# 3

# Algorithm Design Techniques and Strategies

In the field of computing, algorithm design is very important for IT professionals for improving their skills and enabling growth in the industry. The algorithm design process starts with a substantial number of real-world computing problems, which must be clearly formulated for efficiently building the solution using one of the possible techniques from the range of algorithm design techniques available. The world of algorithms contains a plethora of techniques and design principles, mastery of which is required to tackle more difficult problems in the field. Algorithm designs are important in computer science, in general, to efficiently design the solution for a precisely formulated problem since a very sophisticated and complex problem can easily be solved with an appropriate algorithm design technique.

In this chapter, we will discuss the ways in which different kinds of algorithms can be categorized. Design techniques will be described and illustrated, and we will further discuss the analysis of algorithms. Finally, we will provide detailed implementations for a few very important algorithms.

In this chapter, we will look at the following algorithm design techniques:

- Divide and conquer
- Dynamic programming
- Greedy algorithms

# Algorithm design techniques

Algorithm design is a powerful tool for viewing and clearly understanding well-posed, real-world problems. A straightforward, or **brute-force**, approach is available that is very simple, yet effective, for many problems. The brute-force approach is trying all possible combinations of solutions in order to solve any problem. For example, suppose a salesperson has to visit 10 cities across the country. In which order should the cities be visited in order to minimize the total distance traveled? The brute-force approach to this problem will be to calculate the total distance for all possible combinations of routes, and then select the route that provides the smallest distance.

As you might guess, the brute-force algorithm is not efficient.

It can provide useful solutions for limited input sizes, but it becomes very inefficient when the input size becomes large. Therefore, we will break the process down into two fundamental components for finding the optimal solution for a computing problem:

1. Formulate the problem clearly
2. Identify the appropriate algorithm design technique based on the structure of the problem for an efficient solution

That is why the study of algorithm design becomes very important when developing scalable and robust systems. Design and analysis are important in the first instance because they assist in developing algorithms that are organized and easy to understand. Design technique guidelines also help in developing new algorithms easily for complex problems. Moreover, design techniques can also be used to categorize the algorithms and this also helps to understand them better. There are several algorithm paradigms as follows:

- Recursion
- Divide and conquer
- Dynamic programming
- Greedy algorithms

Since we will be using recursion many times while discussing different algorithm design techniques, let us first understand the concept of recursion, and thereafter, we will discuss different algorithm design techniques.

# Recursion

A recursive algorithm calls itself repeatedly in order to solve the problem until a certain condition is fulfilled. Each recursive call itself spins off other recursive calls. A recursive function can be in an infinite loop; therefore, it is required that each recursive function adheres to certain properties. At the core of a recursive function are two types of cases:

1. **Base cases**: These tell the recursion when to terminate, meaning the recursion will be stopped once the base condition is met
2. **Recursive cases**: The function calls itself recursively, and we progress toward achieving the base criteria

A simple problem that naturally lends itself to a recursive solution is calculating factorials. The recursive factorial algorithm defines two cases: the base case when *n* is zero (the terminating condition) and the recursive case when *n* is greater than zero (the call of the function itself). A typical implementation is as follows:

```
def factorial(n):
    # test for a base case
    if n == 0:
        return 1
    else:
    # make a calculation and a recursive call
        return n*factorial(n-1)

print(factorial(4))
```

This produces the following output:

```
24
```

To calculate the factorial of 4, we require four recursive calls, plus the initial parent call, as can be seen in *Figure 3.1*. The details of how these recursive calls work is as follows. Initially, the number 4 is passed to the factorial function, which will return the value 4 multiplied by the factorial of (4-1=3). For this, the number 3 is again passed to the factorial function, which will return the value 3 multiplied by the factorial of (3-1=2). Similarly, in the next iteration, the value 2 is multiplied by the factorial of (2-1 =1).

This continues until we reach the factorial of `0`, which returns `1`. Now, each function returns the value to finally compute `1*1*2*3*4=24`, which is the final output of the function.

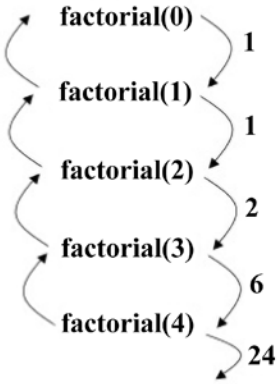


*Figure 3.1: The flow of execution of the factorial 4*

We discussed the concept of recursion, which will be very useful in understanding the implementation of different algorithm paradigms. So, now let us move on to the distinct algorithm design strategies in turn, starting with the divide-and-conquer technique in the next section.

# Divide and conquer

One of the important and effective techniques for solving a complex problem is 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. More specifically, in divide-and-conquer design, the problem is divided into two smaller sub-problems, with each of them being solved recursively. The partial solutions are merged to obtain a final solution. This is a very common problem-solving technique, and is, arguably, the most commonly used approach in algorithm design.

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

Let's have a look at two examples, the binary search and merge sort algorithms, to understand how the divide-and-conquer design technique works.

## Binary search

The binary search algorithm is based on the divide-and-conquer design technique. 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.

Take the example shown in *Figure 3.2*; let's say we want to search for element 4 in the given sorted list of elements. The list is divided in half in each iteration; with the divide-and-conquer strategy, the element is searched $O(logn)$ times.

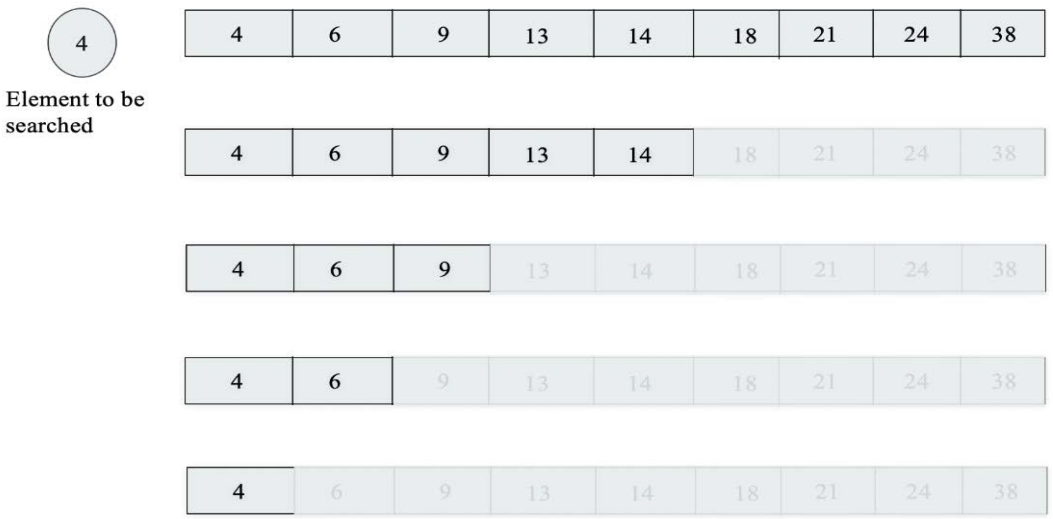


*Figure 3.2: The process of searching for an element using a binary search algorithm*

The Python code for searching for an element in a sorted list of elements is shown here:

```
def binary_search(arr, start, end, key):
    while start <= end:
        mid = start + (end - start)/2
        if arr[mid] == key:
```

```
            return mid
        elif arr[mid] < key:
            start = mid + 1
        else:
            end = mid - 1
    return -1


arr = [4, 6, 9, 13, 14, 18, 21, 24, 38]
x = 13
result = binary_search(arr, 0, len(arr)-1, x)
print(result)
```

When we search for 13 in the given list of elements, the output of the preceding code is 3, which is the position of the searched item.

In the code, initially, the start and end index give the position of the first and last index of the input array `[4, 6, 9, 13, 14, 18, 21, 24, 38]`. The item to be searched that is stored in the variable key is firstly matched with the mid element of the array, and then we discard half of the list and search for the item in another half of the list. The process is iterated until we find the item to be searched, or we reach the end of the list, and we don't find the element.

When analyzing the workings of the binary search algorithm in the worst case, we can see that for a given array of 8 elements, following the first unsuccessful attempt, the list is halved, and then again for an unsuccessful search attempt, the list is of length 2, and finally, only 1 element is left. So, the binary search requires 4 searches. If the size of the list is doubled, in other words, to 16, following the first unsuccessful search, we will have a list of size 8, and that will take a total of 4 searches. Therefore, the binary search algorithm will require 5 searches for a list of 16 items. Thus, we can observe that when we double the number of items in the list, the number of searches required also increments by 1. We can say this as when we have a list of length n, the total number of searches required will be the number of times we repeated halving the list until we are left with 1 element plus 1, which is mathematically equivalent to $(\log_2 n + 1)$. For example, if n=8, the output will be 3, meaning the number of searches required will be 4. The list is divided in half in each iteration; with the divide-and-conquer strategy, the worst-case time complexity of the binary search algorithm is *O(log n)*.

Merge sort is another popular algorithm that is based on the divide-and-conquer design strategy. We will be discussing merge sort in more detail in the next section.

## 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.

In this case, given a list of unsorted elements, we split the list into two approximate halves. We 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. This process is shown in *Figure 3.3*:

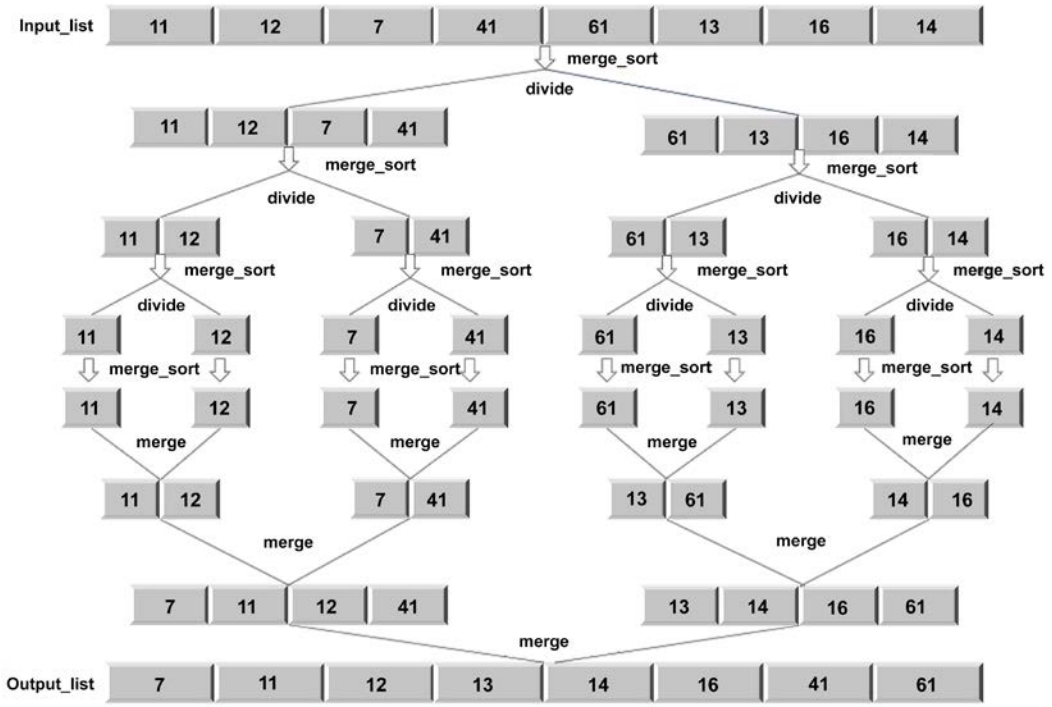


*Figure 3.3: Overview of the merge sort algorithm*

The implementation of the merge sort algorithm is implemented using primarily two methods, namely, the `merge_sort` method, which recursively divides the list. Afterward, we will introduce the `merge` method to combine the results:

```
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)
```

The implementation starts by accepting the list of unsorted elements into the `merge_sort` function. The `if` statement is used to establish the base case, where, if there is only one element in the `unsorted_list`, we simply return that list again. If there is more than one element in the list, we find the approximate middle using `mid_point = len(unsorted_list)//2`.

Using this `mid_point`, we divide the list into two sublists, namely, `first_half` and `second_half`:

```
    first_half = unsorted_list[:mid_point]
    second_half = unsorted_list[mid_point:]
```

A recursive call is made by passing the two sublist to the `merge_sort` function again:

```
    half_a = merge_sort(first_half)
    half_b = merge_sort(second_half)
```

Now, for the merge step, `half_a` and `half_b` are sorted. When `half_a` and `half_b` have passed their values, we call the `merge` function, which will merge or combine the two solutions stored in `half_a` and `half_b`, which are lists:

```
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
```

The `merge` function takes the two lists we want to merge, `first_sublist` and `second_sublist`. The `i` and `j` variables are initialized to `0` and are used as pointers to tell us where we are in the two lists with respect to the merging process.

The final `merged_list` will contain the merged list.

The `while` loop starts comparing the elements in `first_sublist` and `second_sublist`:

```
    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
```

The `if` statement selects the smaller of the two, `first_sublist[i]` or `second_sublist[j]`, and appends it to `merged_list`. The `i` or `j` index is incremented to reflect where we are with the merging step. The `while` loop stops when either sublist is empty.

There may be elements left behind in either `first_sublist` or `second_sublist`. The last two `while` loops make sure that those elements are added to `merged_list` before it is returned. The last call to `merge(half_a, half_b)` will return the sorted list. The following code shows how to pass an array to sort the elements using merge sort:

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

The output will be:

```
[7, 11, 12, 14, 16, 41, 61]
```

Let's give the algorithm a dry run by merging the two sublists `[4, 6, 8]` and `[5, 7, 11, 40]`, shown in *Table 3.1*. In this example, initially, the two sorted sublists are given, and then the, first elements are matched, and since the first element of the first list is smaller, it is moved to `merge_list`. Next, in *step 2*, again, the starting elements from both of the lists are matched, and the smaller element, which is from the second list, is moved to `merge_list`. The same process is repeated until one of the lists becomes empty.

| **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] |

*Table 3.1: Example of merging two lists*

This process can also be seen in *Figure 3.4*:

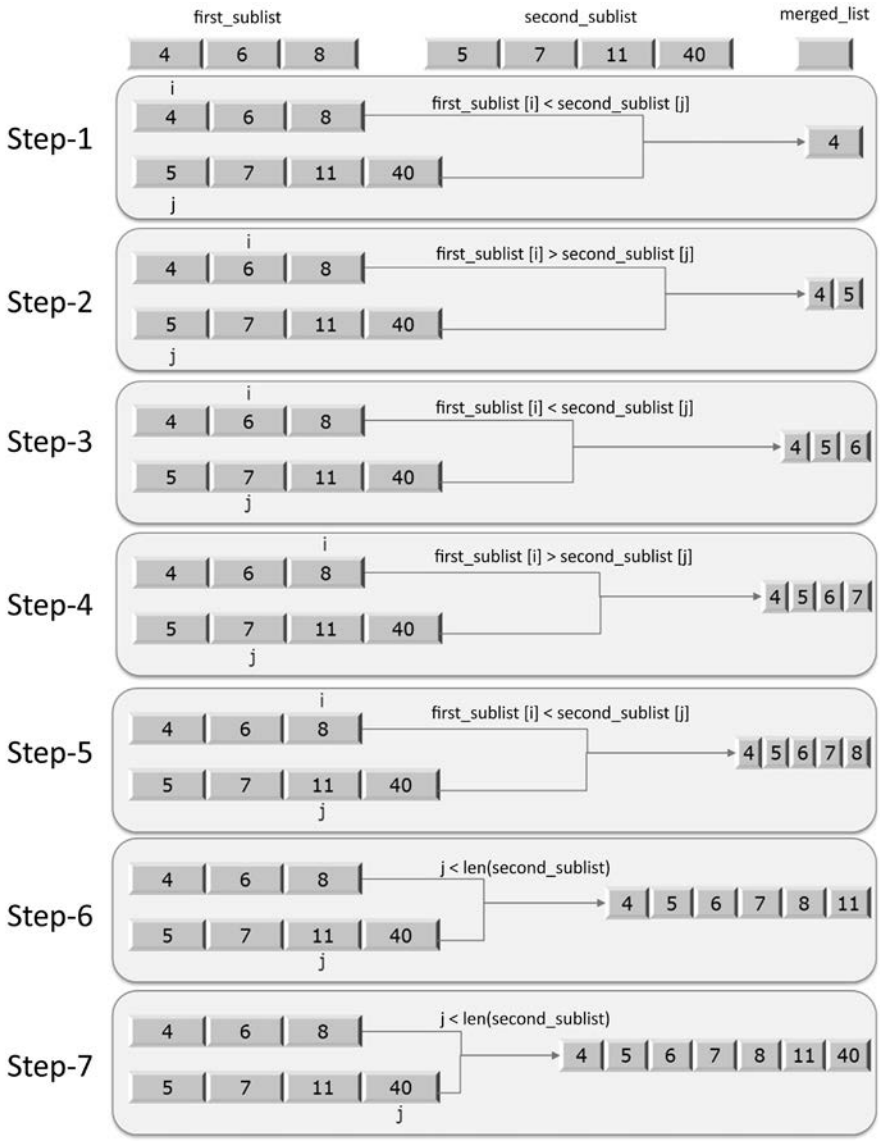


*Figure 3.4: The process of merging the two sublists*

After one of the lists becomes empty, like after *step 4* in this example, at this point in the execution, the third `while` loop in the `merge` function kicks in to move `11` and `40` into `merged_list`. The returned `merged_list` will contain the fully sorted list.

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, which will take $O(log\ n)$, 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 $(n)$ time.

Hence, the merge sort algorithm has a runtime complexity of $O(log\ n)$ $T(n) = O(n) * O(log\ n) = O(n\ log\ n)$.

We have discussed the divide-and-conquer algorithm design technique with the help of a few examples. In the next section, we will discuss another algorithm design technique: dynamic programming.

# Dynamic programming

Dynamic programming is the most powerful design technique for solving optimization problems. Such problems generally have many possible solutions. The basic idea of dynamic programming is based on the intuition of the divide-and-conquer technique. Here, essentially, we explore the space of all the possible solutions by decomposing the problem into a series of sub-problems and then combining the results to compute the correct solution for the large problem. The divide-and-conquer algorithm is used to solve a problem by combining the solutions of the non-overlapping (disjoint) sub-problems, whereas dynamic programming is used when the sub-problems are overlapping, meaning that the sub-problems share sub-sub-problems. The dynamic programming technique is similar to divide and conquer in that a problem is broken down into smaller problems. However, in divide and conquer, each sub-problem has to be solved before its results can be used to solve bigger problems. In contrast, dynamic programming-based techniques solve each sub-sub-problems only once and do not recompute the solution to an already-encountered sub-problem. Rather, it uses a remembering technique to avoid the re-computation.

Dynamic programming problems have two important characteristics:

- **Optimal substructure**: Given any problem, if the solution can be obtained by combining the solutions of its sub-problems, then the problem is said to have an optimal substructure. In other words, an optimal substructure means that the optimal solution of the problem can be obtained from the optimal solution of its sub-problems. For example, the $i^{th}$ Fibonacci number from its series can be computed from $(i-1)^{th}$ and $(i-2)^{th}$ Fibonacci numbers; for example, fib(6) can be computed from fib(5) and fib(4).
- **Overlapping sub-problem**: If an algorithm has to repeatedly solve the same sub-problem again and again, then the problem has overlapping sub-problems. For example, fib(5) will have multiple time computations for fib(3) and fib(2).

If a problem has these characteristics, then the dynamic programming approach is useful, since the implementation can be improved by reusing the same solution computed before. In a dynamic programming strategy, the problem is broken down into independent sub-problems, and the intermediate results are cached, which can then be used in subsequent operations.

In the dynamic approach, we divide a given problem into smaller sub-problems. In recursion also, we divide the problem into sub-problems. However, the difference between recursion and dynamic programming is that similar sub-problems can be solved any number of times, but in dynamic programming, we keep track of previously solved sub-problems, and care is taken not to recompute any of the previously encountered sub-problems. One property that makes a problem an ideal candidate for being solved with dynamic programming is that it has an **overlapping set of sub-problems**. Once we realize that the form of sub-problems has repeated itself during computation, we need not compute it again. Instead, we return a pre computed result for that previously encountered sub-problem.

Dynamic programming takes account of the fact that each sub-problem should be solved only once, and to ensure that we never re-evaluate a sub-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**: This technique starts from the initial problem set and divides it into small sub-problems. After the solution to a sub-program has been determined, we store the result of that particular sub-problem. In the future, when this sub-problem is encountered, we only return its pre computed result. Therefore, 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.

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**: This approach depends upon the "size" of the sub-problems. We solve the smaller sub-problems first, and then while solving a particular sub-problem, we already have a solution of the smaller sub-problems on which it depends. Each sub-problem is solved only once, and whenever we try to solve any sub-problem, solutions to all the prerequisite smaller sub-problems are available, which can be used to solve it. In this approach, a given problem is solved by dividing it into sub-problems recursively, with the smallest possible sub-problems then being solved. Furthermore, 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.

Let's consider an example to understand how dynamic programming works. Let us solve the problem of the Fibonacci series using dynamic programming.

## Calculating the Fibonacci series

The Fibonacci series can be demonstrated using a recurrence relation. Recurrence relations are recursive functions that are used to define mathematical functions or sequences. 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
```

Note that the Fibonacci sequence can be generated by putting the values of $n$ in sequence `[0, 1, 2, 3, 4, ...]`. Let's take an example to generate the Fibonacci series to the fifth term:

```
    1 1 2 3 5
```

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

```
def fib(n):
     if n <= 1:
        return 1
     else:
        return fib(n-1) + fib(n-2)
```

```
for i in range(5):
    print(fib(i))
```

This will produce output like the following:

```
1
1
2
3
5
```

In this 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 3.5*:

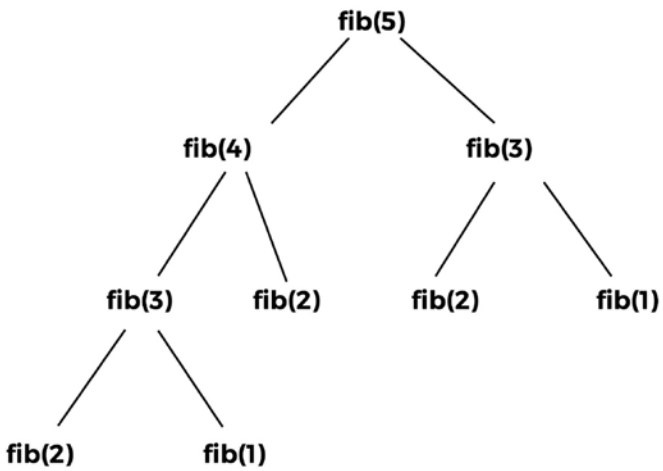


*Figure 3.5: Recursion tree for fib(5)*

We can observe from the overlapping sub-problems from the recursion tree as shown in *Figure 3.6* that 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; for example, the return value for **fib(2)** will always be the same whenever we call it. Likewise, it will also be the same for **fib(1)** and **fib(3)**. So, they are overlapping problems, thus, 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.

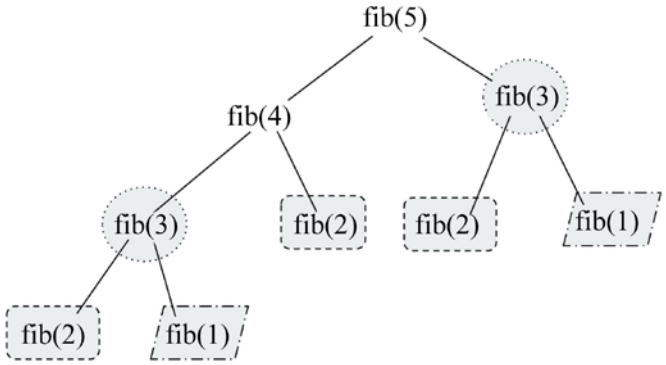


*Figure 3.6: Overlapping sub-problems shown in the recursion tree for fib(5)*

In dynamic programming using the memoization technique, we store the results of the computation of **fib(1)** the first time it is encountered. Similarly, we store return values for **fib(2)** and **fib(3)**. Later, whenever we encounter a call to **fib(1)**, **fib(2)**, or **fib(3)**, we simply return their respective results. The recursive tree diagram is shown in *Figure 3.7*:

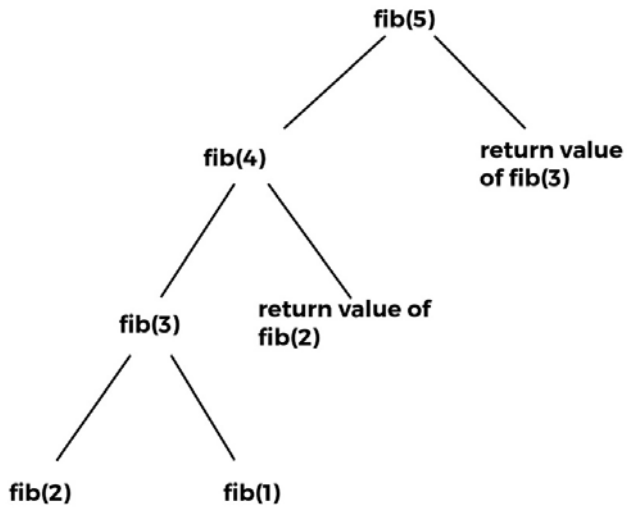


*Figure 3.7: Recursion tree for fib(5) showing re-use of the already computed values*

Thus, in dynamic programming, we have eliminated the need to compute **fib(3)**, **fib(2)**, and **fib(1)** if they are encountered multiple times. This is called the memoization technique, wherein there is no recomputation of overlapping calls to functions when breaking a problem down into its sub-problems.

Hence, the overlapping function calls in our Fibonacci example are **fib(1)**, **fib(2)**, and **fib(3)**. Below is the code for the dynamic programming-based implementation for the Fibonacci series.

```
def dyna_fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    if lookup[n] is not None:
        return lookup[n]

    lookup[n] = dyna_fib(n-1) + dyna_fib(n-2)
    return lookup[n]


lookup = [None]*(1000)

for i in range(6):
    print(dyna_fib(i))
```

This will produce an output like the following:

```
0
1
1
2
3
5
```

In the dynamic implementation of the Fibonacci series, we store the results of previously solved sub-problems in a list (in other words, a lookup in this example code). We 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]`. Otherwise, when we compute its value, it is done through the following code:

```
    if lookup[n] is not None:
        return lookup[n]
```

After computing the solution of the sub-problem, it is again stored in the lookup list. The Fibonacci number of the given value is returned as shown in the following code snippet:

```
lookup[n] = dyna_fib(n-1) + dyna_fib(n-2)
```

Furthermore, in order to store a list of 1,000 elements, we create a list lookup using the `dyna_fib` function:

```
    lookup = [None]*(1000)
```

So, in dynamic programming-based solutions, we use the precomputed solutions in order to compute the final results.

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)**, and then **fib(4)** calls **fib(3)** and **fib(2)**, and so on. Thus, the time complexity for the recursive approach is $O(2^n)$, whereas, 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, we only solve these values once, so the total running time complexity is O(n). Thus, dynamic programming in general improves performance.

In this section, we have discussed the dynamic programming design technique, and in the next section, we discuss the design techniques for greedy algorithms.

# Greedy algorithms

Greedy algorithms often involve optimization and combinatorial problems. In greedy algorithms, the objective is to obtain the optimum solution from many possible solutions in each step. We try to get the local optimum solution, which may eventually lead us to obtain the global optimum solution. The greedy strategy does not always produce the optimal solution. However, the sequence of locally optimal solutions generally approximates the globally optimal solution.

For example, consider that you are given some random digits, say `1`, `4`, `2`, `6`, `9`, and `5`. Now you have to make the biggest number by using all the digits without repeating any digit. To create the biggest number from the given digits using the greedy strategy, we perform the following steps. Firstly, we select the largest digit from the given digits, and then append it to the number and remove the digit from the list until we have no digits left in the list. Once all the digits have been used, we get the largest number that can be formed by using these digits: `965421`. The stepwise solution to this problem is shown in *Figure 3.8*:

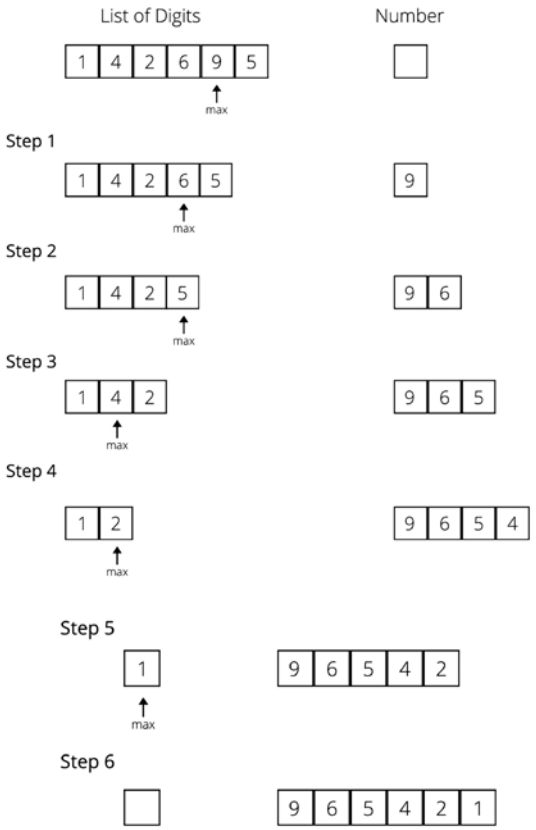


*Figure 3.8: Example of a greedy algorithm*

Let us consider another example to better understand the greedy approach. Say you have to give 29 Indian rupees to someone in the minimum number of notes, giving one note at a time, but never exceeding the owed amount. Assume that we have notes in denominations of 1, 2, 5, 10, 20, and 50. To solve this using the greedy approach, we will start by handing over the 20-rupee note, then for the remaining 9 rupees, we will give a 5-rupee note; for the remaining 4 rupees, we will give the 2-rupee note, and then another 2-rupee note.

In this approach, at each step, we chose the best possible solution and gave the largest available note. Assume that, for this example, we have to use the notes of 1, 14, and 25. Then, using the greedy approach, we will pick the 25-rupee note, and then four 1-rupee notes, which makes a total of 5 notes. However, this is not the minimum number of notes possible . The better solution would be to give notes of 14, 14, and 1. Thus, it is also clear that the greedy approach does not always give the best solution, but a feasible and simple one.

The classic example is to apply the greedy algorithm to the traveling salesperson problem, where a greedy approach always chooses the closest destination first. In this problem, a greedy approach always chooses the closest unvisited city in relation to the current city; in this way, we are not sure that we will get the best solution, but we surely get an optimal solution. This shortest-path strategy involves finding the best solution to a local problem in the hope that this will lead to a global solution.

Listed here are many popular standard problems where we can use greedy algorithms to obtain the optimum solution:

- Kruskal's minimum spanning tree
- Dijkstra's shortest path problem
- The Knapsack problem
- Prim's minimal spanning tree algorithm
- The traveling salesperson problem

Let us discuss one of the popular problems, in other words, the shortest path problem, which can be solved using the greedy approach, in the next section.

## Shortest path problem

The shortest path problem requires us to find out the shortest possible route between nodes on a graph. Dijkstra's algorithm is a very popular method for solving this using the greedy approach. The algorithm is used to find the shortest distance from a source to a destination node in a graph.

Dijkstra's algorithm works for weighted directed and undirected graphs. The algorithm produces the output of a list of the shortest path from a given source node, A, in a weighted graph. The algorithm works as follows:

1. Initially, mark all the nodes as unvisited, and set their distance from the given source node to infinity (the source node is set to zero).
2. Set the source node as the current one.
3. For the current node, look for all the unvisited adjacent nodes, and compute the distance to that node from the source node through the current node. Compare the newly computed distance to the currently assigned distance, and if it is smaller, set this as the new value.

Once we have considered all the unvisited adjacent nodes of the current node, we mark it as visited.

If the destination node has been marked visited, or if the list of unvisited nodes is empty, meaning we have considered all the unvisited nodes, then the algorithm is finished.

We next consider the next unvisited node that has the shortest distance from the source node. Repeat *steps 2* to *6*.

Consider the example in *Figure 3.9* of a weighted graph with six nodes [A, B, C, D, E, and F] to understand how Dijkstra's algorithm works.

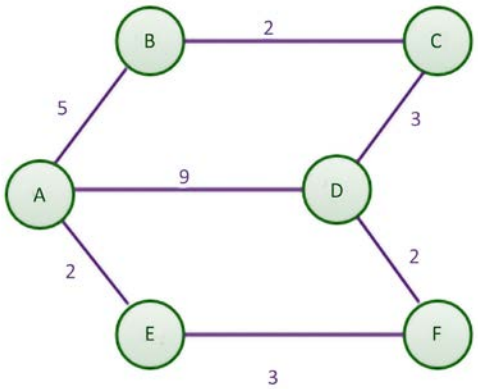


*Figure 3.9: Example weighted graph with six nodes*

By manual inspection, the shortest path between nodes **A** and **D**, at first glance, seems to be the direct line with a distance of 9. However, the shortest route means the lowest total distance, even if this comprises several parts. By comparison, traveling from node **A** to **E**, then from **E** to **F**, and finally to **D** will incur a total distance of 7, making it a shorter route.

We would implement the shortest path algorithm with a single source. It would determine the shortest path from the origin, which in this case is **A**, to any other node in the graph. In *Chapter 9*, *Graphs and Other Algorithms*, we will discuss how to represent a graph with an adjacency list. We use an adjacency list along with the weight/cost/distance on every edge to represent the graph, as shown in the following Python code. The adjacency list for the diagram and table is as follows:

```
graph = dict()
graph['A'] = {'B': 5, 'D': 9, 'E': 2}
graph['B'] = {'A': 5, 'C': 2}
graph['C'] = {'B': 2, 'D': 3}
graph['D'] = {'A': 9, 'F': 2, 'C': 3}
graph['E'] = {'A': 2, 'F': 3}
graph['F'] = {'E': 3, 'D': 2}
```

We will return to the rest of the code after a visual demonstration, but don't forget to declare the graph to ensure the code runs correctly.

The nested dictionary holds the distance and adjacent nodes. A table is used to keep track of the shortest distance from the source in the graph to any other node. *Table 3.2* is the starting table:

| Node | Shortest distance from source | Previous node |
|---|---|---|
| **A** | 0 | None |
| **B** | ∞ | None |
| **C** | ∞ | None |
| **D** | ∞ | None |
| **E** | ∞ | None |
| **F** | ∞ | None |

*Table 3.2: Initial table showing the shortest distance from the source*

When the algorithm starts, the shortest distance from the given source node (**A**) to any of the nodes is unknown. Thus, we initially set the distance to all other nodes to infinity, with the exception of node **A**, as the distance from node **A** to node **A** is **0**. No prior nodes have been visited when the algorithm begins. Therefore, we mark the previous node column of node **A** as **None**.

In *step 1* of the algorithm, we start by examining the adjacent nodes to node **A**. To find the shortest distance from node **A** to node **B**, we need to find the distance from the start node to the previous node of node **B**, which happens to be **A**, and add it to the distance from node **A** to node **B**. We do this for the other adjacent nodes of **A**, these being **B**, **E**, and **D**. This is shown in *Figure 3.10*:

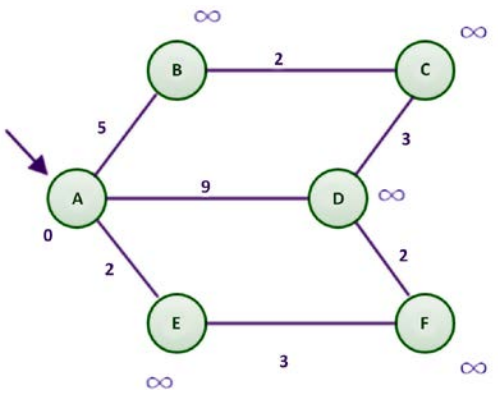


*Figure 3.10: A sample graph for Dijkstra's algorithm*

Firstly, we take the adjacent node **E** as its distance from node **A** is the minimum; the distance from the start node (**A**) to the previous node (**None**) is 0, and the distance from the previous node to the current node (**E**) is **2**.

This sum is compared with the data in the shortest distance column of node **E** (refer to *Table 3.3*). Since **2** is less than infinity (∞), we replace ∞ with the smaller of the two, in other words, **2**. Similarly, the distance from node **A** to nodes **B** and **D** is compared with the existing shortest distance to these nodes from node **A**. Any time the shortest distance of a node is replaced by a smaller value, we need to update the previous node column for all the adjacent nodes of the current node.

After this, we mark node **A** as visited (represented in blue in *Figure 3.11*):

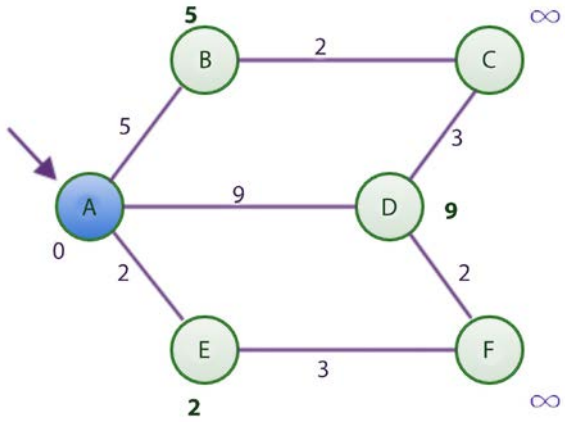


*Figure 3.11: Shortest distance graph after visiting node A using Dijkstra's algorithm*

At the end of *step 1*, the table looks like that shown in *Table 3.3*, in which the shortest distance from node **A** to nodes **B**, **D**,and **E** are updated.

| Node | Shortest distance from source | Previous node |
|---|---|---|
| **A*** | 0 | None |
| **B** | 5 | A |
| **C** | ∞ | None |
| **D** | 9 | A |
| **E** | 2 | A |
| **F** | ∞ | None |

*Table 3.3: Shortest distance table after visiting node A*

At this point, node **A** is considered visited. As such, we add node **A** to the list of visited nodes. In the table, we show that node **A** has been visited by appending an asterisk sign to it.

In the second step, we find the node with the shortest distance using *Table 3.3* as a guide. Node **E**, with its value of 2, has the shortest distance. To reach node **E**, we must visit node **A** and cover a distance of **2**.

Now, the adjacent nodes of node **E** are nodes **A** and **F**. Since node **A** has already been visited, we will only consider node **F**. To find the shortest route or distance to node **F**, we must find the distance from the starting node to node **E** and add it to the distance between nodes **E** and **F**. We can find the distance from the starting node to node **E** by looking at the shortest distance column of node **E**, which has a value of **2**. The distance from nodes **E** to **F** can be obtained from the adjacency list, which is **3**. These two total 5, which is less than infinity. Remember that we are examining the adjacent node **F**. Since there are no more adjacent nodes to node **E**, we mark node **E** as visited. Our updated table and the figure will have the following values, shown in *Table 3.4* and *Figure 3.12*:

| Node | Shortest distance from source | Previous node |
|---|---|---|
| **A*** | 0 | None |
| **B** | 5 | A |
| **C** | ∞ | None |
| **D** | 9 | A |
| **E*** | 2 | A |
| **F** | 5 | E |

*Table 3.4: Shortest distance table after visiting node E*

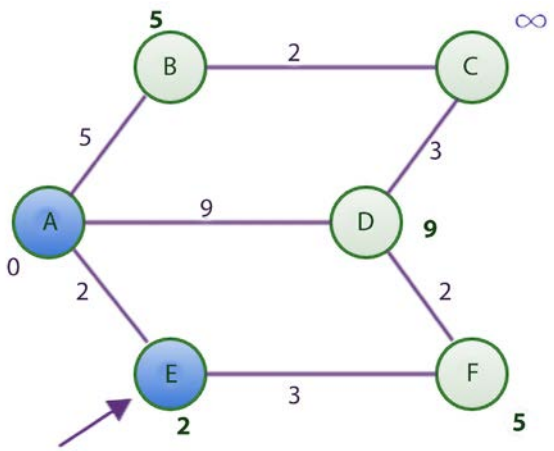


*Figure 3.12: Shortest distance graph after visiting node E using Dijkstra's algorithm*

After visiting node **E**, we find the smallest value in the Shortest distance column of *Table 3.4*, which is 5 for nodes **B** and **F**. Let us choose **B** instead of **F** for alphabetical reasons. The adjacent nodes of **B** are nodes **A** and **C** since node **A** has already been visited. Using the rule we established earlier, the shortest distance from **A** to **C** is 7, which is computed as the distance from the starting node to node **B**, which is 5, while the distance from node **B** to **C** is 2. Since 7 is less than infinity, we update the shortest distance to 7 and update the previous node column with node **B** in *Table 3.4*.

Now, **B** is also marked as visited (represented in blue in *Figure 3.13*).

| Node | Shortest distance from source | Previous node |
|---|---|---|
| **A*** | 0 | None |
| **B*** | 5 | A |
| **C** | 7 | B |
| **D** | 9 | A |
| **E*** | 2 | A |
| **F** | 5 | E |

*Table 3.5: Shortest distance table after visiting node B*

The new state of the table is as follows, in *Table 3.5*:

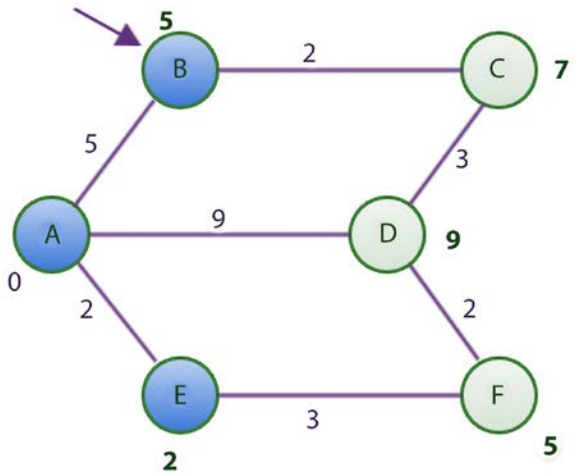


*Figure 3.13: Shortest distance graph after visiting node B using Dijkstra's algorithm*

The node with the shortest distance yet unvisited is node **F**. The adjacent nodes to **F** are nodes **D** and **E**. Since node **E** has already been visited, we will focus on node **D**. To find the shortest distance from the starting node to node **D**, we calculate this distance by adding the distance from nodes **A** to **F** to the distance from nodes **F** to **D**. This totals 7, which is less than **9**. Thus, we update the **9** with **7** and replace **A** with **F** in node **D**'s previous node column of *Table 3.5*.

Node **F** is now marked as visited (represented in blue in *Figure 3.14*).

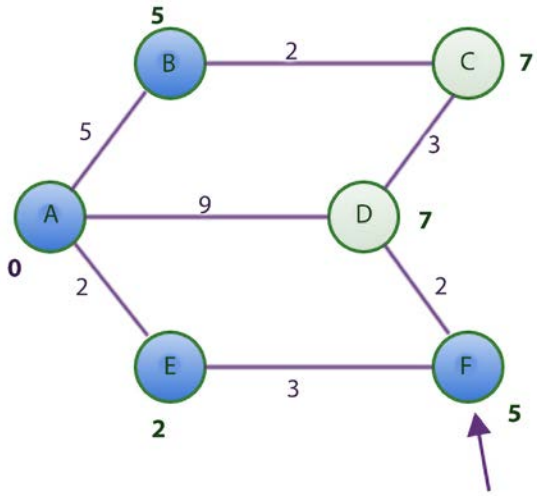


*Figure 3.14: Shortest distance graph after visiting node F using Dijkstra's algorithm*

Here is the updated table, as shown in *Table 3.6*:

| Node | Shortest distance from source | Previous node |
|---|---|---|
| **A*** | 0 | None |
| **B*** | 5 | A |
| **C** | 7 | B |
| **D** | 7 | F |
| **E*** | 2 | A |
| **F*** | 5 | E |

*Table 3.6: Shortest distance table after visiting node F*

Now, only two unvisited nodes are left, **C** and **D**, both with a distance cost of **7**. In alphabetical order, we choose to consider node **C** because both nodes have the same shortest distance from the starting node **A**.

However, all the adjacent nodes to **C** have been visited (represented in blue in *Figure 3.15*). Thus, we have nothing to do but mark node **C** as visited. The table remains unchanged at this point.

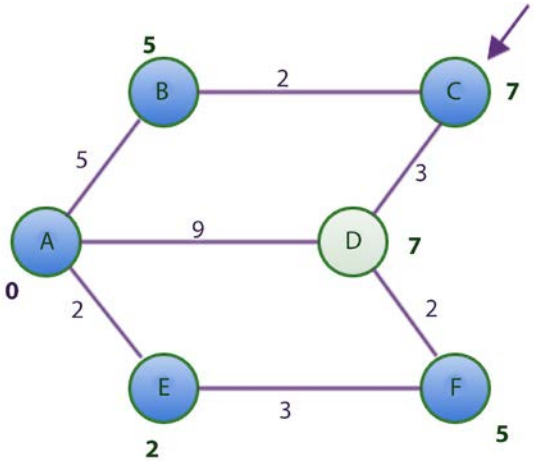


*Figure 3.15: Shortest distance graph after visiting node C using Dijkstra's algorithm*

Lastly, we take node **D** and find out that all its adjacent nodes have been visited too. We only mark it as visited (represented in blue in *Figure 3.16*).

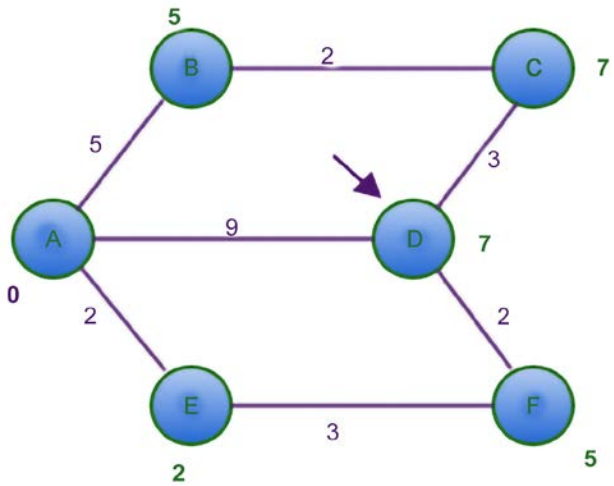


*Figure 3.16: Shortest distance graph after visiting node D using Dijkstra's algorithm*

The table remains unchanged, as shown in *Table 3.7*:

| Node | Shortest distance from source | Previous node |
|---|---|---|
| A* | 0 | None |
| B* | 5 | A |
| C* | 7 | B |
| D* | 7 | F |
| E* | 2 | A |
| F* | 5 | E |

*Table 3.7: Shortest distance table after visiting node F*

Let's verify *Table 3.7* with our initial graph. From the graph, we know that the shortest distance from **A** to **F** is **5**.

According to the table, the shortest distance from the source column for node **F** is 5. This is true. It also tells us that to get to node **F**, we need to visit node **E**, and from **E** to node **A**, which is our starting node. This is actually the shortest path from node **A** to node **F**.

Now, we will discuss the Python implementation of Dijkstra's algorithm to find the shortest path. We begin the program for finding the shortest distance by representing the table that enables us to track the changes in the graph. For the initial *Figure 3.8* that we used, here is a dictionary representation of the table to accompany the graph representation we showed earlier in the section:

```
    table = {
    'A': [0, None],
    'B': [float("inf"), None],
    'C': [float("inf"), None],
    'D': [float("inf"), None],
    'E': [float("inf"), None],
    'F': [float("inf"), None],
}
```

The initial state of the table uses `float("inf")` to represent infinity. Each key in the dictionary maps to a list. At the first index of the list, the shortest distance from the source, node A is stored. At the second index, the previous node is stored:

```
DISTANCE = 0
PREVIOUS_NODE = 1
INFINITY = float('inf')
```

Here, the shortest path's column index is referenced by `DISTANCE`. The previous node column's index is referenced by `PREVIOUS_NODE`.

Firstly, we discuss the helper methods that we will be using while implementing the main function to find the shortest path, in other words, `find_shortest_path`. The first helper method is `get_shortest_distance`, which returns the shortest distance of a node from the source node:

```
def get_shortest_distance(table, vertex):
    shortest_distance = table[vertex][DISTANCE]
    return shortest_distance
```

The `get_shortest_distance` function returns the value stored in index 0 of the table. At that index, we always store the shortest distance from the starting node up to `vertex`. The `set_shortest_distance` function only sets this value as follows:

```
def set_shortest_distance(table, vertex, new_distance):
    table[vertex][DISTANCE] = new_distance
```

When we update the shortest distance of a node, we update its previous node using the following method:

```
def set_previous_node(table, vertex, previous_node):
    table[vertex][PREVIOUS_NODE] = previous_node
```

Remember that the `PREVIOUS_NODE` constant equals 1. In the table, we store the value of `previous_node` at `table[vertex][PREVIOUS_NODE]`. To find the distance between any two nodes, we use the `get_distance` function:

```
def get_distance(graph, first_vertex, second_vertex):
    return graph[first_vertex][second_vertex]
```

The last helper method is the `get_next_node` function:

```
    def get_next_node(table, visited_nodes):
        unvisited_nodes = list(set(table.keys()).difference(set(visited_
nodes)))
        assumed_min = table[unvisited_nodes[0]][DISTANCE]
        min_vertex = unvisited_nodes[0]
        for node in unvisited_nodes:
            if table[node][DISTANCE] < assumed_min:
                assumed_min = table[node][DISTANCE]
                min_vertex = node
```

```
        return min_vertex
```

The `get_next_node` function resembles a function to find the smallest item in a list. The function starts off by finding the unvisited nodes in our table by using `visited_nodes` to obtain the difference between the two sets of lists. The very first item in the list of `unvisited_nodes` is assumed to be the smallest in the shortest distance column of `table`.

If a lesser value is found while the `for` loop runs, `min_vertex` will be updated. The function then returns `min_vertex` as the unvisited vertex or node with the smallest shortest distance from the source.

Now all is set up for the main function of the algorithm, in other words, `find_shortest_path`, as shown here:

```
def find_shortest_path(graph, table, origin):
    visited_nodes = []
    current_node = origin
    starting_node = origin
    while True:
        adjacent_nodes = graph[current_node]
        if set(adjacent_nodes).issubset(set(visited_nodes)):
            # Nothing here to do. All adjacent nodes have been visited.
            pass
        else:
            unvisited_nodes =
                set(adjacent_nodes).difference(set(visited_nodes))
            for vertex in unvisited_nodes:
                distance_from_starting_node =
                    get_shortest_distance(table, vertex)
                if distance_from_starting_node == INFINITY and
                   current_node == starting_node:
                    total_distance = get_distance(graph, vertex,
                                                  current_node)
                else:
                    total_distance = get_shortest_distance (table,
```

```
                    current_node) + get_distance(graph, current_node,
                                                 vertex)

                if total_distance < distance_from_starting_node:
                    set_shortest_distance(table, vertex,
                                          total_distance)
                    set_previous_node(table, vertex, current_node)

        visited_nodes.append(current_node)
        #print(visited_nodes)

        if len(visited_nodes) == len(table.keys()):
            break

        current_node = get_next_node(table,visited_nodes)
    return (table)
```

In the preceding code, the function takes the graph, represented by the adjacency list, the table, and the starting node as input parameters. We keep the list of visited nodes in the `visited_nodes` list. The `current_node` and `starting_node` variables both point to the node in the graph that we choose to make our starting node. The `origin` value is the reference point for all other nodes with respect to finding the shortest path.

The main process of the function is implemented by the `while` loop. Let's break down what the `while` loop is doing. In the body of the `while` loop, we consider the current node in the graph that we want to investigate and initially get all the adjacent nodes of the current node with `adjacent_nodes = graph[current_node]`. The `if` statement is used to find out whether all the adjacent nodes of `current_node` have been visited.

When the `while` loop is executed for the first time, `current_node` will contain node **A** and `adjacent_nodes` will contain nodes **B**, **D**, and **E**. Furthermore, `visited_nodes` will be empty. If all nodes have been visited, we only move on to the statements further down the program, otherwise, we begin a whole new step.

The `set(adjacent_nodes).difference(set(visited_nodes))` statement returns the nodes that have not been visited. The loop iterates over this list of unvisited nodes:

```
    distance_from_starting_node = get_shortest_distance(table, vertex)
```

The `get_shortest_distance(table, vertex)` helper method will return the value stored in the shortest distance column of our table, using one of the unvisited nodes referenced by `vertex`:

```
    if distance_from_starting_node == INFINITY and current_node ==
starting_node:
        total_distance = get_distance(graph, vertex, current_node)
```

When we are examining the adjacent nodes of the starting node, `distance_from_starting_node == INFINITY and current_node == starting_node` will evaluate to `True`, in which case we only have to find the distance between the starting node and vertex by referencing the graph:

```
    total_distance = get_distance(graph, vertex, current_node)
```

The `get_distance` method is another helper method we use to obtain the value (distance) of the edge between `vertex` and `current_node`. If the condition fails, then we assign to `total_distance` the sum of the distance from the starting node to `current_node` and the distance between `current_node` and `vertex`.

Once we have our total distance, we need to check whether `total_distance` is less than the existing data in the shortest distance column of our table. If it is less, then we use the two helper methods to update that row:

```
    if total_distance < distance_from_starting_node:
        set_shortest_distance(table, vertex, total_distance)
        set_previous_node(table, vertex, current_node)
```

At this point, we add `current_node` to the list of visited nodes:

```
    visited_nodes.append(current_node)
```

If all nodes have been visited, then we must exit the `while` loop. To check whether this is the case, we compare the length of the `visited_nodes` list with the number of keys in our table. If they have become equal, we simply exit the `while` loop.

The `get_next_node` helper method is used to fetch the next node to visit. It is this method that helps us find the minimum value in the shortest distance column from the starting nodes using our table. The whole method ends by returning the updated table. To print the table, we use the following statements:

```
shortest_distance_table = find_shortest_path(graph, table, 'A')
for k in sorted(shortest_distance_table):
    print("{} - {}".format(k,shortest_distance_table[k]))
```

This is the output for the preceding code snippet:

```
A - [0, None]
B - [5, 'A']
C - [7, 'B']
D - [7, 'F']
E - [2, 'A']
F - [5, 'E']
```

The running time complexity of Dijkstra's algorithm depends on how the vertices are stored and retrieved. Generally, the min-priority queue is used to store the vertices of the graph, thus, the time complexity of Dijkstra's algorithm depends on how the min-priority queue is implemented.

In the first case, the vertices are stored numbered from 1 to $|V|$ in an array. Here, each operation for searching a vertex from the entire array will take O(V) time, making the total time complexity $O(V2\ V^2 + E) = O(V^2)$. Furthermore, if the min-priority queue is implemented using the Fibonacci heap, the time taken for each iteration of the loop and extracting the minimum node will take $O(|V|)$ time. Further, iterating over all the vertices' adjacent nodes and updating the shortest distance takes $O(|E|)$ time, and each priority value update takes $O(\log|V|)$ time, which makes $O(|E| + \log|V|)$. Thus, the total running time complexity of the algorithm becomes $O(|E| + |V|log\ |V|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges.

# Summary

Algorithm design techniques are very important in order to formulate, understand, and develop an optimal solution to a complex problem. In this chapter, we have discussed algorithm design techniques, which are very important in the field of computer science. Important categories of algorithm design, such as dynamic programming, greedy approach, and divide and conquer, we discussed in detail along with implementations of important algorithms.

The dynamic programming and divide-and-conquer techniques are quite similar in the sense that both solve a bigger problem by combining the solutions of the sub-problems. Here, the divide-and-conquer technique partitions the problem into disjointed sub-problems, solving them recursively, and then combines the solutions of the sub-problems to obtain the solution of the original problem, whereas, in dynamic programming, this technique is employed when sub-problems overlap, and recomputation of the same sub-problem is avoided. Furthermore, in the greedy approach-based algorithm design technique, at each step in the algorithm, the best choice is taken that looks likely to attain the solution.

In the next chapter, we will be discussing important data structures such as `Linked Lists` and `Pointer Structures`.

# Exercises

1. Which of the following options will be correct when a top-down approach of dynamic programming will be applied to solve a given problem related to the space and time complexity?

    a. It will increase both time and space complexity.

    b. It will increase the time complexity, and decrease the space complexity

    c. It will increase the space complexity, and decrease the time complexity

    d. It will decrease both time and space complexities.

2. Dijkstra's single shortest path algorithm is applied on edge weighted directed graph shown in Figure 3.17. What will be the order of the nodes for the shortest path distance path (Assume A as source) ?

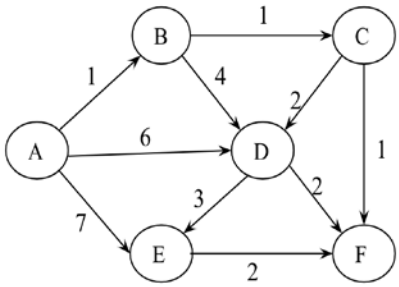


*Figure 3.17: An edge-weighted directed graph*

3. Consider the weights and values of the items listed in *Table 3.8*. Note that there is only one unit of each item.

| Item | Weight | Value |
|---|---|---|
| A | 2 | 10 |
| B | 10 | 8 |
| C | 4 | 5 |
| D | 7 | 6 |

*Table 3.8: The weights and values of different items*

We need to maximize the value; the maximum weight should be 11 kg. No item may be split. Establish the values of the items using a greedy approach.

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

`https://packt.link/MEvK4`