HW 1

Charles Liu
Advanced Data Structures - Dr. Subu Kandaswamy
February 10th 2025

---

1. (4 points) Using induction, prove that the number of nodes at level k of a complete binary tree is exactly 2^k.
   (Note: a complete binary tree is a binary tree where each level is saturated with nodes.)
   (Note: This is a simple problem to test students knowledge of basic induction)

Theorem. The number of nodes at level k of a complete binary tree is exactly 2^k.
Proof.
Base Case: Root node
Let $k = 0$. This means we are at the root node of the binary tree. At this level, there is 1 node. This satisfies our condition that the number of nodes at level k is 2^k, as $2^k = 2^0 = 1$

N-case: Level n
In a binary tree, suppose we are at level n. This means $k = n$. We will assume that the number of nodes at this level is $2^k = 2^n$

N+1-case: Level $n + 1$
Suppose we are at level $k = n + 1$. We want to show that the number of nodes here is $2^k = 2^{n+1}$. We know that for each node at the previous level, there is two child nodes (property of binary tree). As such, the number of nodes at $k = n + 1$ is $2 \times 2^n = 2^{n+1}$, so our assumption is again satisfied.

Through mathematical induction, the number of nodes at level k is 2^k.

---

2. (5 points) Write a simple recursive function to calculate (and return) the height of a binary
   tree T. You should assume that T is pointing to the root of the tree initially at input time. The height of a tree T is defined as the number of levels below the root. In other

words, it is
equal to the length of the longest path from the root to any leaf (i.e., number of edges along
the path from the root to the deepest leaf). Note that the term “nodes” is used to include both internal nodes and leaf nodes. You can assume the following tree node structure:

```
class Node {
Node *left; // points to the left subtree
Node *right; // points to the right subtree
}
```

Your answer can be in C++ syntax or in the form of a generic pseudocode (preferred). Note: that nothing has been specified about this being a complete binary tree or if it is balanced in height, etc. So please don’t make such assumptions. All that matters is that it is a binary tree.

```
int getHeight(Node* T) {
    //return when tree is empty
    if (T == nullptr) {
        return 0;
    }
    //when there is still nodes
    //return one added to the max of height of left and right
    return std::max(getHeight(T->getLeft()), getHeight(T->getRight)) + 1;
}
```

---

3. (5 points)
Solve the following mathematical recurrence. Show all steps (see lecture slides for examples)

$$T(n) = \begin{cases} 3, & n = 1 \\ 3T(n/3) + n, & n > 1 \text{ and } n \text{ is a power of } 3 \end{cases}$$

When $n > 1$,
$T(n) = 3T(n/3) + n$
$= 3(3T(n/9) + n/3) + n$
$= 3^2T(n/3^2) + 2n$
$= 3^2(3T(n/3^3)) + n/3^2) + 2n$

$= 3^3T(n/3^3) + 3n$
$= 3^kT(n/3^k) + kn$ where $k = \log_3(n)$.

$\implies T(n) = 3^{\log_3(n)} \times T(n/3^{\log_3(n)}) + \log_3(n)n$
$= n \times T(1) + n\log_3(n)$
$= 3n + n\log_3(n)$

Thus, $T(n) = 3n + n\log_3(n)$.

---

4. (6 points) `C++` has evolved significantly from its 1990s versions (e.g., `C++98`) to modern `C++`
(`C++11, C++14, C++17`). Among these, `C++11` introduced major changes that shaped the
language for modern programming.

a) List and briefly explain two major improvements in `C++11` that made it a significant upgrade over older versions. (2 points)

1. For-loops which were *range based* so iterating through elements is easier.
2. *Lambda expressions* which could be anonymously defined directly in code allowed more conciseness and flexibility

b) Provide a short `C++11` code snippet (around 3-5 lines) demonstrating one of these improvements and explain how it improves upon older `C++`. (2 points)

```
std::vector<int> num = {4,5,6}
for (int i:numbers) {
    std::cout << num[i];
}
```

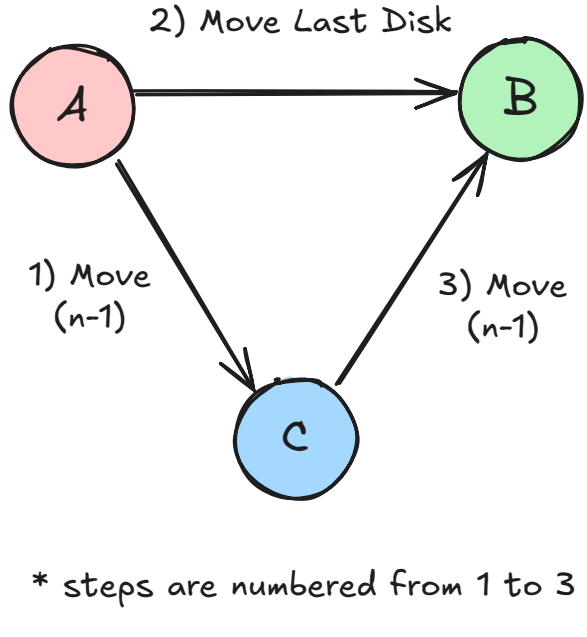
Range-based for loops allowed iterating through elements in containers easier.

c) Mention one key enhancement from `C++14` or `C++17` and how it builds on `C++11`. (2 points)

One key enhancement was structured bindings which built on the `auto` keyword in `C++11`, which deduced variable types. Structured bindings allowed variable deduction in a single line without manual iteration.

5. (10 points)

We discussed an algorithm to solve the Tower of Hanoi problem in class. The approach we
discussed for optimally moving n disks from peg A to peg B using peg C can be illustrated as shown below:



Therefore, the number of moves in Move(n) is:

$$2 \times \text{Move(n-1)} + 1$$

Now consider the following variant of the Tower of Hanoi problem. To the original problem, let us add a new condition: Each disk move now must be performed only in the clockwise direction — ie., $A \rightarrow B$, $B \rightarrow C$, or $C \rightarrow A$. So, for example, if I have to move a disk from A to C then that will involve a minimum of two moves (first, from A to B, and then from B to C); but from C to A needs only one move. All the old conditions from the original version of the problem still apply (i.e., you cannot move a larger disk on top of a smaller disk).

Under this new problem formulation:
6) Move last disk
7) Move $(n-1)$
8) Move $(n-1)$

• Let $Q(n)$ denote the minimum number of moves required to transfer n disks from $A \to B$ using C.
• Let $R(n)$ denote the minimum number of moves to transfer n disks from $A \to C$ clockwise using B.
Prove that:

$$(1) \quad Q(n) = \begin{cases} 1 & n = 1 \\ 2R(n-1) + 1 & n > 1 \end{cases}$$

$$(2) \quad R(n) = \begin{cases} 2 & n = 1 \\ Q(n) + Q(n-1) + 1 & n > 1 \end{cases}$$

Hint 1: To provide the proof you may need to first come up with a corresponding algorithm under the new clockwise condition. That should give you the mathematical recurrence shown
above. Please illustrate your algorithm in the form of a figure—following the example of the
figure shown above in the question, for the original version of the problem.

Hint 2: If $R(n)$ is the number of moves required to move n disks from peg A to peg C (using B), then how many moves will be needed to move n disks from peg B to peg A (using C)?

Hint 3: The idea remains the same. Move $N-1$ disks to helper-peg, Move last peg to target and finally, move $N-1$ disks to the target.

Theorem. (Variation on Tower of Hanoi)

$$(1) \quad Q(n) = \begin{cases} 1 & n = 1 \\ 2R(n-1) + 1 & n > 1 \end{cases}$$

$$(2) \quad R(n) = \begin{cases} 2 & n = 1 \\ Q(n) + Q(n-1) + 1 & n > 1 \end{cases}$$

where $Q(n)$ denotes the minimum number of moves required to transfer n disks from $A \to B$ using C, and $R(n)$ denote the minimum number of moves to transfer n disks from $A \to C$ clockwise using B.

(1)
Case 1: $n = 1$ disk

We move the disk once from A to B (only move), requiring one move. Thus,

$$Q(1) = 1$$

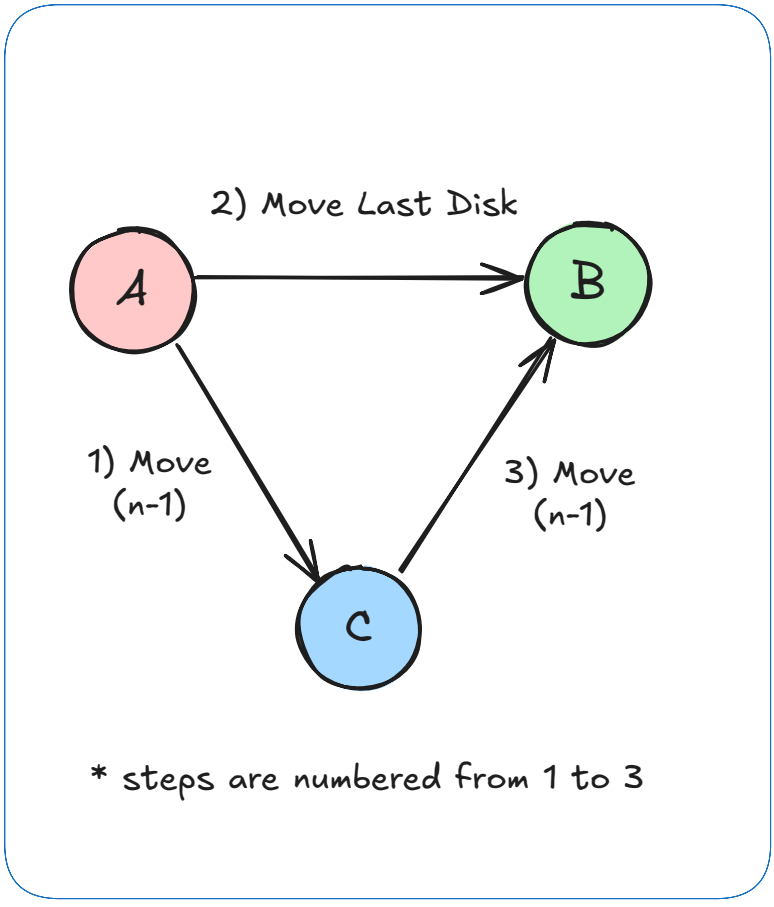
Case 2: $n > 1$ disks

The optimal way to move n disks from A to B is as follows:

9. Move $n - 1$ disks to the helper peg C
10. Move the largest disk to destination B
11. Move $n - 1$ disks to B.

Reasoning

We need to move the biggest disk to C, but to do so we must move all $n - 1$ disks above it. These $n - 1$ disks cannot go to B as they would block the path of the biggest disk, so they must go to C. After moving the biggest disk to B, the $n - 1$ disks must also reach the destination B to complete the tower.



Since we are moving $n - 1$ disks from A to C, $R(n - 1)$ moves are required. We then move the largest disk to B, optimally requiring 1 move. Since clockwise movement cycles, moving $n - 1$ disks from C to B optimally will take an equivalent number of moves as moving $n - 1$ disks from A to C optimally, $R(n - 1)$.

Thus,

$$Q(n) = 2R(n-1) + 1$$

(2)

Case 1: $n = 1$ disk

We move the disk once from A to B (only move), then from B to C (only move), requiring two moves. Thus,

$$R(2) = 2$$

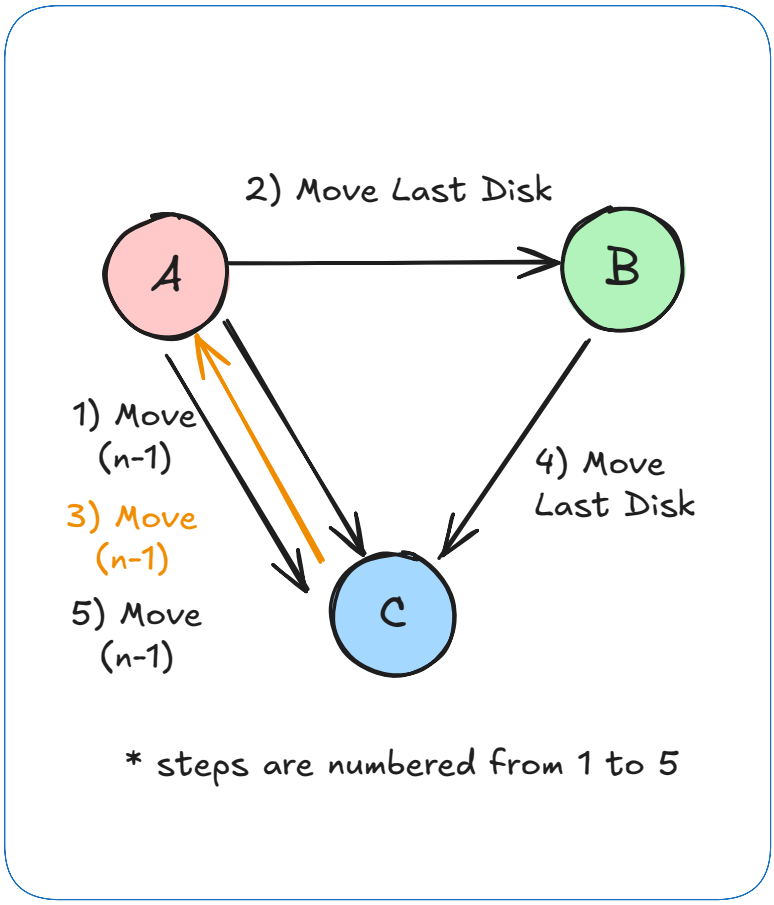
Case 2: $n > 1$ disk

The optimal way to move n disks from A to C is as follows:

12. Move $n-1$ disks from A to C
13. Move the biggest disk from A to B
14. Move $n-1$ disks from C to A
15. Move the biggest disk from B to C
16. Move $n-1$ disks from A to C.

Reasoning

We need to get the biggest disk from A to C (and thus, it must go through B first). Before moving the biggest disk, however, all $n-1$ disks above must be relocated. They cannot go to B, as they will block the path of the biggest disk, so they must go to C. After the biggest disk goes from A to B, the $n-1$ disks must again relocate for the biggest disk to progress. They cannot go to B, as it would prevent the biggest disk from moving, so they must go to A. After the biggest disk moves to C, the $n-1$ disks at A are also moved to the destination C.



We need to first move $n-1$ disks from A to C. This will optimally take $R(n-1)$ moves. We will then move the biggest disk from A to B, optimally taking 1 move. We then move $n-1$ disks from C to A. Since clockwise movement cycles, moving $n-1$ disks from C to A optimally will take an equivalent number of moves as moving $n-1$ disks from A to B optimally, $Q(n-1)$. We then move the biggest disk from B to C, completing its movement and optimally taking 1 move. Finally, we move $n-1$ disks from A to C, optimally taking $R(n-1)$ moves.

Taking the summation of the optimal moves in the process, we get

$$R(n) = R(n-1) + 1 + Q(n-1) + 1 + R(n-1) = 2R(n-1) + 1 + Q(n) + 1$$

.

By (1), we observe that $2R(n-1) + 1 = Q(n)$, so we get

$$R(n) = Q(n) + Q(n-1) + 1$$