CSCI 4470 Algorithms

Part I Foundations

- 1 The Role of Algorithms in Computing
- 2 Getting Started
- 3 Characterizing Running Times
- **4 Divide-and-Conquer**
- 5 Probabilistic Analysis and Randomized Algorithms

Chapter 4 Divide-and-Conquer

```
4 Divide-and-Conquer
  4.1 Multiplying square matrices
  4.2 Strassen's algorithm for matrix multiplication
  4.3 The substitution method for solving recurrences
  4.4 The recursion-tree method for solving recurrences
  4.5 The master method for solving recurrences
  4.6 Proof of the continuous master theorem
  4.7 Akra-Bazzi recurrences
```

Recurrence

- A ***RECURRENCE*** is a function is defined in term of
  - one or more base cases, and itself, with smaller arguments

METHODS FOR SOLVING RECURRENCES

Determine Asymptotic Behavior:

1. Substitution Method:

The **Substitution Method** is a technique employed to solve recurrence relations. The approach involves hypothesizing a closed form solution and subsequently validating its accuracy.

Steps for the Substitution Method

1. **Initial Guess**: Start by making a guess for the solution in the form of a **Closed Form Solution**.

2. **Proof of Correctness**: Prove the correctness of the guess by using mathematical induction or another suitable proof technique.
3. **Substitution**: Substitute the guessed solution into the recurrence equation.
4. **Verification**: Simplify the equation and verify if it matches the guessed solution.
5. **Adjustment**: If the equation matches the guessed solution, the guess is correct and the asymptotic behavior of the recurrence is determined. If not, adjust the guess and repeat the steps.

```
The substitution method for solving recurrences consists of two steps:
1 Guess the form of the solution.
2 Use mathematical induction to find constants in the
form and show that the solution works.
```

2. Recursion Tree Method

The Recursion Tree Method: This method involves representing the recurrence as a tree, where each node represents the costs (time) of a subproblem. By summing up the costs at each level of the tree, you can obtain the total cost and determine the asymptotic behavior.

- The detailed steps of the recursion tree method are:
  i. **Draw the recurrence as a tree** Each node represents the cost of a subproblem.
  ii. **Calculate the cost of each level** Sum up the costs of all nodes at each level.
  iii. **Summarize the total cost of the tree** Add up the costs of all levels.
  iv. **Determine the asymptotic behavior** Based on the total cost of the tree, determine the asymptotic behavior of the recurrence.

3. The Master Method

- **Master Theorem:** a powerful tool for solving recurrences of a specific form. It provides a formula to directly determine the asymptotic behavior based on the coefficients of the recurrence equation.
- Where $a \geq 1, b > 1$ are constants, and $f(n) > 0$.
- The Master Method introduces three cases:
  Case 1: If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, $f(n)$ is polynomially smaller than $n^{log_b(a)}$ then:
  Case 2: If $f(n) = \Theta(n^{\log_b a} \log^k n)$ where $k \geq 0$, $f(n)$ is within a polylog factor of $n^{log_b(a)}$, but not smaller then:
  情况 2: 若 $f(n) = \Theta(n^{\log_b a} \log^k n)$ 其中 $k \geq 0$,

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$

  Case 2: If $f(n) = \Theta(n^{\log_b a})$, where $k \geq 0$, $f(n)$ is within a polylog factor of $n^{log_b(a)}$, but not smaller then:
  Case 3: If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$ and $f(n)$ satisfies the regularity condition $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, $f(n)$ is polynomially greater than $n^{log_b(a)}$ then:

3.1 The Simplified Master Method

- Let $T(n)$ be a monotonically increasing function that satisfies
- Assumptions: $T(n) = aT(\frac{n}{b}) + cn^k$, then $T(1) = constant$
- where $a \geq 1$, $b \geq 2$, $c > 0$. if $f(n) \in \Theta(n^k)$ where $k \geq 0$, then

Case 1: if $a > b^k$, then $T(n) = \Theta(n^{\log_b(a)})$
Case 2: if $a = b^k$, then $T(n) = \Theta(n^k \log(n))$
Case 3: if $a < b^k$, then $T(n) = \Theta(n^k)$

$$T(n) = \begin{cases} \Theta(n^{log_b(a)}) & \text{if } a > b^k \\ \Theta(n^k \log(n)) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k \end{cases}$$

What is the running time of the merge algorithm?

- Use order of growth rather than exact accounting of each different line's timing
- Find an exact expression
  - i.e., Theta instead of big-O
- problem: How can we analyze this recursive algorithm?
  - Clearly recursive calls cannot count as a single step
- solution: Use recurrence equation
  - Let $T(n)$ be the running time of Merge-Sort for an input of size n

Simplifying assumptions:

- In general, we ignore floors and ceilings
- Also, we ignore the boundary (base case) condition
  - For our purposes, the base case is always a constant
- In most cases, these assumptions do not impact the analysis
  - Master method will clarify when these assumptions matter!

Divide and conquer Algorithms

Use Recursive Reasoning:

- Divide problem into sub-problems

```
     {5, 2, 4, 6, 1, 3, 2, 6}
           /               \
 {5, 2, 4, 6}           {1, 3, 2, 6}
     /       \            /       \
   {5, 2}   {4, 6}     {1, 3}   {2, 6}
   /   \      /   \     /   \     /   \
 {5} {2}   {4} {6}   {1} {3}   {2} {6}
```

- solve the sub-problems
- combine solutions

```
Merge Sort: {5, 2, 4, 6, 1, 3, 2, 6}

Divide:

            {5, 2, 4, 6, 1, 3, 2, 6}
           /                         \
   {5, 2, 4, 6}                  {1, 3, 2, 6}
   /          \                  /          \
{5, 2}       {4, 6}            {1, 3}    {2, 6}
  /  \         /  \             /  \       /  \
{5} {2}       {4} {6}         {1} {3}     {2} {6}

Conquer & Merge:

{2, 5}     {4, 6}           {1, 3}    {2, 6}
   \          /               \          /
   {2, 4, 5, 6}              {1, 2, 3, 6}
           \                         /
            {1, 2, 2, 3, 4, 5, 6, 6}
```

An Iterative Function Example

- To find the complexity of an iterative function

```
for(j= 0; j < n; j++) // O(n)
{
  i = n;
  temp = n;           // This could be the correct intent
  while(i > 0){
    temp = temp + 10; // Modifying temp instead of n
    i = i / 2;        // O(log(n))
  }
} // for
```

1. The outer loop runs n times.
2. Inside the loop, the `while` loop performs a halving operation on `i` which makes it run in logarithmic time with respect to n.
3. The combined complexity of these two loops, given that one is nested inside the other, is: $O(n \log n)$
4. Thus, the overall complexity of the function is $O(n \log n)$.

- the complexity of the code is $n \log(n)$

Factorial, The Classic Recursive Function Example

```
int fact(int n)  // T(n)
{
  if(n == 0) // base case O(1)
  {
    return 1;
  } else {
    return n * fact(n-1); // Recursive case T(n-1) always decease until 0
  }
}
```

1. The base case runs in constant time. **Complexity: $O(1)$**
2. For every value of `n`, the function makes one recursive call to itself with `n-1`. **Complexity: $T(n-1)$**

The recurrence relation provided is accurate:

$$T(n) = 1 + T(n-1)$$

Given the recurrence relation, we can unroll it to derive the time complexity:

$$\begin{aligned} T(n) &= 1 + T(n-1) \\ &= 1 + 1 + T(n-2) \\ &= 1 + 1 + 1 + T(n-3) \\ &\vdots \\ &= n + T(0) \end{aligned}$$

Since $T(0)$ is $O(1)$ and there are n operations (each taking constant time) leading up to it, the overall complexity is: $T(n) = O(n)$.

Merge-Sort Algorithm Example

- **Divide**: Divide the list into two separate lists. **(hence the $2T(\frac{n}{2})$).**
- **Conquer**: Sort each smaller list.
  - Use Merge-Sort recursively on the smaller lists.
- **Combine**: Combine the solutions to the sub-problems with linear work **(hence the $\Theta(n)$).**

```
Merge-Sort(A, p, r) // A is an Array, p, r can be index, or sub-array
if p < r     // Size = 1, get combining and sorting, ensure there is more than one element
q = ⌊(p+r)/2⌋ // find the midpoint
Merge-Sort(A, p, q) // recursively sort the left half
Merge-Sort(A, q+1, r) // recursively sort the right half
Merge(A, p, q, r) // merge the two sorted halves
```

Recurrence Relation for Merge-Sort():

$$T(n) = \begin{cases} n & \text{if } n \leq 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- The maximum number of comparisons when merging is n, denoted by $O(n)$.
- The minimum number of comparisons when merging is $\frac{n}{2}$, denoted by $\Omega(n)$.
- For The number of comparisons $\geq \frac{n}{2}$, it is in $\in \Omega(n)$

$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + \Theta(n)$ (This is another representation of the recurrence, taking into account the possible splitting of even and odd-sized arrays).

1. Dividing: We recursively split the array in half until reaching individual elements.
2. Merging: We combine these elements back into sorted arrays.

- $T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$ and $T\left(\left\lceil \frac{n}{2} \right\rceil\right)$ represent dividing the array.
- $\Theta(n)$ accounts for merging, as merging two sub-arrays of length n takes linear time.

MERGE SORTED LISTS

```
Merge (A, p, q. r)
/* merge sorted lists A[p ... q] and A[q+1 ... r] into A[p ... r), where p≤q<r */
n1=q-p+1
n2=r-q
create arrays L[1 ... n1 + 1] and R[1 ... n2 + 1] from A
copy A[p ... p+n1-1] to L[1 ... n1]
copy A[q+1 ... q+n2] to R[1 ... n2]
L[n1 + 1] = R[n2 + 1] = ∞
i=j=1
for k= p to r
  do if L[i] <= R[i]
     then A[k] = L[i]
         i=i+1
     else A[k] = R[i]
         j=j+1
```

- Why the infinity ∞?

```
n1+n2 |____|____|____|_____|_∞_| O(n1+n2)

combine n1+n2
n1 |____|____|_∞_|  n2 |____|____|_∞_|
```

- "Infinity" (`inf`) in `L` and `R` acts as a sentinel.
- It simplifies merging by preventing array overrun.
- It removes the need to check if an array is empty during merging.

Recurrence Relation for Merge-Sort() The Substitution Method Example

$$T(n) = \begin{cases} n & \text{if } n \leq 1 \\ 2T(\frac{n}{2}) + n & \text{if } n > 1 \end{cases}$$

Solving The Recurrence Relation by Substitution Method:

For $n > 1$, Guess: $T(n) = O(n\log(n))$
$T(n) = 2T(\frac{n}{2}) + n$ ... (1)

Find $T(\frac{n}{2})$:substitute n with $\frac{n}{2}$

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$

Substituting for $T\left(\frac{n}{2}\right)$, Assuming $T(n) \leq cn\log(n)$, then $T\left(\frac{n}{2}\right) \leq c\frac{n}{2}\log\left(\frac{n}{2}\right)$ ... (2)

Substitute (2) into (1): $T(n) \leq 2T(\frac{n}{2}) + \Theta(n)$
$T(n) \leq 2\left(c\frac{n}{2}\log\left(\frac{n}{2}\right)\right) + n$

Distribute the 2:
$T(n) \leq 2(c\cdot(\frac{n}{2}))\cdot log(\frac{n}{2}) + n$
$= c \cdot n \cdot log(\frac{n}{2}) + n$

Apply log property: $log(\frac{n}{2}) = log(n) - log(2)$, $log_2(2) = 1$
$\leq c \cdot n \cdot (log(n) - log(2)) + n$
$= c \cdot n \cdot log(n) - c \cdot n \cdot log(2) + n$

Simplify:
$T(n) = cn\log(n) - cn + n$

We started with the simplified recurrence:

$T(n) \leq cn\log(n) - cn + n$

Step 1) Distribute c on the second term:

$T(n) \leq cn\log(n) - c(n) + n$

Step 2) Factor out n:

$T(n) \leq cn\log(n) - n(c) + n$

Step 3) Apply the distributive property:

$T(n) \leq cn\log(n) - n(c-1)$

Given conditions: $log_2(2) = 1$, when $n \geq 2$
$c - 1 > 0$, $c > 1$, $n \geq 2$, the term $cn - n(c-1)$ is positive for all n. Thus, $T(n) = O(n\log(n))$

From the previous steps, we derived the term $cn - n(c-1)$. Since $c - 1 > 0$ and $c > 1$, $cn - n(c-1)$ will be positive for all $n \geq 2$.

The base case is true c is always constant. $c > 1$

$T(n) \leq cn\log(n)$, Which gives us the upper bound:

$T(n) \in O(n\log(n))$

For the base case $n = 1$:
$T(1) = 1 \leq c$ is true, because $c > 1$. Therefore, the lower bound is:

$T(n) = \Omega(n\log(n))$

Combining the upper and lower bounds gives:
$T(n) \in \Theta(nlog(n))$

In summary:
Derived $cn - n(c-1)$ from previous steps, Showed it's positive for $n \geq 2$ based on $c > 1$, Got upper bound $O(n\log(n))$, Showed base case is true giving lower bound $\Omega(n\log(n))$, Combined bounds to get solution $\Theta(n\log(n))$

Recurrence Relation for Merge-Sort()

The Simplified Master Method Example

Case 1: if $a > b^k$, then $T(n) = \Theta(n^{\log_b(a)})$
Case 2: if $a = b^k$, then $T(n) = \Theta(n^k \log(n))$
Case 3: if $a < b^k$, then $T(n) = \Theta(n^k)$

$T(n) = 2T(\frac{n}{2}) + n^k$

$a = 2$, $b = 2$, $k = 1$

Since $a = b^k$, $2 = 2^1$, Case 2 applies. **Case 2:** if $a = b^k$, then $T(n) = \Theta(n^k \log(n))$

Thus we conclude that

$$T(n) \in \Theta(n^k \log(n)) = \Theta(n^1 \log(n) = \Theta(n\log(n))$$

Recurrence Relation for Merge-Sort() Example

The Master Method Example

$$T(n) = \begin{cases} n & \text{if } n \leq 1 \\ 2T(\frac{n}{2}) + n & \text{if } n > 1 \end{cases}$$

Solving The Recurrence Relation by the Master Method:

1. The recurrence is in the form:

$$T(n) = 2T(\frac{n}{2}) + n$$

$$T(n) = aT(\frac{n}{b}) + f(n)$$

Where:

$$a = 2, b = 2, f(n) = n$$

- $f(n) = n$: This represents the work done outside the recursive calls, which in the case of Merge-Sort is the time taken to merge the two halves.

Identifying the case for our recurrence relation:

$f(n) = n$, and $n^{\log_b a} = n^{\log_2 2} = n$.

Since $f(n)$ matches with $n^{\log_b a}$, this puts our recurrence in **Case 2** of the Master Method.

According to **Case 2** of the Master Method:

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$

Here, $k = 0$. Thus:

$$T(n) = \Theta(n \log n)$$

Conclusion:
The time complexity of the Merge-Sort algorithm, based on the Master Theorem, is:

$$T(n) = \Theta(n \log n)$$

Case 1: if $a > b^k$, then $T(n) = \Theta(n^{\log_b(a)})$
Case 2: if $a = b^k$, then $T(n) = \Theta(n^k \log(n))$
Case 3: if $a < b^k$, then $T(n) = \Theta(n^k)$

Solving Using the Simplified Master Method

Given $T(n) = 4T\left(\frac{n}{2}\right) + n$,

Find a, b, k: $a = 4, b = 2, k = 1$

Since $a > b^k$, $4 > 2^1$, Case 1 applies, **Case 1:** if $a > b^k$, then $T(n) = \Theta(n^{\log_b(a)})$

Thus we conclude that

$$T(n) \in \Theta(n^{\log_b(a)}) = \Theta(n^{\log_2(4)}) = \Theta(n^2)$$

$\log_2(4) = \frac{\log(4)}{\log(2)} = \frac{2\log(2)}{\log(2)} = 2$

Solving Using the Master Method

Given the recurrence relation:

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

To apply the Master method, we can match the recurrence to the standard format:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where:

- $a = 4, b = 2, f(n) = n$

For the Master method, we need to compare $f(n)$ with $n^{\log_b a}$. In this case, $a = 4$ and $b = 2$, so:

$$n^{\log_b a} = n^{\log_2 4}$$

1. Express $\log_2(4)$ in terms of natural logarithms:

$$\log_2(4) = \frac{\ln(4)}{\ln(2)}$$

2. Evaluate the logarithm:

$$\log_2(4) = \frac{\ln(2^2)}{\ln(2)} = \frac{2\ln(2)}{\ln(2)} = 2$$

3. Substitute the value of $\log_2(4)$ into our expression:

$$n^{\log_2 4} = n^2$$

Given that $f(n) = n$ and $n^{\log_b a} = n^2$, and since $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, it falls under **Case 1** of the Master method. Therefore, the solution to the recurrence is:

$$T(n) = \Theta(n^2)$$

The Substitution Method

Given $T(n) = 4T\left(\frac{n}{2}\right) + n$,

To use the Master method, the recurrence can be matched to the format:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Assumption:
Assume $T(n) \leq cn^3$, where ‘c’ is a constant, n is real number, $n \geq 1$

Given:

$$T\left(\frac{n}{2}\right) \leq c\left(\frac{n}{2}\right)^3$$

Expanding:

$$T\left(\frac{n}{2}\right) \leq \frac{cn^3}{8}$$

Substituting this into our original equation:

$$T(n) \leq 4 \times \frac{cn^3}{8} + n$$

$$= \frac{cn^3}{2} + n$$

To prove the assumption, you want to show:

$$T(n) \leq cn^3$$

Subtracting $\frac{cn^3}{2} + n$ from both sides:

$$0 \leq \frac{cn^3}{2} - n$$

For our assumption to hold, $\frac{cn^3}{2} - n$ should be greater than 0, or equivalently, $\frac{cn^3}{2} > n$.

Conclusion:

The assumption can be supported if $\frac{cn^3}{2} > n$ for a sufficiently large value of n and some constant c. The solution seems to have a few inconsistencies and may need further exploration to be concrete.

Substitution Method Example

Given the recurrence relation:

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

with the base case:

$$T(1) = \Theta(1)$$

Guess: $T(n) = O(n^3)$

Inductive Hypothesis:
Assume $T(k) \leq ck^3$ for all $k < n$.

Proof:
To prove: $T(n) \leq cn^3$

$$\begin{aligned} T(n) &\leq 4T\left(\frac{n}{2}\right) + n \\ &\leq 4c\left(\frac{n}{2}\right)^3 + n \\ &= cn^3 - \left(\frac{c}{2}n^3 - n\right) \quad \text{(desired - residual)} \\ &\leq cn^3 \end{aligned}$$

This holds true whenever $\frac{c}{2}n^3 - n \geq 0$, for instance, when $c \geq 2$ and $n \geq 1$.

Base Case:
For $1 \leq n \leq n_0$, $T(n) = \Theta(1)$ which is $\leq cn^3$ if we choose a sufficiently large constant c.

Tighter Upper Bound:

We aim to prove: $T(n) = O(n^2)$

Inductive Hypothesis:
Assume $T(k) \leq ck^2$ for all $k < n$.

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{2}\right) + n \\ &\leq 4c\left(\frac{n}{2}\right)^2 + n \\ &= cn^2 + n \quad \text{(desired - residual)} \\ &\leq cn^2 \end{aligned}$$

However, this doesn't hold for any choice of c.

Idea: Strengthen the inductive hypothesis by subtracting a lower-order term.

New Inductive Hypothesis:
Assume $T(k) \leq c_1k^2 - c_2k$ for all $k < n$.

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{2}\right) + n \\ &= 4\left(c_1\left(\frac{n}{2}\right)^2 - c_2\left(\frac{n}{2}\right)\right) + n \\ &= c_1n^2 - 2c_2n + n \\ &= c_1n^2 - c_2n - (c_2n - n) \\ &\leq c_1n^2 - c_2n \quad \text{if } c_2 \geq 1 \end{aligned}$$

Case 1: if $a > b^k$, then $T(n) = \Theta(n^{\log_b(a)})$
Case 2: if $a = b^k$, then $T(n) = \Theta(n^k \log(n))$
Case 3: if $a < b^k$, then $T(n) = \Theta(n^k)$

Example 01

Let $T(n) = 1T(\frac{n}{2}) + \frac{1}{2}n^2 + n$. What are the parameters? a, b, k

$a = 1$, $b = 2$, $k = 2$

Since $a < b^k$,$1 < 2^2$, Case 3 applies. **Case 3:** if $a < b^k$, then $T(n) = \Theta(n^k)$

Thus we conclude that

$$T(n) \in \Theta(n^k) = \Theta(n^2)$$

Example 02

Let $T(n) = 2T(\frac{n}{4}) + \sqrt{n} + 42$. What are the parameters?

$a = 2$, $b = 4$, $k = \frac{1}{2}$

Since $a = b^k$, $2 = 4^{\frac{1}{2}}$, Case 2 applies. **Case 2:** if $a = b^k$, then $T(n) = \Theta(n^k \log(n))$

Thus we conclude that

$$T(n) \in \Theta(n^k \log(n)) = \Theta(\sqrt{n} \log(n))$$

Example 03

Let $T(n) = 3T(\frac{n}{2}) + \frac{3}{4}n + 1$. What are the parameters?

$a = 3$, $b = 2$, $k = 1$

Since $a > b^k$, $3 > 2^1$, Case 1 applies. **Case 1:** if $a > b^k$, then $T(n) = \Theta(n^{\log_b(a)})$

Thus we conclude that

$$T(n) \in \Theta(n^{\log_b(a)}) = \Theta(n^{log_2(3)})$$

Note that $\log_2(3) \approx 1.5849$

The Master Method Example 04

Case 3: If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$ and $f(n)$ satisfies the regularity condition $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, $f(n)$ is polynomially greater than $n^{log_b(a)}$ then:
情況 3: 若 $f(n) = \Omega(n^{\log_b a+\epsilon})$ 對於某個 $\epsilon > 0$ 和 $af(n/b) \leq kf(n)$ 對於某個 $k < 1$ 和足夠大的 n,

$$T(n) = \Theta(f(n))$$

Give that the Recurrence Relation: $T(n) = 3T(\frac{n}{2}) + n^2$

1. Identify Parameters:
   $a = 3$, $b = 2$, $f(n) = n^2$
2. Calculate Critical Exponent
   $n^{log_b(a)} = n^{log_2(3)} = n^{1.58}$

Applying Case 3 that

- pick a random constant like 0.1

$f(n) \geq n^{1.58}$
$f(n) \in \Omega(n^{log_2(3)+0.1})$
$f(n) \in \Omega(n^{1.6+0.1})$

For $c \leq 1$, $af(\frac{n}{b}) \leq cf(n)$, and $f(n) = n^2$

$3(\frac{n}{2})^2 \leq 2n^2)$

$\frac{3n^2}{4} \leq cn^2$, $c = \frac{3}{4} < 1$

Therefore, $T(n) \in \Theta(n^2)$

The Master Method Example 05

Give that the Recurrence Relation $T(n) = T(\frac{2n}{3}) + 1$

$a = 1$, $b = \frac{3}{2}$, $f(n) = 1$

- The log rule $log_a(1) = 0$

Applying Case 2 that $n^{log_{\frac{3}{2}}(1)} = n^0 = 1$

$f(n) = n^{log_b(a)}$
$T(n) \in \Theta(n^{log_b(a)})$

$f(n) = n^k$

$T(n) \in \Theta(n^{log_{-\frac{3}{2}}(1)} log^0(n))$
$T(n) \in \Theta(log(n))$

Absolutely, let's break down each step of the solution:

Master Method Example 05

Given the recurrence relation $T(n) = T\left(\frac{2n}{3}\right) + 1$

We have:

- $a = 1$ (the number of subproblems)
- $b = \frac{3}{2}$ (the factor by which subproblem size is reduced)
- $f(n) = 1$ (the cost of dividing the problem and combining the results)

Now, we apply the master theorem to find a tight bound on the recurrence relation.

1. **Identify the values of a, b, and $f(n)$**:
   - $a = 1, b = \frac{3}{2}, f(n) = 1$
2. **Find $n^{\log_b(a)}$**:
   - Using the formula $n^{\log *b(a)}$, we substitute the values of a and b we have:

$$n^{\log_{\frac{3}{2}}(1)} = n^0 = 1$$

3. **Apply the Master Theorem Case 2**:
   - Since $f(n)$ is a constant function and equals $n^{\log_b(a)}$, we apply case 2 of the master theorem, which gives:

$$T(n) \in \Theta(n^{\log_b(a)} \log^k n)$$

   where $k = 0$ (because $f(n) = n^{\log_b(a)} \cdot 1$)
4. **Find the solution**:
   - Now we substitute the values of $n^{\log *b(a)}$ and k into the formula to find the solution:

$$T(n) \in \Theta(n^{\log *\frac{3}{2}(1)} \log^0 n) = \Theta(\log n)$$

This is how we derive that $T(n)$ is in $\Theta(\log n)$ using the master theorem. Let me know if this helps!

Recursion Tree Example

The recurrence relation for Merge-Sort with the constant c is:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$

The recurrence relation for Merge-Sort with the constant c is:

```
              _________________________
             |           cn            | (Level 0) ---> cn
             |         /    \          |
             |     c(n/2)    c(n/2)    | (Level 1)  ---> cn
             |    /    \    /    \     |
    log(n+1) |c(n/4) ...    ...   ... | (Level 2)  ---> cn
             |   /    \   /    \   /  |
             |c(n/8) ...    ...   ... | (Level 3)  ---> cn
             |   /    \   /    \   /  |
             |c(n/16)...    ...   ... | (Level 4)  ---> cn
             |   /    \   /    \   /  |
             |c(n/32)...    ...   ... | (Level 5)  ---> cn
             |   |                     |
             |   c                     | (Base case) (Level c) ---> c
             |_________________________|
             |          n              |
```

- Based on the Recursion Tree, the Running time: $O(n\log(n))$

Recursion Tree Example 02

For the recurrence relation: $w(n) = 2w(\frac{n}{2}) + n^2$, When reached to LEVEL i, Get that $\frac{n}{2^i} = 1, 2^i = n, i = \log(n)$

```
          _________________________
         |          n^2            | (Level 1)
         |        /    \           |
height=  |  (n/2)^2    (n/2)^2     | (Level 2)
log(n)   |  /     \     /     \    |
         |(n/4)^2 ...    ...   ... | (Level 3)
         |   |      |     |    |   |
         |  w(1)   w(1) w(1) w(1)  | (Base case) (Level log(n))
         |_________________________|
```

Observations:

- The function decreases as we move down the tree, indicating that the size of the problem is changing.

Analysis:

1. **Subproblem Size at Level i**: The size of the subproblem at any level i is $\frac{n}{2^i}$.

2. **Number of Subproblems at Level i**: Since the problem is divided into two at each level, there are 2^i subproblems at level i.

3. **Cost at Level i**: The work done for each subproblem at level i is $\left(\frac{n}{2^i}\right)^2$. Thus, the total cost at level i is $2^i \times \left(\frac{n}{2^i}\right)^2 = \frac{n^2}{2^i}$.

4. **Base Case**: At the last level, where the problem size is 1 (i.e., $\frac{n}{2^i} = 1$), $i = \log(n)$. The cost at this level is $\frac{n^2}{2^{\log(n)}}$.

- **To calculate Cost of Each Level:**

Level	Cost	Term(s)		
L_0	n^2	1 term	$w(n)$	= $\frac{n}{2^0}$
L_1	$2 \cdot (\frac{n}{2})^2$	2 terms	$w(\frac{n}{2})$	=$\frac{n}{2^1}$
L_2	$4 \cdot (\frac{n}{4})^2$	4 terms	$w(\frac{n}{4})$	=$\frac{n}{2^2}$
L_3	$8 \cdot (\frac{n}{8})^2$	8 terms	$w(\frac{n}{8})$	=$\frac{n}{2^3}$
L_4	$16 \cdot (\frac{n}{16})^2$	16 terms	$w(\frac{n}{16})$	=$\frac{n}{2^4}$
L_i	$2^{log(n)}$	i terms	$w(1)$	=$\frac{n}{2^i}$

- The cost of the last level i, find out the size of the problem is at level i.
  - At each level of the recursion, the problem size is halved. So, at level i, the problem size is $\frac{n}{2^i}$.

Now, let's determine how many subproblems of size $\frac{n}{2^i}$ there are at level i. Since the problem is divided into two at each level, there will be 2^i subproblems at level i.

Now, for the cost:

1. The work done for each subproblem at level i is given by the non-recursive term in the recurrence, which is $\left(\frac{n}{2^i}\right)^2$ for each subproblem.
2. Since there are 2^i such subproblems at level i, the total work at this level is $2^i \times \left(\frac{n}{2^i}\right)^2$.

Simplifying:

$$2^i \times \left(\frac{n}{2^i}\right)^2 = 2^i \times \frac{n^2}{2^{2i}} = n^2 \times \frac{2^i}{2^{2i}} = n^2 \times \frac{1}{2^i}$$

So, the cost at level i is $\frac{n^2}{2^i}$.

For the last level, where the problem size is 1 (i.e., $\frac{n}{2^i} = 1$), $i = \log(n)$. Thus, the cost at the last level is $\frac{n^2}{2^{\log(n)}}$.

Total Cost:
The total cost of the recurrence is the sum of the costs at each level:

$$w(n) = \sum_{i=0}^{\log(n)-1} \frac{n^2}{2^i} + 2^{\log(n)}$$

Using the geometric series sum formula, we can simplify this to:

$$w(n) = n^2 \sum_{i=0}^{\log(n)-1} \left(\frac{1}{2}\right)^i + O(n)$$

Given that the sum of an infinite geometric series with a ratio r where $|r| < 1$ is $\frac{1}{1-r}$, the sum becomes:

$$w(n) = n^2 \left(\frac{1}{1-\frac{1}{2}}\right) + O(n) = 2n^2 + O(n)$$

- apply the fraction rule $\frac{1}{\frac{b}{c}} = \frac{c}{b}$
- $\left(\frac{1}{1-\frac{1}{2}}\right) = \left(\frac{1}{\frac{1}{2}}\right) = \frac{2}{1} = 2$

Thus, the solution to the recurrence is:

$$w(n) = O(n^2)$$

Break down the recurrence relation $w(n) = 2w\left(\frac{n}{2}\right) + n^2$ using a recursive tree.

Step 1: Understand the Recurrence.

The recurrence relation $w(n) = 2w\left(\frac{n}{2}\right) + n^2$ can be understood as:

- We divide the problem into 2 subproblems of size $\frac{n}{2}$.
- The cost of dividing and combining the solutions of the subproblems is n^2.

Step 2: Draw the First Level.

At the top level (level 1), the cost is n^2.

Step 3: Draw the Second Level.

The problem is divided into 2 subproblems of size $\frac{n}{2}$. Each subproblem has a cost of $\left(\frac{n}{2}\right)^2$.

Step 4: Draw the Third Level.

Each subproblem from the second level is further divided into 2 subproblems of size $\frac{n}{4}$. Each of these has a cost of $\left(\frac{n}{4}\right)^2$.

Step 5: Continue the Pattern.

Continue this pattern until you reach the base case. The base case is typically when $n = 1$, but it's not explicitly given in this recurrence. For simplicity, we'll assume $w(1) = 1$.

Step 6: Sum the Costs at Each Level.

To find the total cost, sum the costs at each level of the tree.

Recursive Tree:

```
          n^2                          ............. Level 1
        /     \
   (n/2)^2   (n/2)^2                   ............. Level 2
     /   \     /   \
  (n/4)^2 ... ... ... ...              ............. Level 3
     .
     .
     .
  w(1) w(1) w(1) ... w(1)            ............. Level log(n)
```

Step 7: Calculate the Total Cost.

- Level 1: n^2
- Level 2: $2 \times \left(\frac{n}{2}\right)^2 = n^2$
- Level 3: $4 \times \left(\frac{n}{4}\right)^2 = n^2$
- $\ldots$
- Level log(n): $n \times 1 = n$

The cost at each level is n^2, and there are log(n) levels. So, the total cost is $n^2 \times \log(n)$.

Conclusion:

The time complexity of $w(n) = 2w\left(\frac{n}{2}\right) + n^2$ is $O(n^2 \log n)$.

More Challenging Recursion Tree Example:

$$T(n) = T(\frac{n}{4}) + T(\frac{3n}{4}) + cn$$

Recursive Tree Example 03

Consider the recurrence relation $T(n) = 3T(\frac{n}{4}) + cn^2$ for some $constant\ c$. We assume that n is an exact power of 4.

- In the recursion-tree method we expand $T(n)$ into a tree:

```
T(n) -->  cn^2
          / | \
         /  |  \
        /   |   \
  T(n/4)  T(n/4)  T(n/4)
```

- Applying $T(n) = 3T(\frac{n}{4}) + cn^2$ to T (n/4) leads to $T(\frac{n}{4}) = 3T(\frac{n}{16}) + c(\frac{n}{4})^2$, expanding the leaves:

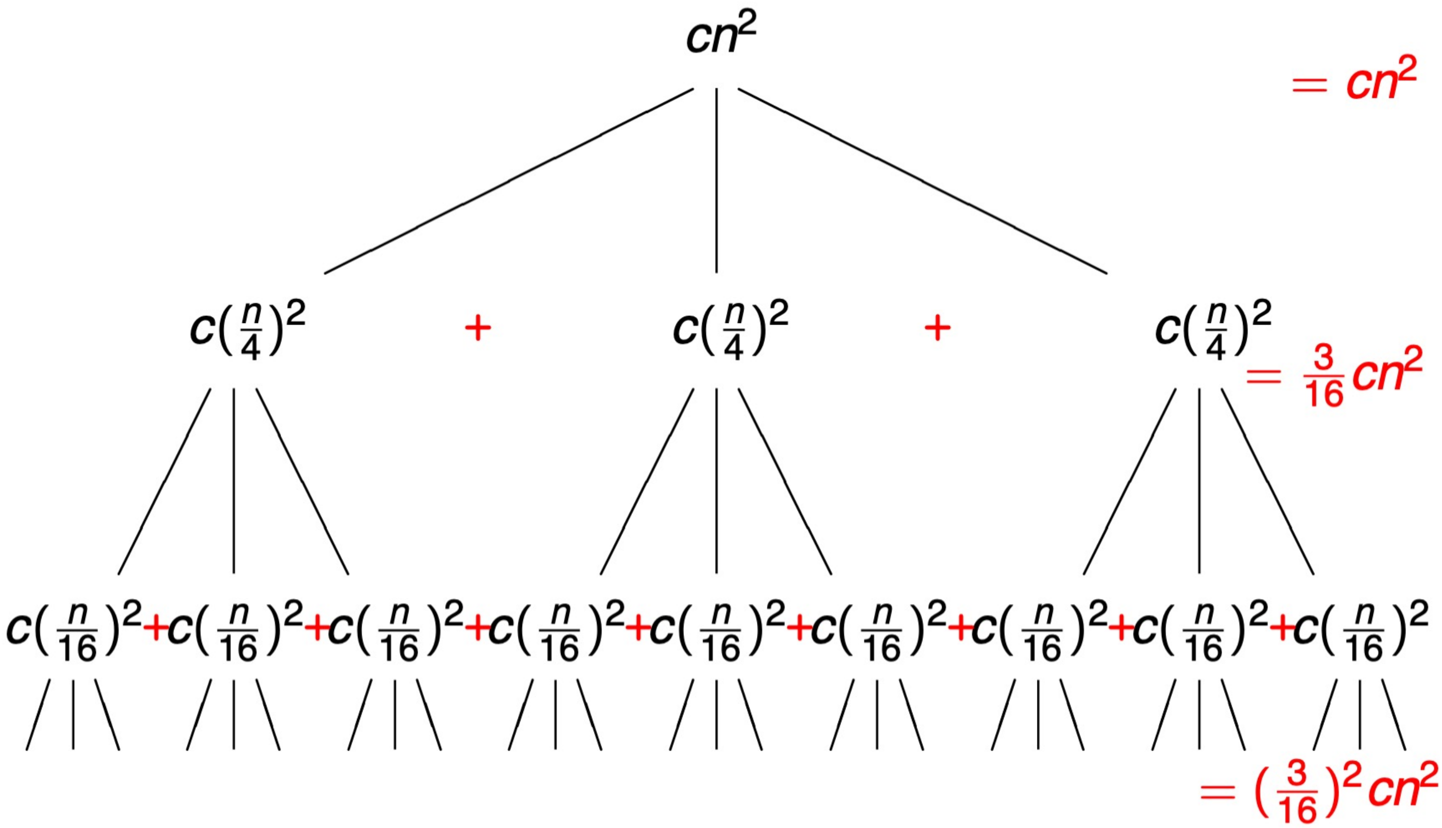
```
T(n) -->        cn^2
           /    |           \
          /     |            \
         /      |             \
        /       |              \
c(n/4)^2      c(n/4)^2      c(n/4)^2
    |             |              |
    |             |              |
3* T(n/16)    3* T(n/16)    3* T(n/16)
```

- Applying $T(n) = 3T(\frac{n}{4}) + cn^2$ to $T(\frac{n}{16})$ leads to $T(\frac{n}{16}) = 3T(\frac{n}{64}) + c(\frac{n}{16})^2$, expanding the leaves:

```
      T(n)
       |
     cn^2
   /   |   \
  /    |    \
 /     |     \
T(n/4) T(n/4) T(n/4)
 |      |      |
c(n/4)^2 c(n/4)^2 c(n/4)^2
   /   |   \    /   |   \    /   |   \
  /    |    \  /    |    \  /    |    \
 /     |     \/     |     \/     |     \
T(n/16) T(n/16) T(n/16) T(n/16) T(n/16) T(n/16) T(n/16) T(n/16) T(n/16)
 |      |      |      |      |      |      |      |      |
c(n/16)^2 ...  c(n/16)^2 ...  c(n/16)^2 ...  c(n/16)^2 ...
```

the cost at each level, We sum the cost at each level of the tree:

Level	Terms	Cost
L_0	cn^2	cn^2
L_1	$c(\frac{n}{4})^2$	$c(\frac{n}{4})^2$
L_2	$c(\frac{n}{4})^2$	$c(\frac{n}{4})^2$



Adding up the costs:

$$T(n) = cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2cn^2 + ...$$

$$cn^2(1 + \frac{3}{16} + (\frac{3}{16})^2) + ...$$

The ... disappear if $n = 16$ or the tree has depth at least 2 if $n \geq 16 = 4^2$.

For $n = 4^k$, $k = log_4(n)$, we have:

$$T(n) = cn^2 \sum_{i=0}^{log_4(n)} (\frac{3}{16})^i$$

Geometric Series, Consider a finite sum first:

$S_n = 1 + r + r^2 + ... + r^n = \sum_{i=0}^{n} r^i$

Applying the Geometric Sum, Applying

$$S_n = \sum_{i=0}^{n} r^i = \frac{r^{n+1} - 1}{r - 1}$$

$$T(n) = cn^2 \sum_{i=0}^{log_4(n)} (\frac{3}{16})^i$$

with $r = \frac{3}{16}$ leads to

$$T(n) = cn^2 \frac{(\frac{3}{16})^{log_4(n)+1} - 1}{\frac{3}{16} - 1}$$

Polishing the result, Instead of $T(n) \leq dn^2$ for some constant d, we have

$$T(n) = cn^2 \frac{(\frac{3}{16})^{log_4(n)+1} - 1}{\frac{3}{16} - 1}$$

recall

$$T(n) = cn^2 \sum_{i=0}^{log_4(n)} (\frac{3}{16})^i$$

to remove the $log_4(n)$ factor, we Consider

$$T(n) \leq cn^2 \sum_{i=0}^{\infty} (\frac{3}{16})^i$$

$$= cn^2 \frac{-1}{\frac{3}{16} - 1} \leq dn^2$$

, for some constant d

Using The Substitution Method

Verifying The Guess, Assuming a bound for T in the form $T(n) \leq dn^2$, is good for $T(n) = 3T(\frac{4}{n}) + cn^2$.

Applying the substitution method:

$$\begin{aligned} T(n) &= 3T(\frac{4}{n}) + cn^2 \\ &\leq 3d\left(\frac{n}{4}\right)^2 + cn^2 \\ &= \left(\frac{3}{16}d + c\right)n^2 \\ &= \frac{3}{16}\left(d + \frac{16}{3}c\right)n^2 \\ &= \frac{3}{16}\left(d + \frac{16}{3}c\right) \leq dn^2 \\ &= \frac{3}{16}(2d)n^2 \ \ , if \ d \geq \frac{16}{3}c \end{aligned}$$

Thus, the bound for $T(n)$ is:

$$T(n) \leq dn^2$$