**Two Pointers**

The **two pointers** pattern is a versatile technique used in problem-solving to efficiently traverse or manipulate sequential data structures, such as arrays or linked lists. As the name suggests, it involves maintaining two pointers that traverse the data structure in a coordinated manner, typically starting from different positions or moving in opposite directions. These pointers dynamically adjust based on specific conditions or criteria, allowing for the efficient exploration of the data and enabling solutions with optimal time and space complexity. Whenever there’s a requirement to find two data elements in an array that satisfy a certain condition, the two pointers pattern should be the first strategy to come to mind.

The pointers can be used to iterate through the data structure in one or both directions, depending on the problem statement. For example, to identify whether a string is a palindrome, we can use one pointer to iterate the string from the beginning and the other to iterate it from the end. At each step, we can compare the values of the two pointers and see if they meet the palindrome properties.

|  |  |
| --- | --- |
| A screenshot of a computer  Description automatically generated | A diagram of a line with text  Description automatically generated |
| A diagram of a diagram  Description automatically generated |  |

The naive approach to solving this problem would be using nested loops, with a time complexity of O(n2). However, by using two pointers moving toward the middle from either end, we exploit the symmetry property of palindromic strings. This allows us to compare the elements in a single loop, making the algorithm more efficient with a time complexity of O(n).

**Does your problem match this pattern?**

Yes, if all of these conditions are fulfilled:

* **Linear data structure:** The input data can be traversed in a linear fashion, such as an array, linked list, or string.
* **Process pairs:** Process data elements at two different positions simultaneously.
* **Dynamic pointer movement:** Both pointers move independently of each other according to certain conditions or criteria. In addition, both pointers might move along the same or two different data structures.

**Real-world problems**

Many problems in the real world use the two pointers pattern. Let’s look at an example.

* **Memory management:** The two pointers pattern is vital in memory allocation and deallocation. The memory pool is initialized with two pointers: the *start* pointer, pointing to the beginning of the available memory block, and the *end* pointer, indicating the end of the block. When a process or data structure requests memory allocation, the *start* pointer is moved forward, designating a new memory block for allocation. Conversely, when memory is released (deallocated), the *start* pointer is shifted backward, marking the deallocated memory as available for future allocations.

**Fast and Slow Pointers**

Similar to the two pointers pattern, the **fast and slow pointers** pattern uses two pointers to traverse an iterable data structure, but at different speeds, often to identify patterns, detect cycles, or find specific elements. The speeds of the two pointers can be adjusted according to the problem statement. Unlike the two pointers approach, which is concerned with data values, the fast and slow pointers approach is often used to determine specific pattern or structure in the data.

The key idea is that the pointers start at the same location and then start moving at different speeds. Generally, the slow pointer moves forward by a factor of one, and the fast pointer moves by a factor of two. This approach enables the algorithm to detect patterns or properties within the data structure, such as cycles or intersections. If there is a cycle, the two are bound to meet at some point during the traversal. To understand the concept, think of two runners on a track. While they start from the same point, they have different running speeds. If the track is circular, the faster runner will overtake the slower one after completing a lap.

Here’s a simple demonstration of how the fast and slow pointers move along a data structure:

|  |  |
| --- | --- |
|  |  |
|  |  |

**Does your problem match this pattern?**

Yes, if the following condition is fulfilled:

* **Linear data structure:** The input data can be traversed in a linear fashion, such as an array, linked list, or string.

In addition, if either of these conditions is fulfilled:

* **Cycle or intersection detection:** The problem involves detecting a loop within a linked list or an array or involves finding an intersection between two linked lists or arrays.
* **Find the starting element at the second quantile:** The problem involves finding the starting element of the second quantile, i.e., second half, second tertile, second quartile, etc. For example, the problem asks to find the middle element of an array or a linked list.

**Real-world problems**

Many problems in the real world use the fast and slow pointers pattern. Let’s look at some examples.

* **Symlink verification:** A symbolic link, or symlink, is simply a shortcut to another file. Essentially, it’s a file that points to another file. Symlinks can easily create loops or cycles where shortcuts point to each other. To avoid such occurrences, a symlink verification utility can be used, and fast and slow pointers are useful in that case.
* **Compiling an object-oriented program:** Compiling object-oriented programs often involves managing dependencies between various modules stored in separate files for easier maintenance. However, these dependencies can sometimes form cyclic relationships, leading to compilation errors. By employing the fast and slow pointers pattern, these cycles can be detected and resolved, ensuring smooth compilation and execution of the program.

**Sliding Window**

The **sliding window** pattern is used to process sequential data, arrays, and strings, for example, to efficiently solve subarray or substring problems. It involves maintaining a dynamic window that slides through the array or string, adjusting its boundaries as needed to track relevant elements or characters. The window is used to slide over the data in chunks corresponding to the window size, and this can be set according to the problem’s requirements. It may be viewed as a variation of the two pointers pattern, with the pointers being used to set the window bounds.

Imagine you’re in a long hallway lined with paintings, and you’re looking through a narrow frame that only reveals a portion of this hallway at any time. As you move the frame along the hallway, new paintings come into view while others leave the frame. This process of moving and adjusting what’s visible through the frame is akin to how the sliding window technique operates over data.

Why is this method more efficient? Consider we need to find k*k* consecutive integers with the largest sum in an array. The time complexity of the naive solution to this problem would be O(kn) , because we need to compute sums of all subarrays of size *k*. On the other hand, if we employ the sliding window pattern, instead of computing the sum of all elements in the window, we can just subtract the element exiting the window, add the element entering the window, and update the maximum sum accordingly. In this way, we can update the sums in constant time, yielding an overall time complexity of O(n). To summarize, generally, the computations performed every time the window moves should take O(1) time or a slow-growing function, such as the log of a small variable.

The following illustration shows a possibility of how a window could move along an array:

|  |  |
| --- | --- |
|  |  |
|  |  |

**Does your problem match this pattern?**

Yes, if all of these conditions are fulfilled:

* **Contiguous data:** The input data is stored in a contiguous manner, such as an array or string.
* **Processing subsets of elements:** The problem requires repeated computations on a contiguous subset of data elements (a subarray or a substring), such that the window moves across the input array from one end to the other. The size of the window may be fixed or variable, depending on the requirements of the problem.
* **Efficient computation time complexity:** The computations performed every time the window moves take constant or very small time.

**Real-world problems**

Many problems in the real world use the sliding window pattern. Let’s look at some examples.

* **Telecommunications:** Find the maximum number of users connected to a cellular network’s base station in every k-millisecond sliding window.
* **Video streaming:** Given a stream of numbers representing the number of buffering events in a given user session, calculate the median number of buffering events in each one-minute interval.
* **Social media content mining:** Given the lists of topics that two users have posted about, find the shortest sequence of posts by one user that includes all the topics that the other user has posted about.

**Merge Intervals**

The **merge intervals** pattern deals with problems involving overlapping intervals. Each interval is represented by a start and an end time. For example, an interval of [10,20][10,20] seconds means that the interval starts at 1010 seconds and ends at 2020 seconds. This pattern involves tasks such as merging intersecting intervals, inserting new intervals into existing sets, or determining the minimum number of intervals needed to cover a given range. The most common problems solved using this pattern are event scheduling, resource allocation, and time slot consolidation.

The key to understanding this pattern and exploiting its power lies in understanding how any two intervals may overlap. The illustration below shows different ways in which two intervals can relate to each other:

|  |  |
| --- | --- |
|  |  |
|  |  |

**Does your problem match this pattern?**

Yes, if both of these conditions are fulfilled:

* **Array of intervals:** The input data is an array of intervals.
* **Overlapping intervals:** The problem requires dealing with overlapping intervals, either to find their union, their intersection, or the gaps between them.

**Real-world problems**

Many problems in the real world use the merge intervals pattern. Let’s look at some examples.

* **Display busy schedule:** Display the busy hours of a user to other users without revealing the individual meeting slots in a calendar.
* **Schedule a new meeting:** Add a new meeting to the tentative meeting schedule of a user in such a way that no two meetings overlap each other.
* **Task scheduling in operating systems (OS):** Schedule tasks for the OS based on task priority and the free slots in the machine’s processing schedule.

**In-Place Manipulation of a Linked List**

The **in-place manipulation of a linked list** pattern allows us to modify a linked list without using any additional memory. **In-place** refers to an algorithm that processes or modifies a data structure using only the existing memory space, without requiring additional memory proportional to the input size. This pattern is best suited for problems where we need to modify the structure of the linked list, i.e., the order in which nodes are linked together. For example, some problems require a reversal of a set of nodes in a linked list which can extend to reversing the whole linked list. Instead of making a new linked list with reversed links, we can do it in place without using additional memory.

The naive approach to reverse a linked list is to traverse it and produce a new linked list with every link reversed. The time complexity of this algorithm is O(n) while consuming O(n) extra space. How can we implement the in-place reversal of nodes so that no extra space is used? We iterate over the linked list while keeping track of three nodes: the current node, the next node, and the previous node. Keeping track of these three nodes enables us to efficiently reverse the links between every pair of nodes. This in-place reversal of a linked list works in O(n) time and consumes only O(1) space.

The following illustration demonstrates the in-place modification of a linked list by depicting the reversal of the given linked list:

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Does your problem match this pattern?**

Yes, if both of these conditions are fulfilled:

* **Linked list restructuring:** The input data is given as a linked list, and the task is to modify its structure without modifying the data of the individual nodes.
* **In-place modification:** The modifications to the linked list must be made in place, that is, we’re not allowed to use more than O(1)*O*(1) additional space.

**Real-world problems**

Many problems in the real world use the in-place manipulation of a linked list pattern. Let’s look at some examples.

* **File system management:** File systems often use linked lists to manage directories and files. Operations such as rearranging files within a directory can be implemented by manipulating the underlying linked list in place.
* **Memory management**: In low-level programming or embedded systems, dynamic memory allocation and deallocation often involve manipulating linked lists of free memory blocks. Operations such as merging adjacent free blocks or splitting large blocks can be implemented in place to optimize memory usage.

**Two Heaps**

The **two heaps** pattern is a versatile and efficient approach used to solve problems involving dynamic data processing, optimization, and real-time analysis. As the name suggests, this pattern maintains two heaps, which could be either two min heaps, two max heaps, or a min heap and a max heap. Exploiting the heap property, the two heaps pattern is a preferred technique for various problems to implement computationally efficient solutions. For a heap containing n elements, inserting or removing an element takes O(logn) time, while accessing the element at the root is done in O(1) time. The root stores the smallest element in the case of a min heap and the largest element in the case of a max heap.

Let’s explore a few example scenarios to gain a better understanding. In some problems, we’re given a dataset and tasked to divide it into two parts to find the smallest value from one part and the largest value from the other part. To achieve this, we can build two heaps: one min heap and one max heap, from these two subsets of data. The root of the min heap will give us the smallest value from its corresponding dataset, while the root of the max heap will provide the largest value from its dataset. In cases where we need to find the two largest numbers from two different datasets, we’ll use two max heaps to store and manage these datasets. Similarly, to find the two smallest numbers from two different datasets, we would use two min heaps. These examples illustrate the versatility of using min heaps and max heaps to efficiently solve different types of problems by facilitating quick access to the smallest or largest values as required.

The following illustration demonstrates how we can build a min heap or a max heap, and how they can be used to solve several tasks, e.g., finding the smallest or largest element from some data:

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Does your problem match this pattern?**

Yes, if both of these conditions are fulfilled:

* **Static or streaming data:** The input could either be:
  + **Linear data:** The input data is linear but not sorted. If the input data is sorted, this pattern won’t apply.
  + **Stream of data:** The input data is a continuous stream of data.
* **Maxima and minima calculation:** The input data can be categorized in two parts, and we need to repeatedly calculate two maxima, two minima, or one maximum and one minimum from the two sets, respectively.

**Real-world problems**

Many problems in the real world use the two heaps pattern. Let’s look at some examples.

* **Video platforms:** As part of a demographic study, we’re interested in the median age of the viewers. We want to implement a functionality whereby the median age can be updated efficiently whenever a new user signs up for video streaming.
* **Gaming matchmaking**: Matching players of similar skill levels is crucial for a balanced and enjoyable gaming experience. By maintaining two heaps (one for minimum skill level and one for maximum skill level), matchmaking algorithms can efficiently pair players based on their skill levels.

**K-way Merge**

The **K-way merge** pattern is an essential algorithmic strategy for merging K sorted data structures, such as arrays and linked lists, into a single sorted data structure. This technique is an expansion of the standard merge sort algorithm, which traditionally merges two sorted data structures into one.

**Note:** For simplicity, we’ll use the term lists to refer to arrays and linked lists in our discussion.

To understand the basics of this algorithm, first, we need to know the basic idea behind the K-way merge algorithm. The K-way merge algorithm works by repeatedly selecting the smallest (or largest, if we’re sorting in descending order) element from among the first elements of the K input lists and adding this element to a new output list (with the same data type as the inputs). This process is repeated until all elements from all input lists have been merged into the output list, maintaining the sorted order.

Now, let’s take a closer look at how the algorithm works. The K-way merge algorithm comprises two main approaches to achieve its goal, both leading to the same result.

Here, we’ll discuss one of the approaches, which uses a minheap:

**Using a min heap**

1. Insert the first element of each list into a min heap. This sets up our starting point, with the heap helping us efficiently track the smallest current element among the lists.
2. Remove the smallest element from the heap (which is always at the top) and add it to the output list. This ensures that our output list is being assembled in sorted order.
3. Keep track of which list each element in the heap came from. This is for knowing where to find the next element to add to the heap.
4. After removing the smallest element from the heap and adding it to the output list, replace it with the next element from the same list the removed element belonged to.
5. Repeat steps 2–4 until all elements from all input lists have been merged into the output list.

The slides below illustrate an example of using this approach with arrays:

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

Besides using a min heap, another effective approach for performing a K-way merge involves grouping and merging pairs of lists. This technique simplifies the merging process by reducing it to a series of two-way merges.

**Making groups of two and repeatedly merging them**

Here are the steps of this method:

1. Start by dividing the K sorted lists into pairs, making groups of two. This organizes our lists into manageable units for merging.
2. For each pair of lists, perform a standard two-way merge operation. This is similar to the merge step in merge sort, where two sorted lists are combined into a single sorted list. This step results in k/2*k*/2 merged lists.
3. If there are an odd number of lists in a group at any point, simply leave one list unmerged in that round. This ensures that no list is left out of the merging process.
4. Repeat the process of pairing up the resulting lists from the previous merge and merging them again until only one sorted list remains, which is the final result.

The slides below illustrate an example of using this approach with arrays:

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Does your problem match this pattern?**

Yes, if one or both of the following conditions are fulfilled:

* **Involves merging sorted arrays or a matrix:** The problem involves a collection of sorted arrays or a matrix with rows or columns sorted in a specific order that needs to be merged. This could be the core of the problem or a step toward the solution.
* **Seeking the kth smallest/largest across sorted collections:** The problem involves identifying the kth smallest or largest element across multiple sorted arrays or linked lists.

**Real-world problems**

Many problems in the real world use the K-way merge pattern. Let’s look at some examples.

* **Patient records aggregation:** In healthcare informatics, patient data often come from multiple sources, such as lab results, physician notes, and imaging reports, each sorted by date or priority. Integrating these data streams into a single, chronologically ordered patient record is essential for providing comprehensive care. The K-way merge pattern can efficiently combine these sorted data streams, ensuring doctors and nurses have timely access to a unified view of patient history, which is crucial for diagnosis and treatment planning.
* **Merging financial transaction streams:** Financial institutions process transactions from multiple sources, including trades, payments, and account transfers. Analysts need these transactions merged into a single stream to perform real-time market analysis or fraud detection. By applying the K-way merge, transactions from different sources can be integrated into a coherent order based on time or transaction ID, enabling more effective monitoring and analysis of financial activities.
* **Log file analysis:** Large-scale web services generate log files from multiple servers, each chronologically ordered. Analyzing these logs for insights into user behavior, system performance, or error diagnosis requires merging them into a single, time-ordered stream. The K-way merge pattern facilitates this by efficiently combining multiple sorted log files, enabling analysts to perform comprehensive log analysis without significant preprocessing.

**Top K Elements**

The **top k elements** pattern is an important technique in coding that helps us efficiently find a specific number of elements, known as k*k*, from a set of data. This is particularly useful when we’re tasked with identifying the largest, smallest, or most/least frequent elements within an unsorted collection.

To solve tasks like these, one might think to sort the entire collection first, which takes O(nlog⁡(n)) time, and then select the top k elements, taking additional O(k) time. However, the top k elements pattern bypasses the need for full sorting, reducing the time complexity to O(nlog⁡k) by managing which elements we compare and keep track of.

Which data structure can we use to solve such problems? A heap is the best data structure to keep track of the smallest or largest k*k* elements. With this pattern, we either use a max heap or a min heap to find the smallest or largest k*k* elements, respectively, because they allow us to efficiently maintain a collection of elements ordered in a way that gives us quick access to the smallest (min heap) or largest (max heap) element.

For example, let’s look at how this pattern operates to solve the problem of finding the top k largest elements (by using min heap) or top k*k* smallest elements (by using max heap):

1. Insert the first k elements from the given set of elements into a heap. If we’re looking for the largest elements, use a min heap to keep the smallest of the large elements at the top. Conversely, for the smallest elements, use a max heap to keep the largest of the small elements at the top.
2. Iterate through the remaining elements of the given set.
   1. For a min heap, if we find an element larger than the top, remove the top element (the smallest of the large elements) and insert the new, larger element. This ensures the heap always contains the largest elements seen so far.
   2. For a max heap, if we find an element smaller than the top, remove the top element (the largest of the small elements) and insert the new, smaller element, keeping the heap filled with the smallest elements seen so far.

The efficiency of this pattern comes from the ability of the heap to insert and remove elements in O(log⁡k) time. Because we only maintain k*k* elements in the heap, these operations are quick, and we can process all n elements in the given set in O(nlog⁡k) time.

It’s important to note that while accessing the top element of the heap can be done in O(1) time, retrieving all k elements, if necessary, involves removing them one by one. This process takes O(klog⁡k) time because each removal necessitates reorganizing the heap.

Let’s look at the following illustration to understand how to use min heap to find the top three largest elements

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

**Does your problem match this pattern?**

* Yes, if both of these conditions are fulfilled:
  + **Unsorted list analysis:** We need to extract a specific subset of elements based on their size (largest or smallest), frequency (most or least frequent), or other similar criteria from an unsorted list. This may be the requirement of the final solution, or it may be necessary as an intermediate step toward the final solution.
  + **Identifying a specific subset:** The goal is to identify a subset rather than just a single extreme value. When phrases like top k, kth largest/smallest, k most frequent, k closest, or k highest/lowest describe our task, it suggests the top k elements pattern is ideal for efficiently identifying a specific subset.
* No, if any of these conditions is fulfilled:
  + **Presorted input:** The input data is already sorted according to the criteria relevant to solving the problem.
  + **Single extreme value:** If only 1 extreme value (either the maximum or minimum) is required, that is, k=1, as that problem can be solved in O(n) with a simple linear scan through the input data.

**Real-world problems**

Many problems in the real world use the top K elements pattern. Let’s look at some examples.

* **Location-based services in ride-sharing apps like Uber:** For a user requesting a ride, the app needs to find the nearest available drivers to ensure a quick pickup. However, it’s not efficient or necessary to consider every driver in the city. Using the top k elements pattern, the app can efficiently determine the n closest drivers to the user’s location. This involves sorting drivers based on their distance from the user but in a way that prioritizes computational efficiency, especially when n is much smaller than the total number of drivers.
* **Performance analysis in financial markets:** We can analyze broker performance by identifying those with the highest transaction volumes or other metrics of success. Given a dataset with broker IDs and their transaction volumes, the top K elements pattern can help sift through the data to highlight the brokers leading the market based on the frequency of their transactions or other relevant performance indicators.
* **Social media trend analysis:** Identifying the most popular or trending topics by analyzing hashtags or keywords. The top K elements pattern can help filter out the top topics based on their frequency over a certain time frame.

**Modified Binary Search**

The **modified binary search** pattern is an extension of the traditional binary search algorithm and can be applied to a wide range of problems. Before we delve into the modified version, let’s first recap the classic binary search algorithm.

**Classic Binary Search**

Binary search is an efficient search algorithm for searching a target value in sorted arrays or sorted lists that support direct addressing (also known as [random access](https://www.educative.io/answers/difference-between-sequential-and-random-access-to-storage)). It follows a divide-and-conquer approach, significantly reducing the search space with each iteration. The algorithm uses three indexes—start, end, and middle—and proceeds as follows:

1. Set the start and end indexes to the first and last elements of the array, respectively.
2. Calculate the position of the middle index by taking the average of the start and end indexes. For example, if start=0*start*=0 and end=7*end*=7, then middle=⌊(0+7)/2⌋=3*middle*=⌊(0+7)/2⌋=3.
3. Compare the target value with the element at the middle index.
4. If the target value is equal to the middle index element, we have found the target, and the search terminates.
5. If the target value is less than the middle index element, update the end index to middle−1*middle*−1 and repeat from step 22 onwards. Because the array is sorted, all the values between the middle and the end indexes will also be greater than the target value. Therefore, there’s no reason to consider that half of the search space.
6. If the target value is greater than the middle index element, update the start index to middle+1*middle*+1 and repeat from step 22. Again, because the array is sorted, all the values between the start and the middle indexes will also be less than the target value. Therefore, there’s no reason to consider that half of the search space.
7. Continue the process until the target value is found or if the search space is exhausted, that is, if the start index has crossed the end index. This means that the algorithm has explored all possible values, which implies that the search space is now empty and the target value is not present.

**Note:** We’re assuming the array is sorted in ascending order. If it’s sorted in descending order, we’ll do the opposite when changing the positions of the start and end pointers. Also, to avoid integer overflow when calculating the middle index, especially in languages with limited integer ranges, we can use start + (end - start) / 2 instead of (start + end) / 2.

The following illustration shows how a binary search operation works using the techniques described above:

|  |
| --- |
|  |
|  |
|  |
|  |

Binary search reaches the target value in O(log(n)) time because we divide the array into two halves at each step and then focus on only one of these halves. If we had opted for the brute-force approach, we would have had to traverse the entire array without any partitioning to search for the target value, which would take O(n) in the worst case.

**Modified Binary Search**

The modified binary search pattern builds upon the basic binary search algorithm discussed above. It involves adapting the traditional binary search approach by applying certain conditions or transformations, allowing us to solve problems in which input data are modified in a certain way.

A few common variations of the modified binary search pattern are:

1. **Binary search on a modified array:** Sometimes, the array may be modified in a certain way, which affects the search process. For example, the array might be sorted and then rotated around some unknown pivot. Alternatively, some elements in a sorted array might be modified based on a specific condition. To handle such scenarios, we can modify the basic binary search technique to detect anomalies in the sorted order.
2. **Binary search with multiple requirements:** When searching for a target satisfying multiple requirements, a modified binary search can be used. It involves adapting the comparison logic within the binary search to accommodate multiple specifications. Examples include finding a target range rather than a single target or finding the leftmost or the rightmost occurrence of a target value.

**Does your problem match this pattern?**

* Yes, if either of these conditions is fulfilled:
  + **Target value in sorted data:** The problem involves locating a specific target value—or identifying its first or last occurrence—within a sorted array or list. This pattern applies to data structures that support direct addressing.
  + **Partially sorted segments:** We may use this pattern when segments of an input array are sorted.
* No, if either of these conditions is fulfilled:
  + **Lack of direct addressing:** The input data structure does not support direct addressing.
  + **Unsorted or inappropriately sorted data:** The data to search is not sorted according to criteria relevant to the search. For example, if we’re looking for a particular date in a list of dates, but the list is sorted in alphabetical order (and not chronologically), we cannot use binary search.
  + **Non-value-based solutions:** The problem does not require identifying a specific value or range of values. For example, if we have a list of student names only and we want to search for a student, modified binary search won’t be a correct choice.

**Real-world problems**

Many problems in the real world use the modified binary search pattern. Let’s look at some examples.

* **Dictionary searches:** A dictionary contains words that are alphabetically sorted. Therefore, we can use a classic binary search to find the required word quickly. If we wanted to find all the words in the dictionary that share a common prefix, we could modify the comparison operation used in the classic binary search algorithm.
* **Range-based filtering:** We can find range-based information by applying a modified binary search algorithm. For example, if we want to filter out YouTube videos on the basis of a certain time range, we can modify our classic binary search algorithm. Another example can also be filtering bank transactions by dates or finding test scores of students within a certain range.
* **String searching algorithms:** Modified binary search can be adapted for string searching algorithms such as search on the suffix array or search on the longest common prefix (LCP) array. These techniques are used in bioinformatics for DNA sequence analysis, text processing, and search engines.

**Subsets**

The **subsets** pattern is an important strategy to solve coding problems that involve exploring all possible combinations of elements from a given data structure. This pattern can be useful when dealing with sets containing unique elements or arrays/lists that may contain duplicate elements. It is used to generate all specific subsets based on the conditions that the problem provides us.

The common method used is to build the subsets incrementally, including or excluding each element of the original data structure, depending on the constraints of the problem. This process is continued for the remaining elements until all desired subsets have been generated.

The following illustration shows how subsets are made from a given array:

|  |
| --- |
|  |
|  |
|  |
|  |

**Note:** We sometimes also use a programming technique known as backtracking to generate the required subsets of a given data structure of elements. Backtracking applies to a broader range of problems where exhaustive search, that is, evaluating all possibilities, is required. These problems may involve various constraints, rules, or conditions that guide the exploration process. Not all of these problems involve finding subsets. That is why it is necessary to discuss Subsets as a separate programming pattern.

**Does your problem match this pattern?**

Yes, if the following condition is fulfilled:

* **Requirement for combinations or subsets:** The problem asks us to generate combinations (or subsets) of the elements from the input data structure. This could be the final solution itself or a step toward reaching the solution.

**Real-world problems**

Many problems in the real world use the subsets pattern. Let’s look at some examples.

* **Custom movie playlists:** Using the subsets pattern, we can generate all possible combinations of movies to meet any viewer’s preference. Imagine we have a list of movies and want to create custom playlists based on different criteria, such as genre or length. For example, generating all combinations of comedies and dramas for a weekend binge-watching session.
* **Test case generation:** In software testing, particularly in automated testing, generating comprehensive test cases is crucial. Using the subsets pattern, we can generate various combinations of input values to test different scenarios and edge cases thoroughly.
* **Feature selection in machine learning:** In machine learning, feature selection plays a vital role in building effective models. By considering subsets of features from a dataset, we can evaluate the performance of different combinations of features and select the subset that optimizes model performance and complexity.

**Greedy Techniques**

An **algorithm** is a series of steps used to solve a problem and reach a solution. In the world of problem-solving, there are various types of problem-solving algorithms designed for specific types of challenges. Among these, greedy algorithms are an approach for tackling optimization problems where we aim to find the best solution under given constraints.

Imagine being at a buffet, and we want to fill the plate with the most satisfying combination of dishes available, but there’s a catch: we can only make our choice one dish at a time, and once we move past a dish, we can’t go back to pick it up. In this scenario, a greedy approach would be to always choose the dish that looks most appealing to us at each step, hoping that we end up with the best possible meal.

**Greedy** is an algorithmic paradigm that builds up a solution piece by piece. It makes a series of choices, each time picking the option that seems best at the moment, the most greedy choice, with the goal of finding an overall optimal solution. They don’t worry about the future implications of these choices and focus only on maximizing immediate benefits. This means it chooses the next piece that offers the most obvious and immediate benefit. A greedy algorithm, as the name implies, always makes the choice that seems to be the best at the time. It makes a locally-optimal choice in the hope that it will lead to a globally optimal solution. In other words, greedy algorithms are used to solve optimization problems.

Greedy algorithms work by constructing a solution from the smallest possible constituent parts. However, it’s important to understand that these algorithms might not always lead us to the best solution for every problem. This is because, by always opting for the immediate benefit, we might miss out on better options available down the line. Imagine if, after picking the most appealing dishes, we realize we’ve filled our plate too soon and missed out on our favorite dish at the end of the buffet. That’s a bit like a greedy algorithm getting stuck in what’s called a local optimal solution without finding the global optimal solution or the best possible overall solution.

However, let’s keep in mind that for many problems, especially those with a specific structure, greedy algorithms work wonderfully. One classic example where greedy algorithms shine is in organizing networks, like connecting computers with the least amount of cable. Prim’s algorithm, for instance, is a greedy method that efficiently finds the minimum amount of cable needed to connect all computers in a network.

**Note:** To sum up, greedy algorithms offer a straightforward and often effective way to solve optimization problems by making the most advantageous choice at each step.

The illustration below shows a simple example of finding the path with the maximum sum of node values that demonstrates the working of a greedy algorithm and also shows how a greedy algorithm doesn’t guarantee an optimal solution.

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

**Does your problem match this pattern?**

* Yes, if both of these conditions are fulfilled:
  + **Optimization problem:** The problem is an optimization problem, where we are looking to find the best solution under a given set of constraints. This could involve minimizing or maximizing some value, such as cost, distance, time, or profit.
  + **Making local choices leads to a global solution:** The problem can be solved by making simple decisions based on the current option or state without needing to look ahead or consider many future possibilities.
* No, if any of these conditions is fulfilled:
  + **Local choices lead to sub-optimal solutions:** Our analysis shows that making local greedy choices leads us to a sub-optimal solution.
  + **Problem lacks clear local optima:** If the problem doesn’t naturally break down into a series of choices where we can identify the best option at each step, a greedy algorithm might not be applicable.

**Real-world problems**

Many problems in the real world use the greedy techniques pattern. Let’s look at some examples.

* **CPU scheduling algorithms:** In operating systems, managing how processes are assigned to the CPU for execution is critical for efficiency and performance. Greedy algorithms are used in CPU scheduling to decide the order of tasks based on specific criteria, such as minimizing waiting time, maximizing utilization, or ensuring fairness among users. By making greedy choices, such as selecting the process with the shortest expected completion time next (Shortest Job Next), systems aim to optimize overall throughput and resource utilization.
* **Network Design in LANs:** For large Local Area Networks (LANs) connecting numerous devices, efficient data transmission is vital. Greedy algorithms help in designing the network’s infrastructure to ensure data packets travel in the most efficient way possible. By finding a Minimum Spanning Tree (MST) for the network, a greedy algorithm like Prim’s or Kruskal’s ensures that all switches are connected with the least total cable length, reducing costs and improving network performance. This approach ensures minimal data transmission times and reduced chances of bottlenecks.
* **Friend recommendations on social networking websites:** Social networking platforms like Facebook, LinkedIn, and X (formerly Twitter) enhance user engagement by suggesting new friends or connections. A key algorithm behind this feature is Dijkstra’s algorithm, a greedy technique used to find the shortest paths between nodes in a graph. In this context, nodes represent users, and edges represent connections between them (like friendships). By finding the shortest path of connections between users, the algorithm can suggest people we may know or should connect with, based on your existing network of friends. This makes the platform more engaging by encouraging the growth of our social network through relevant connections.

**Backtracking**

Imagine we’re planning an exciting road trip through a city, aiming to visit all the places we want to see while covering the shortest distance possible. However, there are some conditions we must follow: we can’t revisit the same place more than once, and we must end up back where we started. This problem, known as the city road trip problem, requires finding the optimal route that satisfies these conditions. It’s a classic example where the concept of backtracking comes into play, allowing us to explore different paths until we find the shortest one that fulfills all the conditions.

Let’s first see how this problem can be solved using a brute-force approach. We can do this by exploring routes in every single way we can visit the places. We have to write down every possible route, check how long each one is, and then pick the shortest one. But as our list of places grows, it makes this approach computationally impractical for a large number of routes.

Now, let’s look at a backtracking approach to solve the same problem. With backtracking, we can start by picking a place and choose the next place to visit that’s close and follows our conditions. We move back (backtrack) to the previous place if the current place has been visited before or if we cannot move forward to any place from here. We check these conditions on each of our choices because we do not want to break any of our road trip rules. We keep doing this, choosing, checking conditions, and backtracking until we’ve visited all the places according to the requirements. At every step, we choose the closest place, ensuring we have chosen the shortest path to visit all the places we want to see.

**Backtracking** is an algorithmic technique for solving problems by incrementally constructing choices to the solutions. We abandon choices as soon as it is determined that the choice cannot lead to a feasible solution. On the other side, brute-force approaches attempt to evaluate all possible solutions to select the required one. Backtracking avoids the computational cost of generating and testing all possible solutions. This makes backtracking a more efficient approach. Backtracking also offers efficiency improvements over brute-force methods by applying constraints at each step to prune non-viable paths.

As seen in the above example, backtracking works by exploring all potential routes toward a solution step-by-step. It can be visualized as traversing a state space tree, where each node represents a partial solution. Starting from the root (an empty solution), backtracking moves deeper into the tree, exploring branches (choices) until it finds a feasible solution or reaches a leaf node that cannot be extended into a complete solution. Upon reaching a dead end, the algorithm backtracks to the previous state and explores a different branch. This process is repeated, with constraints applied at each step to avoid exploring paths that cannot lead to a successful, feasible solution.

Let’s look at the visualization of the space state tree below to better understand the working:

A diagram of a diagram

Description automatically generated

In the visualization above, we start with an initial point, S. From this point, we proceed to explore a potential solution, S1, via an intermediate choice, C1. After evaluating, we determine that S1 does not satisfactorily solve our problem. Therefore, we backtrack to S and then shift our exploration towards another potential choice, C2. This process of exploration and backtracking continues until we identify a successful feasible solution.

In the scenario above, both S1 and C2 fail to provide feasible solutions and only S3 emerges as a successful solution to the problem. This illustrates that the backtracking approach examines all potential combinations until it discovers a successful feasible solution.

The backtracking algorithm can be implemented using recursion. We use recursive calls where each call attempts to move closer towards a feasible solution. This can be outlined as follows after starting from the initial point as the current point:

* **Step 1:** If the current point represents a feasible solution, declare success and terminate the search.
* **Step 2:** If all paths from the current point have been explored (i.e., the current point is a dead-end) without finding a feasible solution, backtrack to the previous point.
* **Step 3:** If the current point is not a dead-end, keep progressing towards the solution, and reiterate all the steps until a solution is found or all possibilities are exhausted.

If all the points have been explore without finding a feasible solution, declare failure; no solution exists.

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Examples**

The following examples illustrate some problems that can be solved with this approach:

**1. Path in binary matrix:** Find a path of 1s from top-left to bottom-right in an n×n binary maze. We are only allowed to move to the right or downward.

A screenshot of a game

Description automatically generated

**2. Check knight tour configuration:** Check if a knight can cover all possible squares once in an n×n chessboard. The initial position of the knight is at the top-left square of the board.

A screenshot of a puzzle

Description automatically generated

**Does your problem match this pattern?**

* Yes, if any of these conditions is fulfilled:
  + **Complete exploration is needed for any feasible solution:** The problem requires considering every possible choice to find any feasible solution.
  + **Selecting the best feasible solution:** When the goal is not just to find any feasible solution but to find the best one among all feasible solutions.
* No, if the following condition is fulfilled:
  + **Solution invalidity disqualifies other choices:** In problems where failing to meet specific conditions instantly rules out all other options, backtracking might not add value.

**Real-world problems**

Many problems in the real world share the backtracking pattern. Let’s look at some examples.

* **Syntax analysis:** In compilers, we use recursive descent parsing. It is a form of backtracking, to analyze the syntax of the program. This analysis involves matching the sequence of tokens (basic symbols of programming language) against the grammar rules of the language. When a mismatch occurs during the analysis, the parser backtracks to a previous point to try a different rule of the grammar. This ensures that even complex nested structures can be accurately understood and compiled.
* **Game AI (Artificial Intelligence):** In games like chess or Go, AI algorithms use backtracking to try out different moves and see what happens. If a move doesn’t work out well, the AI goes back and tries something else. This helps the AI learn strategies that might be better than those used by humans because it can think about lots of different moves and figure out which ones are likely to work best.
* **Pathfinding algorithms:** In pathfinding problems like finding the way through a maze or routing in a network, backtracking is used. It tries out different paths to reach the destination. If it hits a dead end or a spot it can’t pass through, it goes back and tries another path. This keeps happening until it finds a path that works and leads to the destination.

**Dynamic Programming**

Many computational problems are solved by recursively applying a divide-and-conquer approach. In some of these problems, we see an **optimal substructure**, i.e., the solution to a smaller problem helps us solve the bigger one.

Let’s consider the following problem as an example: Is the string “rotator” a palindrome? We can start by observing that the first and the last characters match, so the string *might* be a palindrome. We shave off these two characters and try to answer the same question for the smaller string “otato”. The subproblems we encounter are: “rotator”, “otato”, “tat”, and “a”. For any subproblem, if the answer is *no*, we know that the overall string is not a palindrome. If the answer is *yes* for all of the subproblems, then we know that the overall string is indeed a palindrome.

While each subproblem in this recursive solution is distinct, there are many problems whose recursive solution involves solving some subproblems over and over again (overlappingsubproblems). An example is the recursive computation of the *nth* Fibonacci number. Let’s review the definition of the Fibonacci series:

fib(0)=0

A math equations with numbers

Description automatically generated with medium confidence

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

We can see above that the subproblem fib(2)*fib*(2) is evaluated twice, fib(1)*fib*(1) is evaluated thrice, and fib(0)*fib*(0) is evaluated twice. These are repeated or overlapping subproblems. As every subproblem is reevaluated every time it appears, the naive recursive implementation of this solution will have exponential time complexity.

An optimization of such a recursive solution would be to store and reuse solutions to subproblems, reducing the time complexity to polynomial time. Such an approach is called dynamic programming (DP).

We’ve discussed that we need to save the computations, but how can we save and use them? We use the following two approaches that primarily save our computations by reusing previous calculations of subproblems:

* **Top-down approach:** It is a recursive approach that stores the results of redudant function calls to avoid repeating calculations for the same subproblems.
* **Bottom-up approach:** It is an iterative strategy that systematically fills a table with results of subproblems to solve larger problems efficiently.

**Top-down approach**

The top-down approach is also known as memoization. It is usually implemented as an enhancement of the naive recursive solution. It uses recursion to break down larger subproblems into smaller ones. The smallest one is solved and the result is stored in a lookup table for use in computing larger subproblems.

To take advantage of the stored results of subproblems, in every call, the top-down recursive function *first* checks if a solution to a subproblem already exists. If it does, the result is fetched from the lookup table *instead* of making a recursive call to compute it. Otherwise, the recursive call is made. As we can see in the illustration below, using the top-down approach, we are able to avoid recomputing the subproblems fib(2), fib(1), and fib(0).

Compared to the naive recursive solution, the top-down approach takes up additional space in memory because it stores intermediate results in a lookup table.

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**Bottom-up approach**

The bottom-up approach is also known as tabulation. In this approach, the smallest problem is solved first, the results saved, and then larger subproblems are computed based on the evaluated results. In contrast to the top-down approach, which uses recursion to first break down a larger problem into smaller subproblems, the bottom-up approach starts by solving the smallest subproblem, and then iterates progressively through larger subproblems to reach the overall solution.

We start by initializing a lookup table and setting up the values of the base cases. For every subsequent, larger subproblem, we simply fetch the results of the required preceding smaller subproblems and use them to get the solution to the current subproblem.

For example, to compute the Fibonacci series, we first set up the two base cases, fib(0)*fib*(0) and fib(1)*fib*(1), and then proceed to calculate the larger subproblems:

fib(2)=fib(1)+fib(0)=1+0=]

and

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

and so on.

The slides below present a walkthrough of the bottom-up DP solution to the Fibonacci numbers problem:

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |

**Does your problem match this pattern?**

Yes, if the problem exhibits both of these characteristics:

* **Overlapping subproblems:** We encounter overlapping subproblems, that is, we can use the results of one subproblem when solving another, possibly larger subproblem.
* **Optimal substructure:** In problems where the final solution can be constructed from the optimal solutions to its subproblems.

**Real-world problems**

Many problems in the real world use the dynamic programming pattern. Let’s look at some examples.

* **Optimal route planning in GPS navigation systems:** In GPS navigation systems, dynamic programming plays a key role in determining the best route from one location to another. By analyzing various factors such as distance, traffic conditions, and road constraints, dynamic programming evaluates different route options. It iteratively explores all possible paths to determine the optimal route that minimizes travel time and avoids congested or inefficient routes.
* **Text justification:** For text justification, dynamic programming is employed to determine the optimal arrangement of words within lines, ensuring that the text fits within a specified width while minimizing whitespace and enhancing readability. By iteratively considering various line break points and evaluating the associated costs, dynamic programming finds the best way to arrange the text, making it look nice and easy to read.

**Cyclic Sort**

Imagine we have a classroom with numbered seats, and each student is given a card with a number corresponding to their seat number. To maintain classroom decorum and order, students are required to sit in their assigned seats. However, students have randomly taken seats, so the seating arrangement is all jumbled up. If we have to fix this, we start with the first seat, seat 1, and check if the student sitting here has the right card, i.e., the number 1. If not, we move the student to the seat corresponding to their card number. We repeat this process for each seat until every student is in their proper seat according to the number on their card. After completing this process for all seats, the students will be sitting in ascending order according to their seat numbers.

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

The repeated swapping of students until they find their correct positions is nothing but the cyclic sort. It can be seen as the cycling of elements through the array until they settle into their proper places.

**Cyclic sort** is a simple and efficient in-place sorting algorithm used for sorting arrays with integers in a specific range, most of the time [1 – n] It places each element in its correct position by iteratively swapping it with the element at the position where it should be located. This process continues until all elements are in their proper places, resulting in a sorted array.

But how do we know the correct position of any element? This is where the algorithm makes things even easier: the correct place of any element is simply the value of element - 1. For example, if we have the array [3,1,2][3,1,2], then the correct position of the first element, 3, is (3−1)th index, i.e., index 2 and not 0. Similarly, the correct position for the elements 1 and 2 is index 0 and 1, respectively.

The slides below help visualize how cyclic sort works:

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

Is there a way to determine whether to use cyclic sort on a given unsorted array of integers? The answer is: Yes. We can identify cycles within the array, which are nothing but the subsequences of numbers that are not in sorted order. A cycle occurs when there’s an element that is out of place, and swapping it with the element at its correct position moves another element out of place, forming a cycle.

**Note:** The unsorted arrays with numbers ranging from [1−n], where *n* is the length of the array, are guaranteed to have one or more cycles in them.

The slides below illustrate a traversal technique that is utilized for identifying any cycles present within the array.

**Note:** In practice, the process of identifying cycles and performing cyclic sort is seamlessly integrated and considered as one cohesive operation. The separate illustrations provided are merely for explanatory purposes, simplifying the understanding of the overall process.

Unlike algorithms with nested loops or recursive calls, cyclic sort achieves a linear time complexity of O(n)due to its streamlined approach of iteratively placing each element in its correct position within a single pass through the array. This makes it fast and reliable, especially for small arrays. Moreover, cyclic sort doesn’t require any extra space, as it sorts the elements in place.

Cyclic sort, while efficient in specific scenarios, does come with its limitations. Let’s go over a few:

* **Limited range:**Cyclic sort’s efficiency is contingent upon the range of elements being limited and known beforehand. When dealing with arrays with an unknown or expansive range, the cyclic sort may not perform optimally. For example, if we have an array of prices of goods ranging from 11 to 100100, cyclic sort would efficiently organize them. However, if the price range extended beyond 100100 or was unknown, cyclic sort might not provide optimal results.
* **Not stable:** Cyclic sort lacks stability as a sorting algorithm. This means it may alter the relative order of equal elements during sorting, which could be undesirable in scenarios where maintaining the original order is important. For example, in a task queue where tasks with equal priority should be processed in the order they were received, cyclic sort might reorder tasks with the same priority, disrupting the original sequence.
* **Not suitable for noninteger arrays:** Cyclic sort is optimized for sorting arrays of integers, attempting to use it on noninteger arrays may not produce the desired outcome. Suppose we want to sort an array of names alphabetically, then using cyclic sort is not a good choice.
* **Not suitable when multiple attributes play a significant role:** Cyclic sort is primarily designed for arrays of integers only, so it may not handle cases where the input has multiple attributes associated with it. For example, given an array containing objects representing employees, where each object has attributes such as name, age, and salary, if we need to sort the objects with respect to all three attributes then using cyclic sort on just salary may not produce the desired outcome. This is because the other two attributes play an equal role in deciding the final order, so we can’t just take one attribute while sorting the array.

Normally, it is quite easy to know when to terminate the algorithm for cyclic sort. If we have the input array of length *n* containing all the numbers from 1 to n, then the cyclic sort would end once every element of the array has been traversed from index 00 to n−. However, it gets tricky to determine the termination when the input array of length n doesn’t contain every number from 1 to n. In any such scenario, we’ll end up either with an infinite loop or incorrect placing of elements in their respective positions. In both cases, it’s essential to incorporate appropriate error-handling mechanisms or termination conditions within the cyclic sort algorithm to prevent infinite loops and ensure correct sorting outcomes. This may involve additional checks for duplicates or missing values, as well as adjustments to the sorting logic to accommodate such scenarios.

**Note:** The examples in the section below represent two such scenarios where termination of cyclic sort requires an additional check.

**Does your problem match this pattern?**

* Yes, if either of these conditions are fulfilled:
  + **Limited range integer arrays:** The problem involves an input array of integers in a small range, usually [1−n].
  + **Finding missing or duplicate elements:** The problem requires us to identify missing or duplicate elements in an array.
* No, if any of these conditions is fulfilled:
  + **Noninteger values:**Theinput array contains noninteger values.
  + **Nonarray format:**The input data is not originally in an array, nor can it be mapped to an array.
  + **Stability requirement:** The problem requires stable sorting.

**Real-world problems**

Many problems in the real world share the cyclic sort pattern. Let’s look at some examples.

* **Computational biology:** The species on a planet have n distinct genes numbered 1…n. Find the kth missing​​ gene in a given DNA sequence.
* **Playing card sorting:**If we have a deck of playing cards represented as integers in the range [1−52], and they are randomly shuffled, we can use cyclic sort to rearrange the cards into sorted order efficiently.
* **Data validation:** Cyclic sort can be used for data validation tasks, especially when dealing with datasets that are expected to have distinct values within a certain range. By applying cyclic sort to the dataset, we can quickly identify any missing or duplicate values, which can help in validating the integrity and accuracy of the data.
* **Package delivery routing:** In logistics or package delivery systems, drivers may have a list of addresses to visit. We can map these addresses to integers in a defined range. Then, we can use cyclic sort to optimize the route by rearranging the addresses based on these mapped integers, thus minimizing travel time and fuel consumption.

**Topological Sort**

Topological sorting is a technique used to organize a collection of items or tasks based on their dependencies. Imagine there is a list of tasks to complete, but some tasks can only be done after others are finished. There are many such tasks that depend on each other, or there’s a specific sequence of actions that must be followed. For example, when baking a cake, there are several steps one needs to follow, and some steps depend on others. We can’t frost the cake until it’s baked, and we can’t bake it until we’ve mixed the batter. To ensure that we don’t frost the cake before baking it or mix the batter after preheating the oven, we need to sort the steps based on their dependencies. This is where topological sorting helps us. It figures out the correct sequence of steps to bake the cake efficiently.

The **topological sort** pattern is used to find valid orderings of elements that have dependencies on or priority over each other. These elements can be represented as the nodes of a graph, so in technical terms, topological sort is a way of ordering the nodes of a directed graph such that for every directed edge [a,b] from node a*a* to node b*b*, a*a* comes before b*b* in the ordering.

**Note:** Topological sort is only applicable to directed acyclic graphs (DAGs), meaning there should be no cycles present in the graph.

If we write a recipe for baking a cake, then the list of tasks goes like first mix the batter, then bake the cake, and finally, frost it. These tasks can also be organized in a graph, where each task is a node, and the dependencies between tasks are represented by directed edges.

A diagram of a cake

Description automatically generated

However, if we mistakenly add an additional step in our recipe that contradicts any of the existing steps, it introduces a cycle in our graph. For example, if the recipe goes like mix the batter, frost the cake, bake the cake, and frost the cake, we can’t frost a cake that hasn’t been baked and can’t bake a cake that’s already frosted. Similarly, in a graph with cycles, we can’t establish a clear order of tasks, making topological sorting impossible. Therefore, topological sorting is only applicable to directed acyclic graphs (DAGs), where tasks are organized in a logical sequence without any contradictions or cycles.

A screenshot of a computer

Description automatically generated

Topological sorting is crucial for converting a partial ordering to a complete ordering, especially when certain tasks have defined dependencies while others do not. For instance, consider a research project involving five tasks, labeled as A, B, C, D, and E. Some tasks depend on others: Task A must be completed before Task B and Task C, Task B must be finished before the Task D, and Task C must be done before Task E. However, when it comes to performing Tasks B and C, it’s unclear which one should be done first. This is where topological sorting comes in. It helps us convert this partial ordering of tasks into a complete ordering, providing clarity on the sequence of tasks to ensure efficient execution. There can be multiple possible valid ordering of the elements. For example, based on the given dependencies in our example, some valid orderings of the tasks could be:

A group of letters on a white background

Description automatically generated

Now, let’s have a visual demonstration of the example we discussed above:

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

Because topological sort can be applied to DAGs, it’s important to convert a non-DAG input to a DAG before solving the problem.

**Does your problem match this pattern?**

* Yes, if either of these conditions is fulfilled:
  + **Dependency relationships:** The problem involves tasks, jobs, courses, or elements with dependencies between them. These dependencies create a partial order, and topological sorting can be used to establish a total order based on these dependencies.
  + **Ordering or sequencing:** The problem requires determining a valid order or sequence to perform tasks, jobs, or activities, considering their dependencies or prerequisites.
* No, if either of these conditions is fulfilled:
  + **Presence of cycles:** If the problem involves a graph with cycles, topological sorting cannot be applied because there is no valid linear ordering of vertices that respects the cyclic dependencies.
  + **Dynamic dependencies:** If the dependencies between elements change dynamically during the execution of the algorithm, topological sorting may not be suitable. Topological sorting assumes static dependencies that are known beforehand.

**Real-world problems**

Many problems in the real world share the topological sort pattern. Let’s look at some examples.

* **Course scheduling**: In academic institutions, students need to enroll in courses based on prerequisites. Topological sorting helps determine the order in which courses should be taken to ensure that students meet the prerequisites for each course.
* **Recipe planning in cooking**: Recipes often have steps that must be performed in a specific order. For example, a cake can’t be baked until we’ve mixed the batter. Topological sorting can help plan the steps of a recipe to ensure that they are performed in the correct sequence.
* **Process scheduling in computer systems:** During system boot-up, the operating system needs to initiate various processes, some of which depend on others. These dependencies are represented as ordered pairs. Circular dependencies are not allowed. The operating system selects an order for executing processes, ensuring that each process’s dependencies are met before execution.