning Overview

- ✓ Learn how to solve the N-Queens problem.
- ✓ Learn the principle of solving the Sum-of-Subsets.
- ✓ Learn how to apply backtracking techniques.

Concepts you will need to know from previous units

- ✓ Be able to understand and implement the definition of recursive functions and the termination conditions of recursions.
- ✓ Be able to make conditional branches using conditional statements.
- ✓ Be able to use Python lists.

Unit learning objective (3/3)

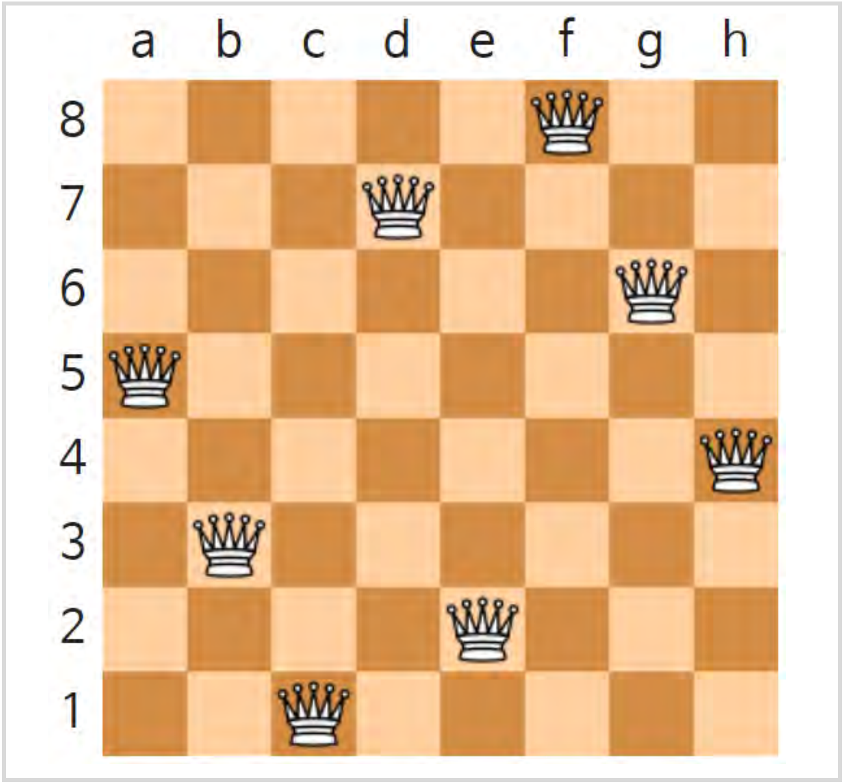
Keywords

- Backtracking
- Depth-First Search
- promising
- pruning

| Mission

1. Real world problem

1.1. The 8-Queens Puzzle



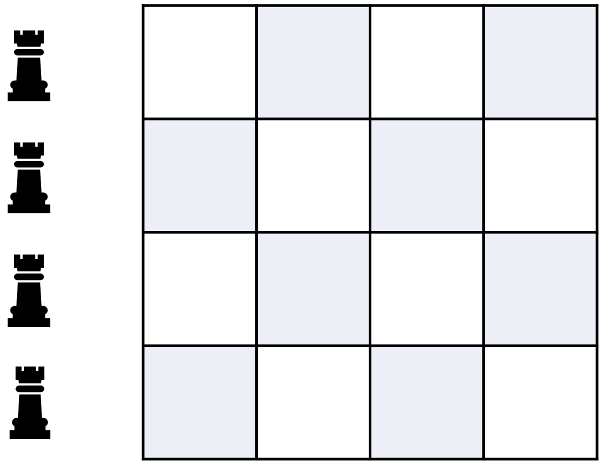
https://en.wikipedia.org/wiki/Eight_queens_puzzle

- The 8-Queens problem is the problem of placing eight queens on an 8*8 chessboard.
- In chess, the queen can move left and right, up and down, and diagonally. Therefore, no other queen can be placed in the same row, column, or diagonal as where another queen is placed.
- To generalize this problem, it becomes an N-Queens problem. That is, it is a problem of arranging N-Queens on the N*N chessboard.
- Let's create an algorithm to solve the N-Queens problem.

2. Mission

2.1. N-Queens Problem

- The N-Queens problem is the problem of placing N queens on the N*N chessboard.
- For example, the 4-Queens problem in the case of N=4 is a problem of placing four queens on a 4*4 chessboard.



3. Solution

3.1. How the N-Queens Solver works

- The N-Queens problem is the problem of placing N-queens on the N*N chessboard.
- For example, the 4-Queens problems in the case of N=4 is a problem of placing four queens on a 4*4 chessboard.

```python
N = int(input("Input the number of queens: "))
n_queens(-1, [-1] * N)
```

```
Input the number of queens: 4
[1, 3, 0, 2]
[2, 0, 3, 1]
```

3. Solution

3.2. N-Queens Solver Final Code

```python
def n_queens(i, col):
    if promising(i, col):
        if i == len(col) - 1:
            print(col)
        else:
            for j in range(len(col)):
                col[i + 1] = j
                n_queens(i + 1, col)
```

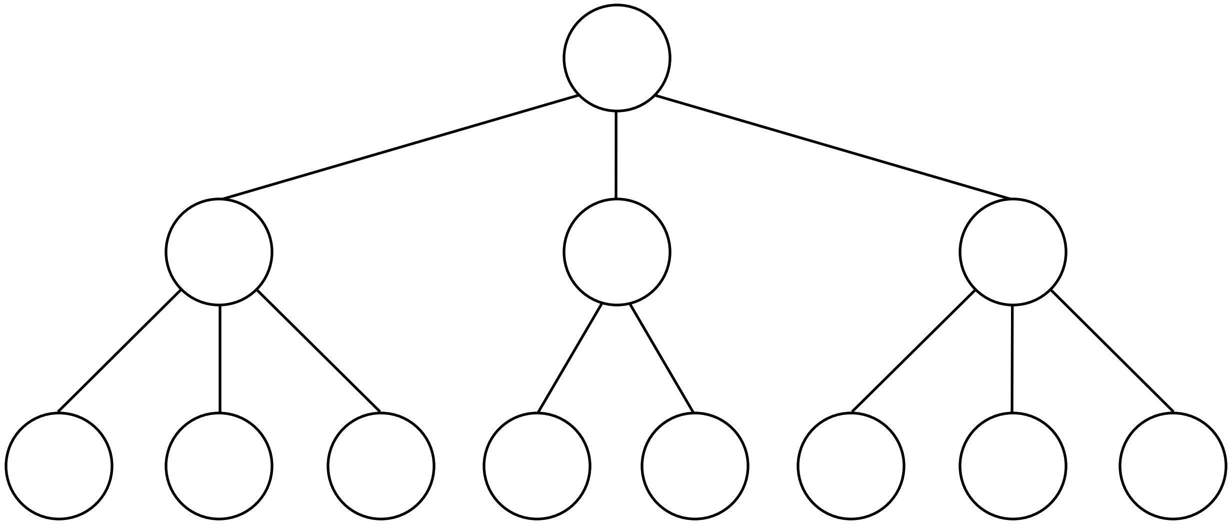
```python
def promising(i, col):
    for k in range(i):
        if col[i] == col[k] or abs(col[i] - col[k]) == (i - k):
            return False
    return True
```

Key concept

1. Backtracking

1.1. Depth First Search

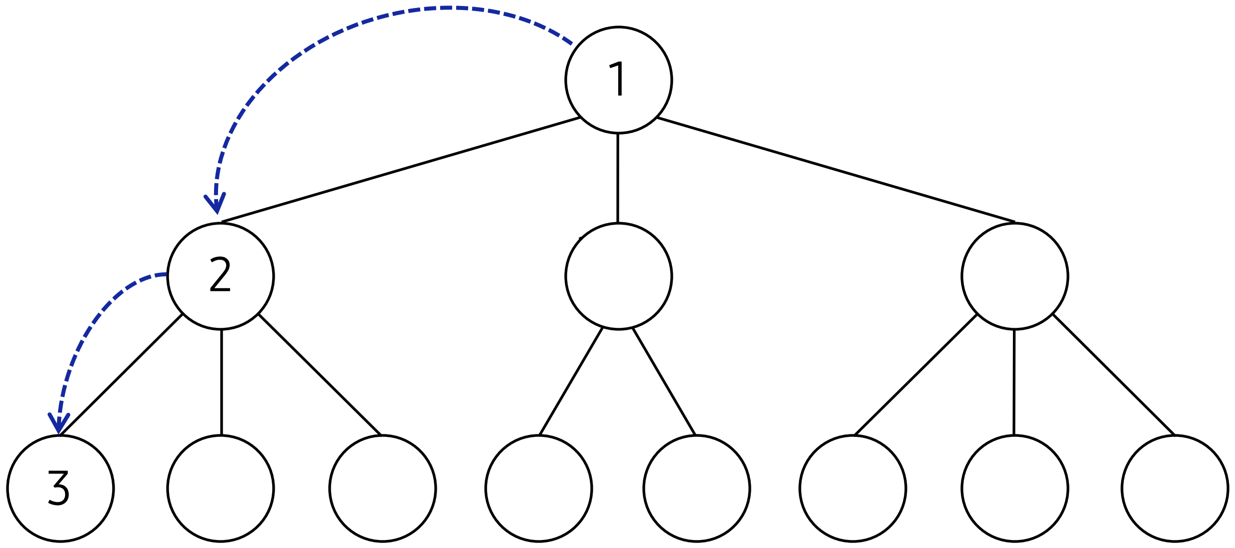
How can one visit all the nodes once in a tree structure as shown below?



1. Backtracking

1.1. Depth First Search

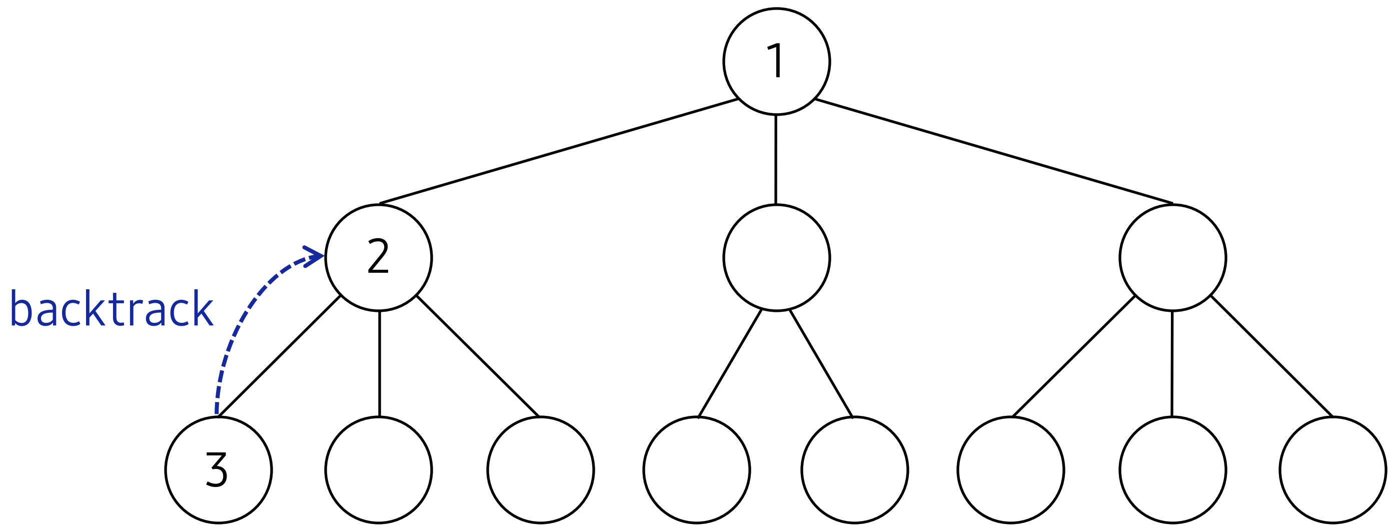
- The Depth-First search refers to a search method in which depth priority is searched as follows.
- First, visit the root node, followed by node 2 and node 3.



1. Backtracking

1.1. Depth First Search

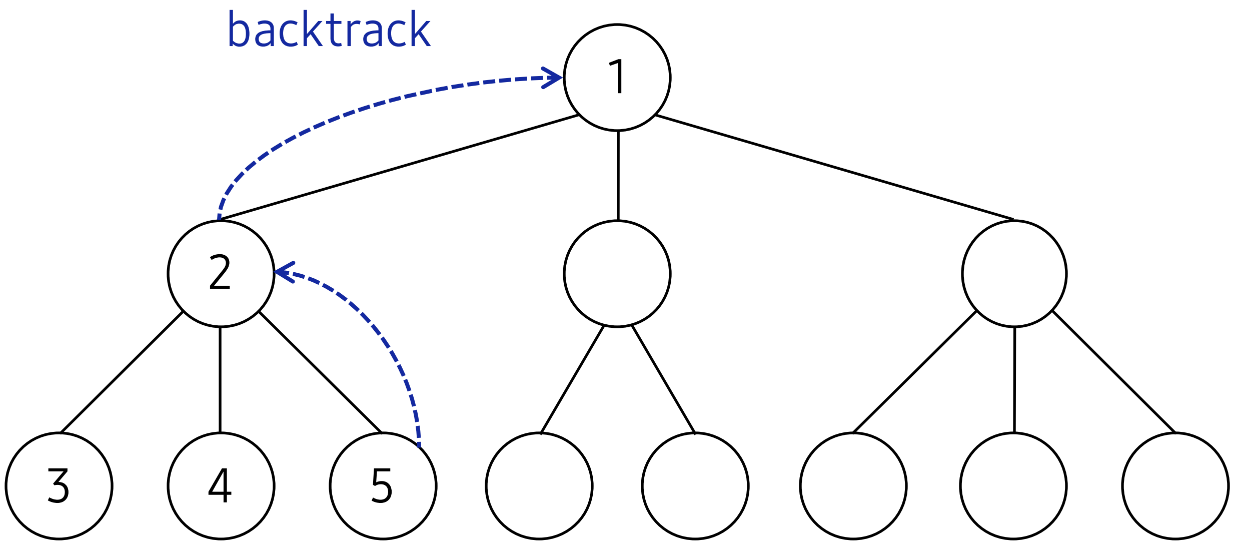
- Node 3 has more nodes to visit, so you have to return the previous node.
- Since there are remaining nodes to visit at node 2, the depth-first search may be continued.



1. Backtracking

1.1. Depth First Search

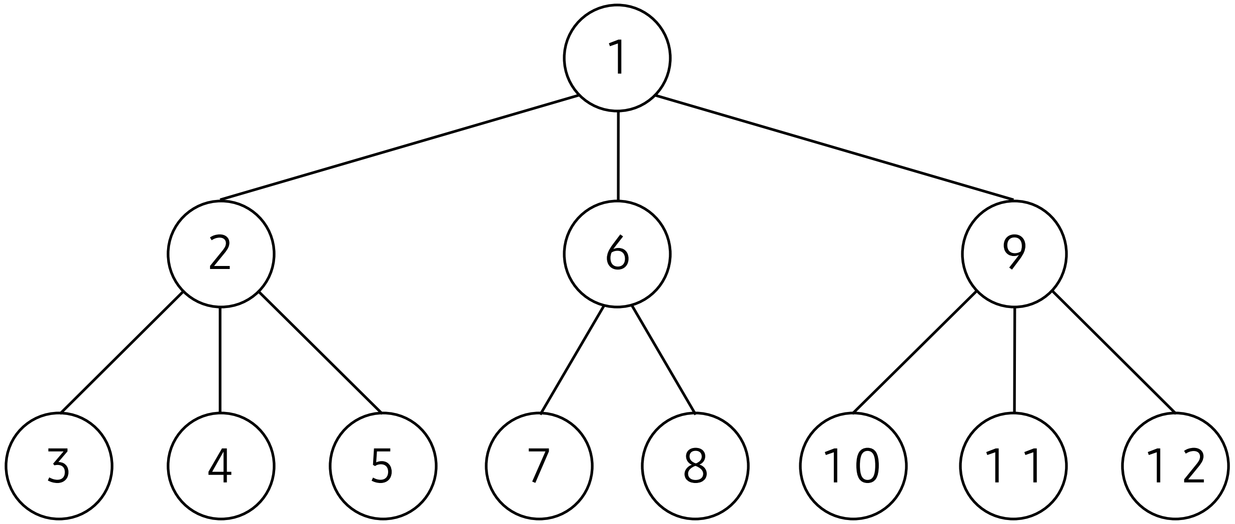
- After visiting nodes 4 and 5, it returns to 2.
- Since all sub-nodes of node 2 have already been visited, they must return to node 1.



1. Backtracking

1.1. Depth First Search

- In this way, all nodes can be visited in the following order.
- The number of each node is the order of nodes visited by DFS.



1. Backtracking

1.1. Depth First Search

The pseudocode of depth-first search can be expressed as follows.

```
def depth_first_search(node v):
    visit v
    for each child u of v:
        depth_first_search(u)
```

1. Backtracking

1.2. Backtracking Strategies

Backtracking is a useful algorithm design strategy that incrementally builds a solution candidate, and backtracks as soon as it determines that the candidate solution cannot possibly lead to a valid solution.

Focus Backtracking solves problems with the following process

- Organize the search space in the form of a tree.
- Explore the search space tree by depth-first search.
- It returns when it is no longer promising.

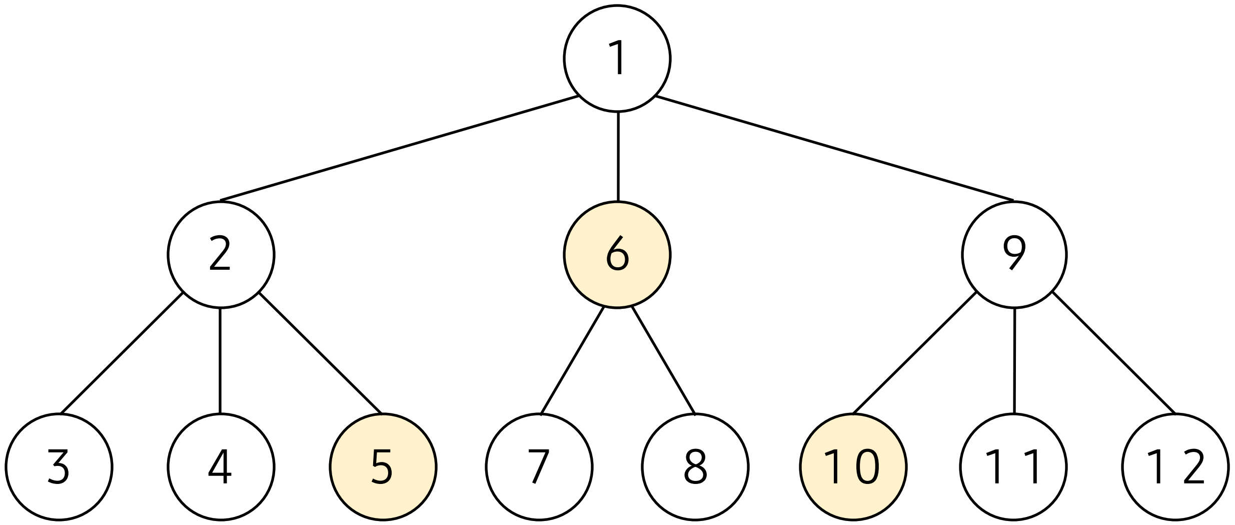
Focus Backtracking uses the pruning technique in the following way.

- Pruning refers to returning without visiting nodes in the subtree which are not promising.
- Promising refers to a state in which a solution may exist in the subtree of a currently visiting node.

1. Backtracking

1.2. Backtracking Strategies

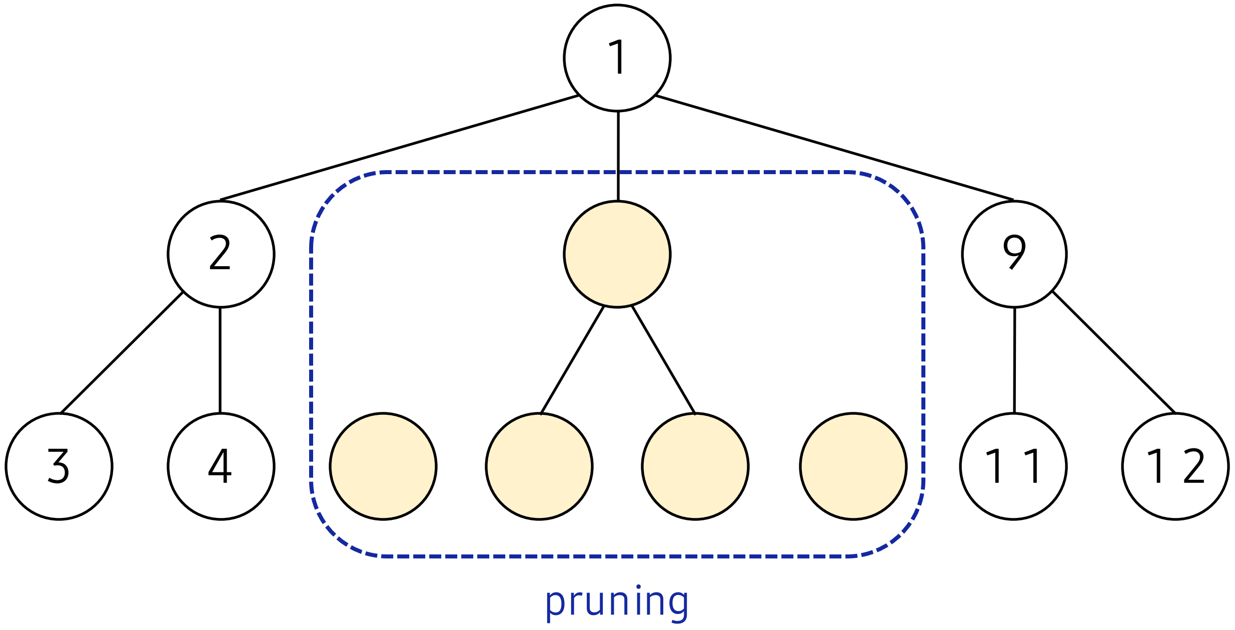
- For example, as shown below, nodes 5, 6, and 10 do not need to visit this node and its sub-nodes if they are not promising.



1. Backtracking

1.2. Backtracking Strategies

- Therefore, the efficiency of the algorithm may be improved by pruning unpromising nodes as follows and thus preventing the visit.

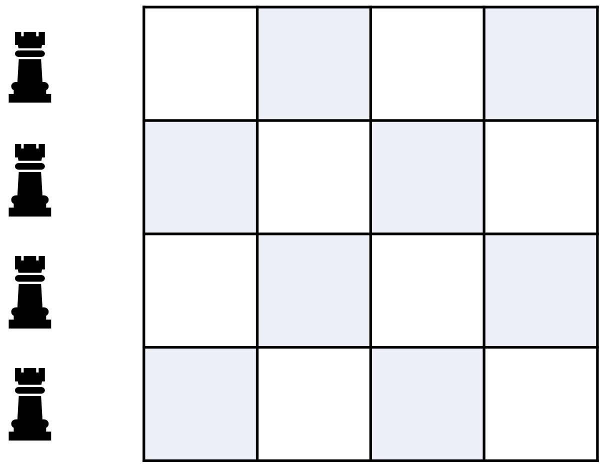


Let's code

1. N-Queens Problem with Backtracking

1.1. Solving N-Queens Problem with Backtracking

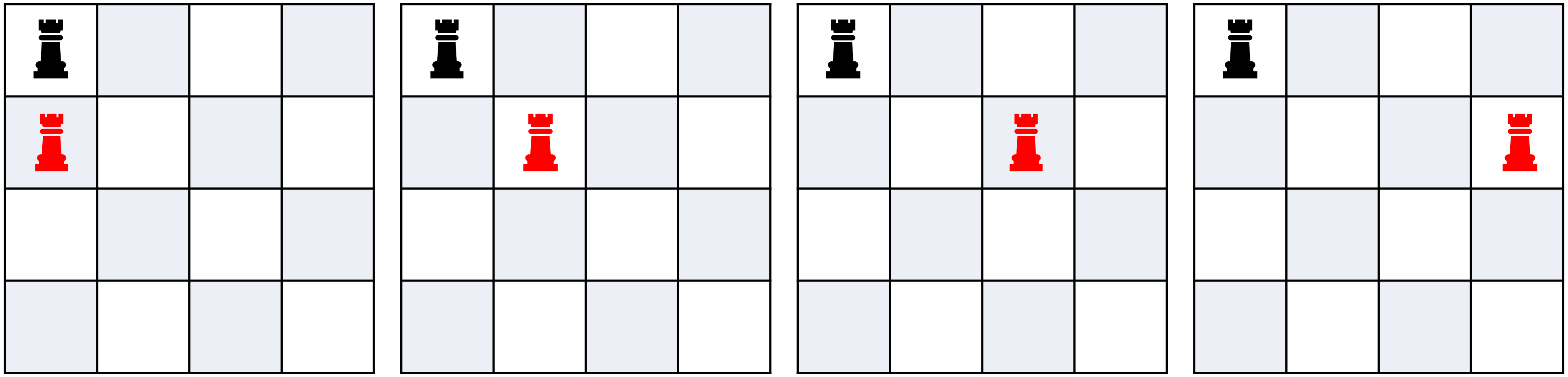
- In order to solve the N-Queens problem, consider the case of N=4. The 4-Queens problem is the problem of placing four queens on a 4*4 chessboard.



1. N-Queens Problem with Backtracking

1.1. Solving N-Queens Problem with Backtracking

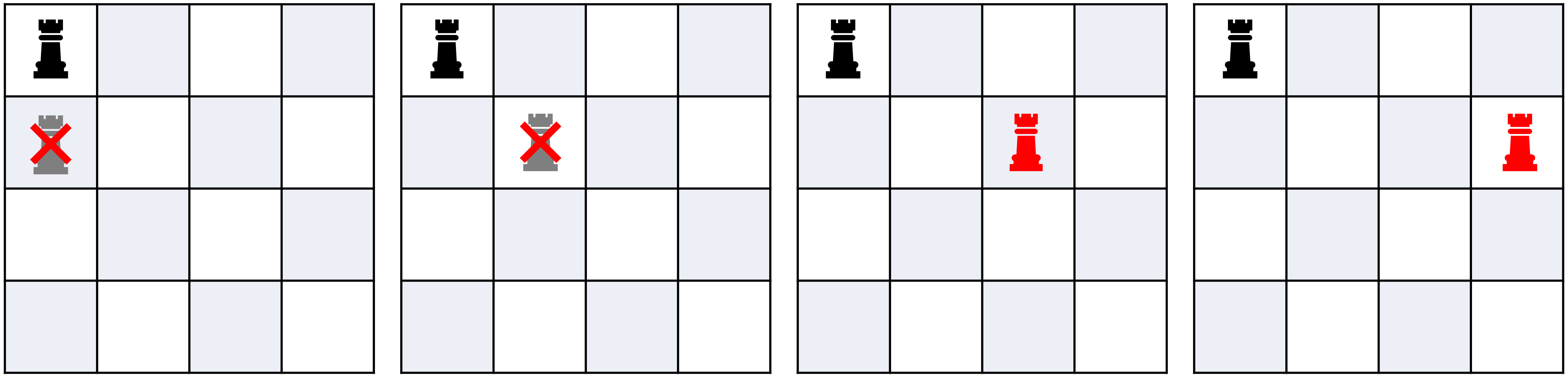
- The easiest way is to place the queen in all configurable locations.
- If a black queen is placed in the position of (0, 0) as shown below, you can try and place the red queen in every possible position in the second row.



1. N-Queens Problem with Backtracking

1.1. Solving N-Queens Problem with Backtracking

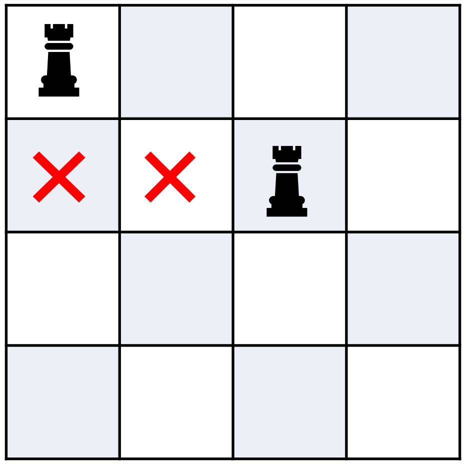
- However, the first and second compartments in the second row cannot be placed.
- This is because the black queen, which has already been placed, cannot coexist with another queen in the same row or diagonal.

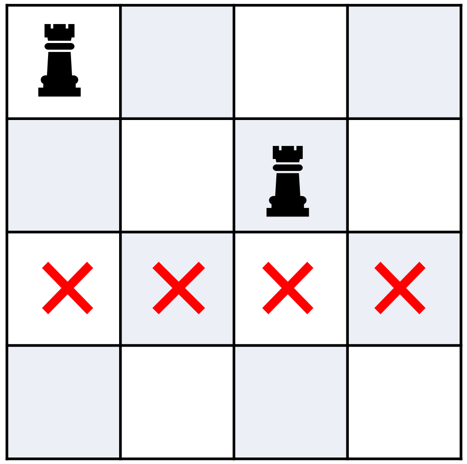


1. N-Queens Problem with Backtracking

1.1. Solving N-Queens Problem with Backtracking

- If, in the second row, the queen is placed in the third column.
- In the third row, queens cannot be placed anywhere.
- This is because it cannot be placed in the same row, column, or diagonal line as the two queens already in position.

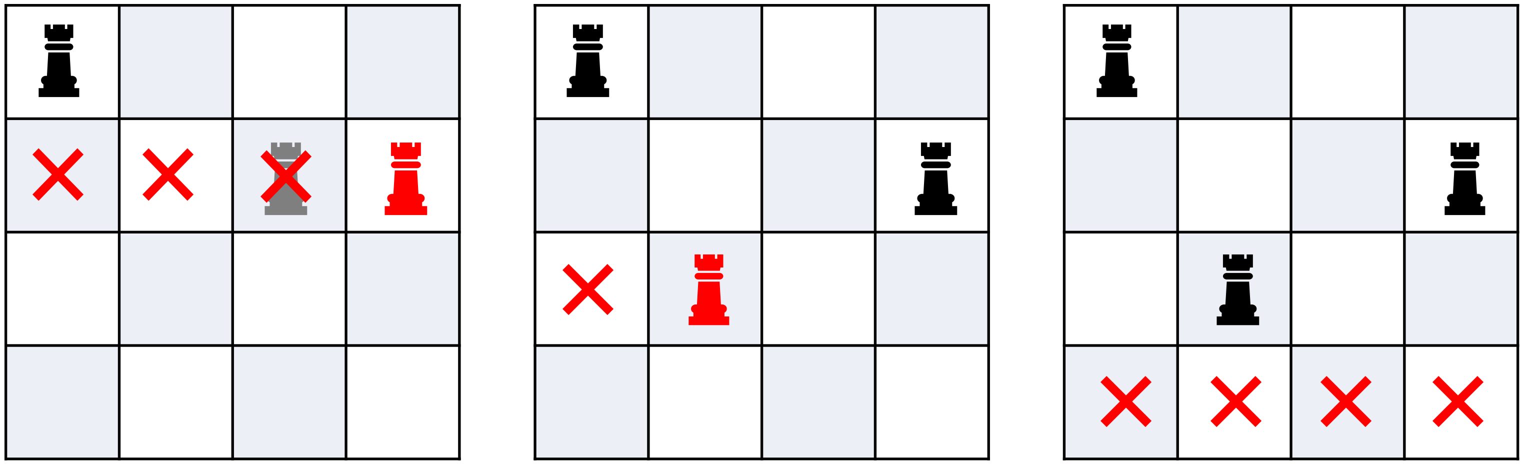




1. N-Queens Problem with Backtracking

1.1. Solving N-Queens Problem with Backtracking

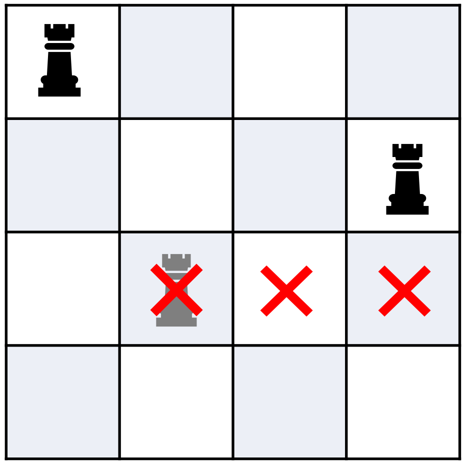
- Let's go back to the second compartment and place the queen in the fourth compartment.
- This time, the queen can be placed in the second column in the third row.
- However, the queen can no longer be placed in the fourth compartment.



1. N-Queens Problem with Backtracking

1.1. Solving N-Queens Problem with Backtracking

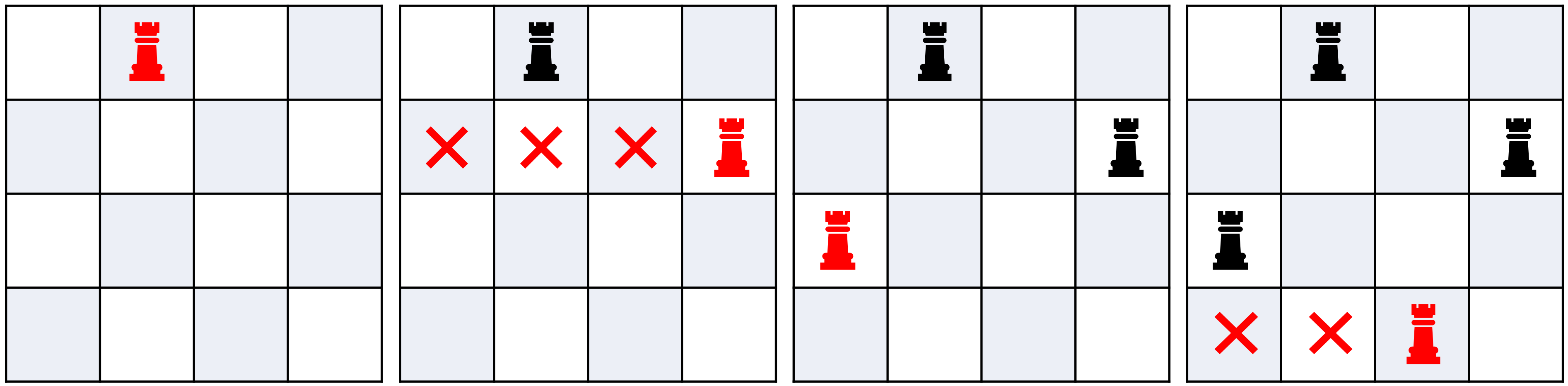
- Let's go back to the third row and place the queen.
- The queen cannot be placed in the third and fourth rows, so we have to go back to the second row again.
- Since we've already tried all the second lines, let's go back to the first line and try the next one.



1. N-Queens Problem with Backtracking

1.1. Solving N-Queens Problem with Backtracking

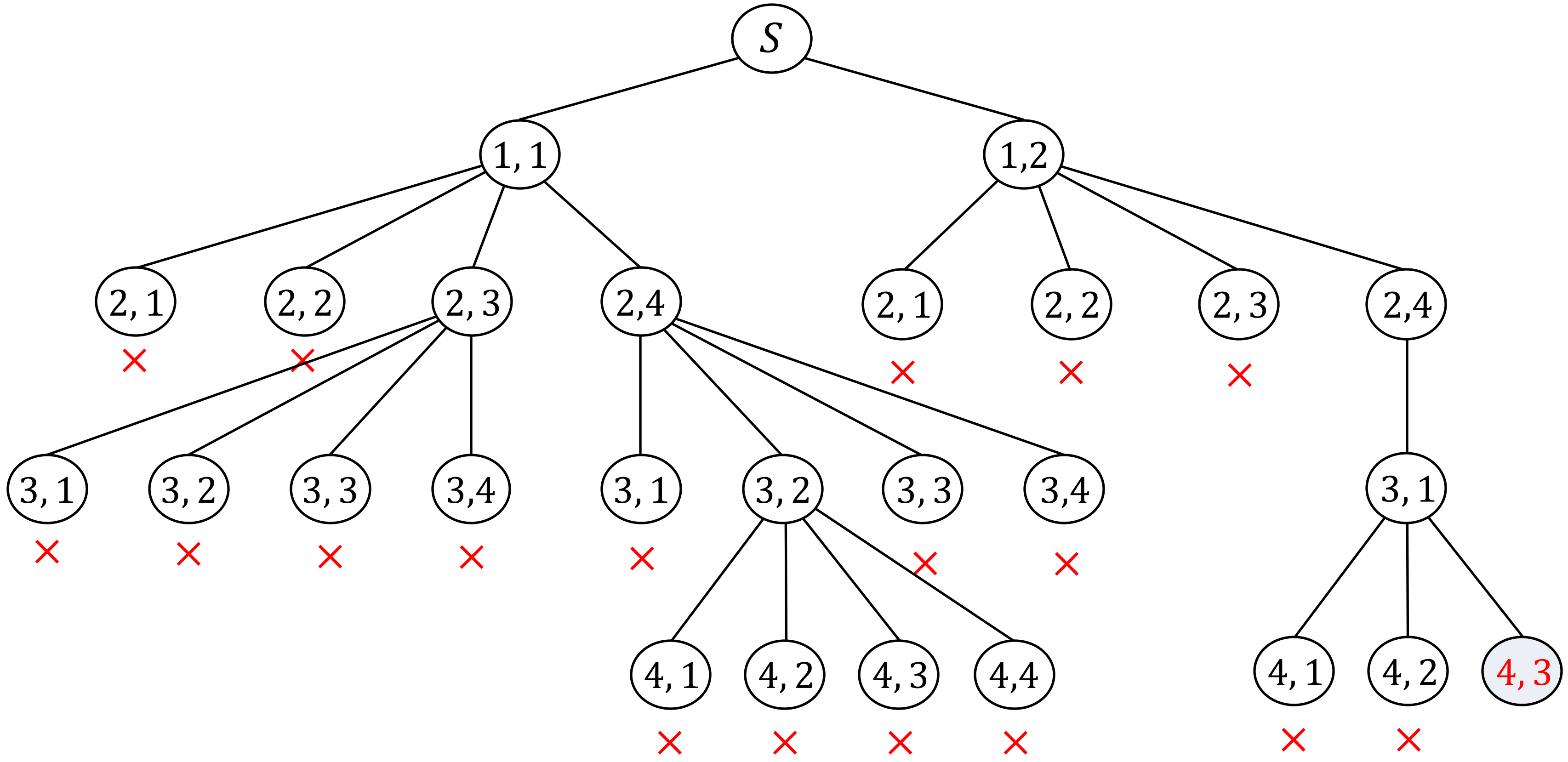
- If the queen is placed on the second column in the first row, the queen can be placed on the fourth column of the second row.
- If you put a queen in the first column of the third row, you can put a queen in the third column of the fourth row.
- We have now resolved this problem because we have all four queens.



1. N-Queens Problem with Backtracking

1.2. Search Space with Pruning

The Search Space Tree of the N-Queens problem can be drawn as follows.



1. N-Queens Problem with Backtracking

1.3. Pseudo-code for N-Queens Solver

The pseudocode for solving this problem can be expressed as follows.

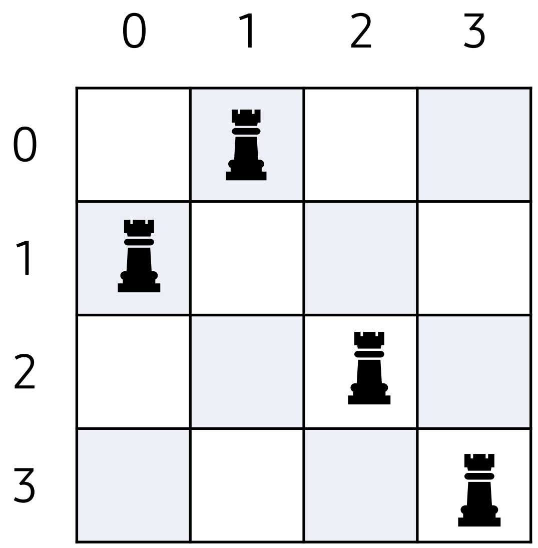
```
def n_queens(node v):
    if (promising(v)):
        if (there is a solution at v):
            write the solution;
        else
            for (each child u of v):
                n_queens(u);
```

1. N-Queens Problem with Backtracking

1.4. Implementation and Coding

- The promising function should check whether two queens exist in the same row, column, or diagonal.
  - Suppose col[i] is the number of the column in the i-th row.
  - That is, col = [1, 0, 2, 3] shows the following state.

col = [1, 0, 2, 3]

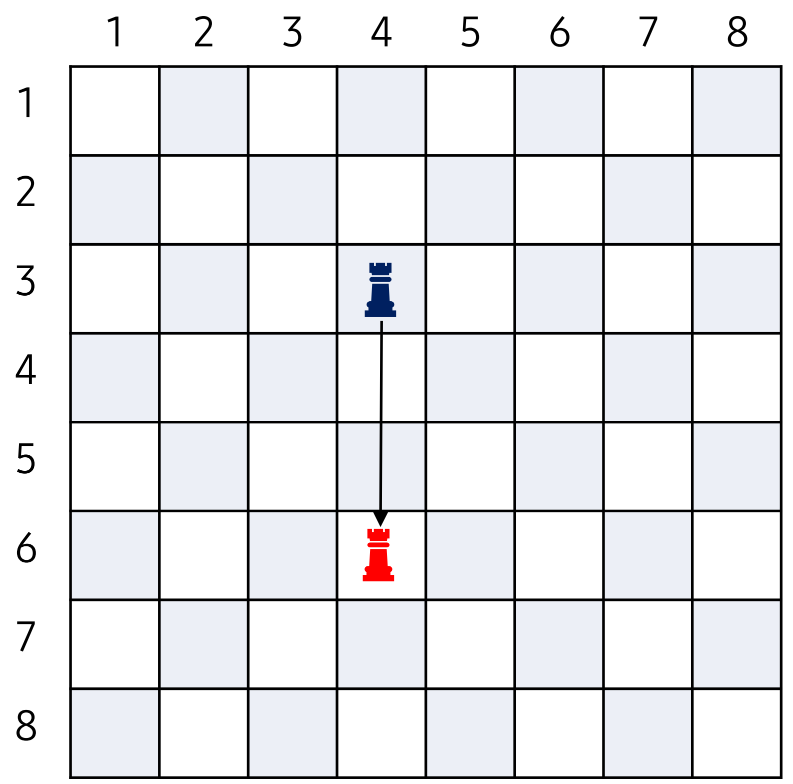


col[0]=1: (0, 1)
col[1]=0: (1, 0)
col[2]=2: (2, 2)
col[3]=3: (3, 3)

1. N-Queens Problem with Backtracking

1.4. Implementation and Coding

- Since we will place queens only in different rows, we can avoid the overlapping of rows.
- In order to check whether the two queens exist in the same column, the following situation must be checked.

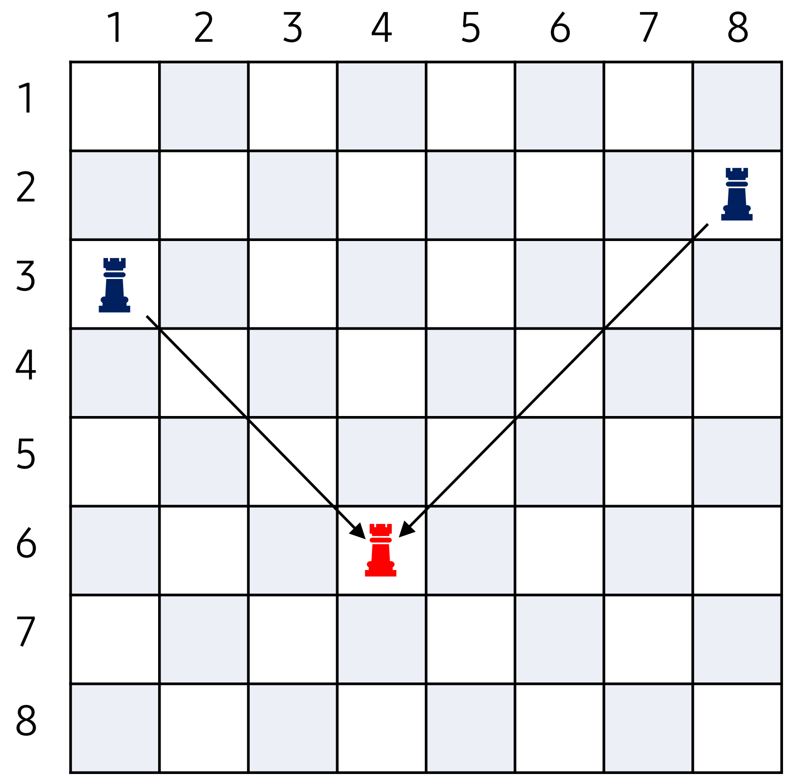


col[i] == col[j]

1. N-Queens Problem with Backtracking

1.4. Implementation and Coding

- In order to determine whether the two queens can exist diagonally, the absolute value of the difference between rows and columns should be checked as follows.



$|col[i] - col[j]| == i - j$

1. N-Queens Problem with Backtracking

1.4. Implementation and Coding

The implementation of the promising function should check whether the i-th column is promising in the given col list.

```
1 def promising(i, col):
2     for k in range(i):
3         if col[i] == col[k] or abs(col[i] - col[k]) == (i - k):
4             return False
5     return True
```

Line 1-2

- Given a col list and index i as input parameters, check whether the i-th column is promising.
- The row on which the queen is already placed is the 0th to i – 1st branch, so it traverses the for loop from 0 to I - 1.

1. N-Queens Problem with Backtracking

1.4. Implementation and Coding

```
1  def promising(i, col):
2      for k in range(i):
3          if col[i] == col[k] or abs(col[i] - col[k]) == (i - k):
4              return False
5      return True
```

Line 3-5

- For each index k, check whether there are already different queens in the same column, or whether there are different queens on the diagonal.
- If there is a queen in the same column or in its diagonal, return False. Otherwise, return True.

1. N-Queens Problem with Backtracking

1.4. Implementation and Coding

The n_queens() function implements a backtracking strategy.

```
1  def n_queens(i, col):
2      if promising(i, col):
3          if i == len(col) - 1:
4              print(col)
5          else:
6              for j in range(len(col)):
7                  col[i + 1] = j
8                  n_queens(i + 1, col)
```

Line 2-4

- Given i and col as input parameters, if the i-th column is not promising, it returns immediately.
- If the i-th column is promising and I is the last row, the solution has been found, so the corresponding col list is input.

1. N-Queens Problem with Backtracking

1.4. Implementation and Coding

```
1 def n_queens(i, col):
2     if promising(i, col):
3         if i == len(col) - 1:
4             print(col)
5         else:
6             for j in range(len(col)):
7                 col[i + 1] = j
8                 n_queens(i + 1, col)
```

Line 5-8

- If the i-th column is promising and I is not the last row, then continue the depth-first search.
- Line 7: Before continuing the depth-first search, place the queen in column i+1st term j of the col list,
- Line 8: and Continue searching for the row I + 1.

1. N-Queens Problem with Backtracking

1.5. Running the N-Queens Solver

After receiving the value of N as an input from the user, the col list initializing the i-th row to -1 and the list elements to -1 is given as a parameter value, and the n_queens() function is called.

```python
N = int(input("Input the number of queens: "))
n_queens(-1, [-1] * N)
```

```
Input the number of queens: 4
[1, 3, 0, 2]
[2, 0, 3, 1]
```

Pop quiz

Q1. If you replace each letter with a number in the following letters, the number corresponding to all five words will be squared. And for each word, the sum of each digit also becomes a square number.
Find the number that each letter represents.

A MERRY XMAS TO ALL

A	M	E	R	R	Y	X	M	A	S	T	O	A	L	L
										8	1			

Q1. If you replace each letter with a number in the following letters, the number corresponding to all five words will be squared. And for each word, the sum of each digit also becomes a square number.
Find the number that each letter represents.

A MERRY XMAS TO ALL

A		M	E	R	R	Y		X	M	A	S		T	O		A	L	L
													8	1				

Hint

```
Note that T and O can be 8 and 1, respectively.
Since 81 is a perfect square number, 81 = 9².
Also, 8 + 1 is a perfect square number, since 9 = 3².
```

Pair programming

Pair Programming Practice

Guideline, mechanisms & contingency plan

Preparing pair programming involves establishing guidelines and mechanisms to help students pair properly and to keep them paired. For example, students should take turns "driving the mouse." Effective preparation requires contingency plans in case one partner is absent or decides not to participate for one reason or another. In these cases, it is important to make it clear that the active student will not be punished because the pairing did not work well.

Pairing similar, not necessarily equal, abilities as partners

Pair programming can be effective when students of similar, though not necessarily equal, abilities are paired as partners. Pairing mismatched students often can lead to unbalanced participation. Teachers must emphasize that pair programming is not a "divide-and-conquer" strategy, but rather a true collaborative effort in every endeavor for the entire project. Teachers should avoid pairing very weak students with very strong students.

Motivate students by offering extra incentives

Offering extra incentives can help motivate students to pair, especially with advanced students. Some teachers have found it helpful to require students to pair for only one or two assignments.

Pair Programming Practice

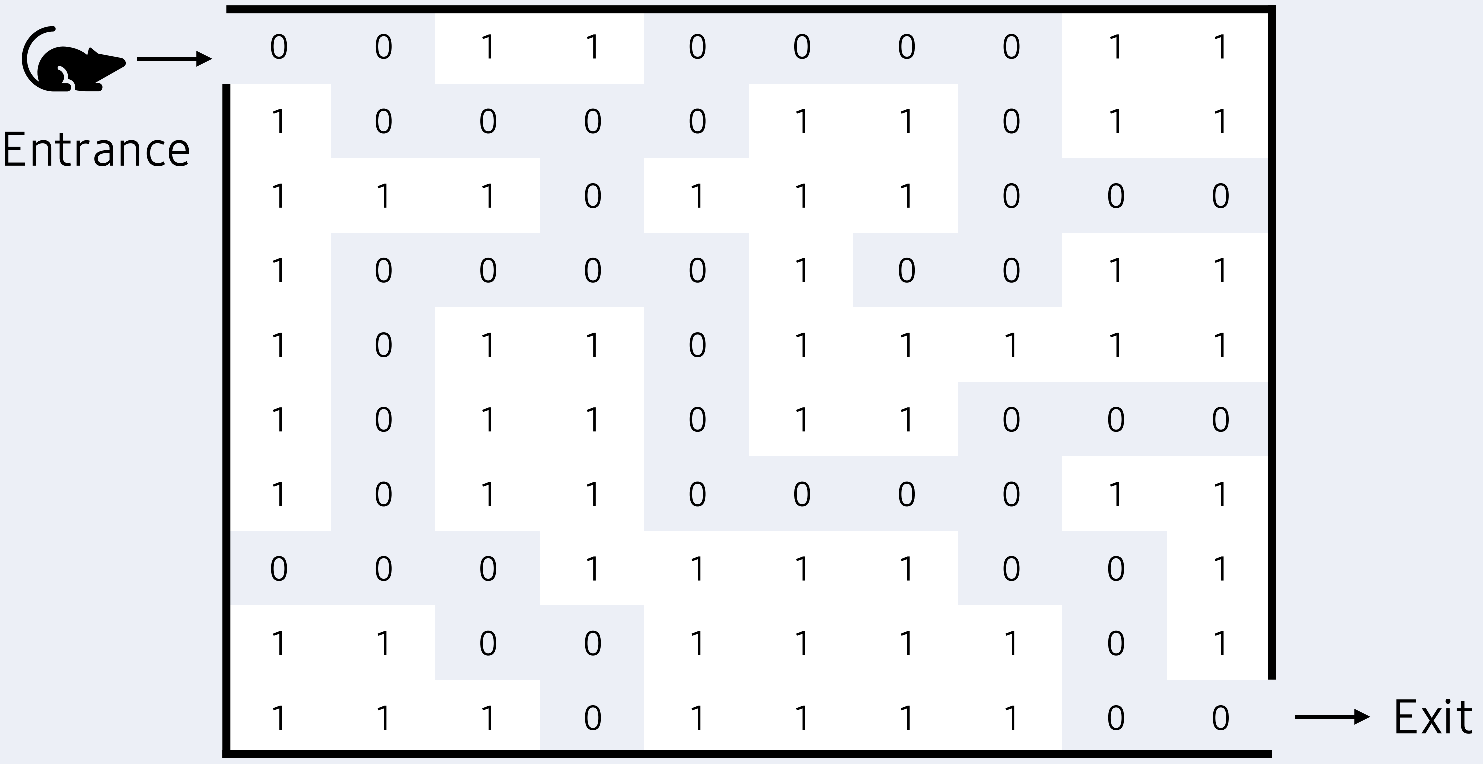
Prevent collaboration cheating

The challenge for the teacher is to find ways to assess individual outcomes, while leveraging the benefits of collaboration. How do you know whether a student learned or cheated? Experts recommend revisiting course design and assessment, as well as explicitly and concretely discussing with the students on behaviors that will be interpreted as cheating. Experts encourage teachers to make assignments meaningful to students and to explain the value of what students will learn by completing them.

Collaborative learning environment

A collaborative learning environment occurs anytime an instructor requires students to work together on learning activities. Collaborative learning environments can involve both formal and informal activities and may or may not include direct assessment. For example, pairs of students work on programming assignments; small groups of students discuss possible answers to a professor's question during lecture; and students work together outside of class to learn new concepts. Collaborative learning is distinct from projects where students "divide-and-conquer." When students divide the work, each is responsible for only part of the problem solving and there are very limited opportunities for working through problems with others. In collaborative environments, students are engaged in intellectual talk with each other.

Q1. The N*N binary matrix is given in a maze as follows, and the mouse must move from the coordinate (0, 0) to the coordinate (N-1, N-1)

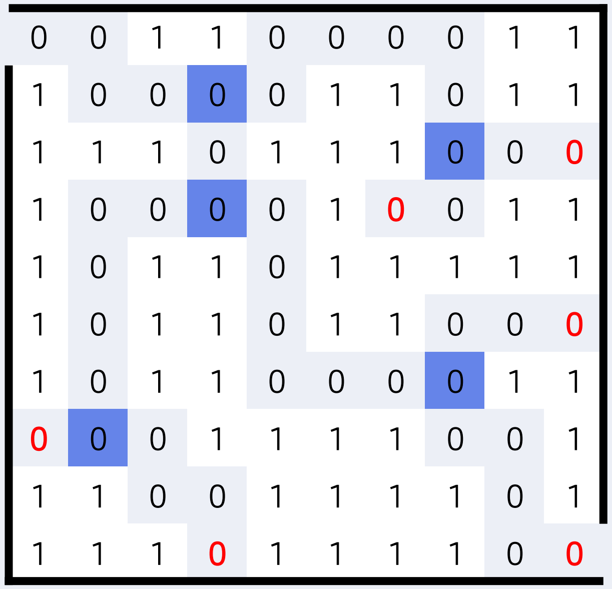


0	0	1	1	0	0	0	0	1	1
1	0	0	0	0	1	1	0	1	1
1	1	1	0	1	1	1	0	0	0
1	0	0	0	0	1	0	0	1	1
1	0	1	1	0	1	1	1	1	1
1	0	1	1	0	1	1	0	0	0
1	0	1	1	0	0	0	0	1	1
0	0	0	1	1	1	1	0	0	1
1	1	0	0	1	1	1	1	0	1
1	1	1	0	1	1	1	1	0	0

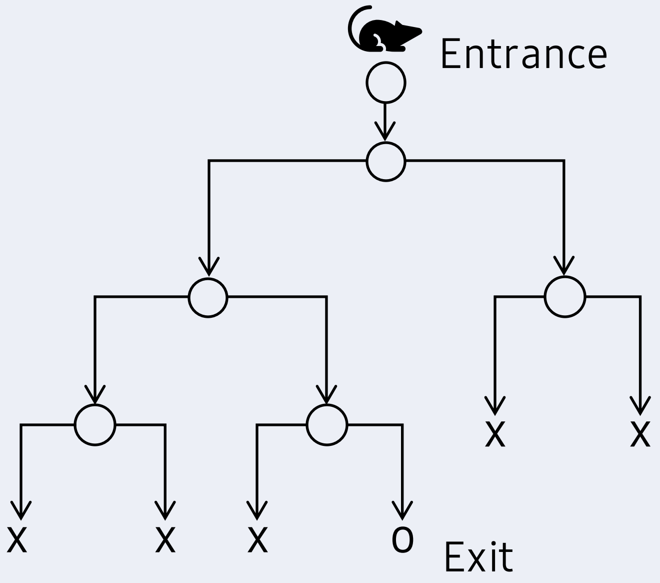
Q1. The N*N binary matrix is given in a maze as follows, and the mouse must move from the coordinate (0, 0) to the coordinate (N-1, N-1)

Hint

- In order to get out of this maze, the mouse must travel to a path with the number 0.
- Continue the search using DFS, return when you reach a dead end, and travel until you arrive at your destination.



0	0	1	1	0	0	0	0	1	1
1	0	0	0	0	1	1	0	1	1
1	1	1	0	1	1	1	0	0	0
1	0	0	0	0	1	0	0	1	1
1	0	1	1	0	1	1	1	1	1
1	0	1	1	0	1	1	0	0	0
1	0	1	1	0	0	0	0	1	1
0	0	0	1	1	1	1	0	0	1
1	1	0	0	1	1	1	1	0	1
1	1	1	0	1	1	1	1	0	0



Q1.

The N*N binary matrix is given in a maze as follows, and the mouse must move from the coordinate (0, 0) to the coordinate (N-1, N-1)

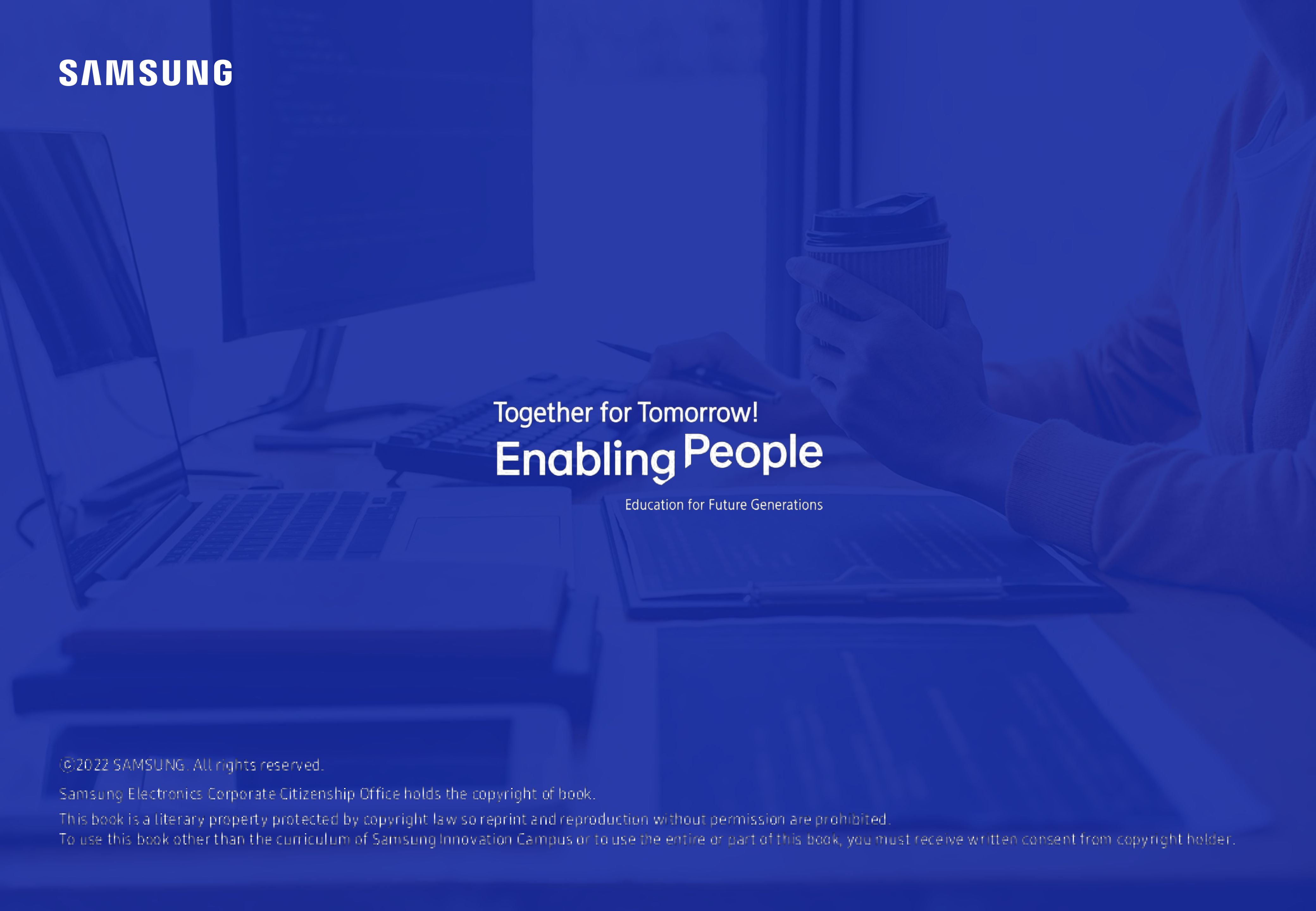
Hint

Given the N * N maze, write a program that outputs the escape route, as shown below.

```python
maze = [ [0, 0, 0, 0],
         [1, 0, 1, 0],
         [1, 0, 1, 1],
         [0, 0, 0, 0] ]
solve(maze)
```

```
0 0 1 1
1 0 1 1
1 0 1 1
1 0 0 0
```

End of Document



SAMSUNG

Together for Tomorrow!

Enabling People

Education for Future Generations

©2022 SAMSUNG. All rights reserved.

Samsung Electronics Corporate Citizenship Office holds the copyright of book.

This book is a literary property protected by copyright law so reprint and reproduction without permission are prohibited.
To use this book other than the curriculum of Samsung Innovation Campus or to use the entire or part of this book, you must receive written consent from copyright holder.