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

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)}.

The last edge $(Aux3, Dest)$ will be referred to as the Final Edge, the first edge $(Start, Aux1)$, will be referred to as the Begin Edge, while the rest of the edges in $\mathcal{G}$ will be considered the Next Edges. The motivation for this distinction is to identify the edges used to liberate the largest disc (the Next edges), to identify the specific edge that will be used to move discs into the cycle made by the Next Edges (the Begin Edge), and to identify the specific edge that will be used to park the largest disc not on the Dest vertex (the Final Edge).

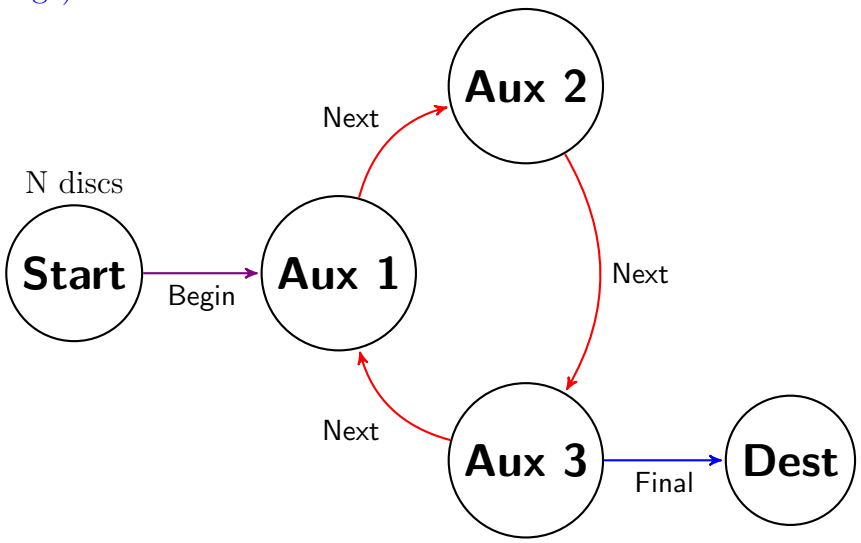
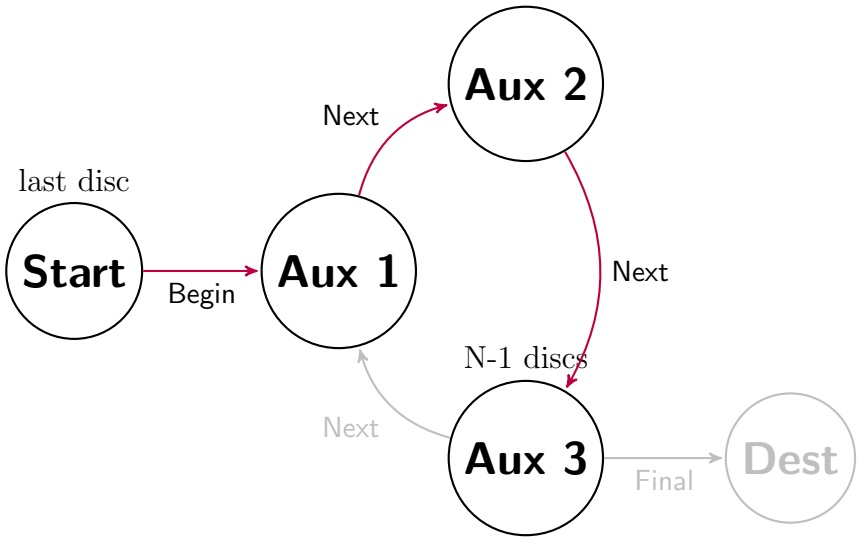


Figure 1.a Tower of Hanoi puzzle for $\mathcal{G}$ =($\mathcal{V}$,$\mathcal{E}$).

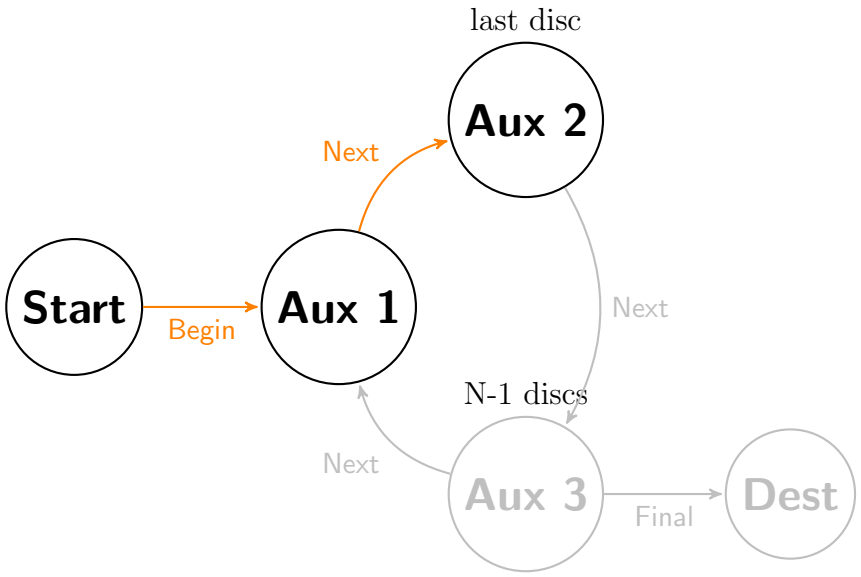
Design Rationale

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.

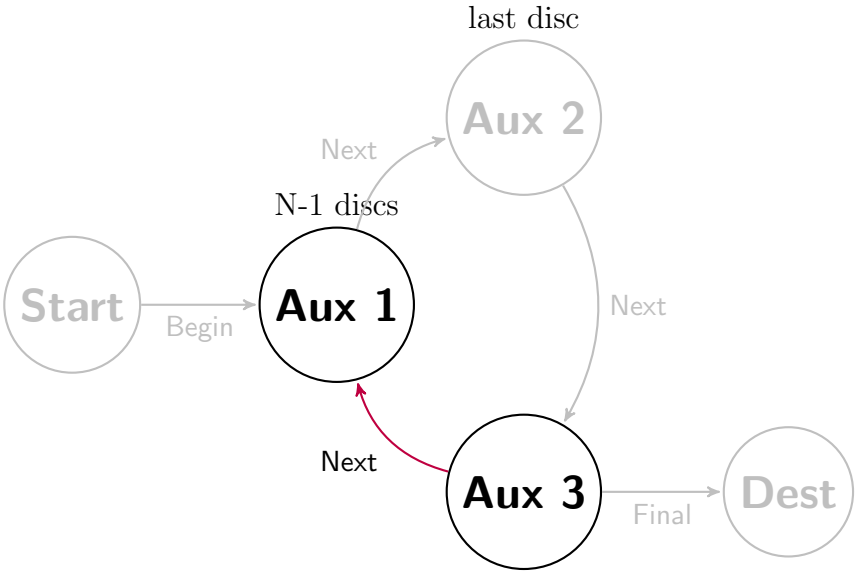
Our goal is to move a stack of N number of discs from Start to Dest. In order to move N number of discs from Start to Dest we must move (N-1) discs to access the last disc. This can be achieved by the following recursive pattern:



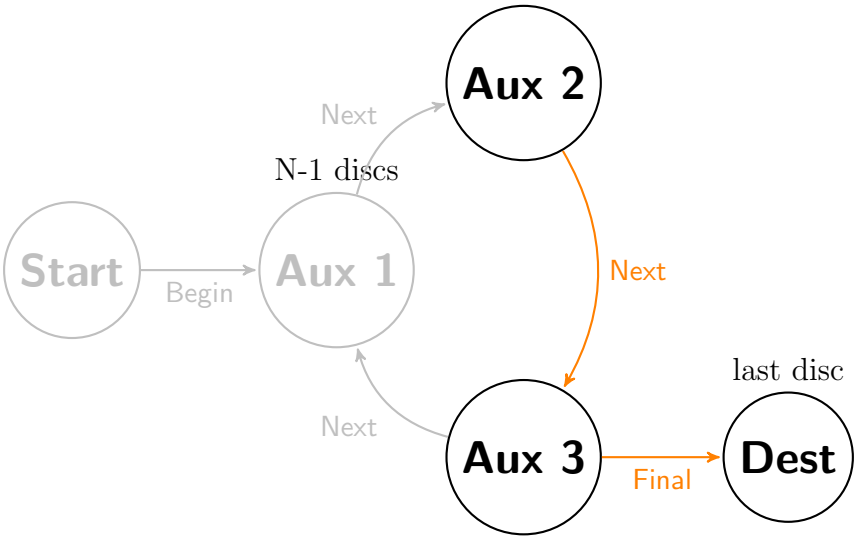
(1) *Figure 1.b* Move (N-1) discs from Start to Aux 3.
This will take T(N-1) operations.



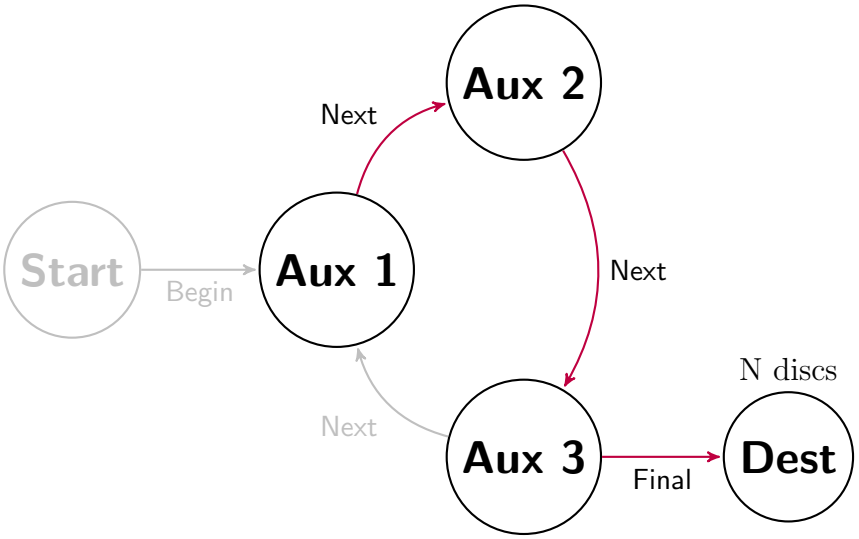
(2) *Figure 1.c* Move the last disc from Start to Aux 2.
This will take a T(1) operation.



(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:

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

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:

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

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){
  if(N = 1)
    if the amount of discs to move is one then take the path:
      A $\xrightarrow{N}$B, B $\xrightarrow{N}$C, C $\xrightarrow{N}$D, D $\xrightarrow{N}$E 4 Atomic Actions = $\theta(4)$
  }if(N > 1) {
    if N is greater than 1 then move then take the path:
    Move N-1 discs from A to D.
    H3(N-1, A, B, C, D, E) 1^{st} 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) 2^{nd} 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) 3^{rd} 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 = 1)
    A $\xrightarrow{K}$B, B $\xrightarrow{K}$C, C $\xrightarrow{K}$D — 3 Atomic Actions = $\theta(3)$
  if(K > 1)
    H3(K-1 ,A, B, C, D, E) — 1^{st} 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) — 2^{nd} Recursive Invocation $\theta(K-1)$
    C$\xrightarrow{K}$D — 1 Atomic Action = $\theta(1)$
    H2(K-1, E, A, B, C, D) — 3^{rd} 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 = 1)
    A $\xrightarrow{L}$B, B $\xrightarrow{L}$C — 2 Atomic Actions = $\theta(2)$
  if(K > 1)
    H2(L-1, A, B, C, D, E) — 1^{st} Recursive Invocation $\theta(L-1)$
    A$\xrightarrow{L}$B — 1 Atomic Action = $\theta(1)$
    H1(L-1, C, A, B, D, E) — 2^{nd} Recursive Invocation $\theta(L-1)$
    B$\xrightarrow{L}$C — 1 Atomic Action = $\theta(1)$
    H2(L-1, A, B, C, D, E) — 3^{rd} 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 = 1)
    A $\xrightarrow{M}$B 1 Atomic Action = $\theta(1)$
  if(M > 1)
    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

```
// Create the 5 Vertices
// new Stack(7, "demo") Create a stack of size 7 with the name "demo."
s = new Node(new Stack(n, "Start"));
a1 = new Node(new Stack(n, "Aux1"));
a2 = new Node(new Stack(n, "Aux2"));
a3 = new Node(new Stack(n, "Aux3"));
d = new Node(new Stack(n, "Dest"));

// Set the next edges
s->next = a1;
a1->next = a2;
a2->next = a3;
a3->next = a1;

// Set the dest edge
// Only use the dest after all of the discs are off of the Start disc.
a3->dest = d;
```

We implemented a Graph and Node class to store discs accordingly. Each node contains three pointers to stacks of discs. One stack is the current peg, and the other two pointers are references to edges in the graph. There is a pointer to the next peg and a pointer to the dest edge. If the peg does not connect to the final tower then the dest pointer will be empty. All other edges in the graph are considered to be next edges.

```
void H1(int n, Node * begin, Node * aux, Node * end) {
    if(n == 1)
        moveNext(begin, end);
    else if(n >= 2) {
        H2(n-1, begin, end, aux);
        moveNext(begin, end);
        H2(n-1, aux, begin, end);
    } recursiveCalls++;
}
```

```
void H2(int n, Node * begin, Node * aux, Node * end) {
    if(n == 1)
        moveNext(begin, end);
    else if(n >= 2) {
        H2(n-1, begin, aux, end);
        moveNext(begin, aux);
        H1(n-1, end, aux, begin);
        moveNext(aux, end);
        H2(n-1, begin, aux, end);
    } recursiveCalls++;
}
```

The H2 and H1 algorithms are implemented almost verbatim as our formal definition on pages 6 and 7 respectively. The H4 and H3 algorithms had to be implemented in two modified versions so that specific pointers to a specific stack would be used.

```
void Hanoi1(int n) {
    if(n >= 1) {
        Hanoi1(n-1);

        moveNext(s, a2);
        H1(n-1, a3, a2, a1);
        moveNext(a2,a3);

        if(!s->v->isEmpty())          // If the Start stack is empty
            H2(n-1, a1, a2, a3);      // Figure 1b Part 1
                                      // Remake the stack on Aux3
    } recursiveCalls++;
}
```

```
void Hanoi2(int n) {
    if(n == 0)
        moveNext(a3,d);
    if(n >= 1) {
        // Figure 1e - Part 2
        // The previous recursive invocation sets this move up
        // Park the largest disc on the Dest peg
        moveNext(a3, d);

        // Figure 1b - Part 2
        // Remake the N-1 stack on Aux 3
        // Move the N-1 stack onto A3
        H2(n-1, a1, a2, a3);

        // Move the penultimate disc onto A2
        // Figure 1c
        moveNext(a1, a2);

        // Figure 1d
        H1(n-1, a3, a2, a1);

        // Figure 1e - Part 1
        // This parks the disc in a spot ready to be used in the next invocatio
        // Park the largest disc on the peg before Dest
        moveNext(a2, a3);

        // Figure 1f
        Hanoi2(n-1);
    } recursiveCalls++;
}
```

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.