CMC UNIVERSITY

CHAPTER 6. ALGORITHM DESIGN AND ANALYSIS TECHNIQUES

CMC UNIVERSITY

Outline

- ✓ Basic concepts
- ✓ Divide and conquer
- ✓ Dynamic programming

CMC UNIVERSITY

DIVIDE AND CONQUER

CMC UNIVERSITY

Divide-and-conquer

- The divide-and-conquer paradigm divides a problem into smaller sub-problems, and then solves these; finally, it combines the results to obtain a global, optimal solution.
- A problem can be found the algorithm by divide and conquer method if:

- Is it possible to define a problem as a combination of problems of the same type but smaller. What does "smaller" mean? (Define the problem decomposition rule)

- What special case of the problem can be considered small enough to be solved directly? (Determine the basic problems)

- Some examples of the divide-and-conquer design technique are as follows: binary search, merge sort, quick sort, algorithm for fast multiplication, Strassen’s matrix multiplication, closest pair of points, …

CMC UNIVERSITY

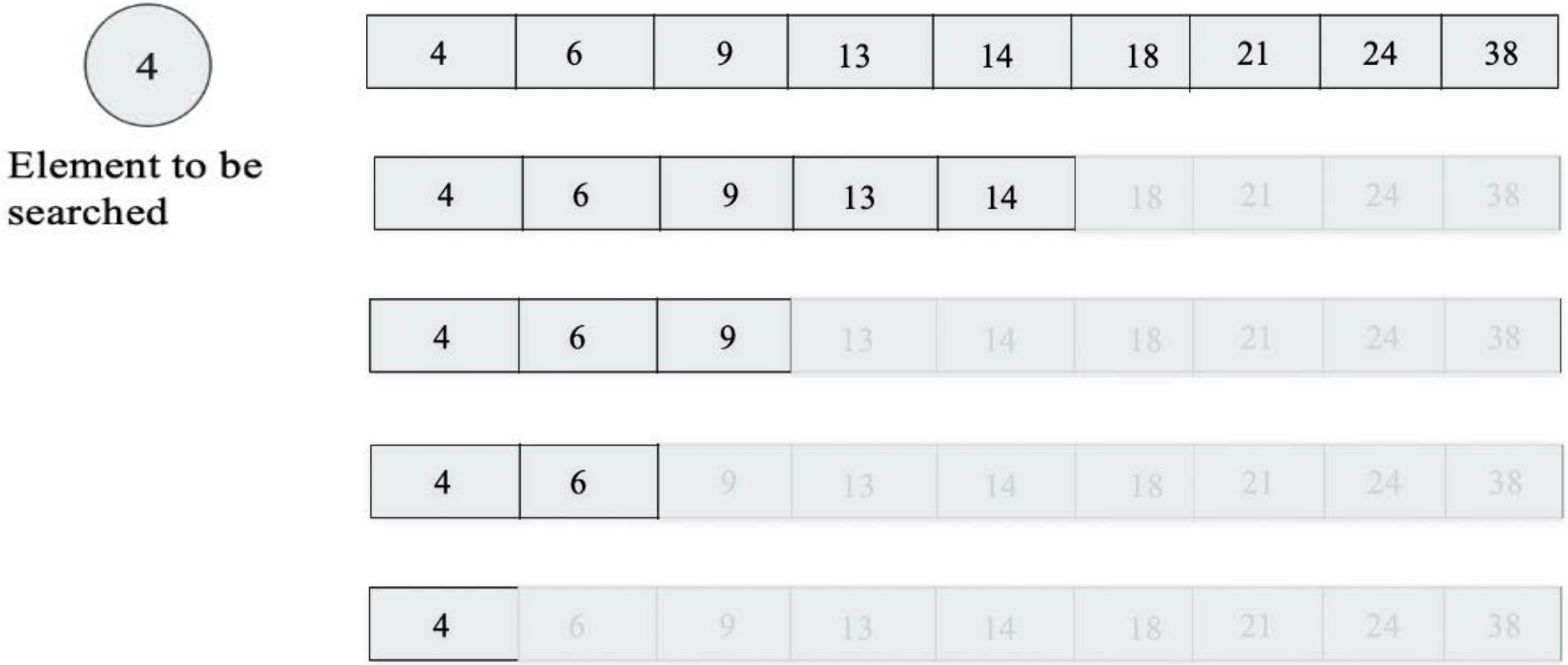
Binary search

This algorithm is used to find a given element from a sorted list of elements.

- It first compares the search element with the middle element of the list; if the search element is smaller than the middle element, then the half of the list of elements greater than the middle element is discarded;
- The process repeats recursively until the search element is found or we reach the end of the list.
- It is important to note that in each iteration, half of the search space is discarded, which improves the performance of the overall algorithm because there are fewer elements to search through.

Binary search

Example: search for element 4 in the given sorted list of elements



CMC UNIVERSITY

Binary search

```
1  def binary_search(arr, start, end, key):
2      while start <= end:
3          mid = start + (end - start)//2
4          if arr[mid] == key:
5              return mid
6          elif arr[mid] < key:
7              start = mid + 1
8          else:
9              end = mid - 1
10     return -1
11
12 arr = [4, 6, 9, 13, 14, 18, 21, 24, 38]
13 x = 13
14 result = binary_search(arr, 0, len(arr)-1, x)
15 print(result)
```

the worst-case time complexity of the binary search algorithm is O(log n)

CMC UNIVERSITY

Merge sort

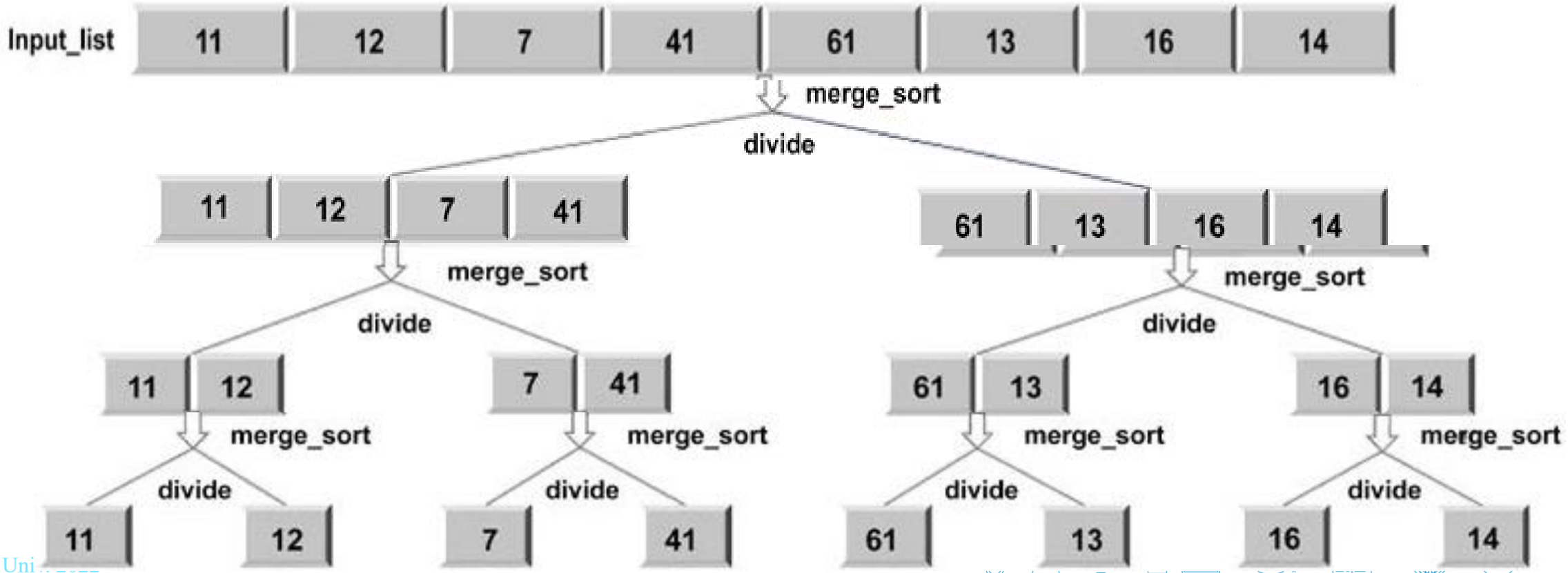
Merge sort is an algorithm for sorting a list of n natural numbers in increasing order.

- Firstly, the given list of elements is divided iteratively into equal parts until each sublist contains one element, and then these sublist are combined to create a new list in a sorted order.
- This programming approach to problem-solving is based on the divide-and-conquer methodology and emphasizes the need to break down a problem into smaller sub-problems of the same type or form as the original problem. These sub-problems are solved separately and then results are combined to obtain the solution of the original problem.

Merge sort

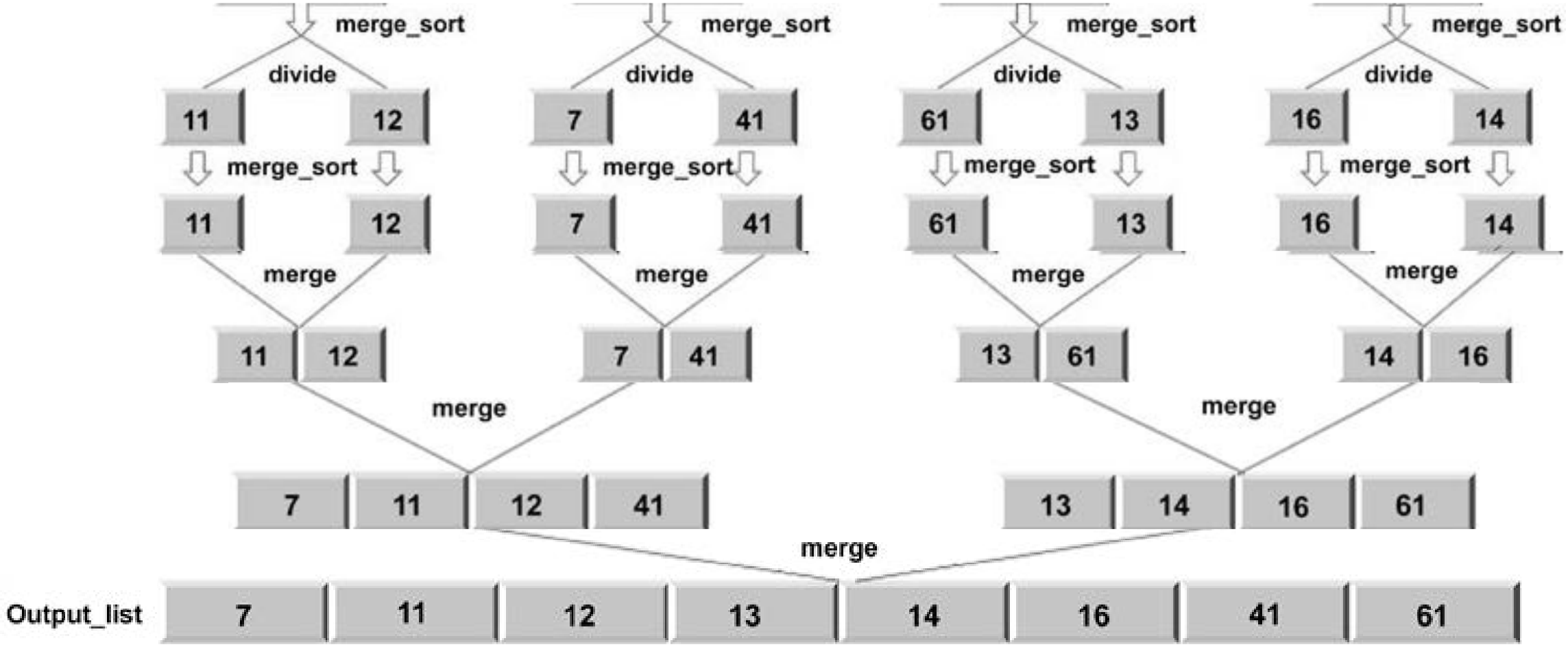
Example 1: given a list of unsorted elements, split the list into two approximate halves, continue to divide the list into halves recursively.

After a while, the sublist created as a result of the recursive call will contain only one element. At that point, we begin to merge the solutions in the conquer or merge step.



CMC UNIVERSITY

Merge sort



CMC UNIVERSITY

Merge sort

The implementation of the merge sort algorithm is implemented using the merge_sort method, which recursively divides the list.

```python
def merge_sort(unsorted_list):
    if len(unsorted_list) == 1:
        return unsorted_list
    mid_point = int(len(unsorted_list)/2)
    first_half = unsorted_list[:mid_point]
    second_half = unsorted_list[mid_point:]
    half_a = merge_sort(first_half)
    half_b = merge_sort(second_half)
    return merge(half_a, half_b)
```

```python
def merge(first_sublist, second_sublist):
    i = j = 0
    merged_list = []
    while i < len(first_sublist) and j < len(second_sublist):
        if first_sublist[i] < second_sublist[j]:
            merged_list.append(first_sublist[i])
            i += 1
        else:
            merged_list.append(second_sublist[j])
            j += 1
    while i < len(first_sublist):
        merged_list.append(first_sublist[i])
        i += 1
    while j < len(second_sublist):
        merged_list.append(second_sublist[j])
        j += 1
    return merged_list

a= [11, 12, 7, 41, 61, 13, 16, 14]
print(merge_sort(a))
```

CMC UNIVERSITY

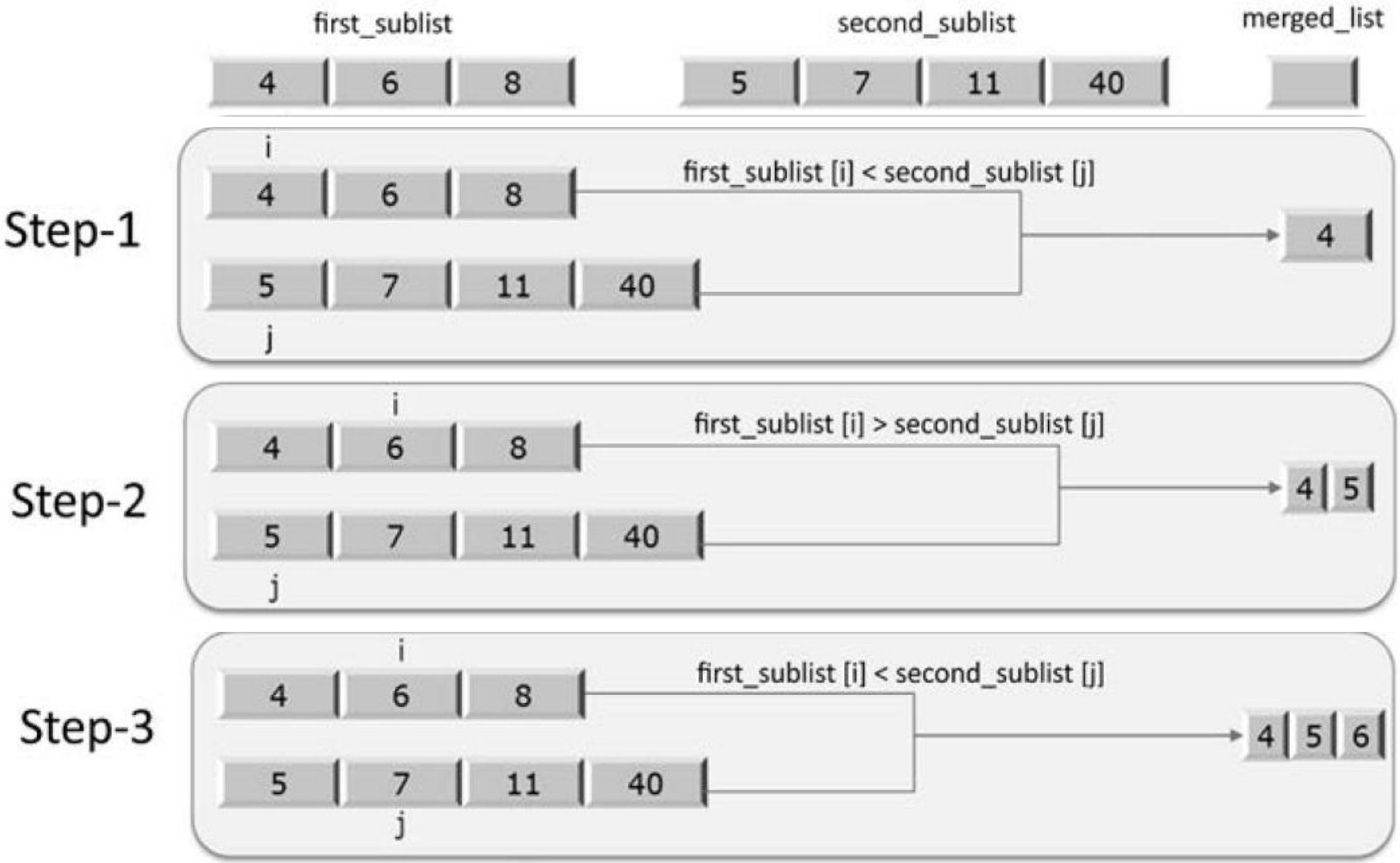
Merge sort

Example 2: Let's give the algorithm a dry run by merging the two sublists [4, 6, 8] and [5, 7, 11, 40], shown in Table:

Step	first_sublist	second_sublist	merged_list
0	[4 6 8]	[5 7 11 40]	[]
1	[6 8]	[5 7 11 40]	[4]
2	[6 8]	[7 11 40]	[4 5]
3	[8]	[7 11 40]	[4 5 6]
4	[8]	[11 40]	[4 5 6 7]
5	[ ]	[11 40]	[4 5 6 7 8]
6	[]	[ ]	[4 5 6 7 8 11 40]

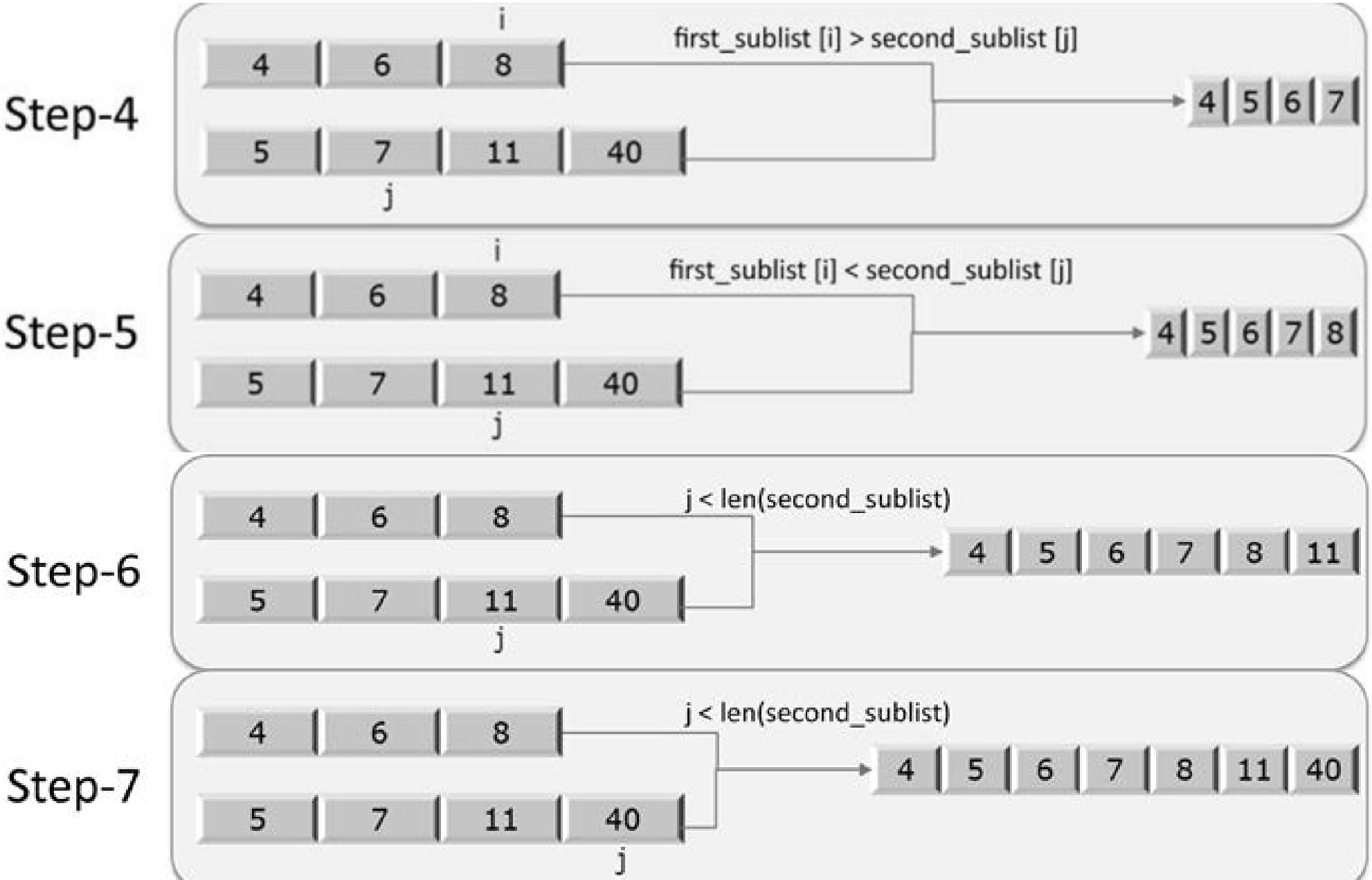
CMC UNIVERSITY

Merge sort



CMC UNIVERSITY

Merge sort



CMC UNIVERSITY

Merge sort

The worst-case running time complexity of the merge sort will depend on the following steps:

1. Firstly, the divide step will take a constant time since it just computes the midpoint, which can be done in O(1) time

2. Then, in each iteration, we divide the list into half recursively, it will take O(logn), which is quite similar to what we have seen in the binary search algorithm

3. Further, the combine/merge step merges all the n elements into the original array, which will take O(n) time.

Hence, the merge sort algorithm has a runtime complexity of:

$$O(n) * O(\log n) = O(n \log n).$$

CMC UNIVERSITY

Exercises

1. Write a recursive algorithm and divide-and-conquer algorithm to calculate the power of a positive integer (Given two positive integers x, n. Let's calculate the value of x^n)

2. Write a recursive algorithm and divide-and-conquer algorithm to find the factorial of a positive integer ($n!$)

3. Write a recursive algorithm and divide-and-conquer algorithm that multiplies two polynomials

Given two polynomials $A(x) = \sum_{i=0}^{m} a_i x^i$ and $\mathrm{B}(x) = \sum_{j=0}^{n} b_j x^j$, find polynomial $C(x) = A(x)B(x) = \sum_{k=0}^{m+n} c_k x^k$

The main task of this problem is to find $c_k = \sum_{i+j=k} a_i b_j, \forall k: 0 \leq k \leq m+n$

CMC UNIVERSITY

DYNAMIC PROGRAMMING

CMC UNIVERSITY

Dynamic programming problem

In mathematics and computer science, ***dynamic programming*** is an effective method of solving optimization problems that has the following three properties:

1. Recursive form: The large problem can be decomposed into similar subproblems, which can be decomposed into even smaller problems…

2. Optimal substructure: The optimal solution of subproblems can be used to find the optimal solution of the large problem.

3. Overlapping subproblems: Two subproblems in the decomposed process may share some other subproblems.

The first and second properties are necessary conditions for a dynamic programming problem.

The third property characterizes a problem that the dynamic programming method is much more efficient than the conventional recursive solution.

CMC UNIVERSITY

Dynamic programming problem

- The problem solved by the dynamic programming method is called the ***dynamic programming problem***.

- The formula for combining solutions of subproblems to get solutions of large problems is called the ***recursive formula*** of dynamic programming.

- The set of smallest problems with immediate solutions from which to solve larger problems is called ***dynamic programming basis***.

- The space to store the solutions of subproblems to find ways to coordinate them is called the ***solution table*** of dynamic programming.

CMC UNIVERSITY

Dynamic programming problem

Solution for dynamic programming problem:

Dynamic Programming = Divide and Conquer + Memoization

1. Analyze the problem: verify the three properties of a dynamic programming problem
2. Divide the given problem into similar subproblems
3. Develop a way to find the solution of the big problem given that we already know the solutions of the subproblems - find the recurrence formula
4. Show an efficient way to store the results of subproblems

CMC UNIVERSITY

Dynamic programming problem

We need an efficient way to store the results of each sub-problem. The following two techniques are readily available:

• **Top-down with memoization**:

- Starts from the initial problem set and divides it into small sub-problems.
- Determine the solution to a sub-program and store its result (In the future, when this sub-problem is encountered, we only return its pre computed result).
- If the solution to a given problem can be formulated recursively using the solution of the sub-problems, then the solution of the overlapping sub-problems can easily be memoized.

CMC UNIVERSITY

Dynamic programming problem

Memoization means storing the solution of the sub-problem in an array or hash table. Whenever a solution to a sub-problem needs to be computed, it is first referred to the saved values if it is already computed, and if it is not stored, then it is computed in the usual manner. This procedure is called memoized, which means it “remembers” the results of the operation that has been computed before.

- **Bottom-up approach**:

- A given problem is solved by dividing it into sub-problems recursively, with the smallest possible sub-problems then being solved.
- The solutions to the sub-problems are combined in a bottom-up fashion to arrive at the solution to the bigger sub-problem in order to recursively reach the final solution.

CMC UNIVERSITY

Calculating the Fibonacci series

The Fibonacci series can be demonstrated using a recurrence relation

For example, the following recurrence relation defines the Fibonacci sequence [1, 1, 2, 3, 5, 8 ...]:

```
func(0) = 1
func(1) = 1
func(n) = func(n-1) + func(n-2) for n>1
```

A recursive-style program to generate the sequence would be as follows:

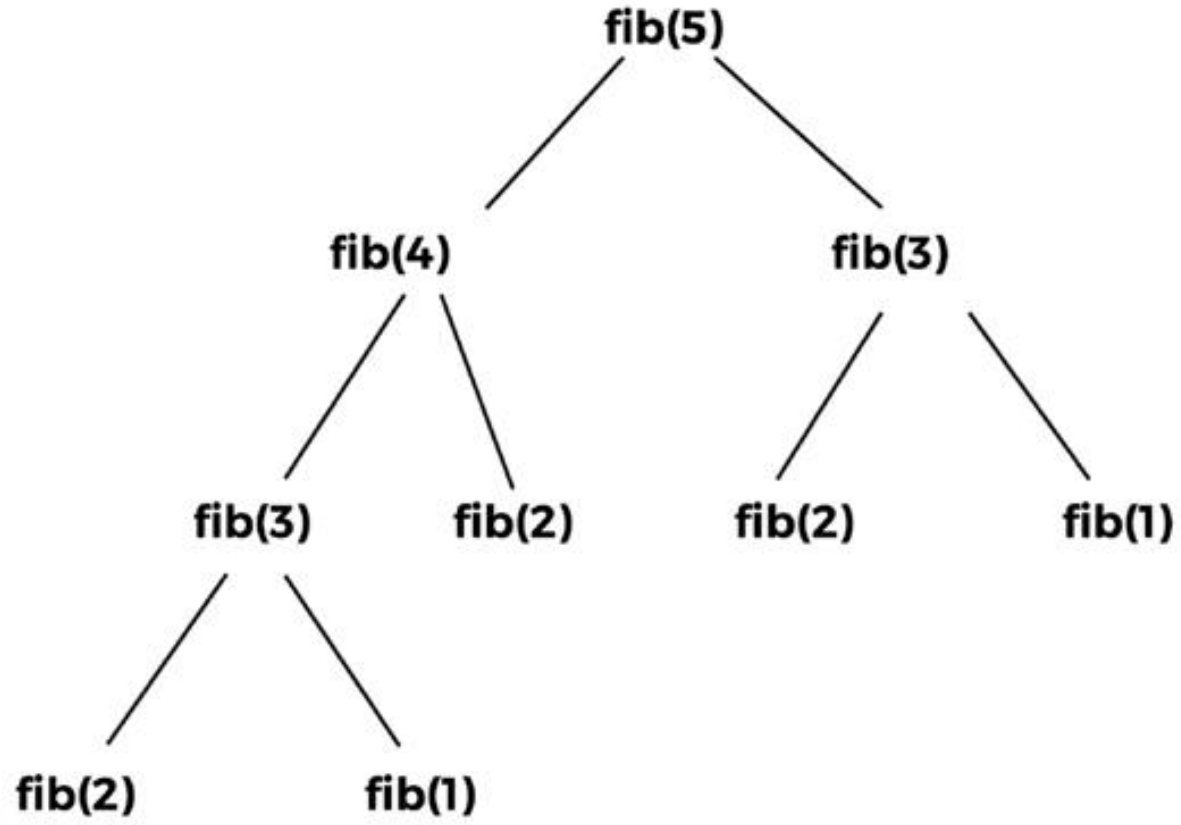
```
1 def fib(n):
2     if n <= 1:
3         return 1
4     else:
5         return fib(n-1) + fib(n-2)
6 for i in range(5):
7     print(fib(i))
```

CMC UNIVERSITY

Calculating the Fibonacci series

In the above code, we can see that the recursive calls are being called in order to solve the problem.

- When the base case is met, the fib() function returns 1. If n is equal to or less than 1, the base case is met.
- If the base case is not met, we call the fib() function again. The recursion tree to solve up to the fifth term in the Fibonacci sequence is shown in Figure:

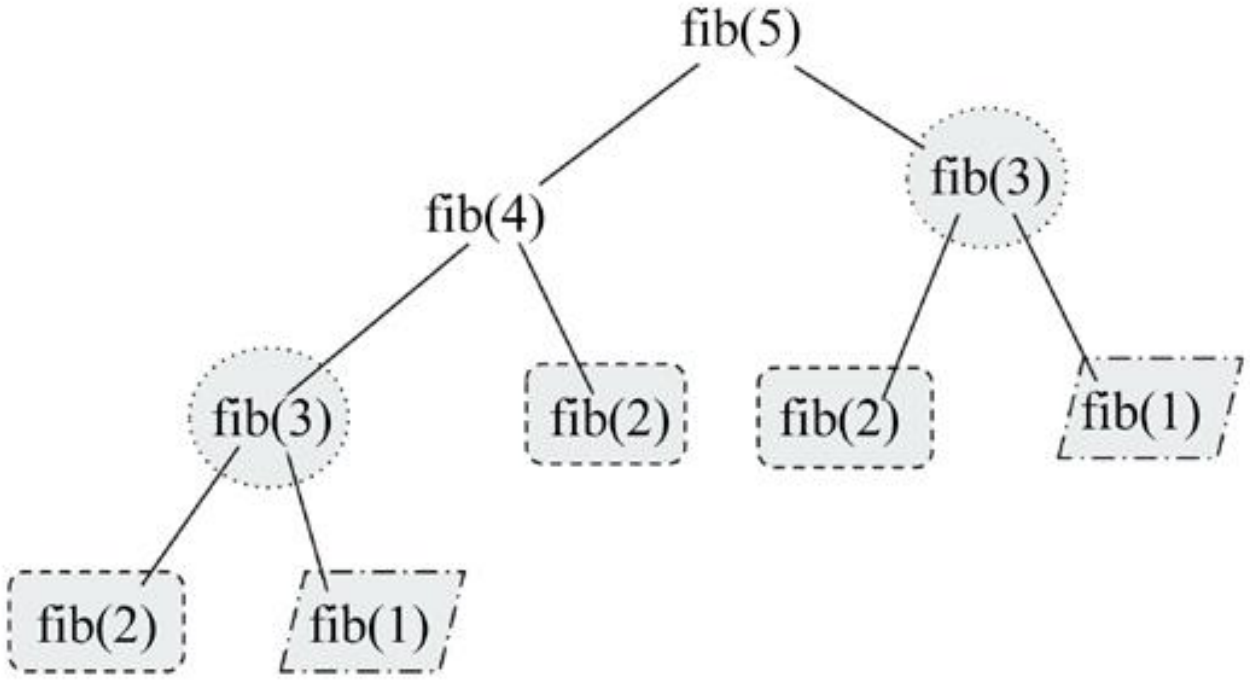


Calculating the Fibonacci series

The overlapping sub-problems from the recursion tree as shown in the following Figure: the call to fib(1) happens twice, the call to fib(2) happens three times, and the call to fib(3) occurs twice. The return values of the same function call never change.

So, fib(1), fib(2), fib(3) are overlapping problems. Computational time will be wasted if we compute the same function again whenever it is encountered.

These repeated calls to a function with the same parameters and output suggest that there is an overlap. Certain computations reoccur down in the smaller sub-problem.



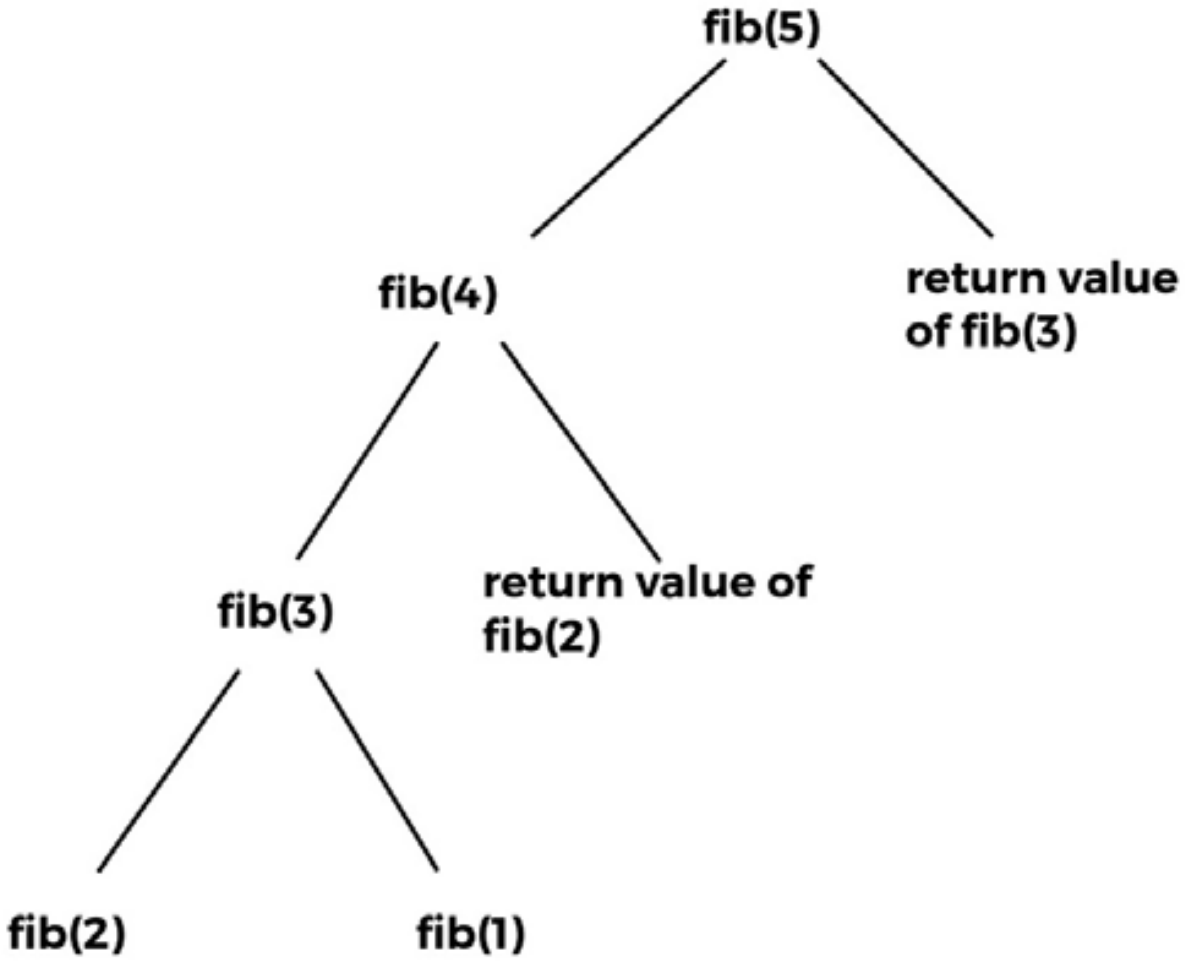
CMC UNIVERSITY

Calculating the Fibonacci series

In dynamic programming using the memoization technique:

- Store the results of the computation of fib(1), fib(2) and fib(3) the first time they are encountered.

- Later, whenever a call to fib(1), fib(2), or fib(3) is encountered, simply return their respective results. The recursive tree diagram is shown in Figure:



CMC UNIVERSITY

Calculating the Fibonacci series

In the dynamic implementation of the Fibonacci series, we store the results of previously solved sub-problems in a lookup list.

First check whether the Fibonacci of any number is already computed:

- If it is already computed, then we return the stored value from the lookup[n].
- After computing the solution of the sub-problem, it is again stored in the lookup list.

```
1  def dyna_fib(n):
2      if n == 0:
3          return 0
4      if n == 1:
5          return 1
6      if lookup[n] is not None:
7          return lookup[n]
8      lookup[n] = dyna_fib(n-1) + dyna_fib(n-2)
9      return lookup[n]
10
11 lookup = [None]*(1000)
12 for i in range(6):
13     print(dyna_fib(i))
```

CMC UNIVERSITY

Calculating the Fibonacci series

Dynamic programming improves the running time complexity of the algorithm.

- In the recursive approach, for every value, two functions are called

For example, fib(5) calls fib(4) and fib(3),

fib(4) calls fib(3) and fib(2),

and so on.

Thus, the time complexity for the recursive approach is $O(2^n)$

- In the dynamic programming approach, we do not recompute the sub-problems, so for fib(n), we have n total values to be computed, in other words, fib(0), fib(1), fib(2), …, fib(n).

Thus, the total running time complexity is O(n).

CMC UNIVERSITY

Knapsack Problem

Problem: An explorer needs to carry a knapsack weighing no more than m. There are n objects that can be carried. The i-th object has a weight w_i and a use value of v_i. What objects should be carried by the explorer so that their total use value is greatest?

Solution:

Let $f[i, j]$ be the largest possible value by choosing among objects $\{1,2, \ldots, i\}$ with weight limit j. Our goal is to find $f[n, m]$: The largest possible value by selecting among n given objects with a weight limit m.

With a weight limit j, choosing from objects $\{1,2, \ldots, i\}$ to get the largest value would be one of two possibilities:

- If the i-th object is not selected, then $f[i, j]$ is the largest possible value by choosing among the products $\{1,2, \ldots, i\text{-}1\}$ with a weight limit of j. That is, $f[i, j] = f[i - 1, j]$

CMC UNIVERSITY

Knapsack Problem

Solution:

- If the i-th object is selected (only consider this case of course when $w_i \leq j$), then $f[i, j]$ is equal to the i-th object value (= v_i) plus the maximum possible value can be obtained by choosing among the products $\{1,2, \ldots, i - 1\}$ with weight limit $j - w_i$ (= $f[i-1, j-w_i]$). That is, in terms of the obtained value, $f[i, j] = f[i-1, j-w_i] + v_i$.

By constructing $f[i, j]$, we deduce the recurrence formula:

$$f[i,j] = \begin{cases} f[i-1,j], & if\ i < w_i \\ \max\{f[i-1,j], f[i-1,j-w_i] + v_i\}, & if\ j \geq w_i \end{cases}$$

To calculate the elements on row i of table f, we need to know the elements on the previous row (row i-1). So if we know row 0 of table f, we can calculate every element in the table. It follows that the dynamic programming basis $f[0, j]$, by definition it is the largest possible value by choosing among 0 objects, so it is obvious that $f[0, j] = 0$.

CMC UNIVERSITY

Knapsack Problem

Solution:

After calculating the table of solutions, we are interested in $f[n, m]$, which is the maximum value obtained when choosing in all n objects with weight limit m. The rest is to indicate the solution to choose the objects.

- If $f[n, m] = f[n - 1, m]$ (the optimal solution does not choose object n), trace to $f[n -1, m]$
- If $f[n, m] \neq f[n - 1, m]$ (the optimal solution does not chooses object n), trace to $f[n - 1, m - w_i]$.

Continue tracing until you reach row 0 of the options table

CMC UNIVERSITY

Knapsack Problem

```python
def knapsack_dynamic_programming(w, v, m):
    n = len(w)
    f = [[0] * (m + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for j in range(m + 1):
            if w[i - 1] <= j:
                f[i][j] = max(v[i - 1] + f[i - 1][j - w[i - 1]],
                    f[i - 1][j])
            else:
                f[i][j] = f[i - 1][j]

    return f[n][m]

w = [2, 3, 4, 5, 6, 7, 8]
v = [3, 4, 5, 6, 9, 15, 20]
m = 20
result = knapsack_dynamic_programming(w, v, m)
print("Maximum value:", result)
```

```
Maximum value: 42
```

0, 1, 5, 6

CMC UNIVERSITY

Exercises

1. Given the natural number $n <= 400$. Tell how many ways to decompose the number n into the sum of the sequence of positive integers (the decomposition ways being permutations of each other, counting only one way).

For example, $n = 5$ has 7 ways of analysis:

1. 5 = 1 + 1 + 1 + 1 + 1
2. 5 = 1 + 1 + 1 + 2
3. 5 = 1 + 1 + 3
4. 5 = 1 + 2 + 2
5. 5 = 1 + 4
6. 5 = 2 + 3
7. 5 = 5

CMC UNIVERSITY

Exercises

2. Given the sequence of integers $A = (a_1, a_2, \ldots, a_n)$. A subsequence of A is a way of selecting from A a number of elements in the same order. Thus, there are 2^n subsequences.

Find the monotonically increasing subsequence of A with the greatest length.

That is, find the largest number k and the index sequence $i_1 < i_2 < \ldots < i_k$ such that $a_{i_1} < a_{i_2} < \ldots < a_{i_k}$

For example,

Sample Input	Sample Output
12 1 2 3 8 9 4 5 6 2 3 9 10	Length = 8 a[1] = 1 a[2] = 2 a[3] = 3 a[6] = 4 a[7] = 5 a[8] = 6 a[11] = 9 a[12] = 10


CMC UNIVERSITY
THANK YOU