When we are given a computational task, we need to ask ourselves what algorithms are available for it, and what sort of time complexities can we expect?

We’ve already seen a few techniques

1. Divide and Conquer
2. Greedy
3. Dynamic Programming

D&C & Greedy usually give fast algorithms, operating in polynomial time.

DP can be fast, but can also work when no fast algorithms are available.

**Polynomial Time -** An algorithm is said to run in polynomial time if it’s complexity can be expressed as O(n^k) for any integer k.

O(n^100) would be slow, but is still a tractable solution

**Exponential Time -** Algorithms that behave with a worst case of O(2^n), O(n^n) or O(n!) are extremely slow and considered to be intractable algorithms.

Exponential algorithms are considered intractable because they grow at a rapid rate with even small increases in N, and require an almost impossible amount of computational resources to solve.

Exponential Algorithms can arise in a few problems, for example:

* Exhaustive Enumeration
* Systematic search of the space of partial solutions using unbounded backtracking

**Example Task**

*Consider an undirected graph to be coloured using k colours, such that each node is allocated a single colour, and nodes linked by an edge have different colours.*

One way to solve this problem would be using **exhaustive enumeration.** That is, simply trying every single possible allocation of colours to each node in the graph, and checking if any solution is a valid colouring.

There are k^n allocations of k colours to the nodes,

and checking each allocation to see if the colour is valid is O(E) (where E is number of edges).

This solution would run with a worst-case time complexity of O(E \* k^n)

Another technique to solve this problem is to use **exhaustive unbounded backtracking**.

* Number the nodes 1-N
* For each node encountered, successively try each colour
* If all colours fail, undo the colour of the previous node and try all new colours on it
* If a colour succeeds, remember it and try to colour the next node.
* Repeat this until we either exhaustively fail, or a combination succeeds.

This method is also exponential in the worst-case, as there are no known algorithms to colour a graph in polynomial time.

Therefor, it goes that Graph Colouring is an example of an NP-Complete problem, that is, a problem that can only be solved in exponential time.

**Another Example**

**Travelling Salesman problem** - Find a tour of N cities in a country, where the tour visits every city just once, returns to the starting point, and is of minimum possible distance.

This is an example of an **optimization problem**, where we are not just concerned with any solution, but only the *best* solution.

As with graph colouring, this problem can be solved again with exhaustive enumeration.

We accomplish this by starting at any city, and enumerating all possible permutations of cities to visit. We then find the total distance of each permutation, and choose the one with minimum distance as the solution.

This solution once again performs in exponential time, O(N!), as there are (N-1)! permutations for N cities.

This problem can also be solved using **Dynamic Programming**, which performs faster in the average case, but also has an exponential worst case.

**Knapsack Problems**

A Knapsack problem requires us to fit a number of items of different sizes into a container of fixed size. The optimal solution should be found by minimising unused space.

Some knapsack problems can be solved in polynomial/linear time, but others only have exponential solutions.

**Job Scheduling, Rostering and Timetabling**

A collection of tasks to be completed by allocating them to abstract “slots”.

The slots can take many forms, including:

* Time periods
* Machines
* Rooms
* People’s availability

The allocation usually has some constraints.

1. Compatibility constraints - Which tasks fit in which slots
2. Scheduling constraints - Some tasks may have to be performed before others can be started
3. Resource Constraints - limited or unlimited resources?
4. Optimization Requirements - We may be asked to maximize or minimize given criteria. For example, perform the scheduling in the minimum possible time, or produce the most optimally distributed spread of tasks.

Examples of scheduling problems:

* Scheduling in multi-tasking operating systems
* Industrial Processing
* Rostering - e.g COMP23420 IBMS task
* Human Timetabling

There is a wide variety of algorithmic solutions to this problem, with some completing in linear/polynomial time, such as greedy methods or first-come-first-served allocation

Other scheduling problems require algorithms which only operate in exponential time, such as exhaustive enumeration

**Heuristics**

Rules which can be used in decision making. Intended to reach optimal solutions.

Heuristics are useful when in general the required algorithm will complete in exponential time, but corners can be cut to produce an approximated solution in polynomial time.