Lecture:DP - 3

Topics Covered:

1. Regular Expression Match
2. Coin Sum Infinite
3. Substring Palindrome Queries
4. Palindrome Partitioning

Detailed Descriptions:

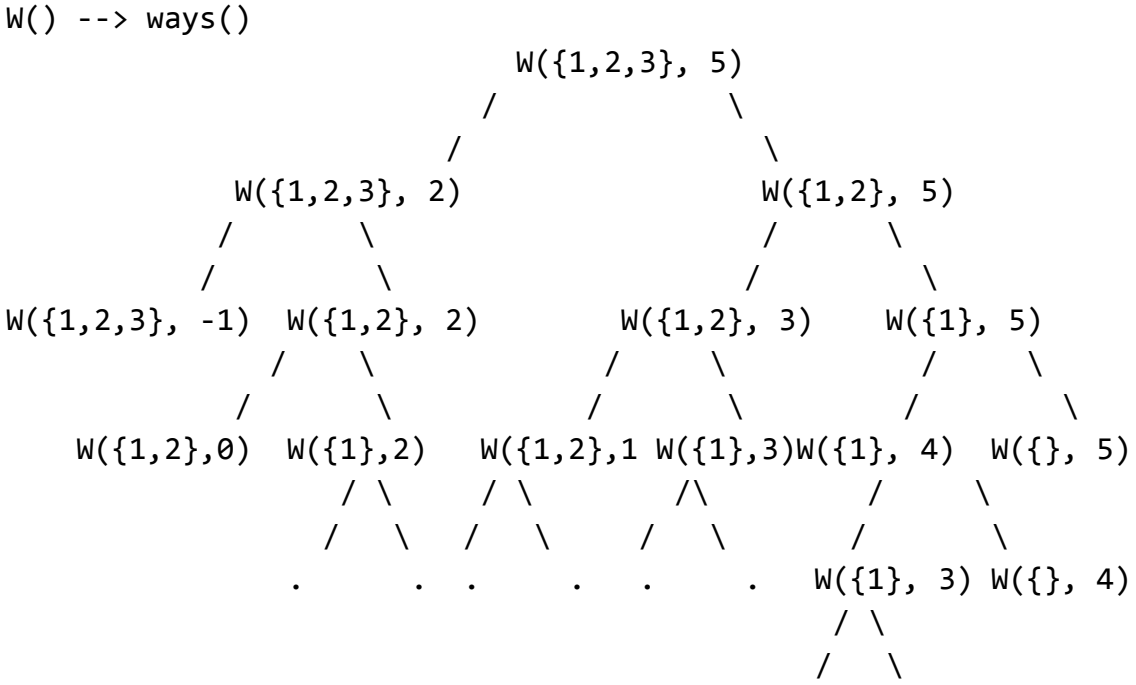
- Coin Sum Infinite:
  - You are given a set of coins `A`. In how many ways can you make sum `N` assuming you have infinite amounts of each coin in the set.
  - Coins in set `A` will be unique.
  - ```
    Input :
          A = [1, 2, 3]
          N = 4

    Output: 4

    Explanation : The 4 possible ways are
    {1, 1, 1, 1}
    {1, 1, 2}
    {2, 2}
    {1, 3}
    ```
  - There are two variants of the question :
    1. If the order of the coin does not matter .For above input we have given the possible ways in the explanation.
    2. If the order is important then the possible way will be:
       ```
       {1,1,1,1}
       {1,1,2}
       {1,2,1}
       {2,1,1}
       {2,2}
       {1,3}
       {3,1}
       ```
       ```

Output: 7

- Build Observation for DP Approach :
    1. To count the total number of ways, we can divide all set solutions into two sets.
        a. Solutions that do not contain `m'th` coin
        b. Solutions that contain at least one `m'th.`
    2. Let `way[N]` be the number of ways to construct a sum of `N` using the coins.
    3. Suppose we include the `m'th one time,` then we need to construct a sum of `N-m` values using the coins.
       `ways[N] = ways[N-A[m]];`
    4. For all the coins in the set the equation will look like this
       ways[N] = ∑(i = 0 to |A|-1) ways[N - A[i]]
    5. The tree of all the possible ways will look like this:



    6. Since the same suproblems are called again, this problem has Overlapping Subprolems property and also has optimal substructure. So the Coin Change problem can be solved using DP.
    7. The above tree is created when order is important as the resulting output will be 7.In the above tree we are considering all the possible permutations .

8. So the above discussed recursive equation will solve the problem for case 2.
9. Code for Case 2:

```
ways[0]=1 // base condition
for(i=1;i<=N;++i){//find all ways from 1 to N
      for(j=0;j<A.length;++j){//try all the coins
whose value is less than sum
            if(A[j]<=i)ways[i]+=ways[i-A[j]];
      }
}
```

10. Time Complexity of above code is O(N*|A|) , |A| is the size of the coin set and N is the required sum
11. Space Complexity O(N)

- For case 1:
  1. We need to avoid the repetition of sets in the answer set.For this we need to fix the order in which we select coins
  2. We can solve this by simply swapping the both loops in the above code.
  3. We fix `i'th` coin and try to use it all the sum range, then we go to the next coin and try the same for it, this will lead to avoiding repetition .
  4. For example we have a set {x,y,z} and a required sum N.We know that N can be constructed using 2 `x` coins and 1 `y` coin.
  5. In the code below it will first use x coin for N and then only it will use y coins the order will be like `{x,x,y}` to construct `N`. No other arrangement of `{x,x,y}` will be generated because we aren't going to use the x coin again nor the `y`.
  6. Code for Case 1:

```
// Base case (If given value is 0)
 way[0] = 1;
 // Pick all coins one by one and update the table[]
values
 // after the index greater than or equal to the value
of the
 // picked coin
 for(int i=0; i<m; i++)
    for(int j=A[i]; j<=n; j++)
        way[j] += way[j-A[i]];
```

7. Dry Run for above code:
    a. `N = 5 , A = [1,3,5], way = [0,0,0,0,0,0]`
    b. Performing the base case by initialising `0'th` index with 1.
    c. Now we fix the first coin valued 1 and try to add it from 1 to N.
    d. Starting from 1 , since `way[1] = way[1-1]=way[0]=1`
    e. Similarly for `way[2] = way[2-1] = 1`
    f. Similarly for 3,4,and 5,the way array will look like
    `way = [1,1,1,1,1,1]`
    g. Now we fix the second coin valued 3 and try to add it from 3 to N.
    h. Staring from 3, `way[3] += way[3-3] += way[0] += 1.`
    `way = [1,1,1,1,2,1,1]`
    i. For 4 , `way[4] += way[4-3] +=way[1] += 1.`
    `way = [1,1,1,1,2,2,1]`
    j. For 5 , `way[5] += way[5-3] +=way[2] += 2`
    `way = [1,1,1,1,2,2,2]`
    k. Now we fix the third coin valued 5 and try to add it from 5 to N.
    l. For 5, `way[5] += way[5-5] += way[0] +=1`
    `way = [1,1,1,1,2,2,3]`
    m. Hence the total number of ways to construct a sum of 5 using the above set of coins is 3 in case 1.
- We can also solve this problem using 2D DP approach the states would have been
  `way[i][j]` = number of ways we can construct `j` sum using first `i` coins.
- But it's necessary to use 2D DP approach as it requires more space.

- Substring Palindrome Queries:
    - Given a string `S , ∀(i,j)` check if `S[i to j]` is a palindrome or not.
    - Output a boolean 2D matrix with row `i` and column `j` defines if the substring `S[i to j]` is palindrome or not.
    - Output "_" if the substring is not possible since `i>j`.
    - Input : S = `"bbcb"`

Output: {{true,true,false,false},
{_,true,false,true},
{_,_,true,false},
{_,_,_,true}}

- Build Observation:
  1. Let's first consider smaller substrings.
  2. Substrings of size 1 will always be a palindrome.As we know every string with length 1 is a palindrome.
  3. Substrings of size 2 will be a palindrome if the first and the second character match with each other.
  4. Substrings of size 3 will be a palindrome if the first and the third character match with each other.
  5. Substrings of size 4 will be a palindrome if the first and the last character is equal and the string from second to third index is already a palindrome. Eg : `"abba"`.
  6. Substrings of size 5 will be a palindrome if the first and the last index characters are equal and the string from second to fourth character is already a palindrome. Eg:`"abcba"`.
  7. We see a general pattern here and we can make a general equation here.
  8. Let `isP[i][j]` be true if substring`(i,j)` is a palindrome else it will be false.
  9. For substring size 1, `isP[i][j] = true.`
  10. For substring size 2,

```
if(s[i]==s[j])
    isP[i][j] = true
else
    isP[i][j] = false.
```

  11. For substring size `N` , if the first and last character are equal we check if the substring of size `(N-2)` inside the current substring is a palindrome or not.

```
if(s[i]==s[j])
 //check if the substring from second index to
 //penultimate index is palindrome or not
     isP[i][j] = isP[i+1][j-1]
else
     isP[i][j] = false.
```

12. The basic idea behind the general equation is that if a string is a palindrome then after adding equal elements at both ends will always result in a palindromic string.

- DP Approach:
    1. In most of the previous DP questions solved we required the data from above, current or diagonal row.
    2. But in this problem according to the recursive equation to get `isP[i][j]`, we require data from the below row i.e. `isP[i+1][j-1]` i.e. diagonally bottom left cell.
    3. To tackle this problem we will not traverse the matrix from the top of the matrix to the bottom of it.
    4. Instead we will first fill substring of size 1, then 2 and then 3 and so on to N i.e. we traverse the matrix diagonally and first fill diagonal of size N , then (N-1) ,then (N-2) and so on to 1.
- Code:

```
// Get the size of the string
    int n = s.size();

    // Initially mark every
    // position as false
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            isP[i][j] = false;
    }

    // For the length
    for (int j = 1; j <= n; j++) {

        // Iterate for every index with
        // length j
        for (int i = 0; i <= n - j; i++) {

            // If the length is less than 2
            if (j <= 2) {

                // If characters are equal
                if (s[i] == s[i + j - 1])
                    isP[i][i + j - 1] = true;
            }
```

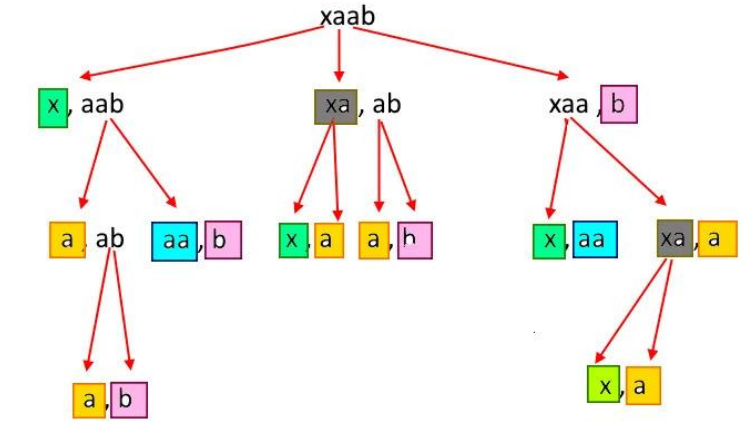
```
                // Check for equal
                else if (s[i] == s[i + j - 1])
                    isP[i][i + j - 1] = dp[i + 1][i + j - 2];
            }
        }
```

- Time Complexity O(N^2) where N is the size of the string
- Space Complexity O(N^2)

- Palindrome Partitioning:
  - Given a string, a partitioning of the string is a *palindrome partitioning* if every substring of the partition is a palindrome.
  - For example, `"aba|b|bbabb|a|b|aba"` is a palindrome partitioning of `"ababbbabbababa"`.
  - Determine the fewest cuts needed for a palindrome partitioning of a given string.
  - For example, minimum of 3 cuts are needed for `"ababbbabbababa"`. The three cuts are `"a|babbbab|b|ababa"`.
  - If a string is a palindrome, then minimum 0 cuts are needed. If a string of length n contains all different characters, then minimum n-1 cuts are needed.
  - Input : `str = "aaaa"`
    Output : 0
    The string is already a palindrome.
  - Input : `str = "abcde"`
    Output : 4
  - Input : `str = "abbac"`
    Output : 1

  - Recursive approach:
    1. In recursion we can try all the possible combinations of cuts and check which combination gives minimum cuts.
    2. Also we will the tree , before making any cut we will check if the cut introduces a palindrome or not.
    3. If a new palindrome is constructed after performing the cut then we will recurse further,else we will avoid the cut at that position.

- Greedy approach:
    1. For example the string `S = "aabbb"`, we start from end and find the largest palindrome from the end.
    2. In the above string the largest palindrome substring from end is `"bbb"`, we make a cut there and find the next largest palindrome from the (N-3) here.
    3. So the next largest palindrome from the made cut is `"aa"` .
    4. So we required 1 cut `"aa|bbb"` here which is optimal.
    5. But this will not always work.Lets see an example.
    6. `S = "baabbb"`, for this string the cuts made according to greedy approach would be `S = "b|aa|bbb"` but the optimal cuts would have been `S = "baab|bb"`.
    7. So a greedy approach is not optimal.
- DP approach:
    1. We can see in the recursion tree that there are overlapping subproblems and we can use DP to store such problems result to use it again.



Recursion will stop once it finds that substring is palindrome

Sub problems are solved repeatedly. a solved 6 times, x solved 4 times, aa solved 2 times etc.

    2. Let's define the DP state,
    `cut[i]` → min cuts in `S[0-i]` required to make every substring of the partition a palindrome.
    3. There will be two situations where `cut[i]` will be 0.
        a. When the string is empty i.e. `cut[0] = 0`.
        b. When the substring starting from `0 to i` is a palindrome then the required cut is 0.

```
if(isP[0][i]==true)
    cut[i] = 0
```

4. Also since we require to know that if a substring(i,j) is palindrome or not , we will make use of the solution in the above question.
5. Before performing the DP approach we will do preprocess and will create a 2D matrix `isP[][]` using the above method which will store if the substring`(i,j)` is palindrome or not.
6. If we are at index i and we found a palindrome starting from i and having a length of 2 then `cut[i] = 1+cut[i-2].`
7. Similarly for palindromes of size 3 it will be `cut[i] = 1+cut[i-3]`.
8. So in general we will make a cut whenever we find a palindrome substring starting from i and also add the required cuts after the palindrome.

- Code:

```
int N;//size of the string;
string S;//input string
int cut[N];
int isP[N][N];

for(int i=0;i<N;++i)
     cut[i] = INT_MAX; //initialize cut array with INT_MAX
//copy paste code from above solution for preprocessing
for(int i=0;i<N;++i){
     if(isP[0][i])//base case
          cut[i]=0;
     for(int j=0;j<i;++j){
          if(isP[j+1][i])
               cut[i] = min(cut[i],1+cut[j]);
     }
}
return cut[N];
```

- Time Complexity O(N^2)
- Space Complexity O(N^2) considering isP matrix else O(N).