Divide-and-Conquer Algorithms

Divide-and-conquer algorithms are recursive algorithms that:

1. Divide problem into k smaller subproblems of the same form
2. Solve the subproblems
3. Conquer the original problem by combining solutions of subproblems

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 \( a_k = T(2^k) \): \( a_k = a_{k-1} + 1 \) \( a_0 = 1 \)
- What's the solution for this recurrence?
- Since \( n = 2^k \), this implies \( T(n) = \log_2 n + 1 \)
- Hence, complexity of binary search: \( \Theta(\log n) \)

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
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)

### 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 $n = 2^k$:

### The Master Theorem

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

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

### Summary

- Recurrence relations for divide-and-conquer algorithms look like:
  
  $T(n) = a \cdot T(\frac{n}{b}) + f(n)$

- 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?

### Revisiting Examples

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

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

- Example 3: Consider recurrence \( T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 3 \)
- Example 4: Consider recurrence \( T(n) = T\left(\frac{n}{2}\right) + n^2 \)

Why is the Master Theorem True?

Consider the recurrence \( T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d \)

- At every level of recursion, \# subproblems multiplied by \( a \)
- But size of subproblem divided by \( b \)
- Let \( f(n) \) be \( c \cdot n^d \)

Proof of Master Theorem

- What is the height \( h \) of this tree?
- Since problem size is 1 in base case, \( \frac{n}{b} = 1 \Rightarrow h = \log_b n \)
- At the \( i \)'th level, we have \( a^i \) subproblems, hence \( a^{\log_a n} \) leaves
- Equal to \( n^{\log_a n} \) — verify by taking \( \log_a \) of both sides

Proof of Master Theorem, cont.

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

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

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

Proof of Master Theorem, cont.
Proof of Master Theorem, cont.

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

- **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 \( \Theta(n^{\log_b a}) \).