Hanoi Algorithm Design, Complexity, & Implementation Analysis by Carlos Flores & George Navarro

Primary Motivation

Implement an algorithm and program it to solve a Tower of Hanoi puzzle for the following graph $\mathcal{G}$ =($\mathcal{V}$,$\mathcal{E}$) with the Vertices, $\mathcal{V}$ = {Start, Aux 1, Aux 2, Aux 3, and Dest}, and Directed Edges $\mathcal{E}$ = {(Start, Aux1), (Aux1, Aux2), (Aux2, Aux3), (Aux3, Aux1), (Aux3, Dest)}.

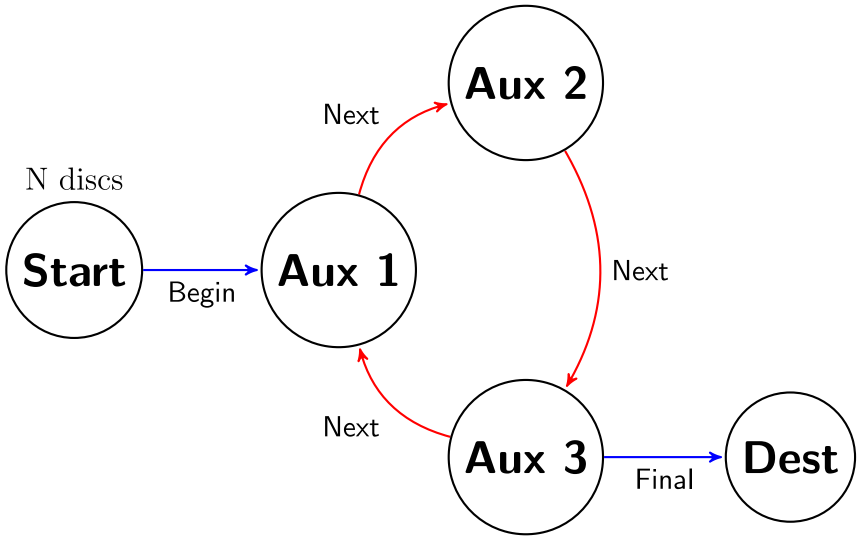
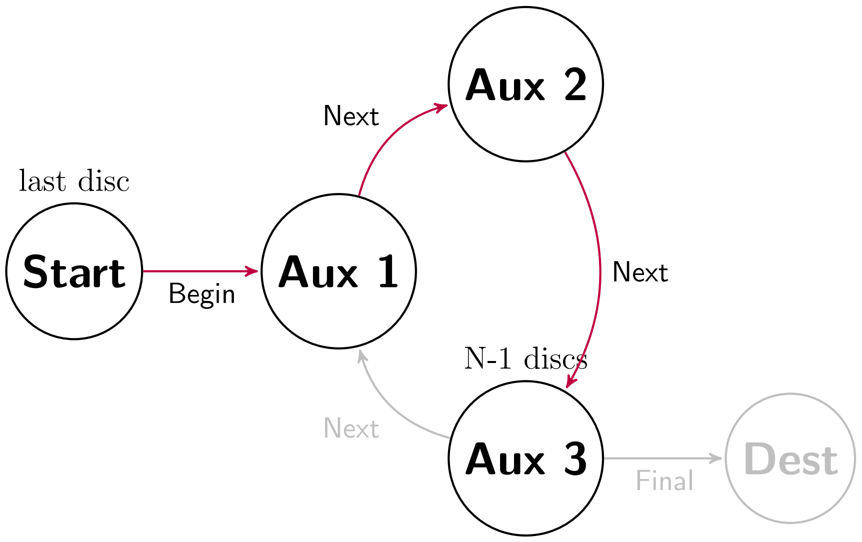


Figure 1.a Tower of Hanoi puzzle for $\mathcal{G}$ =($\mathcal{V}$,$\mathcal{E}$).

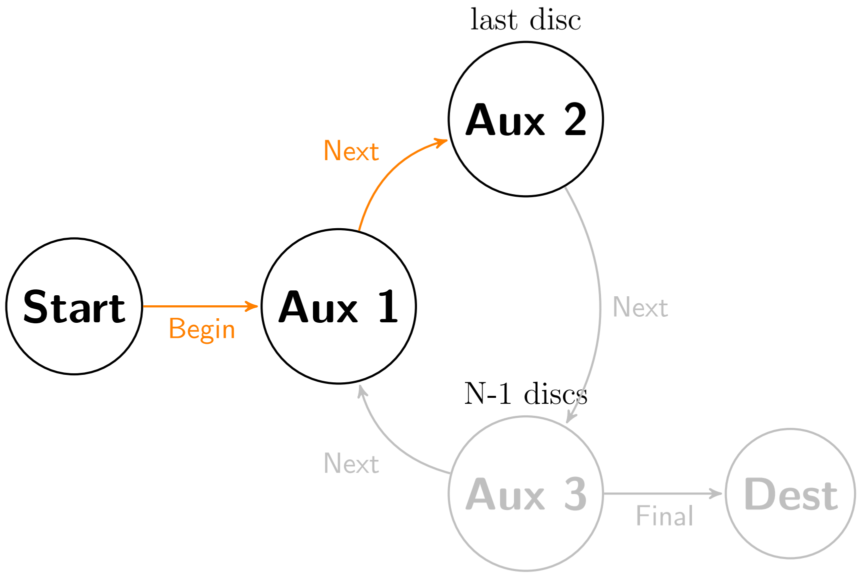
Design Rationale

We can clearly see that the cycle made by the Next edges can be used to move discs around the graph in order to solve the puzzle.

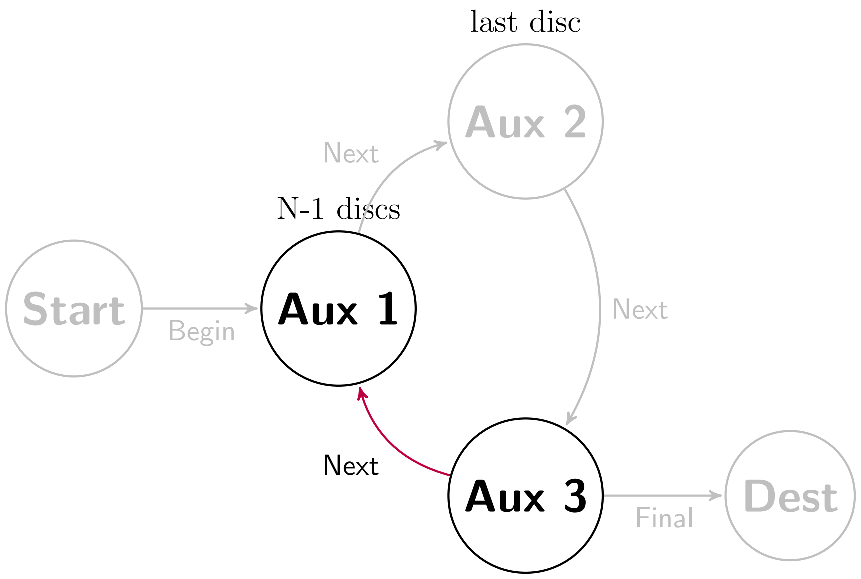
Our goal is to move a stack of N number of discs from Start to Dest. In order to move N number of discs from Start to Dest we must move (N-1) discs to access the last disc. This can be achieved by the following recursive pattern:



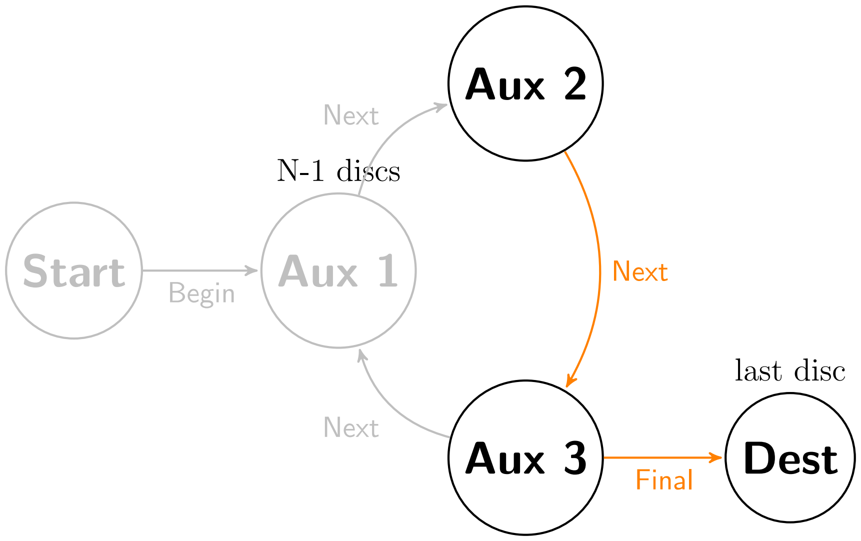
(1) *Figure 1.b* Move (N-1) discs from Start to Aux 3.
This will take T(N-1) operations.



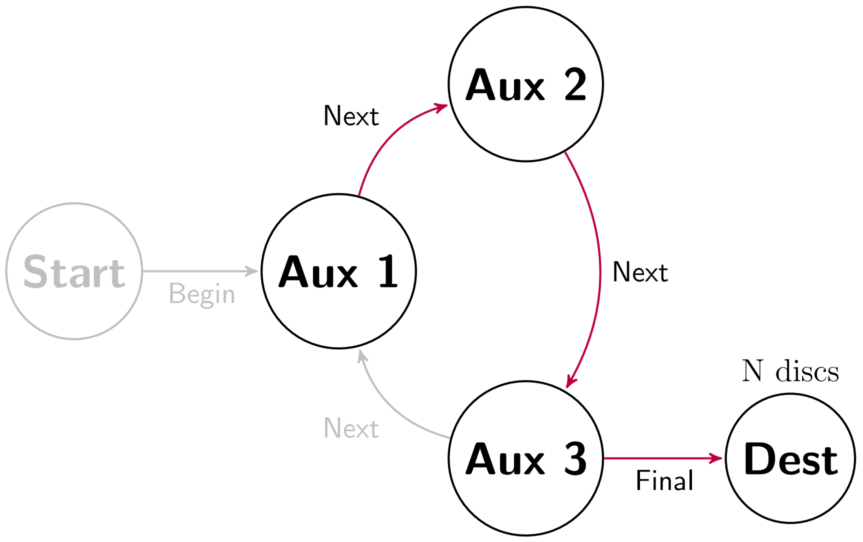
(2) *Figure 1.c* Move the last disc from Start to Aux 2.
This will take a T(1) operation.



(3) *Figure 1.d* Move (N-1) discs from Aux 3 to Aux 1.
This will take T(N-1) operations.



(4) *Figure 1.e* Move 1 disc from Aux 2 to Dest.
This will take a T(1) operation.



(5) *Figure 1.f* Move (N-1) discs from Aux 1 to Dest.
This will take T(N-1) operations.

Now all of the discs have been moved from Start to Dest. We moved the stack of (N-1) discs three times and the original size of the input was decremented by one. The sum of all operations give us the recurrence relation:

$$\{T(N) = 3T(N-1) + 2(1)|\, R = 3, D = 1\}$$

Therefore, according to the Master Theorem of Reduction by Subtraction:

T(N)= $\theta(3^{(N/1)})$ which simplifies to:

$$\boxed{T_{time}(N) = \theta(3^N)} \quad (1)$$

Additionally, the space required to hold the data within memory will remain constant since all recursive invocations free up memory that was used when they terminate. Therefore the space complexity is:

$$\boxed{T_{space}(N) = \theta(N)} \quad (2)$$

Asymptotic Behavior by Rigor

In order to move a stack of N discs 4 vertices from Start to Dest we can use the H4 algorithm as described below:

```
H4(N, A, B, C, D, E){
  N is the number of discs to be moved from Tower A to Tower E via:
  if(N > 0) {
        Move N-1 discs from A to D.
        H3(N-1, A, B, C, D, E)            1st Recursive Invocation $\theta(N-1)$

        Move 1 disc from A to C.
        A $\xrightarrow{N}$B
        B $\xrightarrow{N}$C                          2 Atomic Actions = $\theta(2)$

        Move N-1 discs from D to B.
        H1(N-1, D, B, A, C, E)            2nd Recursive Invocation $\theta(N-1)$

        Move 1 disc from C to E.
        C $\xrightarrow{N}$D
        D $\xrightarrow{N}$E                   2 Atomic Actions = $\theta(2)$

        Move N-1 discs from B to E.
        H3(N-1, B, C, D, E, A)            3rd Recursive Invocation $\theta(N-1)$
}
```

The H4 algorithm has the following recurrence relation:

$$T_4 = \left\{ \begin{array}{ll} \theta(4), & \text{for } N = 1 \\ 2T_3(N-1) + T_1(N-1) + 4, & \text{for } N > 1 \end{array} \right\}$$

We must evaluate the H3 and H1 functions to calculate the complexity of the H4 algorithm. With every invocation of H4 we will recursively invoke two H3 functions and one H1 function. Those two functions must be evaluated to see how the recursion trees grow with respect to the size of the stack (N discs).

In order to move a stack of K discs 3 vertices from A to D (by way of the path ABCD) we can use the H3 algorithm as described below:

```
H3(K, A, B, C, D, E)
  if(K > 0)
       H3(K-1 ,A, B, C, D, E)      1st Recursive Invocation $\theta(K-1)$
       A $\xrightarrow{K}$ B , B $\xrightarrow{K}$ C      2 Atomic Actions $= \theta(2)$
       H1(K-1, D, B, A, C, E)      2nd Recursive Invocation $\theta(K-1)$
       C$\xrightarrow{K}$D      1 Atomic Action $= \theta(1)$
       H2(K-1, E, A, B, C, D)      3rd Recursive Invocation $\theta(K-1)$
```

$$T_3 = \left\{ \begin{array}{ll} \theta(3), & \text{for } K = 1 \\ T_3(K-1) + T_2(K-1) + T_1(K-1) + 3, & \text{for } K > 1 \end{array} \right\}$$

The H3 algorithm will invoke one H1, H2, and H3 call recursively and the arguments will all have diminished by one. The H2 algorithm is defined as a series of functions to move a stack of L discs 2 pegs:

```
H2(L, A, B, C, D, E)
  if(L > 0)
       H2(L-1, A, B, C, D, E)      1st Recursive Invocation $\theta(L-1)$
       A$\xrightarrow{L}$B      1 Atomic Action $= \theta(1)$
       H1(L-1, C, A, B, D, E)      2nd Recursive Invocation $\theta(L-1)$
       B$\xrightarrow{L}$C      1 Atomic Action $= \theta(1)$
       H2(L-1, A, B, C, D, E)      3rd Recursive Invocation $\theta(L-1)$
```

$$T_2 = \left\{ \begin{array}{ll} \theta(2), & \text{for } L = 1 \\ 2T_2(L-1) + T_1(L-1) + 2, & \text{for } L > 1 \end{array} \right\}$$

The H2 algorithm will invoke two recursive H2 algorithms and one H1 algorithm. The last remaining algorithm to define is the H1 algorithm. This function will move a stack of M number of discs 1 peg by performing the following series of actions:

```
H1(M, A, B, C, D, E)
  if(M > 0)
        H2(M-1, A, B, C, D, E)        $1^{st}$ Recursive Invocation $\theta(M-1)$
        $A \xrightarrow{M} B$                        1 Atomic Action $= \theta(1)$
        H2(M-1, C, A, B, D, E)        $2^{nd}$ Recursive Invocation $\theta(M-1)$
```

$$T_1 = \left\{ \begin{array}{ll} \theta(1), & \text{for } M = 1 \\ 2T_2(M-1) + 1, & \text{for } M > 1 \end{array} \right\}$$

Now that we have defined the required recursive algorithm definitions we can analyze the recurrence relationship further:

$$T_4 = \left\{ \begin{array}{ll} \theta(4), & \text{for } N = 1 \\ 2T_3(N-1) + T_1(N-1) + 4, & \text{for } N > 1 \end{array} \right\}$$

$$T_3 = \left\{ \begin{array}{ll} \theta(3), & \text{for } K = 1 \\ T_3(K-1) + T_2(K-1) + T_1(K-1) + 3, & \text{for } K > 1 \end{array} \right\}$$

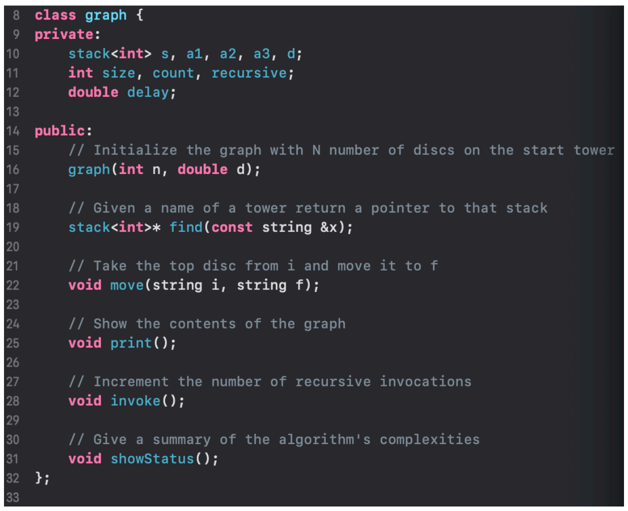
$$T_2 = \left\{ \begin{array}{ll} \theta(2), & \text{for } L = 1 \\ 2T_2(L-1) + T_1(L-1) + 2, & \text{for } L > 1 \end{array} \right\}$$

$$T_1 = \left\{ \begin{array}{ll} \theta(1), & \text{for } M = 1 \\ 2T_2(M-1) + 1, & \text{for } M > 1 \end{array} \right\}$$

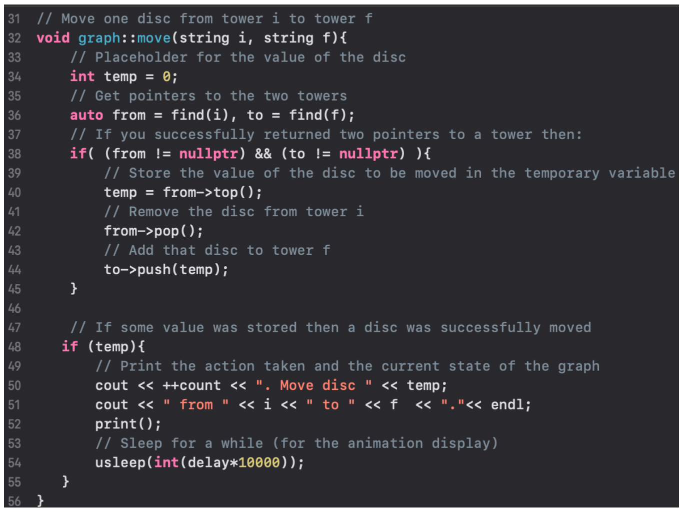
All of the functions trigger three H functions except for the H1 algorithm; that will trigger two H functions. The recursion tree will produce a smaller tree than a 3^N call stack because some recursive invocations will only trigger two functions. While a 3^N tree will always triple with each subsequent call. Additionally the algorithm will always grow faster than a 2^N tree because some functions will triple with each recursive call.

Therefore the asymptotic behavior is bounded from above by 3^N and from below by 2^N

Source Code Breakdown

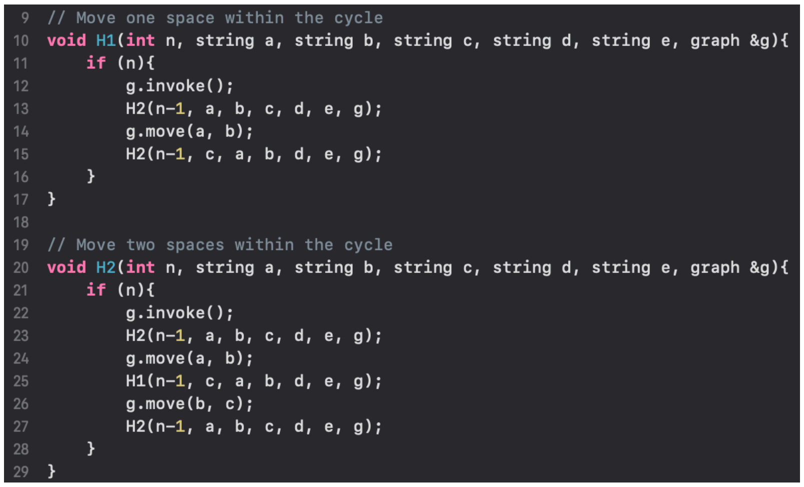


```cpp
class graph {
private:
    stack<int> s, a1, a2, a3, d;
    int size, count, recursive;
    double delay;

public:
    // Initialize the graph with N number of discs on the start tower
    graph(int n, double d);

    // Given a name of a tower return a pointer to that stack
    stack<int>* find(const string &x);

    // Take the top disc from i and move it to f
    void move(string i, string f);

    // Show the contents of the graph
    void print();

    // Increment the number of recursive invocations
    void invoke();

    // Give a summary of the algorithm's complexities
    void showStatus();
};
```



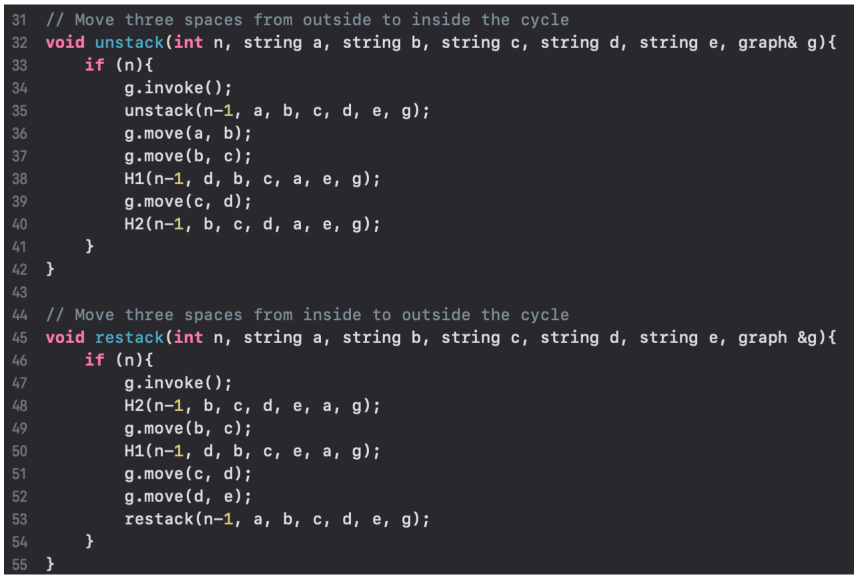
```cpp
// Move one disc from tower i to tower f
void graph::move(string i, string f){
    // Placeholder for the value of the disc
    int temp = 0;
    // Get pointers to the two towers
    auto from = find(i), to = find(f);
    // If you successfully returned two pointers to a tower then:
    if( (from != nullptr) && (to != nullptr) ){
        // Store the value of the disc to be moved in the temporary variable
        temp = from->top();
        // Remove the disc from tower i
        from->pop();
        // Add that disc to tower f
        to->push(temp);
    }

    // If some value was stored then a disc was successfully moved
   if (temp){
       // Print the action taken and the current state of the graph
       cout << ++count << ". Move disc " << temp;
       cout << " from " << i << " to " << f  << "."<< endl;
       print();
       // Sleep for a while (for the animation display)
       usleep(int(delay*10000));
   }
}
```

We used a class with five stacks of integers as member variables to represent the graph. Each stack holds discs independently and represent the towers. None of the edges in the theoretical graph are required to be implemented in code since the algorithm respects the edges by design. Pointers were eschewed in the presence of recursive rigor. Thus, providing an apt example at how algorithmic design can be invaluable and efficient without being careless!



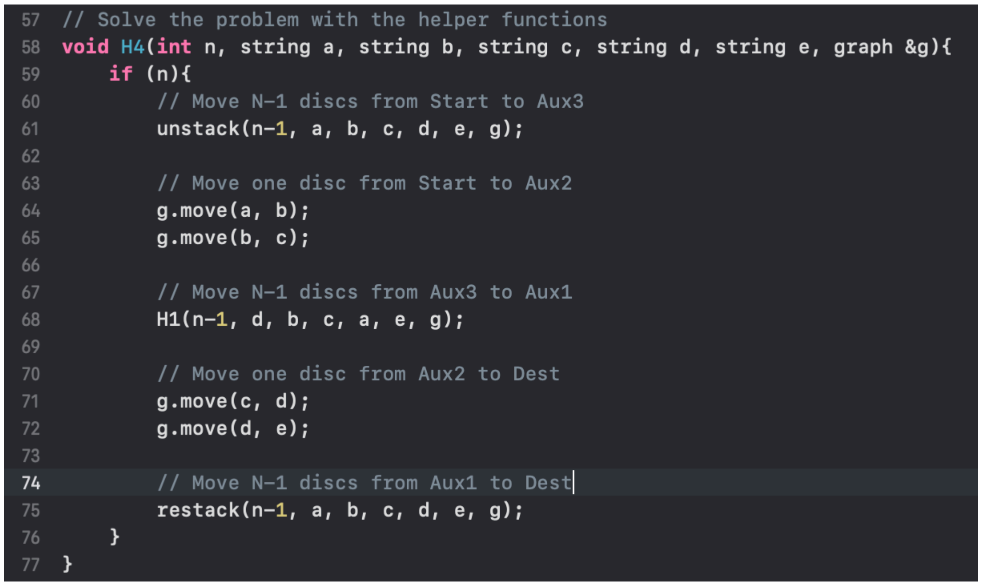
```cpp
// Move one space within the cycle
void H1(int n, string a, string b, string c, string d, string e, graph &g){
    if (n){
        g.invoke();
        H2(n-1, a, b, c, d, e, g);
        g.move(a, b);
        H2(n-1, c, a, b, d, e, g);
    }
}

// Move two spaces within the cycle
void H2(int n, string a, string b, string c, string d, string e, graph &g){
    if (n){
        g.invoke();
        H2(n-1, a, b, c, d, e, g);
        g.move(a, b);
        H1(n-1, c, a, b, d, e, g);
        g.move(b, c);
        H2(n-1, a, b, c, d, e, g);
    }
}
```

The H1 and H2 algorithms are implemented verbatim as our formal logical definition on pages 5 and 6 respectively.



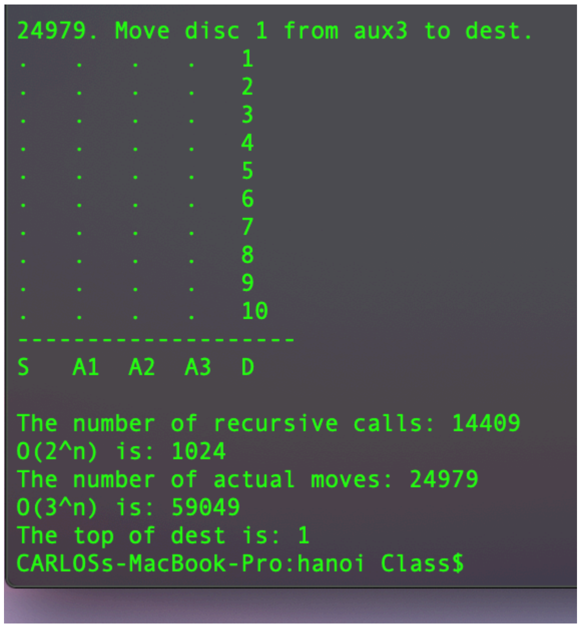
```
// Move three spaces from outside to inside the cycle
void unstack(int n, string a, string b, string c, string d, string e, graph& g){
    if (n){
        g.invoke();
        unstack(n-1, a, b, c, d, e, g);
        g.move(a, b);
        g.move(b, c);
        H1(n-1, d, b, c, a, e, g);
        g.move(c, d);
        H2(n-1, b, c, d, a, e, g);
    }
}

// Move three spaces from inside to outside the cycle
void restack(int n, string a, string b, string c, string d, string e, graph &g){
    if (n){
        g.invoke();
        H2(n-1, b, c, d, e, a, g);
        g.move(b, c);
        H1(n-1, d, b, c, e, a, g);
        g.move(c, d);
        g.move(d, e);
        restack(n-1, a, b, c, d, e, g);
    }
}
```

The H3 algorithm is implemented as unstack() and restack() functions. The former is verbatim to our definition and the latter is the same algorithm but in reverse; it uses H2, H1, then H3 to park the largest disc on Dest. Regardless, the complexities are still unchanged.

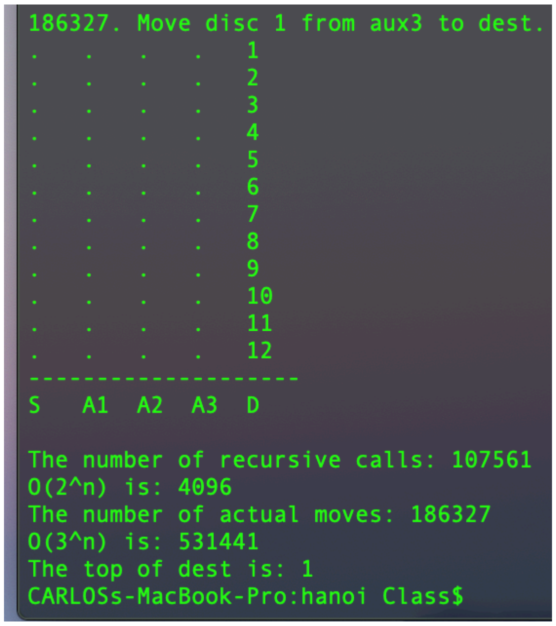


```
// Solve the problem with the helper functions
void H4(int n, string a, string b, string c, string d, string e, graph &g){
    if (n){
        // Move N-1 discs from Start to Aux3
        unstack(n-1, a, b, c, d, e, g);

        // Move one disc from Start to Aux2
        g.move(a, b);
        g.move(b, c);

        // Move N-1 discs from Aux3 to Aux1
        H1(n-1, d, b, c, a, e, g);

        // Move one disc from Aux2 to Dest
        g.move(c, d);
        g.move(d, e);

        // Move N-1 discs from Aux1 to Dest
        restack(n-1, a, b, c, d, e, g);
    }
}
```

The H4 algorithm is the master function to invoke all recursive subroutines.



```
24979. Move disc 1 from aux3 to dest.
.   .   .   .   1
.   .   .   .   2
.   .   .   .   3
.   .   .   .   4
.   .   .   .   5
.   .   .   .   6
.   .   .   .   7
.   .   .   .   8
.   .   .   .   9
.   .   .   .   10
--------------------
S   A1  A2  A3  D

The number of recursive calls: 14409
O(2^n) is: 1024
The number of actual moves: 24979
O(3^n) is: 59049
The top of dest is: 1
CARLOSs-MacBook-Pro:hanoi Class$
```



```
186327. Move disc 1 from aux3 to dest.
.   .   .   .   1
.   .   .   .   2
.   .   .   .   3
.   .   .   .   4
.   .   .   .   5
.   .   .   .   6
.   .   .   .   7
.   .   .   .   8
.   .   .   .   9
.   .   .   .   10
.   .   .   .   11
.   .   .   .   12
--------------------
S   A1  A2  A3  D

The number of recursive calls: 107561
O(2^n) is: 4096
The number of actual moves: 186327
O(3^n) is: 531441
The top of dest is: 1
CARLOSs-MacBook-Pro:hanoi Class$
```

The algorithm has been proven to perform according to our hypothesized complexity and is able to solve the puzzle with varying amounts of discs. It also demonstrates how the Master Theorem of Reduction by Subtraction can be used to identify an exponential complexity.

You can try the program out and see the supplementary materials on GitHub at github.com/cflores713/towerOfHanoi.