



Ain Shams University
Faculty of Engineering
Computer and Systems Engineer

Design and Analysis of Algorithms Project




Course Code: CSE332s

*Course Name: Design and Analysis of
Algorithms*

Omar Mohamed Diaaeldin Ibrahim

1802932



Contents

Task 1	1
Algorithm Method Applied:	1
Detailed Assumptions:	1
Problem Description:	2
Detailed Solution including code and the description of the steps of your solution:	2
Complexity Analysis for the algorithm:.....	4
Sample Output of the solution:	6
A comparison between your algorithm and at least one other:	7
Minimum Number of moves to invert a triangle:.....	8
Conclusion:.....	9
Task2	10
Task 3	17
Problem description:.....	17
Assumptions.....	17
Detailed solution:.....	18
Code in cpp	19
Pseudocode.....	21
Complexity analysis:.....	22
Comparison:	22
Sample Output	23
Conclusion:.....	23
Task 4	24
Answers of the questions in the task:.....	24
Detailed assumptions:	25
Problem Description:	25
Detailed solution in cpp:	25
Complexity analysis:.....	27
Sample output 1:.....	27
Comparison:	28
Complexity analysis of comparison:	29
Sample output of comparison:	29
Conclusion:.....	29

Task 5	30
Description of the problem:.....	30
Steps/pseudocode:	30
Code in cpp	31
Complexity:	32
Comparison:	35
Conclusion:.....	35
Task 6	36
Description of the problem:.....	36
Steps/pseudocode:	36
Code:	37
Complexity:	39
Comparison:	40
Sample of Output:.....	41
Questions on Puzzle:.....	42
Conclusion:.....	42
Bonus(Frame-Stewart Algorithm):.....	43
.....	44
.....	45
References Used For All Tasks:	45

Task 1

Algorithm Method Applied: Iterative Improvement Method.

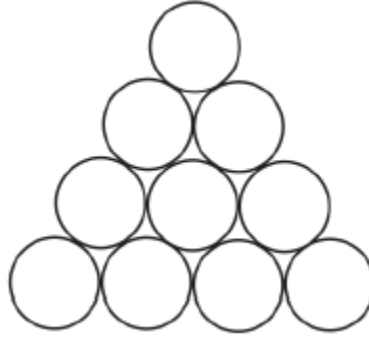


Figure 1: Equilateral Triangle

Detailed Assumptions:

To invert a coin Triangle in the minimum number of moves: (Assuming we have n rows at first)

1. Select one of the horizontal rows as the base of the inverted triangle. (i.e. K)
2. Assuming rows are counted top to bottom, and $1 \leq k \leq n$, so K^{th} row contains k coins.
3. Moving $n-k$ coins into k row, where base of the inverted triangle is k (row selected before applying algorithm).
4. Coins are moved from last row to k row by the following iteration:
 - $\left\lfloor \frac{n-k}{2} \right\rfloor$ right most coins moved to left end of k row.
 - $\left\lceil \frac{n-k}{2} \right\rceil$ left most coins moved to right end of k row.
 - Last iteration will leave in the last row the following number of coins:

$$n - \left(\left\lfloor \frac{n-k}{2} \right\rfloor + \left\lceil \frac{n-k}{2} \right\rceil \right) = n - (n-k) = k \text{ coins in the last row.}$$

5. For the next iteration: $n-k-2$ coins must be moved into row $k+1$ of the triangle.
 - Coins in this row will leave the row by the following iteration:
 - $\left\lfloor \frac{n-k}{2} \right\rfloor - 1$ left most coins moved to right end of $k+1$ row.
 - $\left\lceil \frac{n-k}{2} \right\rceil - 1$ right most coins moved to left end of $k+1$ row.

6. This Operation of continuous improvement goes on till:

➤
$$\left(\left\lfloor \frac{n-k}{2} \right\rfloor - 1\right) - \left\lfloor \frac{n-k}{2} \right\rfloor - 1 = 0 \text{ or } 1$$

depends on whether $(n - k)$ is either even or odd respectively

➤ Then, left most coins from row $n - \left\lfloor \frac{n-k}{2} \right\rfloor - 1$ are moved to the right end of row $k + \left\lfloor \frac{n-k}{2} \right\rfloor - 1$

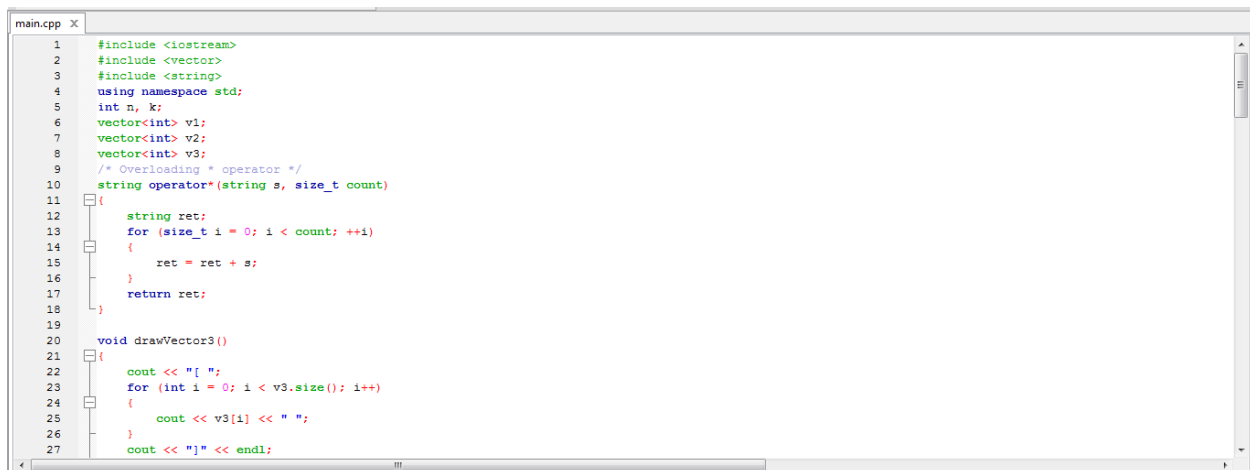
7. Finally, coins from the first $k-1$ rows of the given triangle are moved in the reverse order, from longest to shortest, to form successive rows below the original base row.

Problem Description:

Inverting a Coin Triangle Consider an equilateral triangle formed by closely packed pennies or other identical coins like the one shown in the figure below. (The centers of the coins are assumed to be at the points of the equilateral triangular lattice.)

Use iterative improvement method to design an algorithm to flip the triangle upside down in the minimum number of moves if on each move you can slide one coin at a time to its new position.

Detailed Solution including code and the description of the steps of your solution:



```
main.cpp x
1  #include <iostream>
2  #include <vector>
3  #include <string>
4  using namespace std;
5  int n, k;
6  vector<int> v1;
7  vector<int> v2;
8  vector<int> v3;
9  /* Overloading * operator */
10 string operator*(string s, size_t count)
11 {
12     string ret;
13     for (size_t i = 0; i < count; ++i)
14     {
15         ret = ret + s;
16     }
17     return ret;
18 }
19
20 void drawVector3()
21 {
22     cout << "[ ";
23     for (int i = 0; i < v3.size(); i++)
24     {
25         cout << v3[i] << " ";
26     }
27     cout << "]" << endl;
```

Figure 2: Code Implementation 1

```

main.cpp X
29 void drawVector2()
30 {
31     cout << "[ ";
32     for (int i = 0; i < v2.size(); i++)
33     {
34         cout << v2[i] << " ";
35     }
36     cout << "]" << endl;
37 }
38 void drawVector1()
39 {
40     cout << "[ ";
41     for (int i = 0; i < v1.size(); i++)
42     {
43         cout << v1[i] << " ";
44     }
45     cout << "]" << endl;
46 }
47 void reverse_triangle(vector<int> v)
48 {
49     string data1 = " ";
50     string data2 = " ";
51     for (int i = 0; i <= v.size() - 1; i++)
52     {
53         std::cout << data1 * (v.size() - v[i]) + data2 * (v[i]) << "\n";
54     }
55 }

```

Figure 3: Code Implementation 2

```

main.cpp X
56 void drawTriangle(int n)
57 {
58     int counter = 1;
59     for (int i = 1; i <= n; i++)
60     {
61         for (int x = n - i; x > 0; x--)
62         {
63             cout << " ";
64         }
65         for (int j = 1; j <= i; j++)
66         {
67             cout << "*" << " ";
68         }
69         cout << endl;
70         v1.push_back(counter);
71         counter++;
72     }
73     cout << "the original vector\n";
74     drawVector1();
75 }
76 void Kth(int m, int s)
77 {
78     // m is n & s is k
79     if (s <= 0 || s > m)
80     {
81         cout << "invalid value of the base row please restart the program and enter a valid value" << endl;
82         exit(0);
83     }
84     if ((m - s) % 2 == 0)
85     {
86         for (int i = 0; i <= (n - k) / 2; i++)
87         {
88             int val = m;
89             // cout << val << endl;
90             while (s != val)
91             {
92                 v1[m - 1] = v1[m - 1] - 2;
93                 v1[s - 1] = v1[s - 1] + 2;
94                 // remove MostRight from last row and add it to MostRight of Kth Row ;
95                 // remove MostLeft from last row and add it to MostLeft of Kth Row ;
96                 val = val - 2;
97             }
98             drawVector1();
99             reverse_triangle(v1);
100             m--;
101         }
102     }
103 }

```

Figure 4: Code Implementation 3

```

main.cpp X
79 void Kth(int m, int s)
80 {
81     // m is n & s is k
82     if (s <= 0 || s > m)
83     {
84         cout << "invalid value of the base row please restart the program and enter a valid value" << endl;
85         exit(0);
86     }
87     if ((m - s) % 2 == 0)
88     {
89         for (int i = 0; i <= (n - k) / 2; i++)
90         {
91             int val = m;
92             // cout << val << endl;
93             while (s != val)
94             {
95                 v1[m - 1] = v1[m - 1] - 2;
96                 v1[s - 1] = v1[s - 1] + 2;
97                 // remove MostRight from last row and add it to MostRight of Kth Row ;
98                 // remove MostLeft from last row and add it to MostLeft of Kth Row ;
99                 val = val - 2;
100             }
101             drawVector1();
102             reverse_triangle(v1);
103             m--;
104         }
105     }
106 }

```

Figure 5: Code Implementation 4

```

main.cpp X
106         s++;
107     }
108     // drawVector1();
109 }
110 else
111 {
112     for (int i = 0; i <= ((n - k) / 2) + 1; i++)
113     {
114         // int j = 0;
115         int val = m;
116         while (s != val && s <= m)
117         {
118             v1[m - 1] = v1[m - 1] - (1);
119             v1[s - 1] = v1[s - 1] + (1);
120             val = val - (1);
121         }
122         m--;
123         s++;
124     }
125     drawVector1();
126     reverse_triangle(v1);
127 }
128 for (int i = k - 1; i > 0; i--)
129 {
130     v2.push_back(i);
131 }
132

```

Figure 6: Code Implementation 5

```

main.cpp X
132 }
133 drawVector2();
134
135 for (int i = k; i <= v1.size(); i++)
136 {
137     v3.push_back(v1[i - 1]);
138 }
139 drawVector3();
140 for (int i = 0; i <= v2.size() - 1; i++)
141 {
142     v3.push_back(v2[i]);
143 }
144
145 drawVector3();
146 }
147
148 int main()
149 {
150     cout << "Enter number of rows of the triangle and the base of inverter triangle " << endl;
151     cin >> n >> k;
152     drawTriangle(n);
153     Kth(n, k);
154     reverse_triangle(v3);
155
156     return 0;
157 }
158

```

Figure 7: Code Implementation 6

Complexity Analysis for the algorithm:

Let $M(k)$ be the number of moves made by the iterative algorithm

$$\begin{aligned}
 M(k) &= \sum_{j=0}^{\left\lfloor \frac{n-k}{2} \right\rfloor} (n - k - 2j) + \sum_{j=1}^{k-1} j = \sum_{j=0}^{\left\lfloor \frac{n-k}{2} \right\rfloor} (n - k) - \sum_{j=0}^{\left\lfloor \frac{n-k}{2} \right\rfloor} 2j + \sum_{j=1}^{k-1} j \\
 &= (n - k) \left(\left\lfloor \frac{n-k}{2} \right\rfloor + 1 \right) - \left\lfloor \frac{n-k}{2} \right\rfloor \left(\left\lfloor \frac{n-k}{2} \right\rfloor + 1 \right) + \frac{(k-1)k}{2} * \\
 &\quad \left(\left\lfloor \frac{n-k}{2} \right\rfloor + 1 \right) \left\lfloor \frac{n-k}{2} \right\rfloor + \frac{(k-1)k}{2}
 \end{aligned}$$

➤ If n-k is even

$$M(k) = \left(\frac{n-k}{2} + 1\right) \frac{n-k}{2} + \frac{(k-1)k}{2} = \frac{3k^2 - (2n+4)k + n^2 + 2n}{4}$$

➤ If n-k is odd

$$M(k) = \left(\frac{n-k-1}{2} + 1\right) \frac{n-k+1}{2} + \frac{(k-1)k}{2} = \frac{3k^2 - (2n+4)k + (n+1)^2}{4}$$

• **Minimum value of $M(k)$ is @ $k = (n+2)/3$**

➤ **If k is an integer $\rightarrow (n-k) = (3i+1) - \frac{3i+1+2}{3} = 2i$, which is even & a unique solution**

➤ **If k is not an integer $\rightarrow k^- = \left\lfloor \frac{n+2}{3} \right\rfloor, k^+ = \left\lceil \frac{n+2}{3} \right\rceil$**

$$T(n) = \theta(n^2)$$

Sample Output of the solution:

[illegible]

Figure 8: Sample Output

A comparison between your algorithm and at least one other:

Now we solve the problem using Bubble sort instead of Iterative Improvement so we will replace Kth method with bubble sort to sort the vector v1 produced in the drawTriangle() method finally we use The reverse_triangle() method

- **Code :**

```
}
void bubble_sort(std::vector<int>& v) {
    if(v.size() == 0) return;

    for(int max = v.size(); max > 0; max--) {
        for(int i = 1; i < max; i++) {
            int& current = v[i - 1];
            int& next = v[i];
            if(current < next)
                std::swap(current, next);
        }
    }
    cout<<"the vector after applying bubble sort"<<endl;
    drawVector1();
}

int main() {
    cout << "enter number of rows of the triangle " << endl;
    cin >> n;
    drawTriangle(n);
    bubble_sort(v1);
    reverse_triangle(v1);

    return 0;
}
```

Figure 9(The Bubblesort Method)

- **Complexity Analysis of the Bubble sort Algorithm:**

$$\sum_{i=0}^n \sum_{j=1}^n 1 = \sum_{i=0}^n (n-1) = n(n-1) = n^2 - n$$

$T(n) = \theta(n^2)$ in Average and Worst case but in best case is $T(n) = \theta(n)$

- The sample of the Output in This case:

```

"D:\CSE(Senior 1)\second term\design algorithms and analysis\Bubblesort comparison\Bubblesort\bin\Debug\quicksort.exe"
enter number of rows of the triangle
10
  *
 * *
* * *
* * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
the original vector before applying bubble sort
[ 1 2 3 4 5 6 7 8 9 10 ]
the vector after applying bubble sort
[ 10 9 8 7 6 5 4 3 2 1 ]
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * *
* *
*
*

Process returned 0 (0x0)   execution time : 21.165 s
Press any key to continue.

```

Figure 10(The output in case of using Bubblesort)

Minimum Number of moves to invert a triangle:

Let $M(k)$ be the number of moves made by the iterative algorithm

$$\begin{aligned}
 M(k) &= \sum_{j=0}^{\lfloor \frac{n-k}{2} \rfloor} (n-k-2j) + \sum_{j=1}^{k-1} j = \sum_{j=0}^{\lfloor \frac{n-k}{2} \rfloor} (n-k) - \sum_{j=0}^{\lfloor \frac{n-k}{2} \rfloor} 2j + \sum_{j=1}^{k-1} j \\
 &= (n-k) \left(\left\lfloor \frac{n-k}{2} \right\rfloor + 1 \right) - \left\lfloor \frac{n-k}{2} \right\rfloor \left(\left\lfloor \frac{n-k}{2} \right\rfloor + 1 \right) + \frac{(k-1)k}{2} \\
 &\quad \left(\left\lfloor \frac{n-k}{2} \right\rfloor + 1 \right) \left\lfloor \frac{n-k}{2} \right\rfloor + \frac{(k-1)k}{2}
 \end{aligned}$$

➤ If n-k is even

$$M(k) = \left(\frac{n-k}{2} + 1\right) \frac{n-k}{2} + \frac{(k-1)k}{2} = \frac{3k^2 - (2n+4)k + n^2 + 2n}{4}$$

➤ If n-k is odd

$$M(k) = \left(\frac{n-k-1}{2} + 1\right) \frac{n-k+1}{2} + \frac{(k-1)k}{2} = \frac{3k^2 - (2n+4)k + (n+1)^2}{4}$$

• **Minimum value of $M(k)$ is @ $k = (n+2)/3$**

➤ **If k is an integer $\rightarrow (n-k) = (3i+1) - \frac{3i+1+2}{3} = 2i$, which is even & a unique solution**

➤ **If k is not an integer $\rightarrow k^- = \left\lfloor \frac{n+2}{3} \right\rfloor, k^+ = \left\lceil \frac{n+2}{3} \right\rceil$**

Conclusion:

Using the cut-off triangle makes it easy to modify the iterative improvement algorithm to satisfy the additional requirement that “on each move a coin must be slid to a new position so that to touch two other coins that rigidly determine its new position”.

Rather than moving coins a horizontal row by a horizontal row, we can slide all the coins in a cut-off triangle to their designated locations in the inverted triangle by always taking an outside coin from the triangle given and sliding it to its new location where it touches at least two other coins.

Task2

Problem description:

Consider the one-dimensional version of peg solitaire played on an array of n cells, where n is even and greater than 2. Initially, all but one cell are occupied by some counters (pegs), one peg per cell. On each move, a peg jumps over its immediate neighbor to the left or to the right to land on an empty cell; after the jump, the jumped-over neighbor is removed from the board.

Assumptions:

Assume the elements which contain peg in the array are 1 and the empty elements are 0.

Detailed solution & Pseudocode:

1. Determine the array length from the user input.
2. Put the first element = 0 and the rest of the array = 1.
3. Iterate over the array.
4. If index contain a peg and index + 1 (or index -1) exists and contains a peg and index + 2 (or index - 2) exists and empty, then index + 2 (or index - 2) = 1 and index and index + 1 (or index - 1) = 0.
5. Repeat step 3 and 4 until there is no adjacent pegs with empty element after them.
6. If there is only one peg remaining, then that's a solution.
7. If there is more than one peg remaining, then no solution exists.
8. Repeat the steps from 2 with different empty element till the end of the array.

Code in cpp:

```
//Input: the array size.
//Output: the location of the empty cells and the corresponding location of the
remaining peg for every solution.
#include <iostream>
#include <vector>
using namespace std;
void print(long long x, vector<long long> v) {
    cout << "If the empty position was in " << x + 1 << ", then the possible final peg
positions are : ";
    if (v.size() == 0) { cout << "no possible solution" << endl; return; }
    cout << v[0];
```

```

for (long long i = 1; i < v.size(); i++)
{
    cout << " or " << v[i];
}
cout << endl;
}

long long onlyONE(vector<int>v) {
    long long x = -1;
    for (size_t i = 0; i < v.size(); i++)
    {
        if (v[i] == 1) {
            if (x == -1) { x = i; }
            else { return -1; }
        }
    }
    return x;
}

vector<long long> add(vector<long long> soln, int x) {
    for (size_t i = 0; i < soln.size(); i++)
    {
        if (soln[i] == x) { return soln; }
    }
    soln.push_back(x);
    return soln;
}

long long bin_rep(vector<int> v) {
    long long base = 1;
    long long res = 0;
    for (long long i = v.size() - 1; i >= 0; i--)
    {
        res += v[i] * base;
        base *= 2;
    }
    return res;
}

long long get_index(long long rep, vector<long long> to_get_index) {
    for (size_t i = 0; i < to_get_index.size(); i++)
    {
        if (to_get_index[i] == rep) { return i; }
    }
    return -1;
}

long long sum(vector<int> v) {
    long long res = 0;
    for (size_t i = 0; i < v.size(); i++)
        res += v[i];
    return res;
}

```

```

vector<long long> sol(vector<int> v, vector<long long> soln) {
    static vector<long long> to_get_index;
    static vector<vector<long long>> real_soln;
    long long rep = bin_rep(v);
    long long index = get_index(rep, to_get_index);
    long long x = onlyONE(v);
    if (x != -1) {
        soln = add(soln, x + 1);
        if (index == -1) {
            to_get_index.push_back(rep);
            real_soln.push_back(soln);
        }
        else {
            real_soln[index] = soln;
        }
        return soln;
    }
    //if (index != -1) {
    //    return real_soln[index];
    //}
    vector<int> poss(v.size(), 0);
    for (size_t i = 0; i < v.size(); i++)
    {
        if (i < v.size() - 2) {
            if (v[i] == 1 && v[i + 1] == 1 && v[i + 2] == 0) { poss[i] += 1; }
        }
        if (i > 1) {
            if (v[i] == 1 && v[i - 1] == 1 && v[i - 2] == 0) { poss[i] += 10; }
        }
    }
    if (sum(poss) == 0) {
        to_get_index.push_back(rep);
        real_soln.push_back(soln);
        return soln;
    }
    for (size_t i = 0; i < poss.size(); i++)
    {
        if (poss[i] == 1 || poss[i] == 11) {
            vector<int> vv = v;
            vv[i] = 0; vv[i + 1] = 0; vv[i + 2] = 1;
            soln = sol(vv, soln);
            if (index == -1) {
                to_get_index.push_back(rep);
                real_soln.push_back(soln);
            }
        }
    }
}

```

```

        else {
            real_soln[index] = soln;
        }
    }
}
return soln;
}
int main() {
    long long n;
    cin >> n;
    vector<int> v(n, 1);
    for (long long i = 0; i < n; i++)
    {
        v[i] = 0;
        vector<long long> soln = sol(v, {});
        print(i, soln);
        v[i] = 1;
    }
    return 0;
}

```

Complexity analysis:

In the worst-case scenario, the code will try putting the empty slot in every element of the array then iterate over the array to move the pegs that can be moved then call the function recursively every time the pegs move so the complexity is:

$$F(n) = F(n - 1) + 1$$

$$F(1) = 0$$

$$\sum_{i=0}^n \sum_{j=0}^n (n - 2) = \sum_{i=0}^n (n - 2) * n = n^2(n - 2) = n^3 - 2n^2$$

$\in \theta(n^3)$ where n is the length of the array

Comparison:

Applying Divide & Conquer Algorithm:

```
#include <iostream>
#include <string>
#include <vector>
#include <unordered_map> // hashtable using c++
using namespace std;
bool One_Peg(int arr[], int n){
    int constant = 0;
    for (int i = 0; i < n && constant < 2; i++) if (arr[i] == 1) constant++;
    return (constant == 1);
}
vector<int*> Move(int arr[], int n){ // [ [0, 2], [2, 4] ] => [1, 1, 0, 1, 1]
    vector<int*> x;
    for(int i = 0; i < n - 2; i++){
        if(arr[i] == 1 - arr[i + 2] && arr[i + 1] == 1)
            x.push_back(new int[2]{i, i+2});
    }
    return x;
}
int* move(int arr[], int n, int solit, int erre){
    int* new_arr = new int[n];
    for (int i = 0; i < n; i++){
        if(i >= solit && i <= erre) new_arr[i] = 1 - arr[i];
        else new_arr[i] = arr[i];
    }
    return new_arr;
}
int* PegSolitaire(int arr[], int n){
    //case 1: check if arr has 1 peg only and res is 0
    if (One_Peg(arr, n) == true){
        return arr;
    }
    //case 2: check if there is available moves using pegs
    if (Move(arr, n).size() == 0){ // && One_Peg(arr) == false
        int* arr = new int[1]{-1};
        return arr;
    }
    //recursive case 2^n [1, 1, 0, 1, 1, 1, 1]
    int solit, erre;
    vector<int*> move_availabe;
```

```

move_availabe = Move(arr, n); // [ [0, 1], [2, 4] ]
// vector<int>
for (int i = 0; i < move_availabe.size(); i++){
    //update move
    solit = move_availabe[i][0];
    erre = move_availabe[i][1];
    int* new_arr = move(arr, n, solit, erre);
    int *output = PegSolitaire(new_arr, n);
    if (output[0] == -1){ // or [-1]
        continue;
    }
    else{
        return output;
    }
}
int* result_bad = new int[1]{-1}; //[-1]
return result_bad;
}

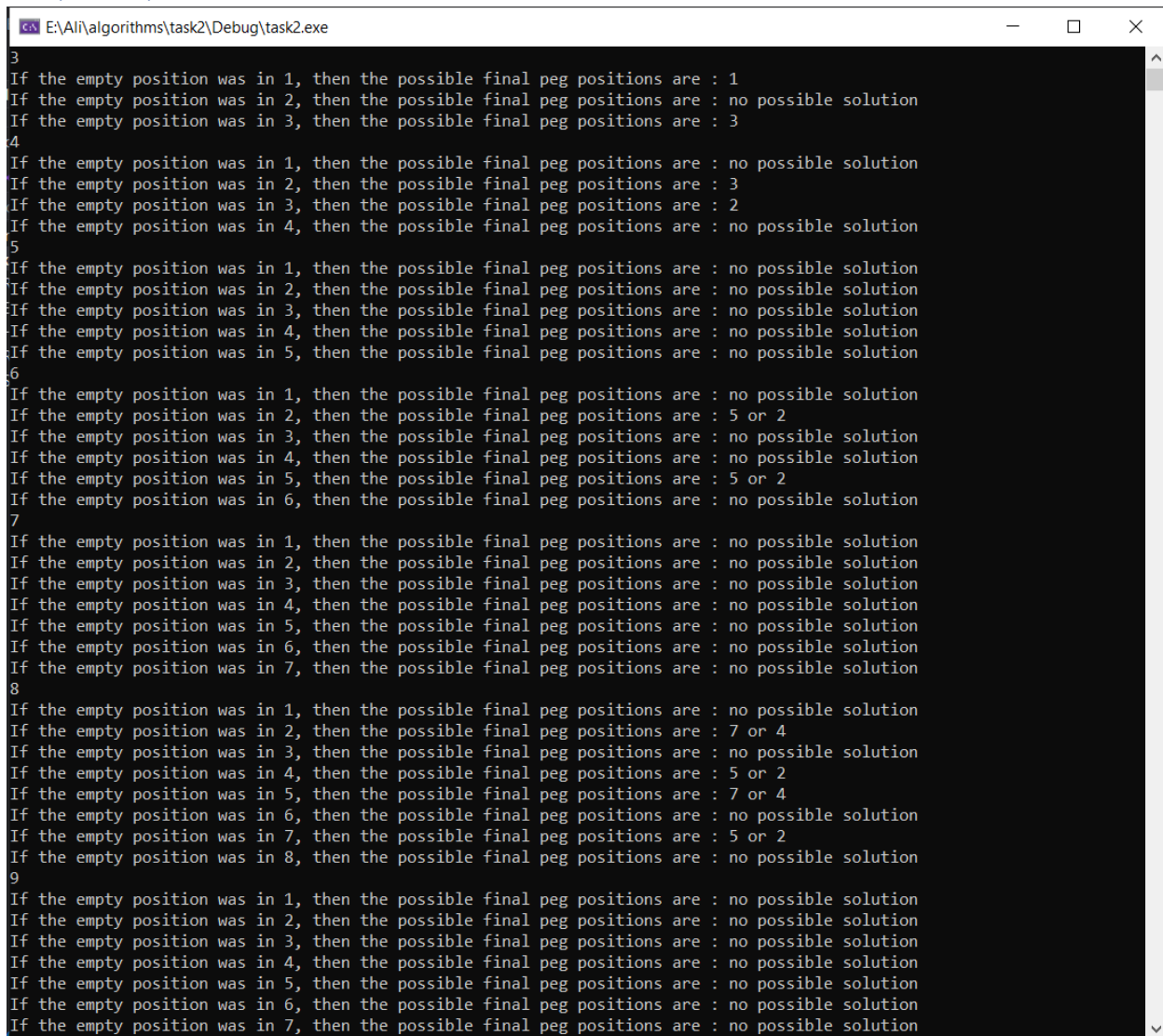
int main(){
    int arr[10] = {1, 1, 1, 0, 1, 1, 1, 1, 1, 1}; //1, 4, 5, 8
    int* output = PegSolitaire(arr, 10);
    if (output[0] == -1){
        cout << "Not existing " << endl;
    }
    else{
        for (int i = 0; i < 10; i++){
            cout << output[i] << ' ';
        }
    }
}

```

Recurrence Relation : $f(n) = 2n + n^2 + nf(n)$

Complexity: $\theta(n^2)$ vs $\theta(n^3)$ in dynamic programming

Sample Output:



```
E:\AI\algorithms\task2\Debug\task2.exe
3
If the empty position was in 1, then the possible final peg positions are : 1
If the empty position was in 2, then the possible final peg positions are : no possible solution
If the empty position was in 3, then the possible final peg positions are : 3
4
If the empty position was in 1, then the possible final peg positions are : no possible solution
If the empty position was in 2, then the possible final peg positions are : 3
If the empty position was in 3, then the possible final peg positions are : 2
If the empty position was in 4, then the possible final peg positions are : no possible solution
5
If the empty position was in 1, then the possible final peg positions are : no possible solution
If the empty position was in 2, then the possible final peg positions are : no possible solution
If the empty position was in 3, then the possible final peg positions are : no possible solution
If the empty position was in 4, then the possible final peg positions are : no possible solution
If the empty position was in 5, then the possible final peg positions are : no possible solution
6
If the empty position was in 1, then the possible final peg positions are : no possible solution
If the empty position was in 2, then the possible final peg positions are : 5 or 2
If the empty position was in 3, then the possible final peg positions are : no possible solution
If the empty position was in 4, then the possible final peg positions are : no possible solution
If the empty position was in 5, then the possible final peg positions are : 5 or 2
If the empty position was in 6, then the possible final peg positions are : no possible solution
7
If the empty position was in 1, then the possible final peg positions are : no possible solution
If the empty position was in 2, then the possible final peg positions are : no possible solution
If the empty position was in 3, then the possible final peg positions are : no possible solution
If the empty position was in 4, then the possible final peg positions are : no possible solution
If the empty position was in 5, then the possible final peg positions are : no possible solution
If the empty position was in 6, then the possible final peg positions are : no possible solution
If the empty position was in 7, then the possible final peg positions are : no possible solution
8
If the empty position was in 1, then the possible final peg positions are : no possible solution
If the empty position was in 2, then the possible final peg positions are : 7 or 4
If the empty position was in 3, then the possible final peg positions are : no possible solution
If the empty position was in 4, then the possible final peg positions are : 5 or 2
If the empty position was in 5, then the possible final peg positions are : 7 or 4
If the empty position was in 6, then the possible final peg positions are : no possible solution
If the empty position was in 7, then the possible final peg positions are : 5 or 2
If the empty position was in 8, then the possible final peg positions are : no possible solution
9
If the empty position was in 1, then the possible final peg positions are : no possible solution
If the empty position was in 2, then the possible final peg positions are : no possible solution
If the empty position was in 3, then the possible final peg positions are : no possible solution
If the empty position was in 4, then the possible final peg positions are : no possible solution
If the empty position was in 5, then the possible final peg positions are : no possible solution
If the empty position was in 6, then the possible final peg positions are : no possible solution
If the empty position was in 7, then the possible final peg positions are : no possible solution
```

Figure 11: Output

Conclusion:

Dynamic Programming is the so suitable for solving optimization problem and saves a lot of time of doing redundant computations.

Task 3

Problem description:

We have six knights on 3x4 chessboard, three white knights on the bottom row and three black knights on the top row.

We want to exchange the white and black knights as shown in Figure 2 using a divide and conquer algorithm in the minimum number of moves.

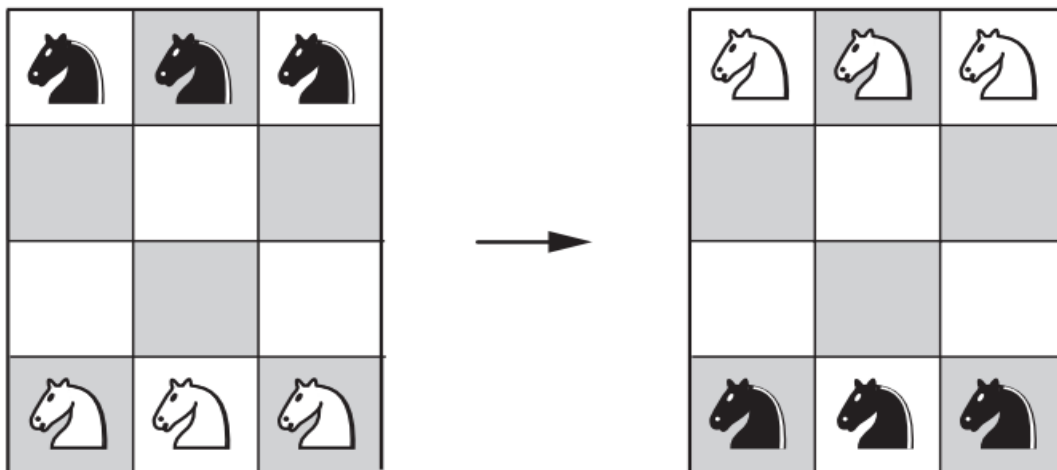


Figure 12: six knights on chessboard

Assumptions: we will assume that the board is numbered as in the following figure so we can follow along with the explanation and the code.

1	2	3
4	5	6
7	8	9
10	11	12

Figure 13: chessboard

Detailed solution:

1. We divide the problem to 2 subproblems: one with the knights on black squares and the other with the knights on the white squares.
2. We invert the board with the knight on the black squares so that the knight on the bottom left corner be on the top left corner and the one on the bottom right corner be on the top right corner and the one on the top center be on the bottom center.
3. Move the knight on square 1 to square 6 then only on the board with the knights on white squares move the knight on square 3 to square 4 then to square 11 then on both boards move the knight on square 6 to square 7.
4. Move the knight on square 11 to square 6 then to square 1.
5. Invert the board with knights on the black squares again and merge the two boards.
6. Move the knight on square 4 to square 3 then move the knight on square 12 to square 7 then to square 2 then move the knight on square 6 to square 7 then to square 12.

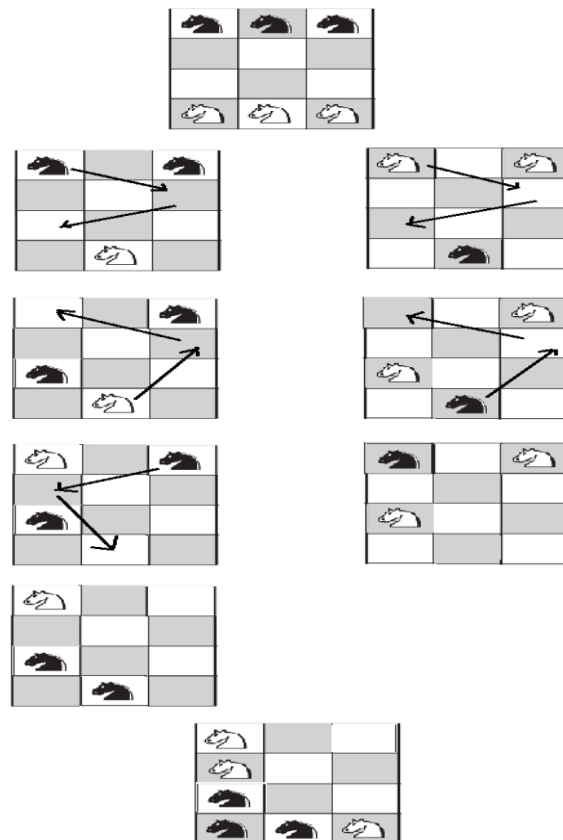


Figure 14: Solution

Code in cpp

```
#include <iostream>
using namespace std;
int** invert(int** arr);
int** divide(int** arr, int count);

int main()
{
    int** board = new int*[4];

    for (int i = 0; i < 4; ++i)
    {
        board[i] = new int[3];
    }
    for (int i = 1; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            board[i][j] = 0;
        }
    }
    for (int i = 0; i < 3; i++) {
        board[0][i] = 1;
    }
    for (int i = 0; i < 3; i++) {
        board[3][i] = 2;
    }
    for (int i = 0; i < 4; i++) {
        cout << endl;
        for (int j = 0; j < 3; j++) {
            cout << board[i][j] << "\t";
        }
    }
    cout << endl;
    board = divide(board,0);
    for (int i = 0; i < 4; i++) {
        cout << endl;
        for (int j = 0; j < 3; j++) {
            cout << board[i][j] << "\t";
        }
    }
}

int** divide(int** arr, int count) {
    if (count == 2) {
        return arr;
    }
    count++;
    arr[1][2] = arr[0][0];
    arr[0][0] = 0;
    arr[2][0] = arr[1][2];
    arr[1][2] = 0;
    arr[1][2] = arr[3][1];
    arr[3][1] = 0;
}
```

```

arr[0][0] = arr[1][2];
arr[1][2] = 0;
if (count == 1) {
    arr[1][0] = arr[0][2];
    arr[0][2] = 0;
    arr[3][1] = arr[1][0];
    arr[1][0] = 0;
}
for (int i = 0; i < 4; i++) {
    cout << endl;
    for (int j = 0; j < 3; j++) {
        cout << arr[i][j] << "\t";
    }
}
cout << endl;
arr = invert(arr);
divide(arr, count);

for (int i = 0; i < 4; i++) {
    cout << endl;
    for (int j = 0; j < 3; j++) {
        cout << arr[i][j] << "\t";
    }
}
cout << endl;
if (count != 1) {
    return arr;
}
arr[0][2] = arr[1][0];
arr[1][0] = 0;
arr[1][2] = arr[2][0];
arr[2][0] = 0;
arr[2][0] = arr[3][2];
arr[3][2] = 0;
arr[0][1] = arr[2][0];
arr[2][0] = 0;
arr[2][0] = arr[1][2];
arr[1][2] = 0;
arr[3][2] = arr[2][0];
arr[2][0] = 0;
return arr;
}

int** invert(int** arr) {
    for (int i = 0; i < 3; i++) {
        int t = arr[0][i];
        arr[0][i] = arr[3][i];
        arr[3][i] = t;
        int s = arr[1][i];
        arr[1][i] = arr[2][i];
        arr[2][i] = s;
    }
    return arr; }

```

Pseudocode

```
Int[][] divide(int[][] arr, int count) {
//Input: 2d array corresponding to the chessboard and a count for the number
of regressions.
//Output: 2d array corresponding to the chessboard after exchanging the black
and white knights.
    if (count == 2) {
        return arr;
    }
    count++;
    arr[1][2] <- arr[0][0];
    arr[0][0] <- 0;
    arr[2][0] <- arr[1][2];
    arr[1][2] <- 0;
    arr[1][2] <- arr[3][1];
    arr[3][1] <- 0;
    arr[0][0] <- arr[1][2];
    arr[1][2] <- 0;
    if (count == 1) {
        arr[1][0] <- arr[0][2];
        arr[0][2] <- 0;
        arr[3][1] <- arr[1][0];
        arr[1][0] <- 0;
    }
    arr <- invert(arr);
    divide(arr, count);
    if (count != 1) {
        return arr;
    }
    arr[0][2] <- arr[1][0];
    arr[1][0] <- 0;
    arr[1][2] <- arr[2][0];
    arr[2][0] <- 0;
    arr[2][0] <- arr[3][2];
    arr[3][2] <- 0;
    arr[0][1] <- arr[2][0];
    arr[2][0] <- 0;
    arr[2][0] <- arr[1][2];
    arr[1][2] <- 0;
    arr[3][2] <- arr[2][0];
    arr[2][0] <- 0;
    return arr;
}

Int[][] invert(int[][] arr) {
    for (int i <- 0; i < 3; i++) {
        int t <- arr[0][i];
        arr[0][i] <- arr[3][i];
        arr[3][i] <- t;
        int s <- arr[1][i];
        arr[1][i] <- arr[2][i];
        arr[2][i] <- s;
    }
    return arr;
}
```

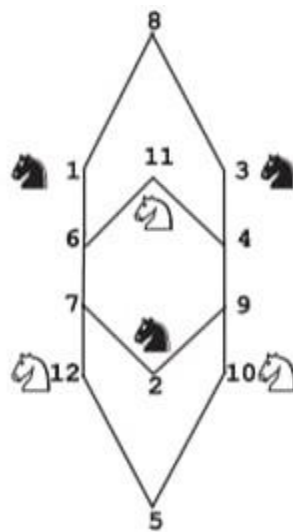

Complexity analysis:

The recursive function divide runs only 2 times and invert iterates over the array so the code complexity will be:

$$\sum_{i=0}^n \sum_{j=0}^m 1 = \sum_{i=0}^n m = n * m \in \theta(n * m)$$

Where n is the number of row and m is the number of columns.

Comparison:



Using brute force and the obtained graph to try different combinations of moves that solve the problem.

We can remove the (8 & 5) nodes from the graph as any moves to these nodes doesn't help with solving the problem and it reduces the number of moves.

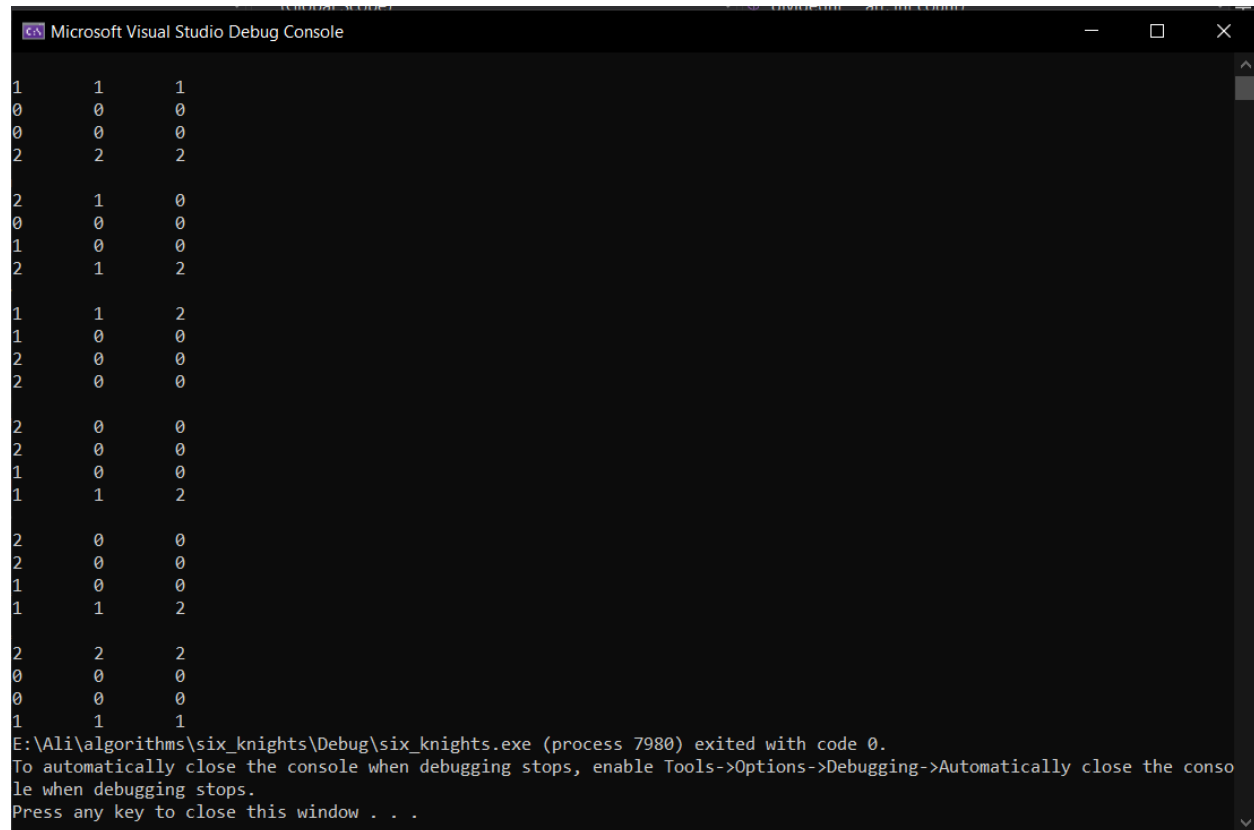
The stop condition for the brute force is if it exceeded 30 moves and if one solution came with a number less than 30 then it will become the next upper bound for the number of iterations.

We can save the route that all the knights took for the solution and chooses the solution with the least moves taken.

Complexity: $O(n!)$ where n is the number of knights.

As the probability of the knights to move to a square becomes less after another knight moves.

Sample Output



```
Microsoft Visual Studio Debug Console

1      1      1
0      0      0
0      0      0
2      2      2

2      1      0
0      0      0
1      0      0
2      1      2

1      1      2
1      0      0
2      0      0
2      0      0

2      0      0
2      0      0
1      0      0
1      1      2

2      0      0
2      0      0
1      0      0
1      1      2

2      2      2
0      0      0
0      0      0
1      1      1

E:\Ali\algorithms\six_knights\Debug\six_knights.exe (process 7980) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Figure 15: Output

Conclusion:

Dividing a problem into smaller subproblems is a better approach than trying to solve it. By dividing this problem we could focus on a solution to an easier one then we could merge these solution to solve the original problem.

Task 4

Answers of the questions in the task:

Does the final distribution of pennies depend on the order in which the machine processes the coin pairs?

No, it doesn't depend on the order because steps may be different from an implementation to another, but the final distribution must be the same.

What is the minimum number of boxes needed to distribute n pennies?

To get the minimum number we will use this expected formula $n = (\log_2(input) + 1)$ and the result of the log we will take the (int) from it without fractions and add a one because the number may be 2^n then it will need an extra box to handle all the numbers.

How many iterations does the machine make before stopping?

We have the following recurrence for the number of iterations needed to place one penny in box i

$$C(i) = 2C(i - 1) + 1 \text{ for } 0 < i \leq k, C(0) = 0.$$

Solving the recurrence by backward substitutions yields the following

$$\begin{aligned} C(i) &= 2C(i - 1) + 1 \\ &= 2(2C(i - 2) + 1) + 1 = 2^2C(i - 2) + 2 + 1 = 2^2(2C(i - 3) + 1) + 2 + 1 \\ &= 2^3 C(i - 3) + 2^2 + 2 + 1 = \dots \\ &= 2^i C(i - i) + 2^{i-1} + 2^{i-2} + \dots + 1 = 2^i \cdot 0 + (2^i - 1) = 2^i - 1 \end{aligned}$$

Hence, the total number of iterations made by the machine before stopping can be found as:

$$\sum_{i=0}^k b_i C(i) = \sum_{i=0}^k b_i (2^i - 1) = \sum_{i=0}^k b_i 2^i - \sum_{i=0}^k b_i = n - \sum_{i=0}^k b_i$$

Detailed assumptions:

- Assumed that the boxes are in an array with n elements.
- Every element of the array is a box from left to right.
- Size of the array is not fixed, and it will be determined from a formula depends on the input pennies.
 - $n = \log_2(input) + 1$
- Which n represents the number of boxes (size of the array).
- The code will iterate element by element and replace every pair of pennies to one in the next box, and this operation will repeat itself on the box until the value in it 0 or 1 then moves to next box and do the same operation.

Problem Description:

A “machine” consists of a row of boxes. To start, one places n pennies in the leftmost box. The machine then redistributes the pennies as follows. On each iteration, it replaces a pair of pennies in one box with a single penny in the next box to the right. The iterations stop when there is no box with more than one coin. For example, see the figure that shows the work of the machine in distributing six pennies by always selecting a pair of pennies in the leftmost box with at least two coins.

Detailed solution in cpp:

```
#include <iostream>
#include <cmath>
using namespace std;
void GreedyTask(int input)
{
    int counter = 0; // counter for the number of iterations
    cout << "the desired number is : " << input << endl;
    cout << "The distribution is: \n";
    if (input < 0)
    {
        cout << "please enter positive number to solve it" << endl;
        return;
    }

    if (input == 1){
        cout << 1 << endl;
    }
}
```

```

    return;
}
int n = log2(input) + 1; // number of boxes by an equation of boxes
int arr[n] = {0};       // array of boxes initially with zeros
arr[0] = input;         // store the input pennies in the first box
for (int i = 0; i < n; i++)
{
    while (arr[i] > 1)
    {
        // repeat if there is a pair in the box
        arr[i] = arr[i] - 2; // decrease the number of pennies by a pair
        arr[i + 1] = arr[i + 1] + 1; // and store one more to the next one
        for (int j = 0; j < n; j++)
        { // for loop to print the updates of the boxes
            cout << arr[j];
            cout << " ";
        }
        cout << "\n";
        counter++; // increase the counter by one after every iteration
        if (arr[i] <= 1)
        {
            break; // if there is no pairs then break and move to next box
        } } }
cout << "Number of iterations = " << counter;
cout << "\n\n";
int main(){
    cout << "Simulation of GreedyTask code " << endl;

    while (true){
        cout << "please enter your input " << endl;
        int n;
        cin >> n;
        GreedyTask(n);
        cout << "if you want to continue the simulation enter 0 else -1 " << endl;
        int x;
        cin >> x;
        if (x == -1)
            break;
        if (x != -1 || x != 0)
        {
            cout << "please enter right number";
            break;
        }
    }

    return 0;}

```

Complexity analysis:

$$\sum_0^n \sum_1^n 1 = \sum_0^n n = n^2 \quad \in \Theta(n^2)$$

Note: code for the printing of the results step by step is not included in the complexity

$\Theta(n^3)$

Note: if we calculate the for loop of printing the array at every iteration then it will be $\Theta(n^2)$

Sample output 1:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS C:\Users\The King 1911> & 'c:\Users\The King 1911\.vscode\extensions-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-Out-h1yjypqq.mpk' '--stderr=Microsoft-MIEngine-Engine-Pid-3itkhook.qfy' '--dbgExe=C:\MinGW\bin\gdb.exe' '--interpret
Simulation of GreedyTask code
please enter your input
3
the desired number is : 3
The distribution is:
1 1
Number of iterations = 1

if you want to continue the simulation enter 0 else -1
0
please enter your input
4
the desired number is : 4
The distribution is:
2 1 0
0 2 0
0 0 1
Number of iterations = 3

if you want to continue the simulation enter 0 else -1
0
please enter your input
5
the desired number is : 5
The distribution is:
3 1 0
1 2 0
1 0 1
Number of iterations = 3
```

```

if you want to continue the simulation enter 0 else -1
0
please enter your input
6
the desired number is : 6
The distribution is:
4 1 0
2 2 0
0 3 0
0 1 1
Number of iterations = 4

if you want to continue the simulation enter 0 else -1
-1

```

The output describes every iteration till stop then calculates the number of iterations.

Comparison:

We use the Brute Force Algorithm to solve this problem to compare it with the main code by Greedy Algorithm.

```

vector<int> GreedyComparison(int n)
{
    vector<int> boxes;
    boxes.push_back(n);
    int position = 0;
    while (boxes[position] > 1)
    {
        if (boxes.size() == (position + 1))
            boxes.push_back(1);
        else
            boxes[position + 1]++;
        boxes[position] = boxes[position] - 2;
        if (boxes[position] < 2)
            position++;
    }
    return boxes;
}

```

Complexity analysis of comparison:

$$T(n) = n - \sum_{i=0}^{\log n} (position)_i = \Theta(n)$$

Sample output of comparison:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Enter the desired number:
1
The Distribution is :
[ 1 ]
Enter the desired number:
2
The Distribution is :
[ 0 1 ]
Enter the desired number:
3
The Distribution is :
[ 1 1 ]
Enter the desired number:
4
The Distribution is :
[ 0 0 1 ]
```

Conclusion:

Greedy algorithm is the most suitable algorithm to solve this problem as wanted to complete all the operations needed in the most right box then move to the next.

Task 5

Description of the problem:

We want Have a set of switches at a certain time they are all on and we would like to turn off a certain switch, but we can't do that unless certain conditions are satisfied.

- 1.If the switch to its immediate right is on and all the other switches to its right, if any, are off.
- 2.We Toggle only one switch at a time.

Steps/pseudocode:

We have a vector which contains either one or zero to indicate the state of the switches (1 for on and 0 for off)

The indices of the vector represent the arrangement of the switches (ascending from left to right)

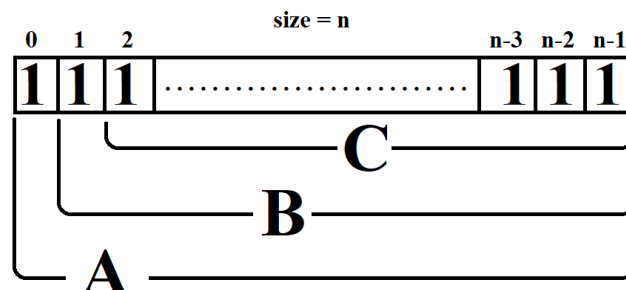
For example, a vector of size 3 be represented as $\begin{pmatrix} 0 & 1 & 2 \\ 1 & 1 & 1 \end{pmatrix}$, 2 is the right most switch.

We Have a Main Method (flip_down) and a Helper method(flip_up).

The "flip_down" and "flip_up" method are recursive methods that call themselves over and over, the base conditions are:

- Closing the last switch.
 - We just flip it as there are no constraints on that
- Closing the next to last one.
 - We flip it down because the last is on and then flip down the last.

If none of the base cases is satisfied, it calls itself again on the switch that's next to the switch that's next to the current switch (**aka. Region C**)



After that we Have all switches off except the current and the one next to it (**Region C OFF**), so we turn current switch off (**Switch 0**).

Now we were able to turn off the current switch (**Switch 0**), but we want to turn off all the switches, so

We turn on all the switches that're next to the switch that's next to the current switch (**Region C ON**).

Then it will call itself with "the vector and the switch next to the current" (**aka. Region B**)

Code in cpp

```
#include <iostream>
#include <vector>
using namespace std;
int x = 0;
void dis(vector<int> v, vector<int> vv) {
    for (int i = 0; i < v.size(); i++) {
        cout << v[i] << " " << vv[i] << endl;
    }
    cout << endl;
}
void dis(vector<int> v) {
    for (int i = 0; i < v.size(); i++) {
        cout << v[i] << " ";
    }
    cout << endl;
}
void flip_u(vector<int>& v, int i);
void flip_d(vector<int>& v, int i) {
    int j = v.size() - 1;
    if (i == j) {
        v[i] = (v[i] == 0) ? 1 : 0;
        x++;
        dis(v);
        return; }
    if (i == j - 1) {
        v[i] = (v[i] == 0) ? 1 : 0;
        dis(v);
        v[j] = (v[j] == 0) ? 1 : 0;
        x++;
        x++;
        dis(v);
        return; }
    flip_d(v, i + 2);
    v[i] = (v[i] == 0) ? 1 : 0;
    x++;
    dis(v);
    flip_u(v, i + 2);
    flip_d(v, i + 1);
    return; }
void flip_u(vector<int>& v, int i) {
    int j = v.size() - 1;
```

```

    if (i == j) {
        v[i] = (v[i] == 0) ? 1 : 0;
        x++;
        dis(v);
        return; }
    if (i == j - 1) {
        v[j] = (v[j] == 0) ? 1 : 0;
        dis(v);
        v[i] = (v[i] == 0) ? 1 : 0;
        x++;
        x++;
        dis(v);
        return; }
    flip_u(v, i + 1);
    flip_d(v, i + 2);
    v[i] = (v[i] == 0) ? 1 : 0;
    x++;
    dis(v);
    flip_u(v, i + 2);
    return; }
int main() {
    /*vector<int> nn;
    vector<int> xx;
    int n=1;
    while (n<=15) {
        nn.push_back(n);
        vector<int> v(n, 1);
        dis(v);
        flip_d(v, 0);
        xx.push_back(x);
        x = 0;
        n++;
    }
    dis(nn,xx);*/
    int n = 10;
    vector<int> v(n, 1);
    dis(v);
    flip_d(v, 0);
    cout << endl << x;
    return 0;}

```

Complexity:

$$d(n) = d(n - 2) + d(n - 1) + u(n - 2) \rightarrow 1$$

$$d(1) = 1$$

$$d(2) = 1$$

$$u(n) = u(n-2) + u(n-1) + d(n-2) \rightarrow 2$$

$$u(1) = 1$$

$$u(2) = 1$$

$$d(n) - u(n) = d(n-1) - u(n-1) \rightarrow 3$$

From 1,2 and 3 we find that $d(n) = u(n)$

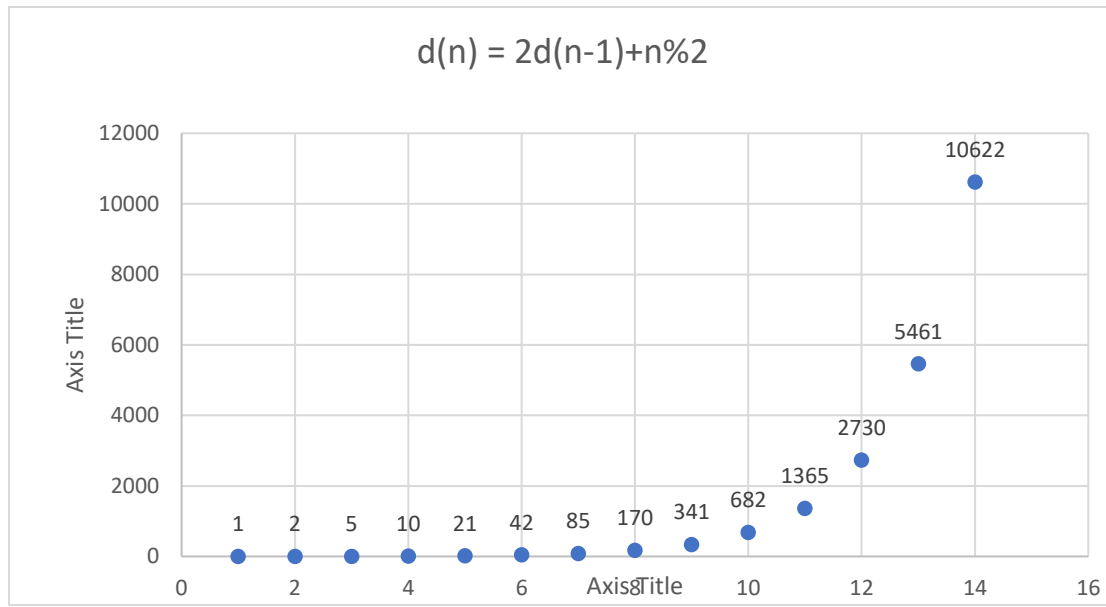
$$\therefore d(n) = 2d(n-2) + d(n-1)$$

And this is similar to the problem of "Tower of Hanoi" as in order to turn all switches off, we need to turn off sub-vector from the original vector and then turn it on after doing something in between.

So $d(n) \in O(2^n)$

And it can be proven by the following Table with numerical approach to:

n	d(n)	d(n) = 2d(n-1)+n%2	2^n
1	1	1	2
2	2	2	4
3	5	5	8
4	10	10	16
5	21	21	32
6	42	42	64
7	85	85	128
8	170	170	256
9	341	341	512
10	682	682	1024
11	1365	1365	2048
12	2730	2730	4096
13	5461	5461	8192
14	10622	10622	16348



```

"D:\CSE(Senior 1)\second term\design algorithms and analysis\lab_test_preparation\Task5\bin\Debug\Task5.exe"
0 0 0 0 0 0 1 0 1 0 0
0 0 0 0 0 0 1 1 1 0 0
0 0 0 0 0 0 1 1 1 0 1
0 0 0 0 0 0 1 1 1 1 1
0 0 0 0 0 0 1 1 1 1 0
0 0 0 0 0 0 1 1 1 0 0
0 0 0 0 0 0 1 1 0 1 0
0 0 0 0 0 0 1 1 0 1 1
0 0 0 0 0 0 1 1 0 0 1
0 0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0 1
0 0 0 0 0 0 1 0 1 1 1
0 0 0 0 0 0 1 0 1 1 0
0 0 0 0 0 0 1 0 1 1 0
0 0 0 0 0 0 1 1 1 1 0
0 0 0 0 0 0 1 1 1 1 1
0 0 0 0 0 0 1 1 1 0 1
0 0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0 1
0 0 0 0 0 0 1 0 1 0 1
0 0 0 0 0 0 1 0 1 1 1
0 0 0 0 0 0 1 0 1 1 0
0 0 0 0 0 0 1 0 1 0 0
0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 0 0 0 1 1 1
0 0 0 0 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 0 0 0 1 1 1
0 0 0 0 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 0 0 0 1 1 1
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0

682
Process returned 0 (0x0)   execution time : 0.973 s
Press any key to continue.

```

Figure 16(Sample output when n=10)

Comparison:

- We will compare our "Divide and conquer" algorithm with "a Brute Force" algorithm we designed.
- Algorithm Security_Switch(int left,int right,bool switches[0.....n-1]){
If(n==1){
Switches[left]=switches[left]^1;
}
If(n==2){
Switches[left]=switches[left]^1;
Switches[right]=switches[right]^1;
}
Arr1→copy(0..... $\frac{n}{2}$);
Arr2→copy($\frac{n}{2}+1$n-1);
Merge(temp,arr1,2, $\frac{n}{2}$,arr2, $\frac{n}{2} + 1$,n);
Security_Switch(temp);
Switches[left]=switches[left]^1;
Security_Switch(temp);
Merge(temp,arr,left,left,temp,0,n);
Security_Switch(temp);
}

$$O(2^{2^{n-1}}n) \text{ Vs. } O(2^n)$$

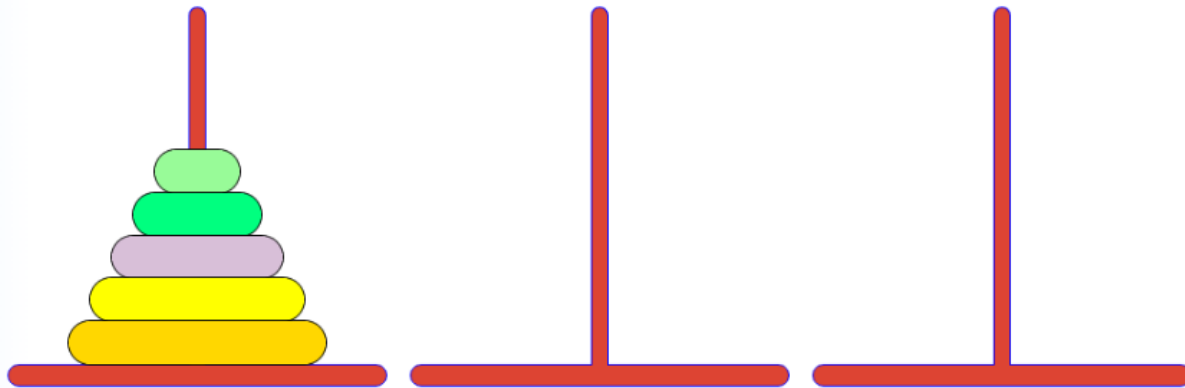
Conclusion:

Divide and Conquer Algorithms has increased performance by $(2^{2^{n-1}}n)$ and was easier to implement.

Task 6

Description of the problem:

The extension of the *Tower of Hanoi* puzzle to more than three pegs was suggested by the French mathematician Édouard Lucas in 1889, who invented the original three-peg version a few years earlier. Under the name *The Reve's Puzzle*, it appeared in Henry E. Dudeney's first puzzle book *The Canterbury Puzzles* [Dud02], where he gave the solutions for $n = 8, 10$, and 21 . Later, more detailed analyses of the above algorithm have produced alternative formulas for the optimal value of the partition parameter k .



Input : number of disks

Output : number of moves

Algorithm Applied : Dynamic Programming

Steps/pseudocode:

Algorithm `Tower_of_hanoi_4rods(discs, arr[])`

// input: Number of discs.

// output: Number of steps.

//Base cases of dynamic programming

if (`arr.search(discs)`) return `arr[discs]`; //check if key is memorized in arr before or not.

if (`discs == 0 || discs == 1`) return `discs`;

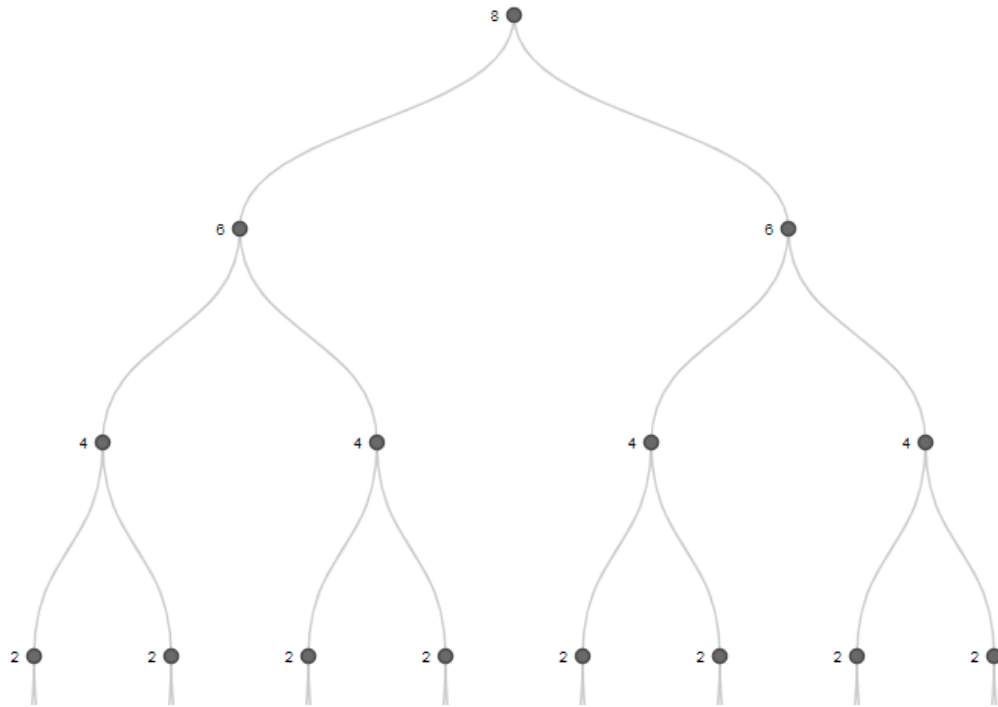
if (`discs == 2`) return 3;

Int `n = 2 * Tower_of_hanoi_4rods(discs - 2, arr) + 3`;

`arr[discs] = n`; // save result of recurrence inside storage element.

return `n`; //returned to parent node.

Here is an example to illustrate operation:



Code:

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

int count = 0;
int* dynamic = nullptr;

void TowersOfHanoi(int n, int r, char at, char to, char arr)
{
    if (n > 0) {
        count++;
        TowersOfHanoi(n - 1, r, at, arr, to);
        cout << "Move disk " << n + r << " at rod " << at << " to rod " << to <<
endl;
        TowersOfHanoi(n - 1, r, arr, to, at);
    }
}
```



```

void TowersOfHanoi_using_4pegs(int n, char at, char to, char arr, char arr2)
{
    if (n == 0)
        return;
    if (n == 1)
    {
        count++;
        cout << "Move disk 1 at rod " << at << " to rod " << to << endl;
        return;
    }

    TowersOfHanoi_using_4pegs(n - 2, at, arr, arr2, to);

    count += 3;

    cout << "Move disk " << n - 1 << " at rod " << at << " to rod " << arr2 <<
endl;
    cout << "Move disk " << n << " at rod " << at << " to rod " << to << endl;
    cout << "Move disk " << n - 1 << " at rod " << arr2 << " to rod " << to <<
endl;

    TowersOfHanoi_using_4pegs(n - 2, arr, to, at, arr2);
}

void OptimumSolutionPuzzle(int n, char at, char to, char arr, char arr2) //
OPTIMUM SOLUTION
{
    int k = n - floor(sqrt(2 * n) + 0.5);
    cout << "-> A has " << n << " pegs" << endl;
    cout << "-> B is our destination" << endl << endl;
    TowersOfHanoi_using_4pegs(k, at, arr2, to, arr);
    TowersOfHanoi(n - k, k, at, to, arr);
    TowersOfHanoi_using_4pegs(k, arr2, to, at, arr);
    cout << endl;
    cout << "-> Total Amount of moves: " << count << endl;
}

void TowersOfHanoi_using_4pegs(int disc, int n, char at, char to, char arr, char
arr2) {
    if (n == 0) {
        return;
    }
    if (n == 1)
    {
        cout << "Move disk 1 at rod " << at << " to rod " << to << endl;

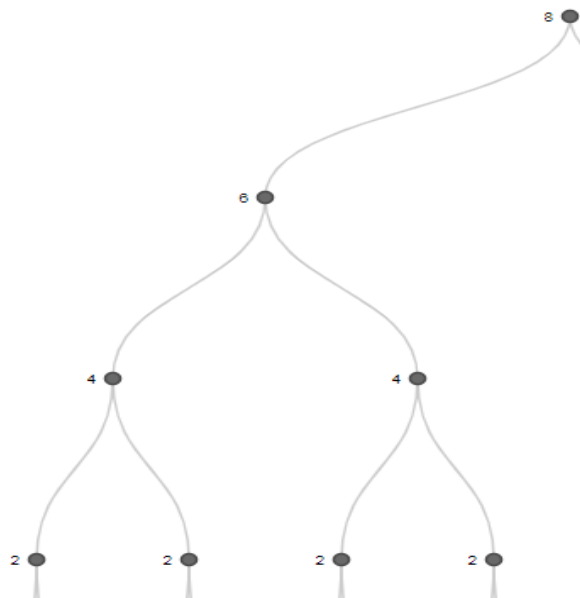
```

```

    return;
}
TowersOfHanoi_using_4pegs(disc, n - 2, at, arr, arr2, to);
if (dynamic == nullptr) {
    int x = (disc < 3) ? 3 : disc;
    dynamic = new int[x + 1];
    dynamic[0] = 0;
    dynamic[1] = 1;
    dynamic[2] = 3;
    dynamic[3] = 5;
}
if (n > 3)
{
    dynamic[n] = (2 * (dynamic[n - 2])) + 3;
}
cout << "Move disk " << n - 1 << " at rod " << at << " to rod " << arr2 << endl;
cout << "Move disk " << n << " at rod " << at << " to rod " << to << endl;
cout << "Move disk " << n - 1 << " at rod " << arr2 << " to rod " << to << endl;
TowersOfHanoi_using_4pegs(disc, n - 2, arr, to, at, arr2);
}
int main() {
    OptimumSolutionPuzzle(8, 'A', 'B', 'C', 'D');
    cout << endl << endl;
    TowersOfHanoi_using_4pegs(8, 8, 'A', 'B', 'C', 'D');
    cout << dynamic[8];
    free(dynamic);
}

```

Complexity:



- Height of tree = 3 which is equivalent to $N/2$. Where each node have 1 Neighborhood so number of nodes (which is the same as number of function call) in terms of input is $N/2 * 2$ function calls so total number of nodes = n :

Time Complexity: $T(n) = \theta(n)$

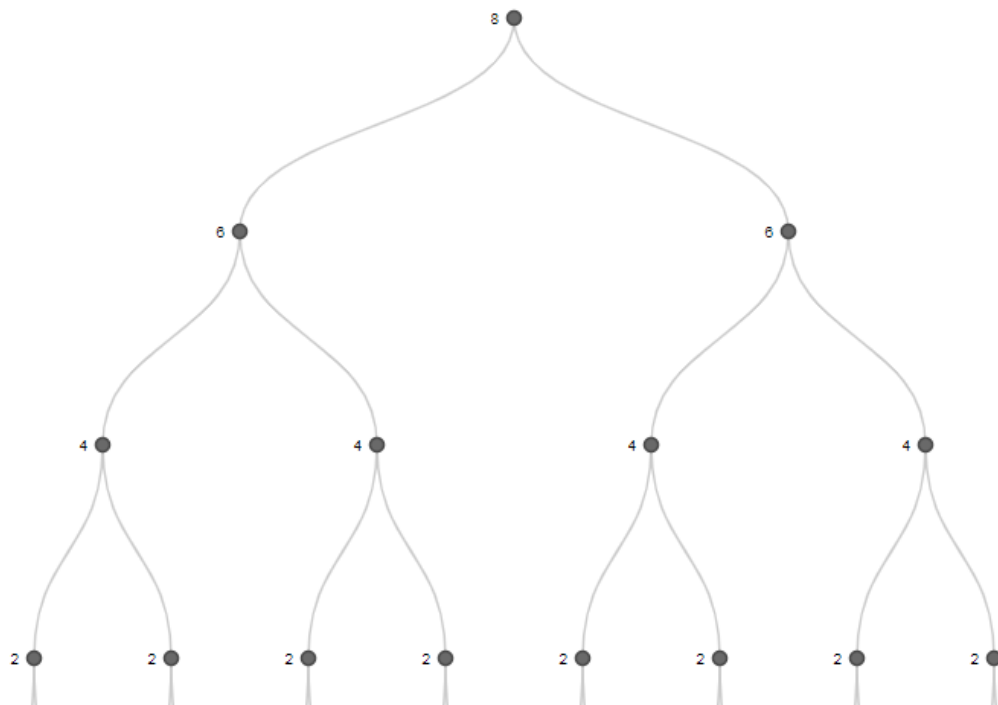
- Space Complexity is number of stack frames + variables saved which is arr + number of stack frames 8 called 6 called 4 called 2 so number of stack frames at t_0 is 4 which is $N/2$ and worst case storage of arr is $N/2$ because we store 8 or 6 or 4 or 2 not all 8 numbers so

$$\text{Space complexity} = \frac{N}{2} + \frac{N}{2} = N$$

Space Complexity: $\theta(n)$

Comparison:

- Using **Divide & Conquer**



-Height of tree = $N/2$ and each subtree has number of nodes = multiples of 2. So, Total number of function calls (which has $O[1]$) = $2 * \text{its self for height times}$, which is $2^{\text{height}} = 2^{N/2} - 1 = 15 \text{ nodes}$

Time Complexity: $T(n) = \theta(2^N)$

Same Number of Stack Frames as previous algorithm so its

Space Complexity: $S(n) = \theta(n)$

```

void TowersOfHanoi_using_4pegs(int n, char at, char to, char arr, char arr2)
{
    if (n == 0)
        return;
    if (n == 1)
    {
        count++;
        cout << "Move disk 1 at rod " << at << " to rod " << to << endl;
        return;
    }

    TowersOfHanoi_using_4pegs(n - 2, at, arr, arr2, to);

    count += 3;

    cout << "Move disk " << n - 1 << " at rod " << at << " to rod " << arr2 << endl;
    cout << "Move disk " << n << " at rod " << at << " to rod " << to << endl;
    cout << "Move disk " << n - 1 << " at rod " << arr2 << " to rod " << to << endl;

    TowersOfHanoi_using_4pegs(n - 2, arr, to, at, arr2);
}

```

Sample of Output:

```

-> A has 8 pegs
-> B is our destination

Move disk 1 at rod A to rod D
Move disk 2 at rod A to rod B
Move disk 1 at rod D to rod B
Move disk 3 at rod A to rod C
Move disk 4 at rod A to rod D
Move disk 3 at rod C to rod D
Move disk 1 at rod B to rod C
Move disk 2 at rod B to rod D
Move disk 1 at rod C to rod D
Move disk 5 at rod A to rod C
Move disk 6 at rod A to rod B
Move disk 5 at rod C to rod B
Move disk 7 at rod A to rod C
Move disk 5 at rod B to rod A
Move disk 6 at rod B to rod C
Move disk 5 at rod A to rod C
Move disk 8 at rod A to rod B
Move disk 5 at rod C to rod B
Move disk 6 at rod C to rod A
Move disk 5 at rod B to rod A
Move disk 7 at rod C to rod B
Move disk 5 at rod A to rod C
Move disk 6 at rod A to rod B
Move disk 5 at rod C to rod B
Move disk 1 at rod D to rod B
Move disk 2 at rod D to rod A
Move disk 1 at rod B to rod A
Move disk 3 at rod D to rod C
Move disk 4 at rod D to rod B
Move disk 3 at rod C to rod B
Move disk 1 at rod A to rod C
Move disk 2 at rod A to rod B
Move disk 1 at rod C to rod B

-> Total Amount of moves: 33

```

```

Move disk 1 at rod A to rod D
Move disk 2 at rod A to rod B
Move disk 1 at rod D to rod B
Move disk 3 at rod A to rod C
Move disk 4 at rod A to rod D
Move disk 3 at rod C to rod D
Move disk 1 at rod B to rod C
Move disk 2 at rod B to rod D
Move disk 1 at rod C to rod D
Move disk 5 at rod A to rod B
Move disk 6 at rod A to rod C
Move disk 5 at rod B to rod C
Move disk 1 at rod D to rod C
Move disk 2 at rod D to rod A
Move disk 1 at rod C to rod A
Move disk 3 at rod D to rod B
Move disk 4 at rod D to rod C
Move disk 3 at rod B to rod C
Move disk 1 at rod A to rod B
Move disk 2 at rod A to rod C
Move disk 1 at rod B to rod C
Move disk 7 at rod A to rod D
Move disk 8 at rod A to rod B
Move disk 7 at rod D to rod B
Move disk 1 at rod C to rod A
Move disk 2 at rod C to rod D
Move disk 1 at rod A to rod D
Move disk 3 at rod C to rod B
Move disk 4 at rod C to rod A
Move disk 3 at rod B to rod A
Move disk 1 at rod D to rod B
Move disk 2 at rod D to rod A
Move disk 1 at rod B to rod A
Move disk 5 at rod C to rod D
Move disk 6 at rod C to rod B
Move disk 5 at rod D to rod B
Move disk 1 at rod A to rod B
Move disk 2 at rod A to rod C
Move disk 1 at rod B to rod C
Move disk 3 at rod A to rod D
Move disk 4 at rod A to rod B
Move disk 3 at rod D to rod B
Move disk 1 at rod C to rod D
Move disk 2 at rod C to rod B
Move disk 1 at rod D to rod B
45
Process returned 0 (0x0)   execution time : 0.437 s
Press any key to continue.

```

Questions on Puzzle:

1. Design a dynamic Programming algorithm to transfer the disks
 - it can be designed but it takes 45 steps.
 - Another optimal solution: Frame-Stewart algorithm.
2. If not design algorithm that make it in 33 steps
 - Divide the pegs into k partitions
 - Put the additional Peg where k can be from 1 to number_of_discs

Conclusion:

1. Dynamic Programming optimize solution and makes a minimal time complexity than normal recurrence method.
2. Although Dynamic Programming optimize solution there is still more optimized solution as at $k = N/2$, 33 steps are needed to move 8 discs and this is the most optimized solution.

Bonus(Frame-Stewart Algorithm):

```
def Tower_Of_Hanoi_3_Rods(discs, arr={}):
    '''
    input: number of discs
    output: number of steps calculated to transfer discs transfer from 1st rod 'A' to
    2nd rod 'B'
    '''
    # case-0 of dynamic programming
    if (discs in arr.keys()):
        return arr[discs]
    # case 1
    if (discs == 0):
        return discs
    # case 2
    if (discs == 1):
        return discs
    # recursion relation:
    Toi = Tower_Of_Hanoi_3_Rods(discs - 1, arr)
    arr[discs] = Toi * 2 + 1
    return Toi * 2 + 1

def Tower_Of_Hanoi_4_Rods(discs, arr={}):
    '''
    input: number of discs
    output: number of steps calculated to transfer discs transfer from 1st rod 'A' to
    2nd rod 'B'
    '''
    # case-0 of dynamic programming
    if discs in arr.keys():
        return arr[discs]
    # case 1
    if discs == 0:
        return discs
    # case 2
    if discs == 1:
        return discs
    # case 3
    if discs == 2:
        return 3
    # recursion relation:
    Toi = Tower_Of_Hanoi_4_Rods(discs - 2, arr)
    arr[discs] = Toi * 2 + 3
    return Toi * 2 + 3
```

```

def Frame_Stewart_algorithm (r, kth):
    nod1 = None
    nod2 = None
    nod3 = None
    if (r <= kth):
        return Tower_Of_Hanoi_3_Rods(r)
    else:
        nod1 = Tower_Of_Hanoi_4_Rods(r - kth)
        nod2 = Tower_Of_Hanoi_3_Rods(kth)
        nod3 = Tower_Of_Hanoi_4_Rods(r - kth)
    return nod1 + nod2 + nod3

def main():
    bestcase = 1e10
    best_kth = 0
    r = int(input("Enter the number of disks needed (r): "))
    print("")
    kth = int(input("Enter the number of kth (kth): "))
    print("")
    print("          Progressing , I'm Still Here with you!!!          ")
    print("")
    for i in range(1,kth+1):
        steps = Frame_Stewart_algorithm(r,i)
        if steps <= bestcase:
            best_kth = i
            bestcase = steps
        print(f"Total of steps for {r} disks is : {steps} with kth = {i}")
    print("Best Kth is :",best_kth)
if __name__=="__main__":
    main()

```

It employs the 4th peg, which has (number of discs – k) discs, and the rest of the pegs have k discs using just 3 pegs, r = number of discs if it's less than the number of partitions, thus no need for partitions, therefore we applied Tower Of Hanoi 3 Rods. However, when discs > number of parts, we put (number of discs – k) discs on the 4th rod, then k (rest of discs) on the 3rd rod, then put (number of discs – k) from the 4th rod to the 3rd rod, and the complexity is still $O[N + N + N]$, so it's still $O(n)$.

Time Complexity: $T(n) = \theta(n)$

Space Complexity: $\theta(n)$

```
PS E:\Second Term\Design and Analysis of Algorithms\Project\Task 6> python -u "e:\Second Term\Design and Analysis of Algorithms\Project\Task 6\Bonus Task 6.py"
PS E:\Second Term\Design and Analysis of Algorithms\Project\Task 6> python -u "e:\Second Term\Design and Analysis of Algorithms\Project\Task 6\Bonus Task 6.py"
Enter the number of disks needed (r): 9

Enter the number of kth (kth): 9

Progressing , I'm Still Here with you!!!

Total of steps for 9 disks is : 91 with kth = 1
Total of steps for 9 disks is : 61 with kth = 2
Total of steps for 9 disks is : 49 with kth = 3
Total of steps for 9 disks is : 41 with kth = 4
Total of steps for 9 disks is : 49 with kth = 5
Total of steps for 9 disks is : 73 with kth = 6
Total of steps for 9 disks is : 133 with kth = 7
Total of steps for 9 disks is : 257 with kth = 8
Total of steps for 9 disks is : 511 with kth = 9
Best Kth is : 4
```

References Used For All Tasks:

Algorithmic Puzzles [Levitin & Levitin 2011-10-14].pdf