1. Subset Sum

a) 1. Generate all possible subsets
2. calculate the sum of each subset's elements
3. If the sum of any subsets = the target sum, return true
4. If not, then return false

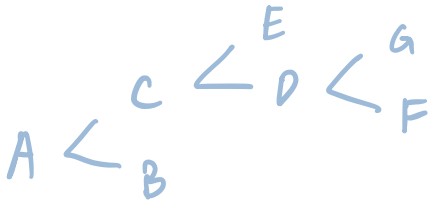
The time complexity of this algorithm is $O(2^n)$. This is because for each element in the list, there is either the choice to include it or exclude it.

b)
- For S(i), a linear sequence will be formed: S(1), S(2), ..., S(n). For S(i, j) a 2D grid will be formed: rows representing elements of the list and columns representing possible sums from 0 to the target sum.
- S(i) depends on S(i-1), S(i,j) depends on S(i-1, j) and S(i-1, j - a[i]) if $j \geq a[i] \rightarrow a[i]$ is the ith element in the list.
- 1. Initialize a 2D dynamic programming table of n+1 and k+1 dimensions, k being the target sum. All elements initially set to False
2. dp[0][0] = true because there is an empty subset that sums up to 0.
3. For i from 1 to n:
   - For j from 0 to k:
     - dp[i][j] = dp[i-1][j]
     - If $j \geq a[i]$ then dp[i][j] = dp[i][j] or dp[i-1][j - a[i]]
4. the final answer is dp[n][k]
- The time complexity of this algorithm is O(nk), n being the number of elements in the input list, and k being the target sum. they are multiplied because we fill the 2D table using a nested loop.

c) true. This is because the brute force algorithm has a time complexity of $O(2^n)$, which becomes impractical as the input size increases. The DP based algorithm has a time complexity of O(nk) which is much more manageable as well as efficient for larger inputs. It exploits the overlapping subproblems which reduces time complexity significantly compared to the brute force approach.

2. Vertex Cover

a) The tree may look like this:



The size of the smallest vertex cover is 3. One potential smallest vertex cover is {C, D, F}.

b)
- The subproblem can be defined as S(v, choice): the size of the minimum vertex cover of the subtree rooted at vertex v, choice either including or excluding vertex v
- The dependancy relationship can be summarized as:
  - S(v, 0): not including vertex v in the cover. The cover must include the children of v to cover their adjacent edges. Depends on S(u, 0) and S(u, 1) for each child u of vertex v.
  - S(v, 1): including vertex v in the cover. The cover can exclude the children of v. Depends on S(u, 0) for each child u of vertex v.
- The algorithm to compute solutions to the subproblems:
  1. Initialize a table 'dp' to store the subproblem solutions and initialize it to a value that indicates no solution has been computed yet
  2. Perform a DFS from any node as the root
  3. For each vertex encountered during the DFS:
     - If dp[v][0] and dp[v][1] are not yet computed recursively calculate the values:
       - dp[v][0]: the minimum vertex cover size when vertex v is not included in the cover. For each child u of v add the minimum of dp[u][0] and dp[u][1] to dp[v][0].
       - dp[v][1]: the minimum vertex cover size when vertex v is included in the cover. For each child u of v add dp[u][0] to dp[v][1].
  4. The minimum vertex cover size for the entire tree after all of the subproblems are solved is the minimum of dp[root][0] and dp[root][1].
- The time complexity of this algorithm is O(n) where n is the number of vertices in the tree. This is due to the DFS traversal which computes the solution for each vertex once resulting in a linear time complexity.

3. Document reconstruction

a) "biggreyelephant": "big" "grey" "elephant" ✓

"nopainnogain": "no" "pain" "no" "gain" ✓

"commondayloff": "common" "day" "lot" "f" ✗

b)
- S(i): True if s[1...i] has a valid reconstruction. I choose this because it encapsulates the idea of checking the validity of various splits of the various splits of the substring and if any of those splits lead to a valid reconstruction.
- The dependancy relationship can be summarized as f(i) depends on S(j) where $0 \leq j < i$ and whether the substring s[j+1...i] is a valid word for each j.
- The algorithm can be represented as:
  1. Initialize a table 'dp' which has size (n+1) and all entries set to False
  2. Set dp[0] to true
  3. For i from 1 to n:
     - For j from 0 to i-1:
       - If dp[j] is true and D(s[j+1...i]) is True (a valid word) then set dp[i] to True.
  4. The final answer is dp[n]
- The time complexity of the algorithm is $O(n^2)$, where n is the length of the input string. This is because at each position of i, we iterate through all possible positions j before i to check the validity of the substring. The constant time lookup is D(s[j+1...i]), so the overall time complexity is quadratic

4. Contigious subsequence

1. Initialize variables max end, maxprogress, start, end, and s all to 0
2. Iterate over each element in the input array arr[i]:
   - add arr[i] to maxend
   - if maxend becomes negative, reset it to 0 and update s to i+1
   - if maxend is greater than maxprogress:
     - update maxprogress with maxend
     - update start with s
     - update end with i
3. The maximum contigious subsequence is inclusive from the indices start to end
4. Return the matimum contigious subsequence

This algorithm iterates through the array linearly, maintaining the maximum sum contigious subsequence. It effectively identifies the subsequence with the maximum sum while taking into account the possibility of negative values in the array. This would give the algorithm a runtime of O(n) where n is the number of elements in the input array.

5. Longest common subsequence

1. Initialize a table 'dp' which has dimensions (n+1)(m+1) to store the lengths of the longest common subsequences between different prefixes of the input strings
2. Initialize the first row and column of dp with zeros (the longest common subsequence with an empty string is 0)
3. Iterate over the characters of both strings using nested loops:
   - for each pair of characters x[i-1] and y[i-1]:
     - If the characters are equal, update dp[i][j] by adding 1 to dp[i-1][j-1].
     - otherwise update dp[i][j] to the maximum of dp[i-1][j] and dp[i][j-1]
4. The value at dp[n][m] contains the length of the longest common subsequence

The time complexity of the algorithm is O(mn) where n is the length of string x and m is the length of string y. This is because this algorithm iterates over each character of both strings exactly once and perform constant time operations for each pair of characters.

6. Spanning intervals

1. Sort intervals based on end points in ascending order
2. Initialize a variable 'end' to store the ending point of the current selected, and Initialize an array 'selected' to store the selected intervals
3. Iterate through the sorted intervals:
   - If the current interval's starting point is greater than or equal to end, add the current interval to the selected array and update end to the current intervals end point
4. Now the selected array contains the minimum number of intervals needed to cover the union of all intervals

The time complexity of this algorithm is $O(n \log n)$ time, which is dominated by the sorting step. n represents the number of intervals, the subsequent iteration over the sorted intervals takes linear time.