

# UE5 Fundamentals of Algorithms

## Lecture 4-5-6: Programming strategies

Ecole Centrale de Lyon, Bachelor of Science in Data Science for Responsible Business

Romain Vuillemot



# Outline

- Definitions of programming strategies
- Divide and conquer
- Greedy algorithms
- Dynamic programming

# Programming strategies

*A programming strategy are algorithms aimed at solving a specific problem in a precise manner.*

Examples of Strategies:

- **Divide and Conquer:** Divide a problem into simpler sub-problems, solve the sub-problems, and then combine the solutions to solve the original problem.
- **Dynamic Programming:** Solve a problem by breaking it down into sub-problems, calculating and memorizing the results of sub-problems to avoid unnecessary recomputation.
- **Greedy Algorithm:** Make a series of choices that seem locally optimal at each step to find a solution, with the hope that the result will be globally optimal as well.

# Divide and conquer

*The **Divide and Conquer** strategy involves breaking a complex problem into smaller, similar subproblems, solving them recursively, and then combining their solutions to address the original problem efficiently.*

1. **Divide:** Divide the original problem into subproblems of the same type.
2. **Conquer:** Solve each of these subproblems recursively.
3. **Combine:** Combine the answers appropriately.

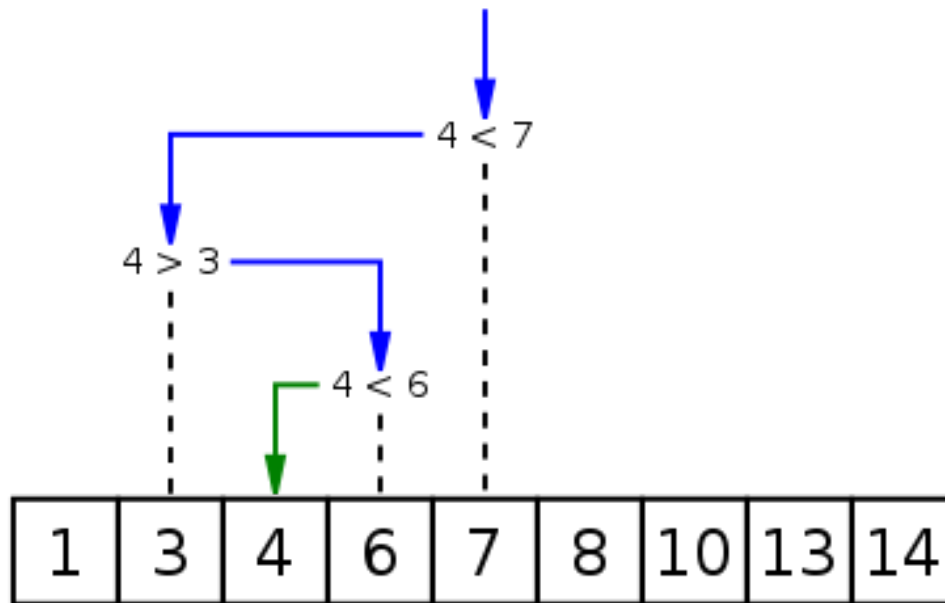
*It is very close to the recursive approach*

## Examples of divide and conquer algorithms:

- Binary search
- Quick sort and merge sort
- Map Reduce
- Others: Fast multiplication (Karatsuba)

# Binary search

*Given a sorted list, find or insert a specific value while keeping the order.*

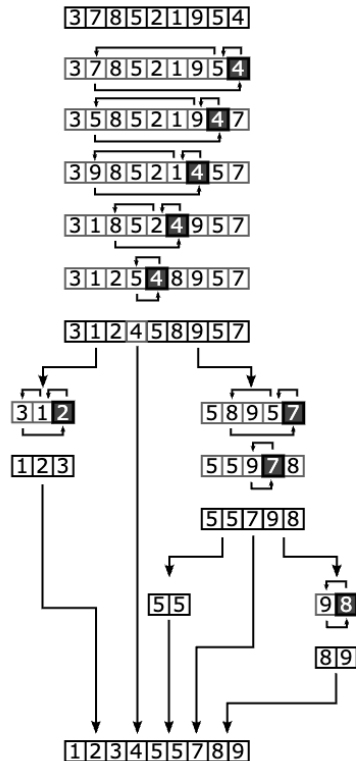


See [the notebook](#).

# Quick sort

Recursive sorting algorithm which works in two steps:

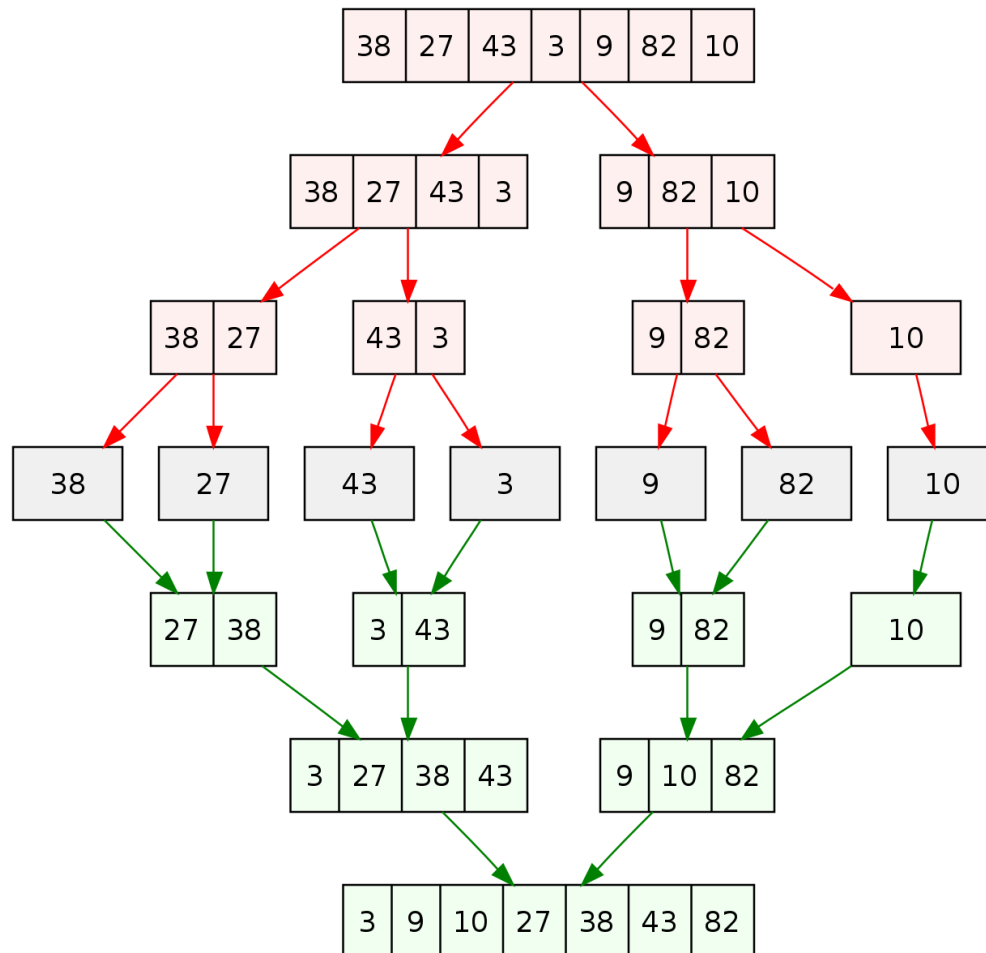
1. select a pivot element
2. partitioning the array into smaller sub-arrays, then sorting those sub-arrays.



# Merge sort

Divide an array recursively into two halves (based on a *pivot* value), sorting each half, and then merging the sorted halves back together. This process continues until the entire array is sorted.

Complexity:  $O(n\log(n))$ .

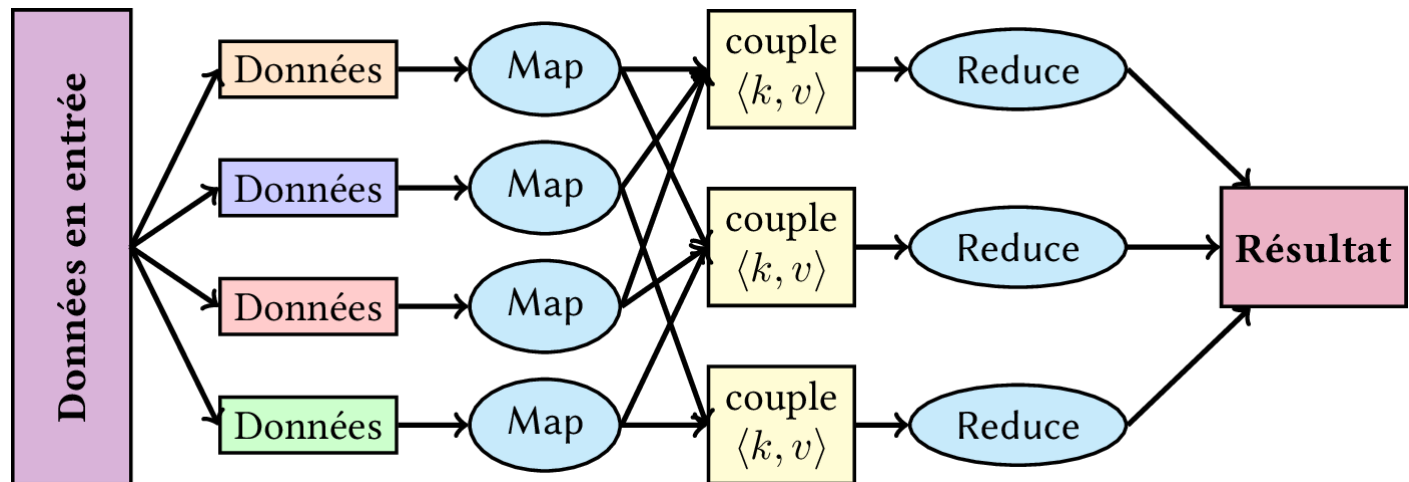




# Map reduce

Divide a large dataset into smaller chunks and processes them independantly. Two main steps:

- the Map stage, where data is filtered and transformed into key-value pairs
- the Reduce stage, where data is aggregated and the final result is produced.



# Map reduce (without map reduce..)

*Calculate the sum of squares values from a list of numerical values.*

```
In [49]: data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

# Map reduce (without map reduce..)

*Calculate the sum of squares values from a list of numerical values.*

```
In [49]: data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
In [60]: result = {}
         for num in data:
             square = num * num
             result[square] = num

         final_result = list(result.items())

         print(final_result)
         print(sum([x[0] for x in final_result]))
```

```
[(1, 1), (4, 2), (9, 3), (16, 4), (25, 5), (36, 6), (49, 7), (64, 8), (81, 9), (100, 10)]
385
```

# Map reduce (Python)

1. Divide the problem in sub-problems
2. Apply the mapping function
3. Reduce the results

# Map reduce (Python)

1. Divide the problem in sub-problems
2. Apply the mapping function
3. Reduce the results

```
In [69]: data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def mapper(numbers):
    result = []
    for num in numbers: # calculate the squares
        result.append((num, num * num))
    return result

def reducer(pairs):
    result = {}
    for key, value in pairs: # sums the squares
        if key in result:
            result[key] += value
        else:
            result[key] = value
    return result.items()
```

# Map reduce (Python)

1. Divide the problem in sub-problems
2. Apply the mapping function
3. Reduce the results

```
In [5]: chunk_size = 2
chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]

mapped_data = [mapper(chunk) for chunk in chunks]

grouped_data = {} # map
for chunk in mapped_data:
    for key, value in chunk:
        if key in grouped_data:
            grouped_data[key].append(value)
        else:
            grouped_data[key] = [value]

reduced_data = [reducer(list(grouped_data.items()))] # reduce
result = sum([x[1][0] for x in final_result])

print(result)
```

```
call last)
```

```
Cell In[5], line 2
```

```
    1 chunk_size = 2  
----> 2 chunks = [data[i:i+chunk_size] for i in range(0, len(data),  
    3 chunk_size)]  
    4 mapped_data = [mapper(chunk) for chunk in chunks]  
    5 grouped_data = {}# map
```

```
NameError: name 'data' is not defined
```

# Discussion on Divide and Conquer

- Similarities with recursion by dividing a problem in a sub-problem
- But with a combination step (which may hold most of the code difficulty)
- Can be implemented in a non-recursive way
- $n\log(n)$  complexity when split the problem and solves each split



# Greedy algorithms

*Algorithms that make a locally optimal choice.*

Examples:

- Change-making problem
- Knapsack problem
- Maze solving
- Graph coloring

# Example: Change-making problem

$$Q_{opt}(S, M) = \min \sum_{i=1}^n x_i.$$

$S$ : all the available coins

$M$ : amount

Greedy solution:

1. Sort the coins in descending order
2. Initialize a variable to count coins used
3. Subtract the number of coins used (if limited)
4. Continue this process until amount becomes zero.

# Example: Change-making problem (Python)

*Greedy solution to return the minimal number of coins necessary.*

```
In [82]: coins = [1, 2, 5]  
         amount = 11
```

# Example: Change-making problem (Python)

*Greedy solution to return the minimal number of coins necessary.*

```
In [82]: coins = [1, 2, 5]
         amount = 11
```

```
In [83]: def coin_change_greedy(coins, amount):
         coins.sort(reverse=True) # important! sort in descending order

         coin_count = 0
         remaining_amount = amount

         for coin in coins:
             while remaining_amount >= coin:
                 remaining_amount -= coin
                 coin_count += 1

         if remaining_amount == 0:
             return coin_count
         else:
             return -1
```

```
In [81]: print(coin_change_greedy(coins, amount)) # 3 (11 = 5 + 5 + 1)
```

# Example: Change-making problem (Python)

*Greedy solution that returns the **list of coins** used.*

Tip: use a list with the same structure as coins.



# Example: Change-making problem (Python)

*Greedy solution that returns the **list of coins** used.*

Tip: use a list with the same structure as coins.

```
In [87]: def coin_change_greedy(coins, amount):
          coins.sort(reverse=True)

          coin_count = 0
          remaining_amount = amount
          used_coins = [0] * len(coins)

          for i, coin in enumerate(coins):
              while remaining_amount >= coin:
                  remaining_amount -= coin
                  coin_count += 1
                  used_coins[i] += 1

          if remaining_amount == 0:
              return coin_count, used_coins
          else:
              return -1, []

          coins = [25, 10, 5, 1]
          amount = 63
          min_coins, coins_used = coin_change_greedy(coins, amount)
```

```
print(f"Minimum coins needed: {min_coins}")  
print("Coins used:", coins_used)
```

Minimum coins needed: 6

Coins used: [2, 1, 0, 3]



# Example: Change-making problem (Python)

*Greedy solution that returns the **list of coins** used from **a limited availability of coins**.*

Tip: use a list of coins availability of same structure as coins.

```
In [92]: coins = [25, 10, 5, 1]
amount = 63
coin_availability = [1, 2, 3, 4]
```



# Example: Change-making problem (Python)

*Greedy solution that returns the **list of coins** used from **a limited availability of coins**.*

Tip: use a list of coins availability of same structure as coins.

```
In [92]: coins = [25, 10, 5, 1]
amount = 63
coin_availability = [1, 2, 3, 4]
```

```
In [91]: def coin_change_greedy(coins, amount, coin_availability):
        coins.sort(reverse=True)

        coin_count = 0
        remaining_amount = amount
        used_coins = [0] * len(coins)

        for i, coin in enumerate(coins):
            while remaining_amount >= coin and used_coins[i] < coin_availability[i]:
                remaining_amount -= coin
                coin_count += 1
                used_coins[i] += 1

        if remaining_amount == 0:
            return coin_count, used_coins
        else:
            return -1, []
```

```
min_coins, coins_used = coin_change_greedy(coins, amount, coin_availabi

print(f"Minimum coins needed: {min_coins}")
print("Coins used:", coins_used)
```

Minimum coins needed: 9  
Coins used: [1, 2, 3, 3]

# Discussion on Greedy algorithms

- Often considered as an *heuristic*
- Easy to understand, implement and communicate
- They often lead to non-optimal solution

# Dynamic programming

***Dynamic programming** involves breaking down a problem into subproblems, solving these subproblems, and combining their solutions to obtain the solution to the original problem. The steps are as follows:*

1. Characterize the structure of an optimal solution.
2. Define the value of an optimal solution recursively.
3. Reconstruct the optimal solution from the computations.

Notes :

- Applies to problems with optimal substructure.
- Also applies to problems where solutions are often interrelated (distinguishing it from divide and conquer).
- Utilizes a memoization approach, involving storing an intermediate solution (e.g., in a table).

## Examples of dynamic programming algorithms

- Fibonacci Sequence
- Rod Cutting
- Sequence Alignment, Longest Subsequence Finding
- Shortest Path Finding

# Fibonnacci (reminder)

To calculate the  $n$ -th number in the Fibonacci sequence, which is determined as follows:

latex Copy code  $fib(n) = fib(n - 1) + fib(n - 2), n \in \mathbb{N}$  Where the sequence starts with 1, 1, and then continues as 2, 3, 5, 8, 13, 21, and so on, to find the 9th number ( $n = 9$ ).

Let's calculate the 9th Fibonacci number step by step:

$$fib(1) = 1$$

$$fib(2) = 1$$

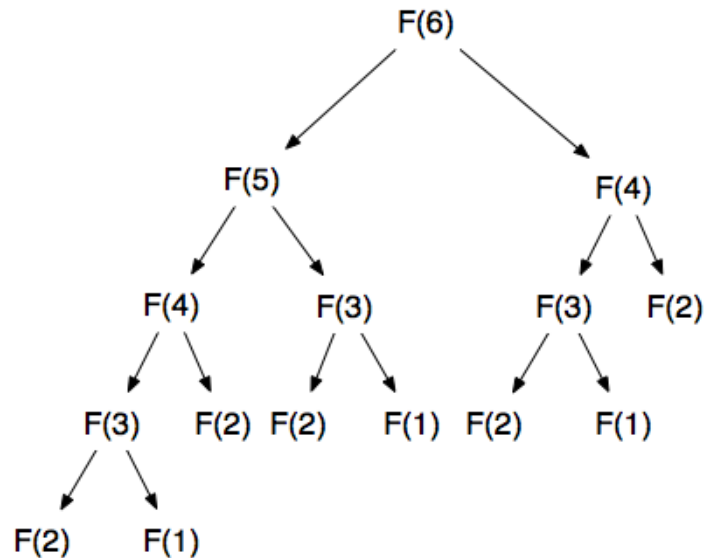
$$fib(3) = fib(2) + fib(1) = 1 + 1 = 2$$



# Fibonacci (naive)

```
In [4]: def fib(n):  
        if n < 2:  
            return n  
        else:  
            return fib(n - 1) + fib(n - 2)
```

Call tree (for  $n = 6$ ):



Requires to calculate the same F-value multiple times.

# Fibonnacci (dynamic programming)

*Optimized using a `lookup` table, which is a data structure to memoize values that have already been computed.*

# Fibonacci (dynamic programming)

Optimized using a `lookup` table, which is a data structure to memoize values that have already been computed.

```
In [97]: def fib(n, lookup):
    if n == 0 or n == 1:
        lookup[n] = n

    if lookup[n] is None:
        lookup[n] = fib(n - 1, lookup) + fib(n - 2, lookup)

    return lookup[n]

def main():
    n = 6

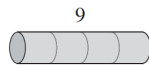
    lookup = [None] * (n + 1)
    result = fib(n, lookup)
    print(f"{n}-th Fibonacci number is {result}")

if __name__ == "__main__":
    main()
```

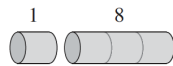
6-th Fibonacci number is 8

# Rod cutting

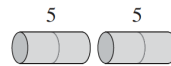
Given a list of cuts and prices, identify the optimal cuts. Given the example below, what is the best cutting strategy for a rod of size 4?



(a)



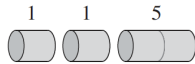
(b)



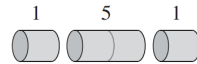
(c)



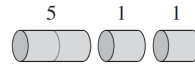
(d)



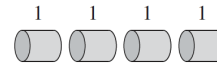
(e)



(f)



(g)

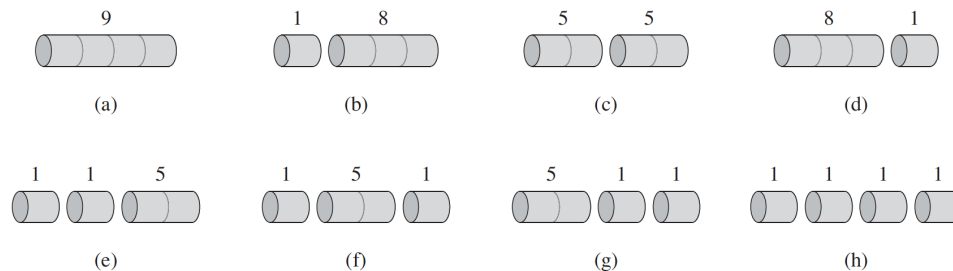


(h)

size (i)	1	2	3	4	5	6	7	8
price (p <sub>i</sub> )	1	5	8	9	10	17	17	20

# Rod cutting

Given a list of cuts and prices, identify the optimal cuts. Given the example below, what is the best cutting strategy for a rod of size 4?



size (i)	1	2	3	4	5	6	7	8
price (p <sub>i</sub> )	1	5	8	9	10	17	17	20

Solution: For a rod of size 4 optimal solution is 2 cuts of size 2 so  $5 + 5 = 10$ .

# Rod cutting: check a solution

Given the previous table of size and price, check the cost of a given solution by defining a function `check_rod_cutting(prices, n)`.

# Rod cutting: check a solution

Given the previous table of size and price, check the cost of a given solution by defining a function `check_rod_cutting(prices, n)`.

```
In [27]: def check_rod_cutting(prices, n):
          table = [0] * (n + 1)

          for i in range(1, n + 1):
              max_price = float('-inf')
              for j in range(1, i + 1):
                  max_price = max(max_price, prices[j] + table[i - j])
              table[i] = max_price

          return table[n]
```

```
In [30]: prices = [0, 1, 5, 8, 9, 10, 17, 17, 20]
          n = 2

          max_total_price = check_rod_cutting(prices, n)
          print(f"The maximum total price for a rod of length {n} is {max_total_p
```

The maximum total price for a rod of length 2 is 5

# Rod cutting (brute force)

Let's solve the rod cutting problem using a brute force (naive) approach.

1. define a value function
2. identify a base case
3. identify a recursion mechanism



# Rod cutting (brute force)

Let's solve the rod cutting problem using a brute force (naive) approach.

1. define a value function
2. identify a base case
3. identify a recursion mechanism

```
In [33]: def cut_brute_force(n, t):  
    if n == 0:  
        return 0  
    max_valeur = float('-inf')  
    for i in range(1, n + 1):  
        valeur_courante = t[i] + coupe_brute_force(n - i, t)  
        max_valeur = max(max_valeur, valeur_courante)  
    return max_valeur
```

# Rod cutting (brute force)

Let's solve the rod cutting problem using a brute force (naive) approach.

1. define a value function
2. identify a base case
3. identify a recursion mechanism

```
In [33]: def cut_brute_force(n, t):  
        if n == 0:  
            return 0  
        max_valeur = float('-inf')  
        for i in range(1, n + 1):  
            valeur_courante = t[i] + coupe_brute_force(n - i, t)  
            max_valeur = max(max_valeur, valeur_courante)  
        return max_valeur
```

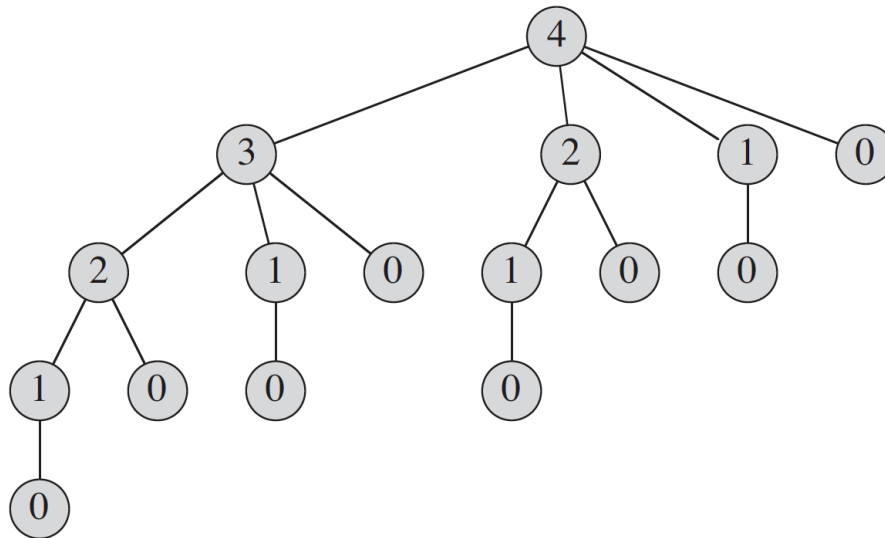
```
In [34]: lengths = [0, 1, 2, 3, 4, 5, 6, 7, 8]  
        values = [0, 1, 5, 8, 9, 10, 17, 17, 20]  
        rod_length = 2  
        max_value = coupe_brute_force(rod_length, values)  
        print(f"The maximum value for a rod of length {rod_length} is {max_value}")
```

The maximum value for a rod of length 2 is 5.

# Rod cutting (dynamic programming)

General case:

- Cutting a rod of length  $i$  optimally.
- Cutting a rod of length  $(n - i)$  optimally.



General case:  $V_n = \max_{1 \leq i \leq n} (p_i + V_{n-i})$

size (i)	1	2	3	4	5	6	7	8
price (pi)	1	5	8	9	10	17	17	20

$$V_3 = \max_{1 \leq i \leq 3} (p_i + V_{3-i})$$

Let's calculate  $v_3$  step by step for each possible value of  $i$ :

1. If  $i = 1$ , we cut the rod into two pieces: one of length 1 and one of length 2.

- $V_1 = p_1 = 2$
- $V_{3-1} = V_2$

2. If  $i = 2$ , we cut the rod into two pieces: one of length 2 and one of length 1.

- $V_2 = p_2 = 5$
- $V_{3-2} = V_1$

3. If  $i = 3$ , we cut the rod into one piece of length 3.

- $V_3 = p_3 = 9$
- $V_{3-3} = V_0$  (Assuming that  $V_0 = 0$  as a base case.)

Now, we can calculate the values for  $v_2$  and  $v_1$  recursively using the same formula:

For  $v_2$ :

$$V_2 = \max(p_1 + V_1, p_2 + V_0) = \max(2 + V_1, 5 + 0) = \max(2 + 2, 5 + 0) = \max(4, 5) = 5$$

For  $v_1$ :

$$V_1 = \max(p_1 + V_0) = \max(2 + 0) = 2$$

So,  $v_2$  is 5 and  $v_1$  is 2.

Now, we can calculate  $v_3$  using the values of  $v_2$  and  $v_1$ :

$$V_3 = \max(p_1 + V_2, p_2 + V_1, p_3 + V_0) = \max(1 + 5, 5 + 2, 9 + 0) = \max(6, 7, 8) = 8$$

# Rod cutting (dynamic programming)



# Rod cutting (dynamic programming)

```
In [36]: INT_MIN = 0

def cutRod(price, n):

    # init cache tables
    val = [0 for x in range(n+1)]
    val[0] = 0

    for i in range(1, n+1):
        max_val = INT_MIN
        for j in range(i):
            max_val = max(max_val, price[j] + val[i-j-1])
        val[i] = max_val

    return val[n]

if __name__ == "__main__":
    arr = [1, 5, 8, 9, 10, 17, 17, 20]
    size = len(arr)
    print("Max size cut " + str(cutRod(arr, size)), len(arr) )
```

Max size cut 22 8



# Change-making problem (dynamic programming)

$$Q_{opt}(S, M) = \min \sum_{i=1}^n x_i.$$

$S$ : all the available coins

$M$ : amount

$$Q_{opt}(i, m)$$

$$= \min \begin{cases} 1 + Q_{opt}(i, m - v_i) & \text{if } (m - v_i) \geq 0 & \text{we use a coin of type } i \text{ of value } v_i \\ Q_{opt}(i - 1, m) & \text{if } i \geq 1 & \text{we do not use coin of type } i, \text{ we use } i - 1 \end{cases}$$

	0	1	2	3	4	5	6	7
$\emptyset$	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
1c	0	1	2	3	4	5	6	7
1c, 3c	0	1	2	1	2	3	2	3
1c, 3c, 4c	0	1	2	1	1	2	2	2

+1

# Lessons on dynamic programming

- It is necessary to study each problem on a case-by-case basis.
- Storing a large number of partial results, which requires significant memory usage.
- Suitable for only certain problems (min, max, counting the number of solutions).

