*#include* <iostream>

*using* *namespace* std;

*void* *fibIter*(*int* n){

*if*(n *<=* 0){

        cout *<<* "Please enter a number greater than 0" *<<* endl;;

*return*;

    }

    cout *<<* "Fibonacci sequence: " *<<* endl;

*if*(n *==* 1){

        cout *<<* 1 *<<* endl;

*return*;

    }

*int* fn *=* 1;

*int* fn1 *=* 0;

*int* fn2 *=* 1;

*for*(*int* i *=* 1; i *<=* n; i*++*){

        cout *<<* fn1 *<<* " ";

        fn *=* fn1 *+* fn2;

        fn2 *=* fn1;

        fn1 *=* fn;

    }

    cout *<<* endl;

}

*int* *fibRec*(*int* n){

*if*(n *<=* 0){

*return* 0;

    }

*if*(n *==* 1){

*return* 1;

    }

*return* *fibRec*(n *-* 1) *+* *fibRec*(n *-* 2);

}

*int* *main*(){

*int* n;

    cin *>>* n;

*fibIter*(n);

*for*(*int* i *=* 0; i *<* n; i*++*){

        cout *<<* *fibRec*(i) *<<* " ";

    }

*return* 0;

}



*# A Huffman Tree Node*

*import* heapq

*class* node:

*def* \_\_init\_\_(self, freq, symbol, left*=*None, right*=*None):

*# frequency of symbol*

*self*.freq *=* freq

*# symbol name (character)*

*self*.symbol *=* symbol

*# node left of current node*

*self*.left *=* left

*# node right of current node*

*self*.right *=* right

*# tree direction (0/1)*

*self*.huff *=* ''

*def* \_\_lt\_\_(self, nxt):

*return* *self*.freq *<* nxt.freq

*# utility function to print huffman*

*# codes for all symbols in the newly*

*# created Huffman tree*

*def* *printNodes*(node, val*=*''):

*# huffman code for current node*

    newVal *=* val *+* str(node.huff)

*# if node is not an edge node*

*# then traverse inside it*

*if*(node.left):

        printNodes(node.left, newVal)

*if*(node.right):

        printNodes(node.right, newVal)

*# if node is edge node then*

*# display its huffman code*

*if*(*not* node.left *and* *not* node.right):

        print(*f*"{node.symbol} -> {newVal}")

*# characters for huffman tree*

chars *=* ['a', 'b', 'c', 'd', 'e', 'f']

*# frequency of characters*

freq *=* [ 5, 9, 12, 13, 16, 45]

*# list containing unused nodes*

nodes *=* []

*# converting characters and frequencies*

*# into huffman tree nodes*

*for* x *in* range(len(chars)):

    heapq.heappush(nodes, node(freq[x], chars[x]))

*while* len(nodes) *>* 1:

*# sort all the nodes in ascending order*

*# based on their frequency*

    left *=* heapq.heappop(nodes)

    right *=* heapq.heappop(nodes)

*# assign directional value to these nodes*

    left.huff *=* 0

    right.huff *=* 1

*# combine the 2 smallest nodes to create*

*# new node as their parent*

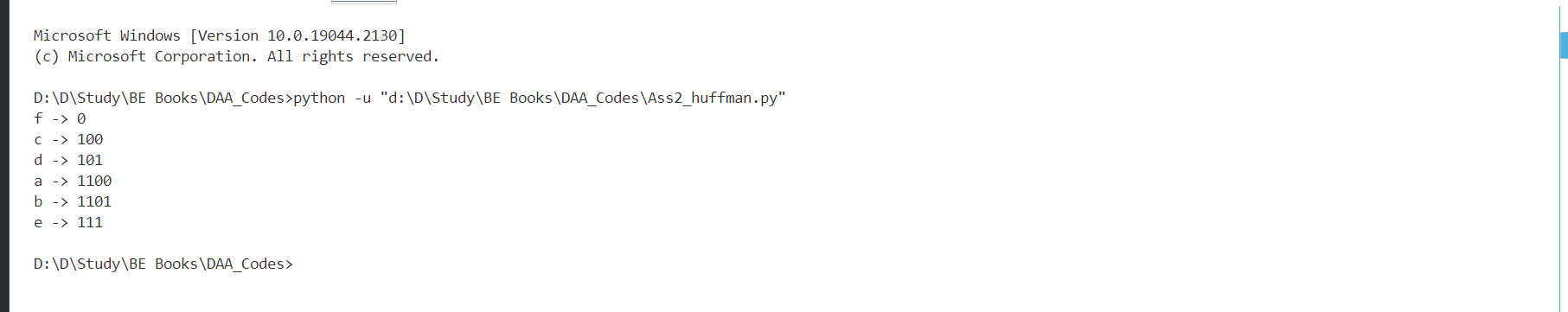
    newNode *=* node(left.freq*+*right.freq, left.symbol*+*right.symbol, left, right)

    heapq.heappush(nodes, newNode)

*# Huffman Tree is ready!*

printNodes(nodes[0])

Output:



*#include* <bits/stdc++.h>

*using* *namespace* std;

*struct* Item

{

*int* *value*;

*int* *weight*;

};

*bool* *static* *comp*(Item a, Item b)

{

*int* *r1* *=* (*int*)a.*value* */* (*int*)a.*weight*;

*int* *r2* *=* (*int*)b.*value* */* (*int*)b.*weight*;

*return* *r1* *>* *r2*;

}

*int* *fractionalKnapsack*(*int* W, Item arr[], *int* n)

{

*sort*(arr, arr *+* n, *comp*);

*int* *curWeight* *=* 0;

*int* *finalvalue* *=* 0.0;

*for* (*int* *i* *=* 0; *i* *<* n; *i++*)

    {

*if* (*curWeight* *+* arr[*i*].*weight* *<=* W)

        {

*curWeight* *+=* arr[*i*].*weight*;

*finalvalue* *+=* arr[*i*].*value*;

        }

*else*

        {

*int* *remain* *=* W *-* *curWeight*;

*finalvalue* *+=* (arr[*i*].*value* */* arr[*i*].*weight*) *\** *remain*;

*break*;

        }

    }

*return* *finalvalue*;

}

*int* *main*()

{

*int* *n* *=* 3, *weight* *=* 50;

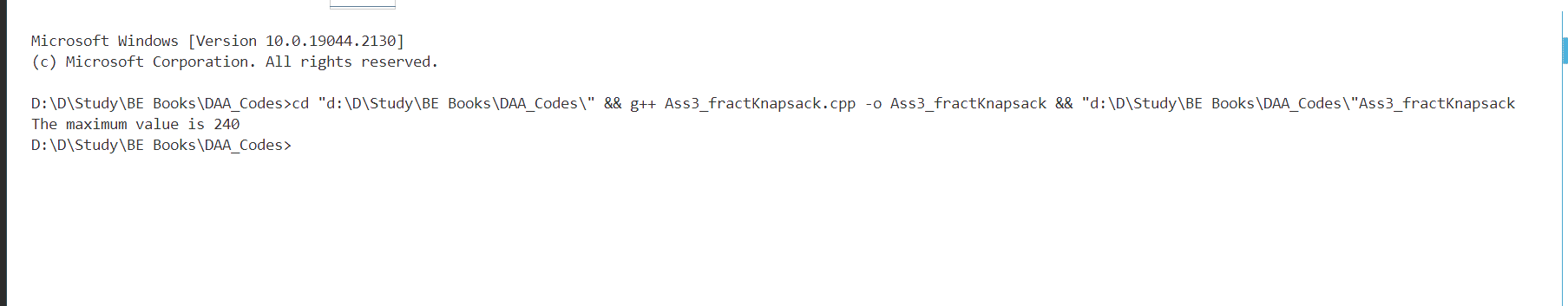
    Item *arr*[*n*] *=* {{100, 20}, {60, 10}, {120, 30}};

*int* *ans* *=* *fractionalKnapsack*(*weight*, *arr*, *n*);

*cout* *<<* "The maximum value is "  *<<* *fixed* *<<* *ans*;

*return* 0;

}



*#include* <bits/stdc++.h>

*using* *namespace* std;

*int* *knapsackUtil*(vector<*int*> *&*wt, vector<*int*> *&*val, *int* ind, *int* W, vector<vector<*int*>> *&*dp)

{

*if* (ind *==* 0)

    {

*if* (wt*[*0*]* *<=* W)

*return* val*[*0*]*;

*else*

*return* 0;

    }

*if* (dp*[*ind*][*W*]* *!=* *-*1)

*return* dp*[*ind*][*W*]*;

*int* *notTaken* *=* 0 *+* *knapsackUtil*(wt, val, ind *-* 1, W, dp);

*int* *taken* *=* *INT\_MIN*;

*if* (wt*[*ind*]* *<=* W)

*taken* *=* val*[*ind*]* *+* *knapsackUtil*(wt, val, ind *-* 1, W *-* wt*[*ind*]*, dp);

*return* dp*[*ind*][*W*]* *=* *max*(*notTaken*, *taken*);

}

*int* *main*()

{

    vector*<int>* *wt* *=* {1, 2, 4, 5};

    vector*<int>* *val* *=* {5, 4, 8, 6};

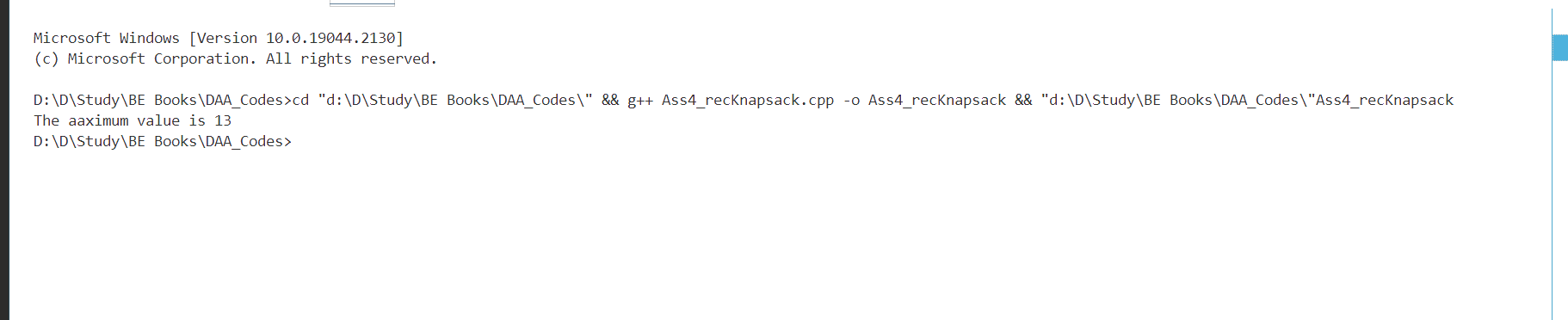
*int* *W* *=* 5;

*int* *n* *=* *wt*.*size*();

    vector*<*vector*<int>>* *dp*(*n*, *vector*<*int*>(*W* *+* 1, *-*1));

*cout* *<<* "The aaximum value is " *<<* *knapsackUtil*(*wt*, *val*, *n* *-* 1, *W*, *dp*);

}



*class* NQBacktracking:

*def* \_\_init\_\_(self, x\_, y\_):

        """self.ld is an array where its indices indicate row-col+N-1

        (N-1) is for shifting the difference to store negative indices"""

*self*.ld *=* [0] *\** 30

        """ self.rd is an array where its indices indicate row+col and used

        to check whether a queen can be placed on right diagonal or not"""

*self*.rd *=* [0] *\** 30

        """column array where its indices indicates column and

        used to check whether a queen can be placed in that row or not"""

*self*.cl *=* [0] *\** 30

        """Initial position of 1st queen"""

*self*.x *=* x\_

*self*.y *=* y\_

*def* *printSolution*(self, board):

        """A utility function to print solution"""

        print(

            "N Queen Backtracking Solution:\nGiven initial position of 1st queen at row:",

*self*.x,

            "column:",

*self*.y,

            "\n",

        )

*for* line *in* board:

            print(" ".join(map(str, line)))

*def* *solveNQUtil*(self, board, col):

        """A recursive utility function to solve N

        Queen problem"""

*# base case: If all queens are placed then return True*

*if* col *>=* N:

*return* True

*# Overlook the column where 1st queen is placed*

*if* col *==* *self*.y:

*return* *self*.solveNQUtil(board, col *+* 1)

*for* i *in* range(N):

*# Overlook the row where 1st queen is placed*

*if* i *==* *self*.x:

*continue*

*# Consider this column and try placing*

*# this queen in all rows one by one*

*# Check if the queen can be placed on board[i][col]*

*# A check if a queen can be placed on board[row][col].*

*# We just need to check self.ld[row-col+n-1] and self.rd[row+coln]*

*# where self.ld and self.rd are for left and right diagonal respectively*

*if* (*self*.ld[i *-* col *+* N *-* 1] *!=* 1 *and* *self*.rd[i *+* col] *!=* 1) *and* *self*.cl[

                i

            ] *!=* 1:

*# lace this queen in board[i][col]*

                board[i][col] *=* 1

*self*.ld[i *-* col *+* N *-* 1] *=* *self*.rd[i *+* col] *=* *self*.cl[i] *=* 1

*# recur to place rest of the queens*

*if* *self*.solveNQUtil(board, col *+* 1):

*return* True

*# If placing queen in board[i][col]*

*# doesn't lead to a solution,*

*# then remove queen from board[i][col]*

                board[i][col] *=* 0  *# BACKTRACK*

*self*.ld[i *-* col *+* N *-* 1] *=* *self*.rd[i *+* col] *=* *self*.cl[i] *=* 0

*return* False

*def* *solveNQ*(self):

        board *=* [[0 *for* \_ *in* range(N)] *for* \_ *in* range(N)]

        board[*self*.x][*self*.y] *=* 1

*self*.ld[*self*.x *-* *self*.y *+* N *-* 1] *=* *self*.rd[*self*.x *+* *self*.y] *=* *self*.cl[

*self*.x

        ] *=* 1

*if* *not* *self*.solveNQUtil(board, 0):

            print("Solution does not exist")

*return* False

*self*.printSolution(board)

*return* True

*if* \_\_name\_\_ *==* "\_\_main\_\_":

    N *=* 8

    x, y *=* 3, 2

    NQBt *=* NQBacktracking(x, y)

    NQBt.solveNQ()

