**Week 6 Submit Task**

**Part A**

For these questions, take the time to practise your exam writing skills.

1. Define backtracking and explain its basic operation. (3 marks)

Backtracking is an algorithm design pattern that, similar to brute force, builds up candidate solutions and abandons each partial candidate if it has been deemed that it can possibly be completed to give the correct solution. The general process is to try each different possibility and stop if a possibility is/will be unsuccessful, and then backtrack to try all other possibilities.

1. Explain the concept of recursion and its role in backtracking algorithms. (2 marks)

Backtracking can be said to be a form of recursion because the process of finding a solution from various options is repeated recursively until we either cannot proceed or reach the final state. As such, backtracking can also be thought of as a DFS across a certain search space.

1. What is the purpose of pruning in backtracking algorithms? Provide an example of how pruning can improve the efficiency of a backtracking solution. (2 marks)

Pruning in backtracking algorithms is the process of removing entire branches of the search tree once it is determined that they cannot possibly lead to a valid solution. This process helps reduce the size of the problem. For example, when trying to solve a Sudoku puzzle, there is no point in continuing to try out possibilities if it is already known that a certain row or square is invalid. As such, whenever we reach an invalid state, the algorithm can “prune” that entire branch of the search tree which reduces the amount of possible arrangments that need to be considered.

1. What are some common applications of backtracking algorithms in computer science or real-world scenarios? (3 marks)

Backtracking can commonly be used to solve puzzles, such as Sudoku, the N-Queens problem, crosswords or verbal arithmetic. This works quite well because the puzzles often involve finding a solution that meets certain constraints out of all possible arrangements, and backtracking provides a good way to systematically explore all these arrangements and abandon the ones that will not meet the constraints. Other problems that involve finding the best possible solution from a finite set of possible solutions (such as the knapsack problem or the TSP) are also well suited to backtracking as any solution that is worse than the best solution so far can easily be discarded.

1. Describe the graph colouring problem and explain how backtracking can be used to solve it. (3 marks)

The graph colouring problem asks, “given an undirected graph, what is the minimum number of colours we would need to use such no two adjacent nodes are assigned the same colour?”. A naïve approach to this problem could be to try every single possible arrangement of colours with increasing values of , the number of colours that can be used. Once a valid solution is found, the algorithm could terminate. To improve the efficiency of this approach, we could use backtracking to remove candidate solutions as they come along. As such, the algorithm would only attempt “legal moves” (no adjacent colours being the same) and abandon the current candidate if there are no legal moves available, letting us traverse the search space while trying out much fewer candidate solutions.

1. Discuss the time complexity of the graph colouring problem when solved using backtracking. (2 marks)

The time complexity of the graph colouring problem when solved using backtracking would not be any better than if it was solved with brute force, because the worst case is that all candidate solutions need to be explored and none can be “pruned”. Even though this still leaves us exploring all combinations in time, it will in general likely see better real-world performance.

**Part B**

The Subset Sum problem involves, for a given set of integers and target sum, determining whether there is a subset of the given set whose elements sum up to the target value. This can be solved with backtracking.

Complete the pseudocode below for the backtrack function. It should return the current subset if the target is found, or null if the target cannot be found. Otherwise add the next element from the set to the current subset and call recursively. If the subset is not found, the element should be excluded and the call repeated with the next element.

function subsetSum(set, targetSum):

return backtrack(set, targetSum, [], 0)

function backtrack(set, targetSum, currentSubset, currentIndex):

if set is empty:

return null

if sum(currentSubset) == targetSum:

return currentSubset

else:

currentSubset.add(set[currentIndex])

if sum(currentSubset) <= targetSum:

return backtrack(set, targetSum, currentSubset, currentIndex + 1)

else:

// we have exceeded the sum -> abandon candidate

return backtrack(set.without(currentSubset.first), targetSum, [], 0)