Name:
Advanced Programming in C++
Lab Exercise 5/6/2025

In this lab, we will look at three classic recursive Computer Science algorithms; Tower of Hanoi, Knapsack problem, and Permutation of Passwords.

Tower of Hanoi

The **Tower of Hanoi** (also called the **Tower of Brahma** or **Lucas' Tower**and sometimes pluralized) is a mathematical game or puzzle. It consists of three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No disk may be placed on top of a smaller disk.

With 3 disks, the puzzle can be solved in 7 moves. The minimal number of moves required to solve a Tower of Hanoi puzzle is $2^n - 1$, where n is the number of disks.

Here is a solution to get you started:

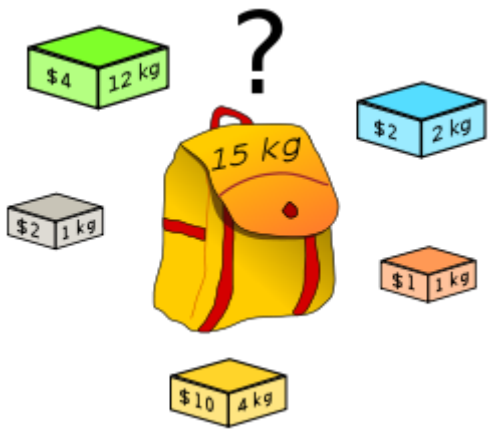
```
#include <iostream>
using namespace std;

void hanoi(int, int, int, int);

int main()
{
hanoi(4, 1, 2, 3);
   return 0;
}

//Recursive function that solves Tower of Hanoi problem in 2^depth - 1 moves
void hanoi(int depth, int from, int to, int alternate)
{
if(depth == 0)
   {
return;
   }
hanoi(depth-1, from, alternate, to);
cout<< "Move ring " << depth << " from pole " << from << " to pole " << to <<endl;
hanoi(depth-1, alternate, to, from);
}
```

Your task is to modify the above program so that it reports the number of moves it takes. With large numbers of disks, the time to solve the puzzle is made significantly longer by displaying the result of each move.

For example, your output might look like:

With 15 disks, the problem solved in 32767 moves.

The Knapsack Problem



The **knapsack problem** or **rucksack problem** is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

The problem often arises in resource allocation where there are financial constraints and is studied in fields such as combinatorics, computer science, complexity theory, cryptography, applied mathematics, and daily fantasy sports.

Here is an implementation of the algorithm. I have added test code to test the function.

```
#include<iostream>

using namespace std;

const int SIZE = 3;

int max(int, int);
intknapsack(int, int [], int [], int);

// Driver program to test function
int main()
{
//Test Code
   int val[] = { 60, 100, 120 };
   int wt[] = { 10, 20, 30 };
   int W = 50;

cout<< "The value of the material in the knapsack is $"<<knapsack(W, wt, val, SIZE) <<endl;

   return 0;
}

// A utility function that returns maximum of two integers
int max(int a, int b)
{
   return (a > b) ? a : b;
}
```

```
// Returns the maximum value that can be put in a knapsack of capacity W
intknapsack(int W, int wt[], int val[], int n)
{
   // Base Case
   if (n == 0 || W == 0)
      return 0;

   // If weight of the nth item is more than Knapsack capacity W, then
   // this item cannot be included in the optimal solution
   if (wt[n - 1] > W)
      returnknapsack(W, wt, val, n - 1);

   // Return the maximum of two cases: (1) nth item included (2) not included
   else
      returnmax(val[n - 1] + knapsack(W - wt[n - 1], wt, val, n - 1),
knapsack(W, wt, val, n - 1));
}

//Output
//The value of the material in the knapsack is $220

//TO DO:
//Get rid of the test code and add code to allow the user to enter values as shown below
//Sample output
//Enter value and weight for item 1:4 6
//Enter value and weight for item 2:5 4
//Enter value and weight for item 3:13 5
//Enter the capacity of knapsack? 10
//The value of the material in the knapsack is $18
```

Permutation Of Passwords

In this lab, we will look at a method of finding all permutations of a world. As in yesterday's examples, these algorithms are recursive in nature. Here is some code for a function that will generate all permutations of a word with a given size.

```
void permute(char *a, int k, int size)
{
   if (k == size)
   {
      for (int i = 0; i< size; i++)
      {
cout<< *(a + i);
      }
cout<<endl;
   }
   else
   {
      for (int i = k; i< size; i++)
      {
         int temp = a[k];
         a[k] = a[i];
         a[i] = temp;

permute(a, k + 1, size);

         temp = a[k];
         a[k] = a[i];
         a[i] = temp;
      }
   }
}
```

This function requires 3 parameters; a C-style string, the starting point (normally 0), and the size of the string.

1. Write a main function that can use this permute function.
2. Modify the permute function to display the number a ways a word can be permuted.

When you have completed these three programs, submit your source code, a screenshot of the program running, and attach to this sheet to turn in.