SC1007 Data Structures and Algorithms

Solution 6: DFS,Backtracking and DP

Q1 Give a pseudocode of finding a simple path connecting two given vertices in an undirected graph by Depth-First-Search.

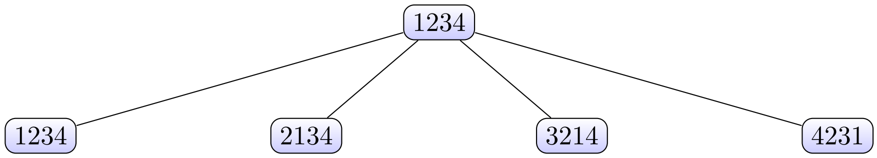
Algorithm 1 Depth First Search (DFS)

S1

```
function SimplePath(Graph G, Vertex v,Vertex w)
    create a Stack, S
    push v into S
    mark v as visited
    while S is not empty do
        peek the stack and denote the vertex as x
        if x == w then
            while S is not empty do
                pop a vertex from S
                peek the stack
                print the link
            end while
            return Found
        end if
        if no unvisited vertices are adjacent to x then
            pop a vertex from S
        else
            push an unvisited vertex u adjacent to x
            mark u as visited
        end if
    end while
    return Not Found
end function
```

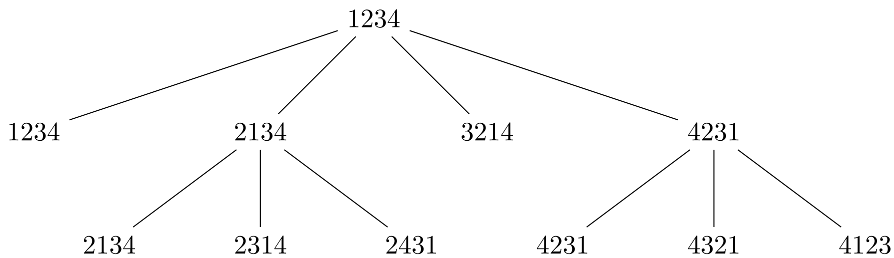
Q2 Give a pseudocode of a backtracking algorithm to print out all possible permutation of a given sequence. For example, input is given as "1234". The 24 output permutations are printed out from "1234" to " 4321".

S2 To systematically print out all the permutations, we first swap the first element with each other

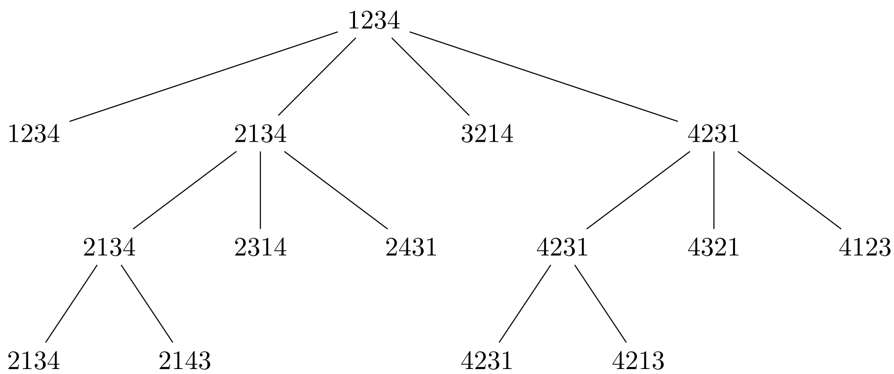


element.

Next we need to swap the second element with each other element (except the element in the first position). Here we only case the second and the fourth cases.



Next we need to swap the third element with the following element (in the example, we only leave the last element to swap).



You can observe that printing out all the permutation we need to iterative swap one element with each other element and recursively do so on its smaller sequence (reduce by one element) until we reach the last element. The pseudocode is as following:

Algorithm 2 Backtracking algorithm for Permutation

```
function PERMUTATION(char[]seq, sInx, eIdx)
    if sInx == eIdx then
        print seq
    else
        for i ← sInx to eInx do
            swap the sInx^th character and the i^th character in seq
            Permutation(seq,sInx+1,eIdx)
            swap the sInx^th character and the i^th character in seq        ▷ backtracking
        end for
    end if
end function
```

Q3 Find length of longest substring of a given string of digits, such that sum of digits in the first half and second half of the substring is same. For example, if the input string is "142124", the whole string is the answer. The sum of the first 3 digits = the sum of the last 3 digits (1+4+2) = 1+2+4. Thus, the length is 6. If the input is "12345678", then the output is 0. If the input is "9430723", then the output is 4 (4307).

S3 Here two possible approaches to be discussed here. The first one is a brute force solution. First of all, the length of the substrings must be even number. The brute force approach will check all the substrings of even length.

Algorithm 3 The Brute Force Solution

```
function MaxSubString(char[]seq)
    maxLen ← 0
    for i ← 0 to length of seq do
        for j ← i+1 to length of seq step 2 do
            len ← length of the substring between indices i and j
            if maxLen >= len then                ▷ maxLen > length of substring, do nothing
                continue
            end if
            for k ← 0 to len/2 do
                lSum to sum of digits in the first half
                rSum to sum of digits in the second half
            end for
            if lSum == rSum then
                maxLen ← len
            end if
        end for
    end for
    return maxLen
end function
```

The time complexity of this brute force approach is $\mathcal{O}(n^3)$. If you observe the algorithm carefully, you can see that many substrings can be overlapping.

If we build a 2-D table that stores sum of substrings, then the time complexity can be improved.

Let $sum[i][j]$ be the sum of digits from i to j and assume that the matrix has been initialized and the lower triangular of the matrix will not be used (when $i > j$)

Algorithm 4 The DP Solution

```
function MaxSubStringDP(char[]seq)
    maxLen ← 0
    for len = 2 to n do
        for i = 0 to n − len + 1 do        ▷ pick i and j to make the length of substring be len
            j ← i + len − 1
            k ← ⌊len/2⌋
            sum[i][j] ← sum[i][j − k] + sum[j − k + 1][j]        ▷ calculate sum[i][j] from table
            if len mod 2 == 0 and sum[i][j − k] == sum[j − k + 1][j] and len > maxLen then
                maxLen ← len        ▷ Update maxLen
            end if
        end for
    end for
    return maxLen
end function
```

In the dynamic programming approach, the time complexity will be $\mathcal{O}(n^2)$ but additional $\mathcal{O}(n^2)$ space is required.