Towers of Hanoi

- Given 3 pegs where first peg contains $n$ disks
- Goal: Move all the disks to a different peg (e.g., second one)
- Rule 1: Larger disks cannot rest on top of smaller disks
- Rule 2: Can only move the top-most disk at a time
- What is a recursive algorithm for solving this problem?
- Question: How many steps does it take to move all $n$ disks?

Towers of Hanoi, cont.

- Recurrence relation:
- Initial condition:
- Now find closed form for $T_n$
- What is a particular solution?
- Solution for homogeneous recurrence:
- Solve for $\alpha$:
- Solution for recurrence:

Example I: Binary Search

- Problem: Given sorted array of integers, is $i$ in the array?
- Binary search algorithm:
  1. Compare $i$ with middle element $m$ of array
  2. If $i > m$, then recursively search right half
  3. Otherwise, recursively search left half
- Classic divide-and-conquer algorithm

Binary Search, cont.

- Question: What is the worst-case complexity of binary search?
- Let $T(n)$ denote # of steps taken on input array of size $n$
- Write recurrence relation for $T(n)$:
- Initial condition:
- How do we get a Big-O estimate from this recurrence?
- Idea: Solve the recurrence and then find Big-O estimate for it
Solving Recurrence for Binary Search

\[ T(n) = T\left(\frac{n}{2}\right) + 1 \quad T(1) = 1 \]

- Not in a form we can immediately solve, but can massage it!
- Let \( n = 2^k \): \( T(2^k) = T(2^{k-1}) + 1 \)
- Now, let \( a_k = T(2^k) \): \( a_k = a_{k-1} + 1 \quad a_0 = 1 \)
- What’s the solution for this recurrence?
- Since \( n = 2^k \), this implies \( T(n) = \log_2 n + 1 \)
- Hence, complexity of binary search: \( \Theta(\log n) \)

Summary

- Recurrence relations for divide-conquer algorithms look like:
  \[ T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \]
- These are called divide-and-conquer recurrence relations
- To determine complexity of a divide-and conquer algorithm:
  1. Write corresponding recurrence relation
  2. Solve it exactly
  3. Obtain \( \Theta \) estimate
- Can we obtain a \( \Theta \) estimate without solving recurrence exactly?

Example II: Merge Sort

- Problem: Sort elements in array
- Merge sort solution:
  1. Recursively sort left half of array
  2. Recursively sort right half of array
  3. Merge the two sorted arrays

Recurrence Relation for Merge Sort

What is worst-case complexity of Merge Sort?

Let \( T(n) \) be \# operations performed to sort array of length \( n \)

- What is a recurrence relation for \( T(n) \)?
- As before, let \( n = 2^k \):

How to Merge Two Sorted Arrays?

- Input: Two sorted arrays \( A_1, A_2 \)
- Output: New sorted array that includes all elements in \( A_1, A_2 \)
- Idea: Pointers to current elements in \( A_1, A_2 \) (initially first)
- Copy smaller element to output array and advance pointer
- If combined size of \( A_1, A_2 \) is \( n \), merging takes \( 4n \) steps (compare, advance two pointers, copy)

Solving Recurrence Relation

\[ a_k = 2 \cdot a_{k-1} + 4 \cdot 2^k \quad a_0 = 1 \]

- Particular solution form:
- Particular solution:
- Solution for homogeneous recurrence:
  - Solve for \( \alpha \): \( \alpha \cdot 2^0 + 0 \cdot 2^2 = 1 \Rightarrow \alpha = 1 \)
  - Solution:
    - Plug in \( k = \log_2 n \):
    - Hence, algorithm is \( \Theta(n \cdot \log n) \)
The Master Theorem

Consider the recurrence \( T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d \) where \( a, c \geq 1 \), \( d \geq 0 \), and \( b > 1 \). Then:

1. \( T(n) = \Theta(n^d) \) if \( a < b^d \)
2. \( T(n) = \Theta(n^d \log n) \) if \( a = b^d \)
3. \( T(n) = \Theta(n^{\log_b a}) \) if \( a > b^d \)

Revisiting Examples

- Example 1: Recurrence for binary search: \( T(n) = T\left(\frac{n}{2}\right) + 1 \)
- Here, \( a = 1, b = 2, d = 0 \), Hence \( a = b^d \)
- By Case 2 of Master Thm, \( T(n) = \Theta(n^0 \log n) = \Theta(\log n) \)

- Example 2: Recurrence for merge sort: \( T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 4n \)
- Here, \( a = 2, b = 2, d = 1 \), Hence \( a = b^d \)
- By Case 2 of Master Thm, \( T(n) = \Theta(n \cdot \log n) \)

Proof of Master Theorem

- Example 3: Consider recurrence \( T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 3 \)
- At every level of recursion, \# subproblems multiplied by \( a \)
- But size of subproblem divided by \( b \)
- Let \( f(n) = c \cdot n^d \)

- Example 4: Consider recurrence \( T(n) = T\left(\frac{n}{2}\right) + n^2 \)
- Total amount of work:

\[
T(n) = \Theta(n \log n) + \sum_{i=0}^{\log_b n - 1} a^i \cdot c \cdot \left(\frac{n}{b^i}\right)^d
\]

- Can be rewritten as:

\[
T(n) = \Theta(n \log n) + \sum_{i=0}^{\log_b n - 1} c \cdot \left(\frac{n}{b^i}\right)^d
\]
Proof of Master Theorem, cont.

\[ T(n) = \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b n-1} c \cdot \left( \frac{a}{b} \right)^i \cdot n^d \]

- **Case 1:** \( \frac{a}{b^d} < 1 \). In this case, \( T(n) \) is of the form:
  
  \[ T(n) = \Theta(n^{\log_b a}) + c \cdot n^d \cdot \sum_{i=0}^{\log_b n-1} r^i \text{ for } |r| < 1 \]

- Hence: \( T(n) = \Theta(n^{\log_b a}) + \Theta(n^d) \)

- Since \( \frac{a}{b^d} < 1 \), we have \( \log_b a - d < 1 \). Thus \( T(n) = \Theta(n^d) \)

Proof of Master Theorem, cont.

\[ T(n) = \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b n-1} c \cdot \left( \frac{a}{b} \right)^i \cdot n^d \]

- **Case 2:** \( a = b^d \). In this case, \( T(n) \) is of the form:
  
  \[ T(n) = \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b n-1} c \cdot n^d \]

- Hence: \( T(n) = \Theta(n^{\log_b a}) + \Theta(n^d \cdot \log_b n) \)

- Since \( n^{\log_b n} = n^d \), this is \( \Theta(n^d \cdot \log_b n) \)

Proof of Master Theorem, cont.

\[ T(n) = \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b n-1} c \cdot \left( \frac{a}{b} \right)^i \cdot n^d \]

- **Case 3:** \( a > b^d \). In this case, \( n^{\log_b a} > n^d \).

- Use closed formula for geometric series to expand summation:
  
  \[ c \cdot n^d \cdot \frac{1 - \left( \frac{a}{b^d} \right)^{\log_b n-1}}{1 - \frac{a}{b^d}} \]

- This can be rewritten to \( c' \left( a^{\log_b n} - n^d \right) \) for some constant \( c' \)

- Since \( a^{\log_b n} = n^{\log_b a} \), \( T(n) \) is \( \Theta(n^{\log_b a}) \)