What is an Algorithm?

An algorithm is simply a set of procedures that we do to achieve a desired task. For example, the steps listed out in a recipe book is an algorithm to cook food. The steps which a school going child follows to go to school, like waking up, brushing teeth, getting ready, having breakfast is also an algorithm. Someone who investigates crime has a procedure to discern what happened in an incident and who could be at fault. Most people have the perception of algorithms as some complex code which can be understood by experts only. However, this is certainly not the case. As we move forward, we will learn to design algorithms on our own and realise how simple yet powerful they are. We apply algorithms in our day to day life without even realising it. Some of the above examples are a clear evidence. Almost every real world problem is solved using an algorithm. Be it the problem of finding a shortest path in a city or giving most accurate results on a search engine, algorithms are everywhere behind the scenes. Everytime we book a cab, make a google search, find friends on social networking sites, play a game like tic-tac-toe against the computer, there is an inherent and elegant algorithm associated with it which helps us to seamlessly achieve our desired task.

In today's digital world, algorithms are the back-bone of almost every industry. It helps us to save resources like time and money and optimise the problem.

Algorithms have some essential characteristics like what does it take as its input, what does it outputs, how much time does it take, does it give the correct output, etc. The most important factor in designing an algorithm is its correctness and the time it requires to produce the correct output. A simple searching method is also an algorithm. If we are given a list of numbers and we have to find whether a number belongs to the list, we simply search through the list one by one and check whether the number is present. This algorithm is popularly called linear search. There are ways in which we can optimise on this algorithm such that we do not have to check each and every number in the list. Computer Scientists and programmers all over the world are constantly working day and night to optimise the algorithms so that we can solve complex problems in as little time as possible.

Characteristics of an Algorithm:

Any algorithm should have the following properties:

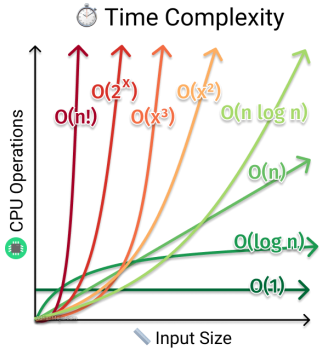
i) **Input & Output specified:** The algorithm should clearly specify what is its input and output. For e.g. a sorting algorithm which sorts a list of numbers takes its input as the list of numbers and outputs the same numbers in sorted order.

ii) **Time taken** by an algorithm should be finite. The most important factor in determining the effectiveness and efficiency of an algorithm is the time it requires to produce the output. The time complexity of algorithms is most commonly expressed using the big O notation.

iii) **Memory required** by the algorithm is also one of the essential features of an algorithm. The algorithm should use a finite amount of memory so that it is practically feasible.

The time required by an algorithm to produce the output is also known as its time complexity. It is denoted in various ways like **big O notation**, **big theta notation** etc. The most widely used is the big O notation. The big O notation tells us that the algorithm takes no longer to complete than the expression inside the bracket. For example, if an algorithm runs in O (n) time on an input size of n , then it means that the algorithm will take atmost n steps to terminate. Consider a list of numbers and we have to find a particular number in that list. The natural searching algorithm would be start from the first number in the list, check if it is the desired number, if it is then we found the number and if not then repeat the process for all the numbers left. One important point to notice is that we could get the number in the first attempt, or second attempt or the nth attempt. This means the algorithm is guaranteed to produce the output after n steps. This can be also phrased as the algorithm uses atmost n steps to produce the output. Therefore, we can say that the algorithm runs in O(n) time.

If the input numbers are in sorted order, then we can apply a much effective form of searching known as the binary search. It runs in O(log n) time if the length of the list is n. This means that it will take atmost log n steps to produce the output. Considering an input size of 2^{64} which is approximately 10^{18} numbers, the algorithm can output in atmost 64 steps which is a huge improvement over the linear search which will use atmost 10^{18} steps. We are talking about the worst case time required of both the algorithms. It may happen for some input, that the linear search gets the number in the first attempt (i.e. the first number is the required number) while the binary search uses 4-5 steps. However, if we take the average case time required then the binary search method is much more efficient than linear search given the input numbers are sorted.



The figure in the right shows the number of CPU operations required for algorithms running in various worst case complexities. When it requires O(1) time, then the number of CPU operations is constant and does not depend on the input size. However, when the worst case time required is n!, the number of cpu operations explodes exponentially with the input size n. Therefore, we always aim to design polynomial time running algorithm to get the output in a reasonable amount of time.

Greedy Algorithms

Consider a situation where you have many meetings on the same day due to some earlier commitments and you cannot postpone them further. But since you are a human being and not a superhuman , you can attend only one meeting at a time. Since it is clear to you that you can't attend all the meets during the day, you decide to attend as many meets as you can.

This is a situation which many of us can face in real life and have to deal with it. Therefore, We introduce a class of algorithms called greedy algorithms which will come to rescue and minimize the chaotic schedule consequences.

As the name suggests, Greedy algorithm involves being greedy i.e. it involves making the best available choice at any given moment. It is one of the simplest and most intuitive algorithm in the world of algorithms. It is quite simple to think of because it does not worry about future, just picks whatever it finds the most optimal at that point.

Therefore, it obtains step by step local optimization which ultimately leads to the optimal solution to whole problem. However, as we shall see further, the greedy strategy does not always produce optimal solutions but is widely used because of its speed and because of the fact that it can give solutions to reasonable approximations where it does not produce optimal solution.

Let us understand the essence of greedy algorithms through an example. Suppose you have Rs.1,5,10 and 20 denominations. You want to return the change to customer with minimum number of coins possible (as it will be easy for you to count less coins). And during one instance, the customer expects Rs.46 return from you. To return him Rs. 46 in minimum number of coins, the first choice that naturally comes to your mind is Rs.20 coin! This is because the 20 rs coin will enable you to decrease the number of coins you have to pay as it is the best available choice at that time. Now, Rs. 23 is left to be paid. Then you again select a 20rs coin so that you can pay the maximum amount in one coin. Now only 3 rs is left which can be paid using 3 coins of 1 rs each. This shows how you chose the best option available at each step until the full amount was paid. The total number of coins (minimum) required is 5.

However, one point to be noted as stated earlier, greedy does not always produce optimal solution. If we had denominations of rs. 27, rs. 10, rs.5, rs. 1 in a hypothetical situation , our greedy approach would require 8 coins(27+10+5+1+1+1+1) whereas optimal answer would be 6 coins (10+10+10+10+5+1).

As we observed in the previous example, greedy algorithms are very simple and intuitive. If applicable, they tend to give results very fast and are therefore used extensively in a wide variety of optimization problems. Regardless of the nature of problem, all greedy algorithms follow a common structure and are therefore easy to implement.

There is a greedy choice at each step that satisfies the constraints of the original problem. We must be able to make a choice that is locally optimum at that point. This means that at every step, we choose the option that is best for the problem at that instant. After choosing the required option, we reduce the problem into a smaller subproblem and apply the greedy choice again and continue this till the problem is solved.

If a problem has following two properties, then it can be solved by the greedy method:

i) **Greedy Choice property:** The global optimal solution can be arrived at by choosing locally optimum solution at each step. In other words, an optimal solution can be obtained by making greedy choices.

ii) **Optimal Substructure property:** When the optimal solution of a problem contains the optimal solution of the sub problem, it is called the optimal sub structure property. This simply means that any optimal solution to the problem is also the optimal solution to its subproblem.

The most difficult part in designing a greedy algorithm is proving its correctness. Its fairly simple to come up with a greedy algorithm but proving it's correctness seems to be an equally difficult task. It involves lot of creativity and thinking skills. . In order to prove the correctness of a greedy algorithm, we must show that it is never beneficial to take less than the maximum benefit at any step of the process. This is most often done by focusing on two specific steps, and showing that if the benefit of one decreases and the benefit of the other increases, the overall benefit is decreased, and so taking the maximum benefit at the first step is most beneficial.

One good question that you can ask yourself is, "Can we get a globally optimal solution by not choosing locally optimum option at any stage?", or "In what ways can the locally optimal choice be bad for the global solution?". These questions might help one come up with a good counterexample to the greedy algorithm.

General Structure of a Greedy Algorithm:

```
While (the problem is not completed):
    Select the best available choice according to the constraints of problem.
    Add it to solution list
Return solution
```

This is simply what we were discussing till now. At each step select the best possible option (greedy step) and add it to solution list. At last when the problem is completely solved, return the solution set.

There are some characteristics of greedy algorithms which are worth mentioning:

1. The greedy algorithm divides the original problem into smaller problems and solves them recursively. For example, in the change problem that we discussed previously, we selected Rs.20 coin first, and then our new problem became returning Rs.26 to the customer with minimum number of coins. Here, after making the greedy choice, we expressed the original problem as a sub-problem of the same type.

2. At each step of the algorithm, the choice it makes does not depend on any of the choices it previously made. It does not "remember" the choices it made and hence does not require much space as compared to algorithms like dynamic programming which we will see later.

3. The first choice that it makes must be present in any of the global optimum solution(if there exists more than 1) so that it can proceed further to express the problem as a sub-problem of the same instance. However, this does not guarantee that the solution it provides will be optimal.

The term greedy algorithm was first coined by Dutch Computer Scientist Edsger W. Dijsktra when he wanted to calculate the minimum spanning tree of a graph. The greedy algorithms first started coming into the picture in the 1950s. The then scientists, Prim and Kruskal also achieved the optimization techniques for minimizing the costs of graphs during that decade.

A few years later, in the 1970s, many American researchers proposed a recursive strategy for solving greedy problems. In 2005, the NIST records registered the greedy paradigm as a separate optimization strategy.

Greedy algorithms are applicable to a wide range of problems and we can get a fair estimate of how powerful yet simple they are:

1. Activity Selection: This is the problem with which we started with. In this problem, given a set of activities with start time and finish time, find the maximum number of activities a person can perform assuming he can perform only 1 activity at a time.
2. Shortest path from a vertex in a graph: This is a very commonly encountered problem in real life where one has to go from one place to another using the shortest possible path to save on time. This was solved by Dijkstra in 1956 and he came up with the algorithm in just 20 minutes!
3. Graph Colouring Problem: We have to find the minimum possible number of colours that can be used to colour the nodes of a graph such that no two nodes have the same colour.
4. Huffman Encoding is a technique to compress large amount of texts. It is also widely used to compress formats like GZIP, BZIP2, and image formats like JPEG, PNG etc. This is also based on greedy method.

The greedy method also solves many other problems like minimum spanning tree in a graph, scheduling problems, fractional knapsack etc. We shall see some of these problems in detail and how to apply the greedy method to solve them.

Activity Selection Problem:

Let us start with the activity selection problem with which we started and realise that how greedy elegantly solves this seemingly tough problem.

So the problem is as follows: Consider the following set of activities with their respective start and finish times. Find the maximum number of activities a person can perform fully assuming he can perform one activity at a time.

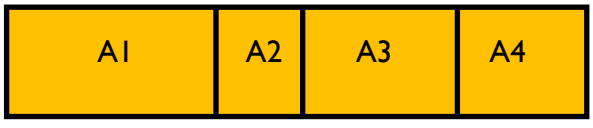
Activity	A	B	C	D	E	F	G	H	I	J
Start Time	1	3	2	4	8	7	9	11	9	12
End Time	3	4	5	7	9	10	11	12	13	14

The constraint in this problem is we have to select activities is such a way that they do not overlap otherwise a person would have to be present at both place simultaneously which is not possible. Since the timing of the activities can collapse, so it might not be possible to complete all the activities and thus we need to schedule the activities in such a way that the maximum number of activities can be finished.

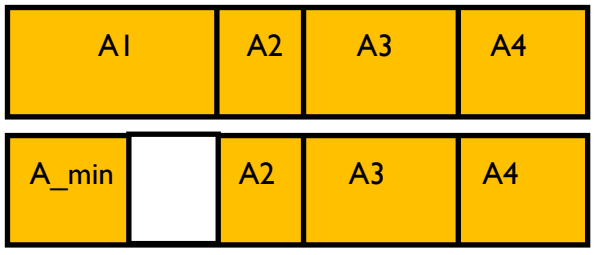
The greedy approach to this problem is fairly simple. Since we want to adjust maximum number of activities, choosing the activity which is going to finish first will leave us maximum time to adjust remaining activities. This is an intuition that greedily choosing the activity with earliest finish time will give us an optimal solution.

Let us prove that this intuition is correct.

If possible, let the activity with the minimum finish time be not included in the optimal solution. Then in the optimal solution, we can always replace the activity with the minimum finish time with the activity which has minimum time in the whole set because this will not lead to an overlap and the number of activities also remains same. Therefore, we are guaranteed to have the minimum finish time activity in one of the optimal answers.



Optimised solution which does not contain the activity with minimum finish time



Optimised solution which contains the activity with minimum finish time

We observe that when we replace the activity with minimum finish time in the solution set with the activity with minimum finish time globally, the solution remains valid and is optimal. Therefore, the greedy choice property holds for this problem.

Now we show that the optimal substructure property also holds. If A is an optimal solution containing the greedy choice (we just showed that atleast one optimal solution contains the greedy choice), then B = A \ {A_min}, where A_min is the activity with least finish time globally is an optimal solution to the problem S' where all activities have start time >= the finish time of A_min. This holds true because, if it were not the case then, we can pick a solution B' to S' with more activities than A' containing the greedy choice for the new problem. Then we can add A_min to B to obtain a solution for the original problem which has more activities than A, which means A is not an optimal solution, a contradiction.

Therefore, the activity selection problem has both the properties required for the greedy algorithm to be applicable. Hence, we can apply the greedy algorithm and obtain the optimal solution easily and quickly. As mentioned earlier, the most challenging part in designing a greedy algorithm for a problem is proving its correctness. This is done by proving that the problem has both the properties (i.e. greedy choice property and optimal substructure property).

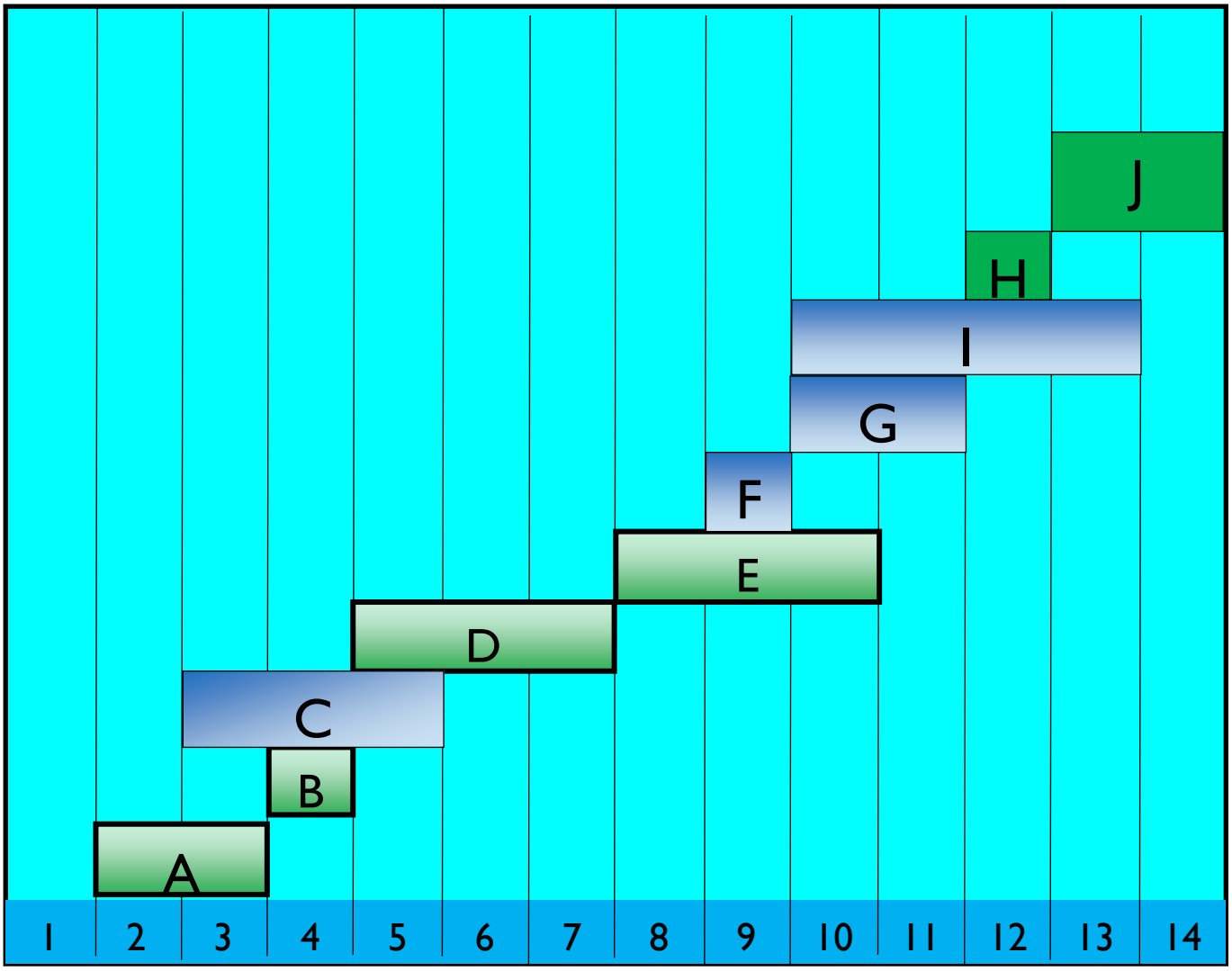


Illustration of Activity selection problem. The boxes marked in green are selected by the greedy algorithm and constitute an optimal solution

Let us apply the greedy method. We sort the activities in ascending order of their finish time as this will enable us to choose the next activity easily (In the given problem, the activities are already sorted according to their finish times) .We select activity A which has the minimum finish time. Now our problem is reduced to selecting maximum activities from B to J. Now, we select activity B which has the least finish time and does not overlap with A. Again our problem is reduced to selecting maximum activities from C to J such that it does not overlap with B. This is how the greedy algorithm solves the problem recursively by selecting the best choice at every step and reducing the problem into a sub problem of the same instance. Continuing in a similar fashion, we get the following set of activities:

A, B , D, E, H, J

Therefore, we can perform at max 6 activities from the given list of activities. The time which the greedy method requires is linear in number of activities. Using the asymptotic big O notation, it requires time of O(n) (Assuming the activities are already sorted according to their finish times). If we have to sort them, then the time required is O(n log n), as sorting takes more time than the selection. This is a very fast solution to a not-so-trivial problem. Most modern PCs can execute about 2-3 billion instructions per second. Therefore, given a list of activities whose number is in the order of hundreds of millions, it can produce the optimal set in 1 second !

Sometimes, there is a priority or importance associated with each activity. For example, a meeting may be more important than the other. In such cases, we want to maximise the total weight or priority of all the activities selected instead of the number of activities. However, greedy method fails in such problems because there is no guarantee that the activity with the highest priority will be there in the optimal solution.

Consider the following set of activities:

Activity	A	B	C
Start Time	1	2	11
Finish Time	10	5	20
Priority	100	50	200

Using the greedy method as in the previous example, we would first select Activity B then C which would give a total priority of 250. However, we observe that selecting activity A and then C will give total priority of 300 which is better than the previous selection made by the greedy algorithm. Therefore, greedy algorithm fails here. This is due to the fact that there is no guarantee that the activity with least finish time will be there in the optimal solution. The greedy choice property does not hold and hence greedy algorithm does not give the optimal result. This problem can be solved using another technique called dynamic programming which we shall see later.

Shortest Path Problem:

Another common situation we may face in our everyday life is travelling from one place to another using the shortest path possible. This will save us both time and fuel costs. This problem was first solved by Dijkstra in 1956. He used the greedy approach to solve this problem. He designed this algorithm in 20 mins when he was in a café ! We can visualise a city as a graph with each junction as a node and each path connecting them as an edge between them. Then we can apply Dijkstra algorithm to get the shortest path from our source vertex to destination vertex.

However, the Google maps we use today which is based on GPS technology uses a highly optimised algorithm called A* algorithm. Using Dijkstra algorithm would not be suitable as the number of nodes in Google maps is very large (In billions) which would take very long time to process and mammoth space is also required to store various data structures. A* is similar to Dijkstra's algorithm, which uses a heuristic function (It is a function which helps to decide which path to follow next) to navigate a better and more efficient path. Even if memory requirements and operations per node are more in A*, it is meant to be faster than Dijkstra algorithm because it explores a lot less nodes and the trade-off is always favourable.

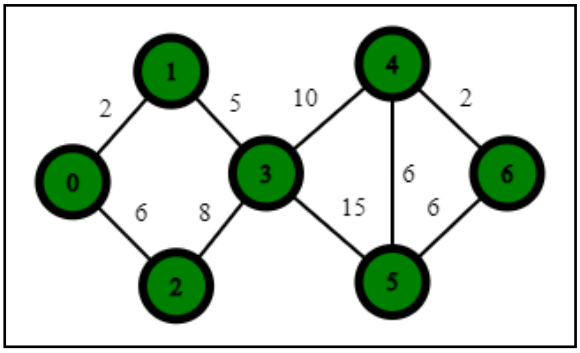
The shortest path algorithm is not limited to just finding the shortest path in a graph. It has a wide variety of applications in various fields (particularly where networking is involved) such as social networking. While using social networking sites, we often notice that the site suggests friends which we may know. This is achieved by employing the shortest path algorithm where each user is treated as a node and connections between users as edges. Dijkstra's algorithm is also widely used in the routing protocols .The algorithm provides the shortest cost path from the source router to other routers in the network.

More on Heuristics:

Heuristic is a technique which is used to solve a problem more quickly when common methods are too slow, or they can also be used to find approximate solutions in situations where we cannot find any exact solution. They are useful in solving very complex problems where finding exact solutions would take so long time that it is practically not feasible to wait for that long. Generally, Heuristics trades off accuracy and correctness for rapid processing. In contrast to algorithms, heuristics make an educated guess which serves as a guide for further explorations. The results of a heuristic are not predictable since it does not explore all possible states of a problem. Nevertheless, they do provide acceptable solutions within acceptable time limits and hence are widely used in industries.

First we study Dijkstra's algorithm and then prove it's correctness.

Consider the given graph. We have to find the shortest path from node 0 to all other nodes in the graph.



Step 1:

We create a set of nodes and their current corresponding distances from the source node 0. We also create the set of visited nodes which contain the nodes for which we know the shortest distance from node 0(source). Hence, our sets in the beginning would look like this:

Node	Distance (from 0)
0	0
1	Infinity
2	Infinity
3	Infinity
4	Infinity
5	Infinity
6	Infinity

Visited Nodes: {0}

In the beginning, we don't know the distance to any of the nodes except the source node itself (which is 0 trivially). We assume the distances to be infinite to all other nodes and then proceed to next step.

Step 2: (Relaxing of nodes)

Node	Distance (from 0)
0	0
1	2
2	6
3	Infinity
4	Infinity
5	Infinity
6	Infinity

From the current node, calculate distances of all neighbouring nodes not in the set of visited nodes from the nodes in the set of visited nodes and update it in the table only if the calculated distance is less than the distance in the table. From 0, the distance to it's neighbouring nodes 1 and 2 are 2 units and 6 units respectively. Both these distances are less than infinity, therefore we have to update the table. This is also known as relaxation of nodes. Here, we have relaxed node 1 and 2.

Step 3: (Greedy Step)

We choose the node which has the minimum distance from the source node and add it to the list of visited nodes. This is the minimum distance of the node from the source. In this case, we select 1 and add it to the set of visited nodes. Therefore, Visited Nodes: {0,1}

Step 4:

Repeat steps 2 and 3 until all nodes are added in the set of visited nodes.

In the given problem, the algorithm would look as follows:

Node	Distance (from 0)
0	0
1	2
2	6
3	7
4	Infinity
5	Infinity
6	Infinity

Visited Nodes: {0,1}

Relaxed Nodes : 3

Select the node with minimum distance → Select 2 → Visited Nodes: {0,1,2}

Node	Distance (from 0)
0	0
1	2
2	6
3	7
4	Infinity
5	Infinity
6	Infinity

Visited Nodes: {0,1,2}

Relaxed Nodes: None

Select the node with minimum distance → Select 3 → Visited Nodes: {0,1,2,3}

Node	Distance (from 0)
0	0
1	2
2	6
3	7
4	17
5	22
6	Infinity

Visited Nodes: {0,1,2,3}

Relaxed Nodes: 4,5

Select the node with minimum distance → Select 4 → Visited Nodes: {0,1,2,3,4}

Node	Distance (from 0)
0	0
1	2
2	6
3	7
4	17
5	22
6	19

Visited Nodes: {0,1,2,3,4}

Relaxed Nodes: 6

Select the node with minimum distance → Select 6 → Visited Nodes: {0,1,2,3,4,6}

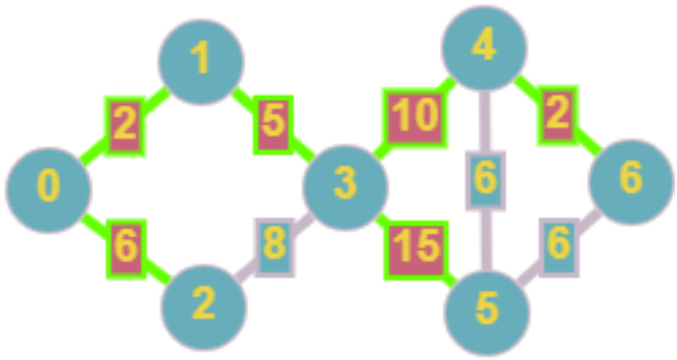
Node	Distance (from 0)
0	0
1	2
2	6
3	7
4	17
5	22
6	19

Visited Nodes: {0,1,2,3,4,6}

Relaxed Nodes: None

Select the node with minimum distance → Select 5 → Visited Nodes: {0,1,2,3,4,5,6}

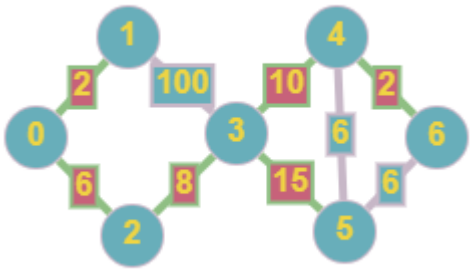
Since we added all nodes from the graph, we stop as we calculated shortest path from node 0 to all other nodes in graph.



The green lines show the selected shortest path from source node 0 to each of the other nodes.

As we keep mentioning again and again, the most challenging part of a greedy algorithm is not to design it but to prove it's correctness. Dijsktra's algorithm is greedy because of the fact that it always chooses the best available node(node with minimum distance) at each iteration.

We can observe that greedy choice property does not hold. The first node that we select may or may not be in the optimal solution. Consider the following graph with source as 0 and destination as 6:

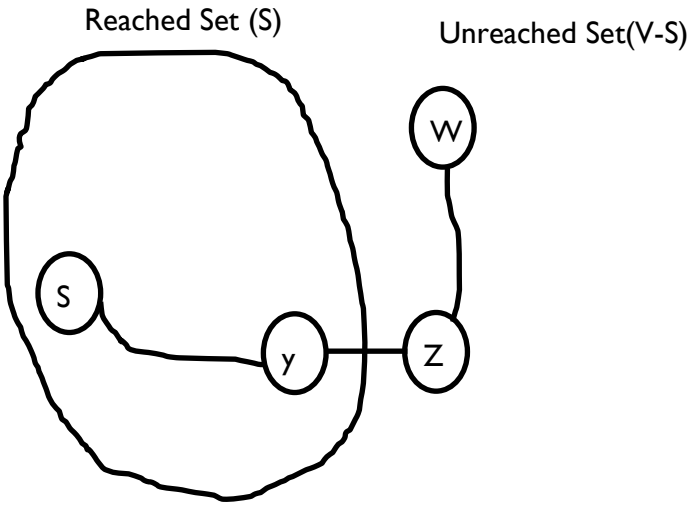


The first choice of selecting node 1 is not in the final shortest path from 0 to 6(0->2->3->4->6).

Nevertheless, Dijkstra's algorithm always produces the shortest path from a given source to destination and we prove it's correctness through induction on the set of visited nodes.

Let us denote the shortest path to any node v as d[v]. Let the set of visited nodes for which the shortest distance is known is S. Assume that we have included n nodes in S. Now we add a new vertex w into the set of visited nodes and claim that the distance calculate from Dijkstra is d[w]. If possible, let there be another path which is the shortest and the nodes which connect the two partitions of the graph (S and V-S) are y and z respectively. Since Dijsktra chooses the minimum weight edge among all the edges that go out from the visited set S, l(x,y) <= l(v,w).

D[y] = d[y] as y is in S and we assumed that we have calculate all distances till now correctly. Also, D[z] = d[y] + l(y,z). And since we are including w before z in set S, we can say that D[w] <= D[z]. We also know that the shortest path has an optimal substructure property, therefore, d[w] = d[z] + shortest distance from z to w. Hence, we conclude that: D[w] <= D[z] = d[y] + l(y,z) <= d[y] + l(y,z) + shortest distance from z to w = d[w]. Hence D[w] <= d[w]. Therefore, the path found by Dijkstra is the shortest.



The time required in Dijsktra's algorithm depends upon relaxing the edges and selecting the edge which corresponds to minimum distance from source. Each node can be connected to other n-1 nodes, so it will have n-1 edges. For each node, we process all its edges and then select the node which has the minimum distance from the source. For a graph with adjacency matrix representation and using linear search to get the minimum distance node, we can approximate the time taken as:

O(|E|(time taken to relax an edge) + |V|(time taken to get the minimum distance node))

To relax an edge, we only perform a constant number of operations(check if current distance is less than the previous distance and if yes update it). To get the minimum distance node, we perform a linear search through the set of nodes not in visited set. This can take upto O(|V|). Therefore, our time reduces to:

O(|E|.1 + |V|.|V|)

= O(|E| + $|V|^2$) = O($|V|^2$) (Since for any graph, no of edges |E| is O($|V|^2$).

However, if adjacency list representation of graph is used along with a priority queue for relaxing edges and selecting the minimum, the time can be reduced to O((|E| + |V|)log|V|).

Optimisation Of Dijkstra:

Dijkstra's Algorithm doesn't work well on huge graph sets where the number of nodes are mammoth. If directly applied on the urban road network, this algorithm will require gigantic amounts of computational resources and cannot meet the dynamic needs either. Dijkstra's Algorithm can be optimised using heuristic function that gives an "estimate" of the distance from the source to target node initially. The advantage of the A algorithm is that it does not need to traverse all nodes, but instead proceeds in the direction of the desired road (the target node that needs to be experienced) according to the selected heuristic function. It uses the cost function f(n) = g(n) + h(n), where g(n) is the actual distance to node n and h(n) is the distance estimated through heuristics. The heuristic function controls the behaviour of A* algorithm immensely:*

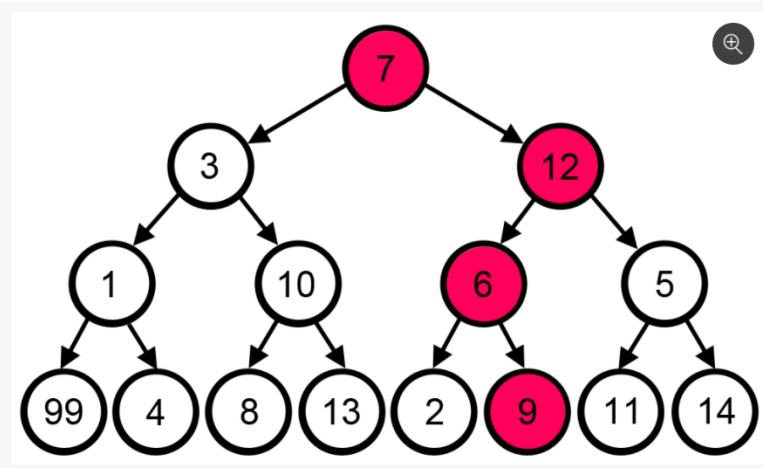
- *On one hand, if the heuristic function is 0, then only g(n) plays a role and the algorithm is equivalent to Dijkstra's Algorithm.*
- *On the other extreme, if the heuristic function is very high, only h(n) plays a role and the algorithm turns into a best first search and will not give optimal path value.*
- *If h(n) is always lower than the actual path length between the source and destination, then the algorithm is guaranteed to give an optimal path. However, the lower h(n) is, the more node A* has to explore making it slower (extreme case is equivalent to Dijkstra as in point 1).*

These are very interesting properties of the heuristic function and the A algorithm in general. In some situations, we may prefer to have a "good" path rather than a "perfect" path especially when "perfect" path takes very long to compute while the "good" path is calculated in a fraction of second. Also, the choice of the heuristic function is very important in the success of this algorithm.*

Not so Greedy !

Activity Selection & Shortest path problems are two prime examples of how powerful a greedy algorithm can be. It can solve very complex problems with ease. However, the greedy method always doesn't come handy as observed in the coin change problem (when denominations were changed), activity selection problem (when weights were assigned to activities) and Shortest path problem (when there are negative weights associated with the edges). Similarly, There are cases when the greedy algorithm might actually give the worst possible solution.

Frequently, there are problems where locally optimal choices are not globally optimal so we have to use the greedy algorithm with great care. For example, consider the following tree.



If we need to find the path such that we get the maximum sum of numbers along the path, the greedy algorithm would choose the largest number at each step. Therefore, it would follow the red path and the sum of the numbers would come out to be 7+12+6+9 = 34. However, the best possible path would be 7->3->1->99 which would give 110 as the sum.

The greedy algorithm fails to solve this problem because it makes decisions purely based on what the best answer at the time is: at each step it did choose the largest number. However, since there could be some huge number that the algorithm hasn't seen yet, it could end up selecting a path that does not include the huge number. The solutions to the subproblems for finding the largest sum or longest path do not necessarily appear in the solution to the total problem. The optimal substructure and greedy choice properties don't hold in this type of problem. Dynamic Programming, another widely used algorithmic technique is used to deal with these types of problems.

Greedy algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data. They can make commitments to certain choices too early which prevent them from finding the best overall solution later. Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum. If a greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the method of choice because it is faster than other optimization methods like dynamic programming, divide-and-conquer etc. Greedy algorithms are usually easy to think of, easy to implement and run fast.

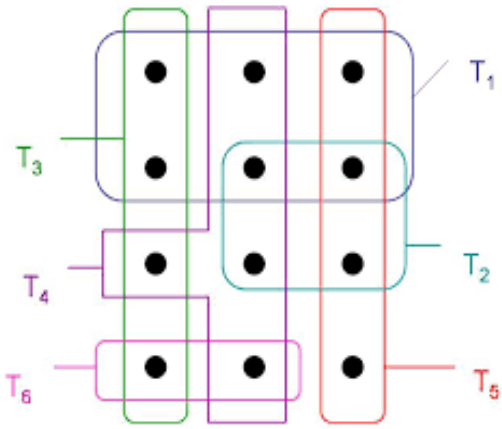
Approximation using Greedy Algorithm:

Many a times, the greedy strategy fails to find an optimum solution to a problem but it finds a "reasonable" approximation to the actual solution. This interesting property of greedy algorithm can be exploited to find approximate solutions to NP hard problems which do not have any polynomial running time algorithm to solve them. One such good example is set cover.

The set cover problem is to select the minimum cost selection of sets from a given list of subsets such that the selected sets contain all the elements of the set where each subset has a cost associated with it. This problem has many interesting applications. For instance, IBM applied the set cover problem to find computer viruses. The elements were viruses and the sets were substrings of 20 or more from viruses (about 9000 sets). A set cover of size 180 was found. This means that it is sufficient to search for these 180 substrings in a code to verify the existence of known computer viruses !

However, as mentioned before the set cover problem is NP hard and thus cannot be solved in polynomial time. However, the greedy algorithm provides a very good approximation of the actual solution and the advantage is that it solves the problem very fast.

The greedy strategy suggests to select that set which has maximum number of not already added elements (since this would make us select less sets to cover the whole set of elements). Since cost is also involved with each set, we can think of the ratio Cost(i)/number of new elements(i) which is to be minimized. The naïve algorithm would try out all the subsets and therefore take exponential time. However, the greedy strategy will produce a result in reasonable time within a bound of log N of the optimal solution where N is the number of elements.



The proof involves involved mathematical calculations. We prove it for a simpler version of the problem where cost of each set is 1 unit. The problem then reduces to finding the minimum number of sets. Let the optimal solution uses k sets. Then, there has to be atleast one set which covers atleast 1/k fraction of the elements (because if there were no such set, then the k sets cannot cover all the elements). After t selections made by the greedy algorithm, it will have atmost $n(1-1/k)^t$ elements left. When we put $t = k \ln n$, the number of elements left becomes less than 1 and hence all elements are covered. This means that the greedy algorithm selects ln n factor of sets more than the optimal algorithm in worst case.

Dynamic Programming

As we observed, the greedy algorithm is applicable for a very specific range of problems. Dynamic programming is an algorithmic paradigm which is applicable for a wide variety of problems and is relatively fast as compared to other algorithms (however, slow as compared to greedy). This technique was developed by American Mathematician Richard Bellman who also discovered the Bellman-Ford algorithm to find the shortest path in graphs with negative edge weights.

Dynamic Programming is used to solve optimization problems by breaking a problem into simpler sub-problems and storing the solution to each problem so that it is solved only once and its result can be used again while calculating another big problem.

Consider a simple problem:

1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1

If we were to find the sum, then we have to add the number of ones. This would give us 9 as the answer. If we add another 1 as:

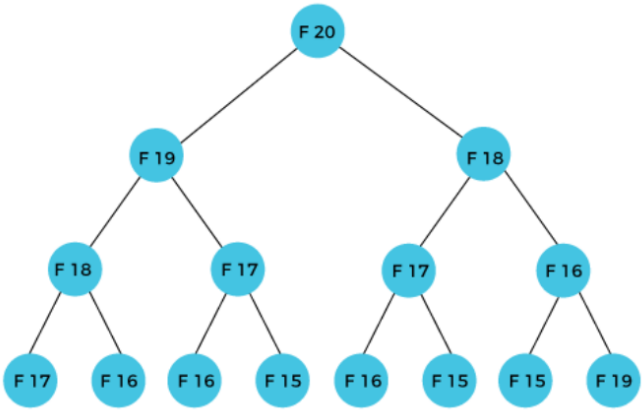
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1

We can simply tell the answer as 10 without having to recalculate all the terms again because we already know the previous sum as 9 or we can say that we "remembered" the solution to previous problem. This is what dynamic programming does. We can store the solutions whenever they are calculated and use them further to solve the problems ahead.

Let us consider the famous Fibonacci series whose nth term is given by f(n) = f(n-1) + f(n-2). The first few terms of the series are:

0,1,1,2,3,5,8,13,21,34,55,89,144….

Suppose we want to calculate F(20). We observe that for calculating F(20), we need to calculate F(19) and F(18).



Consider the tree shown. We observe that F(18) needs to be calculated 2 times, F(17) 3 times, F(16) 4 times, F(15) 5 times and so on. This makes the time required to calculate F(20) exponential due to the fact that we are calculating the same value again and again. To overcome this problem, dynamic programming stores the corresponding value whenever it finds it. And when it needs that value again, it just simply uses the value which it has stored. This reduces the calculation ef-

The first method which calculates the same value again and again is popularly known as recursion. Dynamic Programming is recursion with memorization so that it doesn't require to calculate the value again. No doubt, this increases the space required by the algorithm upto a very large extent as compared with simple recursion, but nevertheless whenever there is a tradeoff between space and time, we generally choose time as time once lost is lost forever. We can get a 1TB hard-disk storage easily but waiting for 10 days for getting the solution to our problem is not feasible. Therefore, although the algorithm requires very large space depending upon the size of input, it converts many exponential time algorithms to polynomial time (generally linear or quadratic).

The basic steps that Dynamic programming follows are:

- Breaks down the problem into simpler sub-problems: This is the most important step of this algorithmic technique. We try to express the problem as some combination of its subproblems (for e.g. sum of its subproblems as in the case of Fibonacci numbers, maximum of values its subproblems, etc.) and then solve these subproblems optimally and store the solution to each subproblem to avoid redundant calculations.
- Use the stored solutions to subproblems and combine them to solve the original problem.

Dynamic programming is efficiently used when the problem possess the following two properties:

- Overlapping Subproblems: It suits the problems where solution of one subproblem is needed repeatedly. It can simply get the solution to the subproblem from its "memory" and use it to solve many large problems. The computed solutions are stored in a table, so that these don't have to be re-computed. For e.g. in case of Fibonacci numbers, the value of F(3) is needed for calculating both F(4) and F(5). This means the larger the number of bigger problems share a subproblem, the more overlapping subproblems are and hence it is more optimised by solving through dynamic programming.
- Optimal Sub-Structure Property: This is the same property we saw in case of greedy algorithms. The optimal solution to a problem is also an optimal solution to any of its subproblems. It implies that the optimal solution can be obtained from the optimal solution of its subproblem.

There are two ways in which dynamic programming can be realised:

- Top Down approach: This approach uses recursion with memoization. It breaks it into simpler sub-problems, remembers the solutions to these subproblems and uses it to solve the problem. This is basically what we discussed till now. One thing to notice is it solves the subproblem only when required. This is generally the simplest way to implement dynamic programming as we solve the subproblems as and when we see them and store them. The calculation of Fibonacci numbers as shown in the tree diagram is an example of top-down approach where we don't know the value of any f(n) (except the base case) and then calculate and store them. For instance, for calculating F(20), we need F(19) and F(18). Once we calculate F(18), then we know what F(18) and F(17) are. We simply use these to calculate F (19) and then use these two to calculate F(20).

Next time if we want to find F(21), we simply use the stored values of F(20) and F(19) unlike recursion where we would have to calculate F(21) from beginning.

- Bottom Up approach: This approach is also known as the tabulation or table filling approach. In this approach, we solve all the previous subproblems in a particular order and move towards the original problem. It starts with the simplest problems (solutions to which are trivial) and then builds up solution to higher and higher problems in a particular order eventually solving the original problem. In the case of Fibonacci numbers, we start from f(2), then calulate f(3), then f(4) till we reach f(20). The subproblems have to be calculated in such a way that all the subproblems have been already solved which will be required to solve that subproblem. For instance, we have to solve the Fibonacci problem in such a way that we always know the value of the numbers 1 less than and 2 less than the current number. If we follow any other order, then we cannot solve the problem. This eliminates the recursion and instead solves the problem iteratively.

Consider the following two statements to better understand the difference between the two approaches:

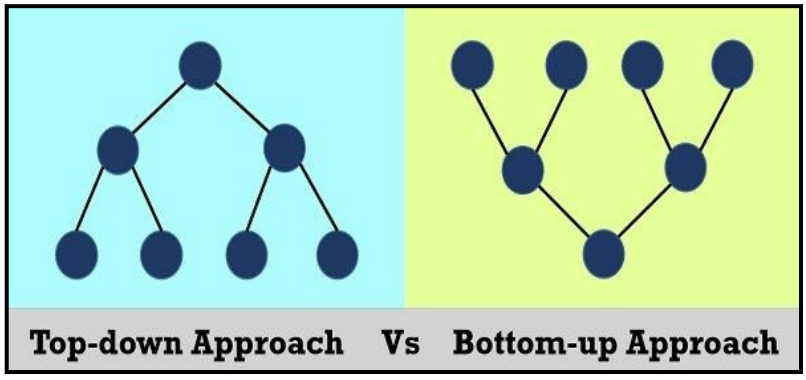
To bake the cake, I will have to read the recipe and to read the recipe I will have to purchase the recipe book.

I will buy recipe book, read the recipe and then bake the cake.

The first statement is analogous to Top-Down approach while the second statement is analogous to Bottom-Up approach.

Comparison between the two approaches:

Memoization(Top down) is indeed the natural way of solving a problem, so implementation is easier in memoization when we deal with a complex problem. Coming up with a specific order while dealing with lot of conditions might be difficult in the tabulation. Memoization is also useful in situations where we do not need to calculate solutions to all subproblems. However, when a lot of recursive calls are required, memoization may cause memory problems because it might have stacked the recursive calls to find the solution of the deeper recursive call but we won't deal with this problem in tabulation.



Generally, memoization is also slower than tabulation because of the large recursive calls. For example, if we want to calculate F(10^6) then it has to make a large number of recursive calls and we may run out of stack space. Both the approaches has its own pros and cons, and depending upon the problem, one might be suited better than the other. However, both approaches encompass the basic essence of dynamic programming: Divide the problems into subproblems and remember the solutions to subproblems to use it when needed again.

Now let us apply dynamic programming to solve some problems which greedy failed to solve. First, we start with the minimum number of coins that make a given value problem. During the discussion of greedy algorithms, we found that there are certain denominations of the coin for which the greedy algorithm provides the optimal answer whereas it fails on some. Dynamic programming is used to overcome this shortcoming and is applicable on any denominations of the coins. It is guaranteed to give the optimal solution. Let us revisit the problem once again.

We have n coins $D_1, D_2, \ldots, D_n$, and we have to find the minimum number of coins such that the sum of these coins equal the given value (say S). To solve this problem, we observe that this problem possess both the optimal substructure property and the overlapping subproblems property.

We look at what the subproblems are. Lets say M_n is the minimum number of coins required to sum upto n. If we choose D_1 as the first coin, then the total number of coins required is $1 + M_{n-d\_1}$ (One for the coin D_1 we just selected and M_{n-d_1} for the minimum number of coins to sum upto $n - D_1$). Similarly, we can select D_2 as the first coin, then the total number of coins required is $1 + M_{n-d\_2}$. Since, we don't know which coin we should select first to minimize the number of coins, we try over all possible coins from D_1 to D_n. And obviously, we want to consider only those coins whose value is less than the sum S. Therefore, we can write M_n in terms of its subproblems as:

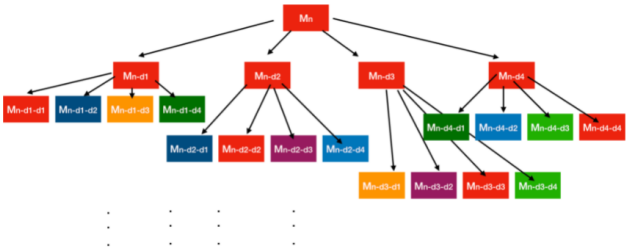
$$M_n = min_{i:d_i \leq n}\{1 + M_{n-d_i}\}$$

Optimal Substructure Property (Optimal solution to a problem also contains optimal solution to its subproblem) :

Suppose we found a optimal solution to M_n with the first coin as D_i . If the solution does not contain the optimal solution to the subproblem M_{n-d_i} (i.e. there exists a solution which uses lesser coins to sum up $n-D_i$ than our solution) then we can use this solution and add 1 to get the solution to our original problem which has lesser number of coins than our original solution. This is a contradiction to the fact that original solution was optimal. Therefore, there exists no solution which uses lesser number of coins for the subproblem M_{n-D_i} and hence this problem has the optimal substructure property.

Overlapping Subproblem Property (Many subproblems share a common subproblem):

To calculate a subproblem M_{n-d_1}, we would have to calculate $M_{n-d\_1-d\_4}$ which would also be used in the calculation of M_{n-d_4}. Therefore, we can store the solutions to these subproblems and use it whenever we require again avoiding the overhead of calculating it again and thereby boosting the time required. The following figure shows this property evidently:



All set, now we are ready to apply the actual algorithm and get the answer beautifully. We follow the bottom-up approach i.e. constructing solutions to the smallest subproblems (trivial ones) and building more and more complex problems until we reach the solution to the original problem.

We store the minimum number of coins required for each value from 1 to n and denote it as M_i where 1<=i<=n. Our final answer is M_n. M_0 is trivially 0 (since we require 0 coins to sum upto 0). Now, we construct the solution piece by piece elegantly:

For every i in 1 to n, do the following:

consider minimum coins(M_i) to be infinity.

Then, for each denomination D_j <= n do the following:

$M_i = \min(M_i, 1 + M_{i - D\_j})$

Return M_n

We observe that a 4 line algorithm gives the answer to such a seemingly complex problem so easily. This shows how powerful our algorithm is! On careful analysis, we observe that we are building solutions for M_1, M_2, M_3, … , M_n sequentially and store it. We are calculating the values of Mi in such a way that we already know the values required to calculate it (since we are storing the solutions to all the calculated subproblems). This solution is also able to tell if we can get the sum using these coins at the first place. If the value of M_n is infinity, then there is no possible way we can reach that sum using the given coins. Let us understand this through a concrete example:

Let the denominations be 1,5,6,9 and the required sum be 13. So we tabulate each step of our algorithm and observe how it changes.

Sum	No of coins (minimum)
0	0

This is the trivial case, where we need 0 coins.

Sum	No of coins (minimum)
0	0
1	1

Here i = 1. Now we check for all coins which are less than or equal to1. Only Coin with denomination 1 satisfies this condition. Now we update, $M_1 = \min(\text{infinity}, 1 + M_0) = \min(\text{infinity}, 1) = 1$. Since no other coins are needed to be checked, we conclude that $M_1 = 1$.

Sum	No of coins (minimum)
0	0
1	1
2	2

Here i = 2. Now we check for all coins which are less than or equal to 2. Only Coin with denomination 1 satisfies this condition. Now we update, $M_1 = \min(\text{infinity}, 1 + M_1) = \min(\text{infinity}, 1+1) = 2$. Since no other coins are needed to be checked, we conclude that $M_2 = 2$.

Using similar reasoning, we can figure out for sum = 3 & 4 also such that $M_3 = 3$, $M_4 = 4$.

Sum	No of coins (minimum)
0	0
1	1
2	2
3	3

Sum	No of coins (minimum)
0	0
1	1
2	2
3	3
4	4

Now for sum = 5, we figure out that coins with denominations 1 and 5 both satisfy the criteria of being less than or equal to sum. So we have to check for both. First we check with 1 and update $M_5 = \min(\text{infinity}, 1 + M_4) = \min(\text{infinity}, 1+4) = 5$. Now, we check with 5 and update $M_5 = \min(5, 1 + M_0) = \min(5, 1) = 1$. Therefore, we can sum to 5 using only one coin. Using similar logic, we can construct the whole table till 13.

Sum	No of coins (minimum)
0	0
1	1
2	2
3	3
4	4
5	1

Sum	No of coins (minimum)
0	0
1	1
2	2
3	3
4	4
5	1
6	1

With Coin 1:

$M_6 = \min(\text{infinity}, 1 + M_5) = \min(\text{infinity}, 1 + 1) = 2$.

With Coin 5:

$M_6 = \min(2, 1 + M_1) = \min(2, 2) = 2$.

With Coin 6:

$M_6 = \min(2, 1 + M_0) = \min(2,1) = 1$.

Hence, $M_6 = 1$.

Sum	No of coins (minimum)
0	0
1	1
2	2
3	3
4	4
5	1
6	1
7	2

With Coin 1:

$M_7 = \min(\text{infinity}, 1 + M_6) = \min(\text{infinity}, 1 + 2) = 3$.

With Coin 5:

$M_7 = \min(3, 1 + M_2) = \min(3, 3) = 3$.

With Coin 6:

$M_7 = \min(3, 1 + M_1) = \min(3, 2) = 2$.

Hence, $M_7 = 2$.

Complete Table:

Sum	Coins Checked				No of coins (minimum)
	Coin 1	Coin 5	Coin 6	Coin 9	
0	-	-	-	-	0
1	Min(inf, 1 + M_0) = 1	-	-	-	1
2	Min(inf, 1+ M_1) = 2	-	-	-	2
3	Min(inf, 1+ M_2) = 3	-	-	-	3
4	Min(inf, 1+ M_3) = 4	-	-	-	4
5	Min(inf, 1+ M_4) = 5	Min(5 , 1+ M_0) = 1	-	-	1
6	Min(inf, 1+ M_5) = 2	Min(2 , 1+ M_1) = 2	Min(2 , 1+ M_0) = 1	-	1
7	Min(inf , 1+ M_6) = 2	Min(2 , 1+ M_2) = 2	Min(2 , 1+ M_1) = 2	-	2
8	Min(inf , 1+ M_7) = 3	Min(3 , 1+ M_3) = 3	Min(3 , 1+ M_2) = 3	-	3
9	Min(inf , 1+ M_8) = 4	Min(4 , 1+ M_4) = 4	Min(4 , 1+ M_3) = 4	Min(4 , 1+ M_0) = 1	1
10	Min(inf , 1+ M_9) = 2	Min(2 , 1+ M_5) = 2	Min(2 , 1+ M_4) = 2	Min(2 , 1+ M_1) = 2	2
11	Min(inf , 1+ M_{10}) = 3	Min(3 , 1+ M_6) = 2	Min(2 , 1+ M_5) = 2	Min(2 , 1+ M_2) = 2	2
12	Min(inf , 1+ M_{11}) = 3	Min(3 , 1+ M_7) = 3	Min(3 , 1+ M_6) = 2	Min(2 , 1+ M_9) = 2	2
13	Min(inf , 1+ M_{12}) = 3	Min(3 , 1+ M_8) = 3	Min(3 , 1+ M_7) = 3	Min(3 ,1+ M_4) = 3	3

Table illustrating the algorithm at each step and how does it choose the minimum number of coins

Thus, we find that we can sum 13 using only 3 coins. The greedy method would have selected Coin 9 and then 4 times Coin 1 to get a total of 5 coins. This shows that greedy method fails in this problem but the dynamic programming method is guaranteed to produce an optimal solution everytime regardless of the denominations and the sum.

We observe that this algorithm uses two loops: One for calculating minimum number of coins for each number and checking all possible denominations for each number. Therefore, the total running time is O(nd), where n is the sum and d is the number of denominations.

We can also make a small modification in the algorithm to actually get the coins needed for making the change as the algorithm only tells us about the number of coins which is not very useful. Suppose, we know the first coin needed to get the optimal solution to each value from 1 to n. Now, we know the first coin needed to make n (say d_1). We subtract this coin from n and get a new value n-d_1. Since we know the first coin to get the optimal solution to each value from 1 to n, we also know the first coin to get the optimal solution to n - d_1(say d_2) which is also the second coin to make n optimally (since the problem has optimal substructure property). Proceeding in a similar fashion till the number becomes 0 or less than 0, we can obtain all the coins needed to make n optimally.

The only thing remaining is to store the first coin to get each of the values 1 to n optimally. This we can easily do by making some changes to the minimum calculating step. Let S_i denote the first coin needed to make i optimally. Then,

For every i in 1 to n, do the following:

consider minimum coins(M_i) to be infinity.

Then, for each denomination D_j <= n do the following:

If(1 + M_{i-d_j} < M_i):

$M_i = 1 + M_{i-d\_j}$

$S_i = d_j$

If we construct the S_i table for the previous example, we would get:

Value	0	1	2	3	4	5	6	7	8	9	10	11	12	13
S_i	0	1	1	1	1	5	6	1	1	9	1	5	6	1

Now, for n = 13, S_{13} = 1

13 - 1 = 12, S_{12} = 6

12 - 6 = 6, S_6 = 6

6 - 6 = 0, S_0 = 0

Therefore, we can pay 13 optimally using one 1 rupee coin and two 6 rupees coin.

There can be a situation where we have a limited number of supplies for each coin. In such a scenario, This algorithm would not work as we assume that we can use any coin any number of times. Therefore, to solve this problem we combine dynamic programming with backtracking to cut the partial solutions which doesn't lead to the optimal solution.

Dynamic programming is very useful in solving these types of problems which have both optimal substructure and overlapping subproblems.

Now, we consider the activity selection problem which we discussed during greedy algorithms. The greedy algorithm works wonderfully when the activities do not have a "weight" associated with them. However, problems arise when they have a weight associated with them. We saw that the greedy algorithm may not produce the correct optimal result in such cases and hence we turn to dynamic programming for help.

Consider the following set of activities with their start and finish times and weights associated with them. We have to select activities in such a way that the total weight of the activities is maximised.

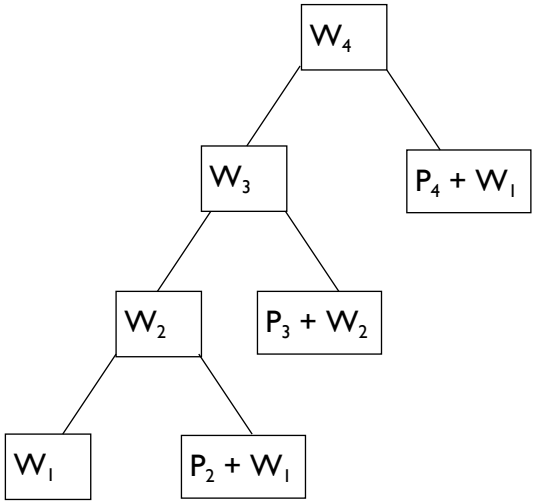
Jobs	Job 1(A)	Job 2(B)	Job 3(C)	Job 4(D)
Start Time	1	3	6	2
Finish Time	2	5	19	100
Weight	50	20	100	200

First, we look at the subproblems. Let us say that W_i is the maximum weight which can be obtained by selecting from first i activities from the set of sorted activities. Now to maximise W_i, we can either select the ith activity or leave it. When we leave it, W_i is the same as W_{i-1} (i.e. the weight obtained by selecting out of i-1 activities) . When we select the ith activity, then the total weight obtained is $P_i + W_k$ (where P_i is the weight of the ith activity and k is the latest activity which does not overlap with i). This follows from the fact that, if we select the ith activity, then the total weight has to be the sum of the weights of activities that give maximum weights without overlapping with i and the weight of ith activity itself. Since we need to maximise the weight, we take the maximum of these two quantities.

$$W_i = max\{W_{i-1}, P_i + W_k\}$$

Now we show that this problem has the optimal substructure property. Consider an optimal solution to an instance of the problem. If the optimal solution does not contain the nth activity, then the solution is same as the solution to the activities from 1 to n-1. If this sub-solution has any other optimal solution which selects the activities that contains more weight than the current solution, then this solution is optimal which contradicts that our original solution is optimal. This means that if the solution is optimal for 1...n activities, it is also optimal for 1…(n-1) activities. Consider the second case, when the optimal solution contains the nth activity. Then we are taking the subproblem as P_i+W_k where W_k is the set of activities which give the maximum weight and do not overlap with nth activity. If this set of activities had a solution which had more weight, then we could choose this and add P_i giving a solution having more weight than our optimal solution which is again a contradiction. Hence, in both cases optimal substructure property holds true.

This problem has also the overlapping subproblems feature since the solutions to previously computed sub-problems are needed by many other larger problems. (For instance, W_5 requires W_4 and possibly W_3 (in case, 3 is the latest activity that does not overlap 5), and W_4 requires W_3).



From the tree for the current problem, we can observe that we require W_1 3 times, W_2 2 times, W_3 and W_4 each 1 time. Since our example is small, we do not observe much overlapping. But problems with hundreds and thousands of activities can lead to overlapping of the order of millions and we certainly don't want to calculate the same value a million times when we can just store it and use whenever we want!

Since the problem has both the features, we can use dynamic programming to solve it in an efficient manner. We again construct the table using bottom up approach since it more easier and intuitive to grasp.

The algorithm is as follows:

Sort the activities in increasing order.

$W_0 = 0$

$W_1 = 1$. (The activity itself is the maximum weight activity)

For each activity in 1...n:

j = largest index such that finish time of W_j <= start time of W_i

$W_i = \max(W_{i-1}, P_i + W_j)$

Return W_n

Again we observe that the solution is very elegant, simple and short. We are building the solutions from W_1 to W_n sequentially so that we know the answers to subproblems needed to calculate the current problem. We can construct the following table for an even clear picture:

Index upto which answer is to be found(i)	Latest Non-overlapping index	Activities Considered	W_i
1	0	$\text{Max}(W_0, 50 + W_0)$	50
2	1	$\text{Max}(W_1, 20 + W_1)$	70
3	2	$\text{Max}(W_2, 100 + W_2)$	170
4	1	$\text{Max}(W_3, 200 + W_1)$	250

Therefore, we find that the answer $W_4 = 250$ is the maximum weight we can get by scheduling these activities. We also note that greedy algorithm fails to get the optimum solution since it would choose activity A then B and then C giving a total weight W = 170. This method only gives the total weight of activities. As observed previously, we can make a slight modification and thereby also get the activities selected.

We can just store the previous activity selected for each of the activities say S_i and get the list of activities we selected. Also to check if the current activity was included in the list or not, we can maintain a Boolean array B where a value of 1 denotes that it was included and vice-versa. The following modified table shows it clearly:

Index upto which answer is to be found(i)	Latest Non-overlapping index	Activities Considered	W_i	S	B
1	0	Max(W_0, 50 + W_0)	50	0	1
2	1	Max(W_1, 20 + W_1)	70	1	1
3	2	Max(W_2, 100 + W_2)	170	2	1
4	1	Max(W_3, 200 + W_1)	250	1	1

Now to recover the activities selected, we just check if the Boolean index is 1 or not. If it is 1, it means the current index is also included in the list and if it is 0, the current index is not included in the list. Then we can print the value of S_i and then make the index equal to this value (since this the index now for which we have to find the list of activities).

```
While(index >= 1)
    if(B_index is 1):
        Print index
    if(S_index >= 1)
        Print S_index
    index = S_index
```

For the current example,
Print 4 => Print S_4 = 1 (now index becomes 1) => Print 1 => (Since S_{index} = 0 we don't print it).

Therefore, we recover the activities 1 and 4 as the activities selected to get the maximum weight of 250.

The time taken by this algorithm depends on two things: i) the number of activities to process (its quite trivial that the larger the number of activities the more time it will take to produce the results) ii) the time taken to find the latest activity that does not overlap with current activity

For a simple linear search (i.e. scanning through all the activities from beginning to current activity and comparing the finish and start time), it would be O(n) for each search. And we have to perform this search for each of the activities. Therefore, the total complexity would be $O(n^2)$. However, we can further improve this by using binary search in ii) which takes O(log n) time and the total complexity is then reduced to O(n logn).

Another interesting application of the Dynamic Programming approach is the solution to 0/1 knapsack problem. Suppose you wake up and find yourself in the middle of a mysterious island. You see various valuable items like gemstones, gold bars, coins etc. in front of you. You also have a bag that can be used to carry these items. However, the bag has a limitation on the weight of items you can put in it and exceeding the weight will tear apart the bag into pieces and then you cant carry any of the items with you. Therefore, you want to choose the items in such a way that the bag does not falls apart, and at the same time the value of the items you select is maximum. This is known as the knapsack problem. We consider the case when there is only one item of each kind and you can either choose to take that item or not choose to take that item.

Item	1	2	3	4
Weight(W_i)	3	2	4	1
Value(v_i)	8	3	9	6

We have to keep track of both weight and value collected. Therefore, when we look at the subproblems, we have to incorporate both these parameters while calculating the optimal value. While considering the ith item, we check that if its weight is less than the current allowed weight w. If it is greater, then we do not consider this item and the solution is same as with that of i-1 items and weight limit w. If it is smaller, then we have two choices: either include or leave and we take the maximum of two choices to get the optimal solution. Therefore, if F(i,w) represents the optimal value for first i items with weight limit w, then we can write F(i,w) as:

$$F(i,w) = \begin{cases} F(i-1,w), & \text{if } w_i > w \\ max\{F(i-1,w), (F(i-1,w-w_i)+v_i)\}, & \text{if } w_i \leq w \end{cases}$$

Using similar arguments as in the previous situations, we can argue that this problem has the optimal substructure property. We can also observe overlapping subproblems from the following diagram:

The trivial cases are F(i , 0) which is equal to 0 for any i since we assume that any item has weight atleast 1 unit. So, no item can be accommodated in a bag with weight limit 0 units. Also F(0, w) for any w is also 0 because we don't get any value if we don't pick any item. Therefore, currently the table would look like this:

i W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

For i=1, we cant choose it if weight is less than 3. Therefore, F(1,1) = F(1,2) = F(0,w) = 0. When w=3, we can choose item 1 or decide not to choose. If we choose it we get the value as 8 and if we do not choose it then we get the value as 0. We take the maximum of these 2 and therefore choose item 1. Again for W = 4, we get the maximum value of 8 by choosing 1. Therefore , the table is now:

i W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	8	8	8
2	0					
3	0					
4	0					

Similarly, we update the values for item2, item 3, item4 and get the final table as:

i W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	8	8	8
2	0	0	3	8	8	11
3	0	0	3	8	9	11
4	0	6	6	9	14	15

Clearly, the answer is F(4,5) = 15 i.e. considering all items with the weight limit of the bag. The algorithm can be written as :

For all items in 1 to n:

F(i,0) = 0

For all items in 1 to n (index = i):

For all weights in 1 to W_{bag} (index = j):

If(W_i <= j):

F(i,j) = max{F(i-1,j) , F(i-1,j-W_i)}

else:

F(i,j) = F(i-1,j);

The time taken by the algorithm is calculated by taking into account 2 things: i) For a particular I, we find F(i,w), where w varies from 1 to W_{bag}. ii) We have to repeat this for every i from 1 to n where n is the number of items. Therefore, the total time is O(W_{bag} n). Since this algorithm only gives the total value and not the actual items we picked, we make a small modification as in the previous cases to incorporate the items as well.

We can get the items we selected for maximum value from the cost matrix itself. Consider the following cost matrix:

i W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	8	8	8
2	0	0	3	8	8	11
3	0	0	3	8	9	11
4	0	6	6	9	14	15

Consider the last cell in green. If we move up the column, we observe that the value decreases from 15 to 11. This means that including item 4 in the final list of items is necessary because not including it will decrease the total value of the items selected. After selecting item 4, we have now a weight limit of 5-1 = 4. Now, we move to column with label 4 and a row up since we already selected 4. This lands us to the yellow cell. Here, we apply the same logic and and observe that when we move up, the total value of items selected is decreased. Therefore, we have to select item 3 compulsorily. Now we have the weight limit of 4-4 = 0. Since, we cant add any more value as the bag is full now, we stop here. Therefore, selecting item 3 and item 4 will give the optimum value of 9+6 = 15. The algorithm for getting the items is:

```
if cost[i, j] != cost[i-1, j] // if cost changes then include the item and update the weight remaining
  print i
  j = j-w[i]
  i = i-1
else
  i - i-1 //don't include item, just move upward
```

Here, i and j denote the rows and columns of the cost matrix respectively.

So far we saw some amazing applications of dynamic programming to seemingly complex problems and its beauty lies in the fact that it always produces the optimal solution, if applicable. We also observed that the algorithms produced by dynamic programming for solving problems is very small in size (8-10 lines) yet so powerful. We also emphasised on the fact that why dynamic programming is much more efficient than simple recursion because of the fact that it stores the solutions to already calculate subproblems to avoid recalculating it again and again. To stress on this fact, we can look at the example of Fibonacci number, where the calculation of the one millionth Fibonacci number using recursion would take approximately millions of years on a modern computer, whereas using dynamic programming would take merely 1 second ! The major advantages that this algorithmic technique provides over other algorithms are:

i) It provides both local and global optimal solution in every case(where applicable) unlike greedy.

ii) It generally has a polynomial time complexity in any problem(which is pretty good compared to exponential time complexity in recursion).

iii) It is applicable to a comprehensive range of problems such as computing the edit distance between two strings, finding shortest path from each node to every other node in a graph, parenthesizing a set of numbers to get maximum product etc.

The disadvantages of using dynamic programming are:

i) It generally is a memory intensive algorithm. If there are heavy constraints on the memory capabilities of the system, then it becomes irrelevant.

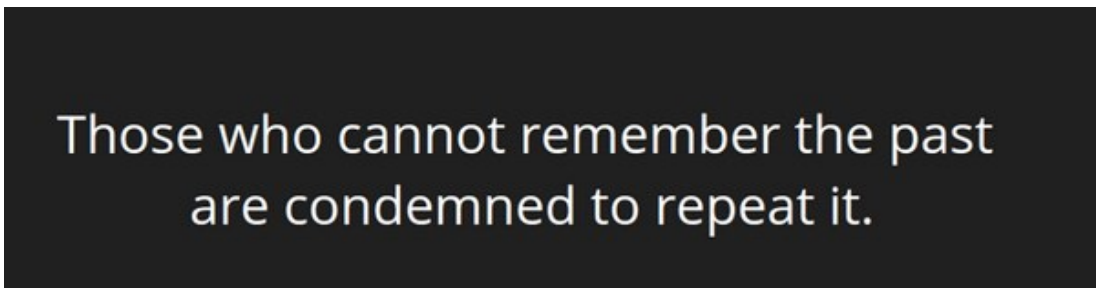
ii) In the top-down approach of Dynamic Programming , functions are called recursively. The stack memory where the functions are called again and again till it reaches the base condition may overflow due to excessive recursive calls.

iii) It fails in cases where a problem is not expressible as an expression of its subproblems.

There are many instances where dynamic programming cannot be applied. For example, the problem of finding the longest path in a graph doesn't hold the optimal substructure property and hence cannot be solved using dynamic programming. This problem is in fact NP hard which means no efficient algorithm has been developed till date to solve this problem.

However, due to its applicability on wide range of problems, it is extensively used in industries in production planning, in molecular biology for DNA sequencing, in prediction tools, in water resource managements and much more.

In conclusion, dynamic programming is an exceptional variant of recursion that compensates for its drawbacks. The following quote is an apt summarization of dynamic programming:



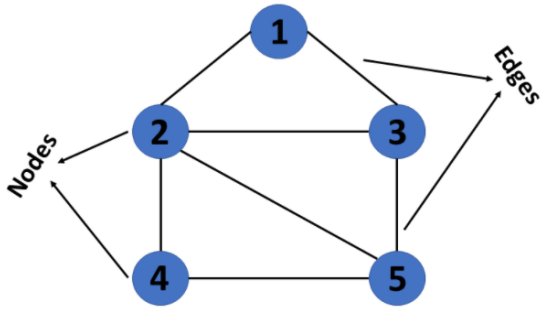
Greedy Algorithms can be thought of as a special case of Dynamic Programming. In dynamic programming we examine a set of solutions to smaller problems and pick the best among them. In a greedy algorithm, we believe one of the solutions to smaller problems to be the optimal one (depending on certain properties of the problem) and don't examine other solutions. The greedy approach is more efficient than the dynamic programming method due to the fact that we don't consider all possible solutions at any stage but assume that the best possible solution at that stage would lead to the optimal solution globally. However, although a little less efficient, the dynamic programming approach always guarantees an optimal solution which is one of the most notable features of the algorithm.

When we compare the two algorithms, we have to keep a number of things in mind. Firstly, they both solve different kind of problems. A problem that can be solved by greedy can be definitely soved by dynamic programming but not the opposite. The greedy algorithm is more intuitive and simpler to implement as compared to dynamic programming and also yields the most efficient solution if applicable. However, it is applicable only to a limited number of problems which satisfy the optimal substructure and greedy choice properties. It is not guaranteed to produce the optimal solution for problems not having the greedy choice property. On the other hand, dynamic programming makes sure that the solution is always optimal though it is less efficient than greedy in terms of time and space taken. Both the algorithms are super powerful and are widely used to solve optimization problems some of which we saw in the previous pages. However, there are limitations to both these techniques and some problems require an altogether different approach to solve them.

The decision (choice) made by Greedy method depends on the decisions (choices) made so far and does not rely on future choices or all the solutions to the subproblems. However, Dynamic programming makes decisions based on all the decisions made in the previous stage to solve the problem. Hence, the major differences can be summarized in the following table:

Greedy	Dynamic Programming
Makes the best possible choice at each step	Solves all the subproblems at each step and takes the best out of it
It is more simple and intuitive to implement	It requires careful thinking and defining the subproblems and their relation with the larger prob-
It requires much less space and time.	It generally requires more time and space than greedy but still is polynomial in time.
It may not provide the optimal solution but can give good approximations to optimal solution.	It always provides the optimal solution.

Graphs:



Graphs are one of the most versatile and widely used data structure that are used to represent real world data in computers. A graph is made up of nodes and nodes are connected through edges. Each edge represents some kind of relation in the real world like a path between two nodes, a link between two websites, a link between 2 persons etc.

Graph may be directed or undirected i.e. the edges may denote a direction from one node to another or simply an undirected edge showing both ways connection.

Graphs can be represented in one of the two ways:

i) Adjacency Matrix Representation: This is a matrix of dimensions nxn where n is the number of nodes in the graph. Whenever there is an edge between nodes i and j, the entry m_{ij} in the matrix is 1. For undirected graphs, the entry m_{ji} is also 1.

ii) Adjacency List Representation: This is an array of length n where each index points to a list of nodes to which the index is connected through an edge.

The graph above can be represented in both the formats as follows:

Node	1	2	3	4	5
1		1	1	0	0
2	1		1	1	1
3	1	1		0	1
4	0	1	0		1
5	0	1	1	1	

Node	Neighbours
1	2,3
2	1,3,4,5
3	1,2,5
4	2,5
5	2,3,4

Both the representations have certain advantages over the other. In the adjaceny matrix representation, we can check the presence of an edge between two nodes quickly by just checking the m_{ij} value. However, in the adjacency list representation, we would have to traverse all the neighbours of node i or j which is O(n) in case of a complete graph. On the other hand, if we want to add a new node to a graph, we can just add another entry in the adjacency list and add its corresponding neighbours. But in the adjacency matrix, we have to add a full row and column and fill each of the new entries created with 0 or 1 depending on the edges. The memory usage is also high in adjacency matrix representation ($O(V^2)$) whereas the adjacency list only stores the actual edges and hence memory required is O(E). The adjacency list is preferred over the adjacency matrix because it allows faster traversal of graph and hence is suitable for building algorithms using it.

Graph Algorithms

There are countless instances of real life where graphs are used. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. They are powerful, versatile, widely spread and used by everyone, without even knowing it. They are also used extensively by social networking giants like Facebook and linkedIn to store and represent data. The Windows file system uses graph structures to organise files and folders. The internet is a humongous graph which is a connection of hosts and servers and users. Google maps uses graphs to store cities and links between them. The recommendations on e-commerce websites also uses graph for recommending relevant products . Apart from the IT field, graphs are also extensively used in linguistics to extract meaningful data out of sentences, in physics and chemistry to model atoms and molecules, in social sciences to explore rumour spreading, in biology to represent migration patterns of various species etc.

Owing to their significant applications in almost every major field, it naturally becomes essential to devise efficient graph algorithms that can process the graphs and produce quick and meaningful results.

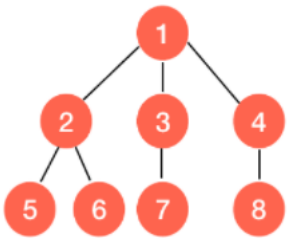
There are many graph algorithms that are developed till now. Some of the most important ones include:

- Breadth First Search & Depth First Search (It refers to the way in which we traverse our graph).
- Shortest path algorithms : These include Dijkstra's Algorithm, Bellman Ford algorithm and Floyd's algorithm.
- Minimum Spanning Tree: This algorithm to find the minimum weight tree of a graph was developed independently by Prim and Kruskal.
- Maximum Flow Algorithm : Used to model traffic flow and water flow in a connected pipeline system
- Strongly Connected components : These are used to identify a group of people who have similar interests or background and suggest friends or pages to them accordingly by the social networking site. Kosaraju's algorithm can be used to find strongly connected components efficiently.

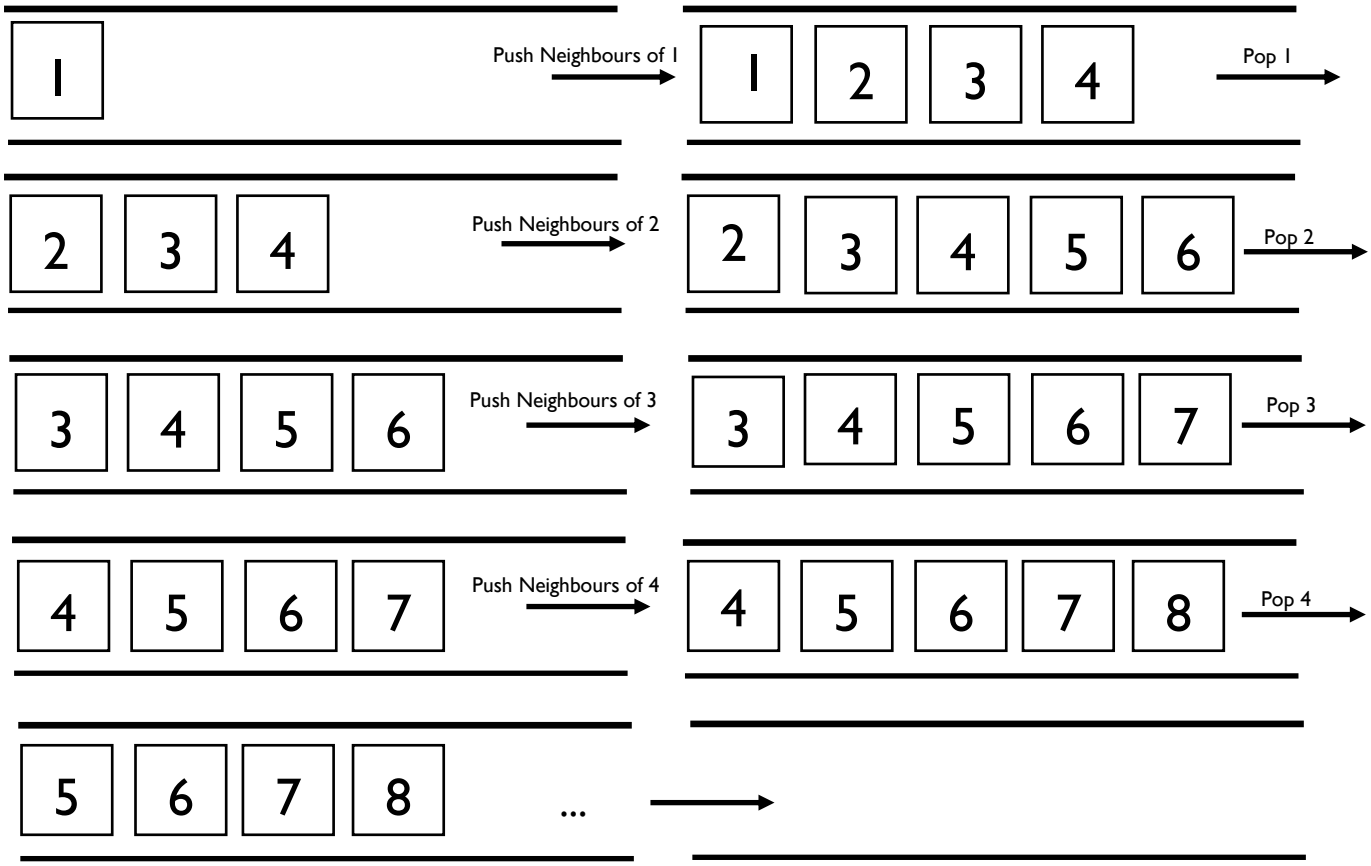
These are the few algorithms which are most commonly used. Apart from these, there are numerous other algorithms which are used to solve real world problems using graph.

Breadth First Search:

Breadth First Search(BFS) is a way of travelling the graph such that we visit a node and then all its adjacent node and then again the adjacent nodes of these nodes and so on. If we consider the following graph:



We can observe that if we start travelling from node 1, we would then visit node 2,3,4 and then 5,6,7,8 in BFS. To travel in BFS, we use a queue in which we push all the neighbouring nodes of current node and mark them as discovered to avoid visiting them more than once. When we pop the node from the queue, it implies that all its neighbouring nodes are either in the queue or have been visited already. In the beginning, the queue is empty. We push the node 1 in queue. Then add all the neighbouring vertices of node 1 into the queue i.e. 2,3 and 4. Then we pop node 1 from the queue indicating that it has been visited. Now, the first element of queue is node 2. We push all the neighbours of 2 i.e. 5 and 6 in the queue and pop node 2 again indicating that it is visited. We repeat these steps till we reach the end of the graph. The following queue simulates the above algorithm.



BFS traversal of the graph. Using the adjacency list representation of graph, we can traverse the graph in O(V + E), since each edge is enqueued only once and each edge is examined atmost twice, once for each of the vertices it's incident on.

The BFS algorithm is used in the following systems:

i) Crawlers in Search Engines: Crawling is a process used by search engine bots to visit and download a page and extract its links in order to discover additional pages. The algorithm starts traversing from the source page and follows all the links associated with the page. Here each web page will be considered as a node in a graph.

ii) Shortest path: The Dijkstra's algorithm for shortest path uses BFS traversal as it explores all the neighbouring nodes of the current nodes and then chooses the one which has the least distance from the source.

iii) Broadcasting: Networking makes use of what we call as packets for communication. These packets follow a traversal method to reach various networking nodes. One of the most commonly used traversal methods is BFS. It is being used as an algorithm that is used to communicate broadcasted packets across all the nodes in a network.

iv) Peer to Peer Networking: Breadth-First Search can be used as a traversal method to find all the neighbouring nodes in a Peer to Peer Network. For example, BitTorrent uses Breadth-First Search for peer to peer communication.

Depth-First Search:

DFS is another way of traversing a graph in which we explore the depth of a graph first and upon reaching the end, it backtracks to the node from which it was started and then do the same with the sibling node.