Lecture 8: Solving Recurrences

CSE 373: Data Structures and Algorithms
What’s the theta bound for the runtime function for this piece of code?

```
public void method1(int n) {
    if (n <= 100) {
        System.out.println(":3");
    } else {
        System.out.println(":D");
        for (int i = 0; i<16; i++) {
            method1(n / 4);
        }
    }
}
```

```
𝑇(𝑛) = \begin{cases} 
    \text{constant work} & \text{if } 𝑛 ≤ 100 \\
    16𝑇\left(\frac{𝑛}{4}\right) + \text{constant work} & \text{otherwise}
\end{cases}
```

- \(a = 16, \ b = 4, \ c = 0\)
- \(\log_4 16 = 2\)
- \(\log_4 16 > 0\)

\(T(n) \in \Theta(n^{\log_4 16}) = \Theta(n^2)\)
Announcements

Exercise 1 – Algorithm Analysis – Due Friday April 16th
Project 1 – Deques – Due Wednesday April 14th
Project 2 Goes out this Friday, due Wednesday April 28th
Midterm goes out Friday April 30th
Questions
Modeling Recursive Code
Meet the Recurrence

A recurrence relation is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s)

It’s a lot like recursive code:
- At least one base case and at least one recursive case
- Each case should include the values for n to which it corresponds
- The recursive case should reduce the input size in a way that eventually triggers the base case
- The cases of your recurrence usually correspond exactly to the cases of the code

\[ T(n) = \begin{cases} 
5 & \text{if } n < 3 \\
2T\left(\frac{n}{2}\right) + 10 & \text{otherwise} 
\end{cases} \]
Recursive Patterns

Modeling and analyzing recursive code is all about finding patterns in how the input changes between calls and how much work is done within each call.

Let’s explore some of the more common recursive patterns:

**Pattern #1**: Halving the Input

**Pattern #2**: Constant size input and doing work

**Pattern #3**: Doubling the Input
Review Why Include Non-Recursive Work?

Think of it this way:

\[ T(n) = \begin{cases} 
2 & \text{if } n < 3 \\
3T\left(\frac{n}{3}\right) + n & \text{otherwise} 
\end{cases} \]

“work that happens if we enter base case”

“work that happens if we enter recursive case”

Non-recursive parts of recursive cases are sometimes where the bulk of the work takes place!

```java
public int recurse(int n) {
    if (n < 3) {
        return 80;
    }
    for (int i = 0; i < n; i++) {
        System.out.println(i);
    }
    int val1 = recurse(n / 3);
    int val2 = recurse(n / 3);
    int val3 = recurse(n / 3);
    return val1 + val2 + val3;
}
```
Recurrence to Big-Θ

\[ T(n) = \begin{cases} 
2 & \text{if } n < 3 \\
2T\left(\frac{n}{3}\right) + n & \text{otherwise} 
\end{cases} \]

It’s still really hard to tell what the big-O is just by looking at it.

But fancy mathematicians have a formula for us to use!

**Master Theorem**

\[ T(n) = \begin{cases} 
d & \text{if } n \text{ is at most some constant} \\
aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} 
\end{cases} \]

Where \( f(n) \) is \( \Theta(n^c) \)

- If \( \log_b a < c \) then \( T(n) \in \Theta(n^c) \)
- If \( \log_b a = c \) then \( T(n) \in \Theta(n^c \log n) \)
- If \( \log_b a > c \) then \( T(n) \in \Theta(n^{\log_b a}) \)

\( a=2 \ b=3 \text{ and } c=1 \)

\( y = \log_b x \text{ is equal to } b^y = x \)

\( \log_3 2 = x \Rightarrow 3^x = 2 \Rightarrow x \approx 0.63 \)

\( \log_3 2 < 1 \)

We’re in case 1

\( T(n) \in \Theta(n) \)
Understanding Master Theorem

Master Theorem

\[ T(n) = \begin{cases} 
  d & \text{if } n \text{ is at most some constant} \\
  aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise}
\end{cases} \]

Where \( f(n) \) is \( \Theta(n^c) \)

- If \( \log_b a < c \) then \( T(n) \in \Theta(n^c) \)
- If \( \log_b a = c \) then \( T(n) \in \Theta(n^c \log n) \)
- If \( \log_b a > c \) then \( T(n) \in \Theta(n^{\log_b a}) \)

The \( \log_b a < c \) case
- Recursive case does a lot of non recursive work in comparison to how quickly it divides the input size
- Most work happens in beginning of call stack
- Non recursive work in recursive case dominates growth, \( n^c \) term

The \( \log_b a = c \) case
- Recursive case evenly splits work between non recursive work and passing along inputs to subsequent recursive calls
- Work is distributed across call stack

The \( \log_b a > c \) case
- Recursive case breaks inputs apart quickly and doesn’t do much non recursive work
- Most work happens near bottom of call stack

- A measures how many recursive calls are triggered by each method instance
- B measures the rate of change for input
- C measures the dominating term of the non recursive work within the recursive method
- D measures the work done in the base case
Recursive Patterns

Pattern #1: Halving the Input
   Binary Search $\Theta(\log n)$

Pattern #2: Constant size input and doing work
   Merge Sort

Pattern #3: Doubling the Input
Merge Sort

Divide

Conquer

Combine

0  1  2  3  4  5  6  7  8  9

0  1  2  3  4  5  6  7  8  9

0  1  2  3  4  5  6  7  8  9

CSE 373 SP 18 - KASEY CHAMPION
Merge Sort

```java
mergeSort(input) {
    if (input.length == 1)
        return
    else
        smallerHalf = mergeSort(new [0, ..., mid])
        largerHalf = mergeSort(new [mid + 1, ...])
        return merge(smallerHalf, largerHalf)
}
```

\[
T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
2T(n/2) + n & \text{otherwise} 
\end{cases}
\]

**Pattern #2** – Constant size input and doing work

Take 1 min to respond to activity

www.pollev.com/cse373activity

Take a guess! What is the Big-O of worst case merge sort?
Merge Sort Recurrence to Big-Θ

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

**Master Theorem**

$$T(n) = \begin{cases} 
d & \text{if } n \text{ is at most some constant} \\
aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise}
\end{cases}$$

Where \( f(n) \) is \( \Theta(n^c) \)

- If \( \log_b a < c \), then \( T(n) \in \Theta(n^c) \)
- If \( \log_b a = c \), then \( T(n) \in \Theta(n^c \log n) \)
- If \( \log_b a > c \), then \( T(n) \in \Theta(n^{\log_b a}) \)

**If**

- \( a = 2 \)
- \( b = 2 \)
- \( c = 1 \)

\[ y = \log_b x \text{ is equal to } b^y = x \]

- \( \log_2 2 = x \Rightarrow 2^x = 2 \Rightarrow x = 1 \)
- \( \log_2 2 = 1 \)

We’re in case 2

\( T(n) \in \Theta(n \log n) \)
Recursive Patterns

Pattern #1: Halving the Input
   **Binary Search** $\Theta(\log n)$

Pattern #2: Constant size input and doing work
   **Merge Sort** $\Theta(n \log n)$

Pattern #3: Doubling the Input
   **Calculating Fibonacci**
Calculating Fibonacci

public int fib(int n) {
    if (n <= 1) {
        return 1;
    }
    return fib(n-1) + fib(n-1);
}

• Each call creates 2 more calls
• Each new call has a copy of the input, almost
• Almost doubling the input at each call

Pattern #3 – Doubling the Input
Calculating Fibonacci Recurrence to Big-Θ

public int f(int n) {
    if (n <= 1) {
        return 1;
    }
    return f(n - 1) + f(n - 1);
}

T(n) = \begin{cases} 
    d & \text{when } n \leq 1 \\
    2T(n - 1) + c & \text{otherwise}
\end{cases}

Can we use master theorem?

Master Theorem

\[
T(n) = \begin{cases} 
    d & \text{if } n \text{ is at most some constant} \\
    aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise}
\end{cases}
\]

Uh oh, our model doesn’t match that format...

Can we intuit a pattern?

T(1) = d
T(2) = 2T(2-1) + c = 2(d) + c
T(3) = 2T(3-1) + c = 2(2(d) + c) + c = 4d + 3c
T(4) = 2T(4-1) + c = 2(4d + 3c) + c = 8d + 7c
T(5) = 2T(5-1) + c = 2(8d + 7c) + c = 16d + 25c

Looks like something’s happening but it’s tough
Maybe geometry can help!
Calculating Fibonacci Recurrence to Big-$\Theta$

How many layers in the function call tree?
How many layers will it take to transform “n” to the base case of “1” by subtracting 1
For our example, 4 -> Height = n

How many function calls per layer?

<table>
<thead>
<tr>
<th>Layer</th>
<th>Function calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

How many function calls on layer $k$?

$2^{k-1}$

How many function calls TOTAL for a tree of $k$ layers?

$1 + 2 + 3 + 4 + \ldots + 2^{k-1}$

$T(n) = \begin{cases} 
d \text{ when } n \leq 1 \\
2T(n-1) + c \text{ otherwise}
\end{cases}$
Calculating Fibonacci Recurrence to Big-$\Theta$

Patterns found:

How many layers in the function call tree? $n$

How many function calls on layer $k$? $2^{k-1}$

How many function calls TOTAL for a tree of $k$ layers?

$$1 + 2 + 4 + 8 + \ldots + 2^{k-1}$$

Total runtime = (total function calls) x (runtime of each function call)

Total runtime = $(1 + 2 + 4 + 8 + \ldots + 2^{k-1})$ x (constant work)

$$1 + 2 + 4 + 8 + \ldots + 2^{k-1} = \sum_{i=1}^{k-1} 2^i = \frac{2^k - 1}{2 - 1} = 2^k - 1$$

Summation Identity

Finite Geometric Series

$$\sum_{i=1}^{k-1} x^i = \frac{x^k - 1}{x - 1}$$

$$T(n) = 2^n - 1 \in \Theta(2^n)$$
Recursive Patterns

Pattern #1: Halving the Input
   Binary Search $\Theta(\log n)$

Pattern #2: Constant size input and doing work
   Merge Sort $\Theta(n \log n)$

Pattern #3: Doubling the Input
   Calculating Fibonacci $\Theta(2^n)$
Questions?
Code Analysis Process

- **Best case**: model of best-case runtime $f(n)$
  - Best-case upper bound $O(n)$
  - Best-case lower bound $\Omega(n)$
  - Best-case tight fit $\Theta(n)$

- **Worst case**: model of worst-case runtime $f(n)$
  - Worst-case upper bound $O(n)$
  - Worst-case lower bound $\Omega(n)$
  - Worst-case tight fit $\Theta(n)$

- If code is recursive:
  - Recurrence
  - Closed Form
  - Master Theorem

- **Case analysis**
  - Code analysis
  - Code modeling

- **Asymptotic analysis**
Recurrence to Big Θ Techniques

A recurrence is a mathematical function that includes itself in its definition.

This makes it very difficult to find the dominating term that will dictate the asymptotic growth.

Solving the recurrence or “finding the closed form” is the process of eliminating the recursive definition. So far, we’ve seen three methods to do so:

1. **Apply Master Theorem**
   - Pro: Plug and chug convenience
   - Con: only works for recurrences of a certain format

2. **Unrolling**
   - Pro: Least complicated setup
   - Con: requires intuitive pattern matching

3. **Tree Method**
   - Pro: Plug and chug
   - Con: Complex setup

Master Theorem

\[
T(n) = \begin{cases} 
  d & \text{when } n \leq 1 \\
  2T(n - 1) + c & \text{otherwise}
\end{cases}
\]

| \(T(1) = d\) |
| \(T(2) = 2T(2-1) + c = 2(d) + c\) |
| \(T(3) = 2T(3-1) + c = 2(2(d) + c) + c = 4d + 3c\) |
| \(T(4) = 2T(4-1) + c = 2(4d + 3c) + c = 8d + 7c\) |
| \(T(5) = 2T(5-1) + c = 2(8d + 7c) + c = 16d + 25c\) |
**Tree Method**

Draw out call stack, what is the input to each call? How much work is done by each call?

**How much work is done at each layer?**

64 for this example -> n work at each layer

Work is variable per layer, but across the entire layer work is constant - always n

**How many layers are in our function call tree?**

Hint: how many levels of recursive calls does it take *binary search* to get to the base case?

Height = $\log_2 n$

It takes $\log_2 n$ divisions by 2 for n to be reduced to the base case 1

$log_2 64 = 6$ -> 6 levels of this tree

**Merge Sort**

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

... and so on...
Tree Method

\[ T(n) = \begin{cases} 
1 & \text{when } n \leq 1 \\
2T\left( \frac{n}{2} \right) + n & \text{otherwise}
\end{cases} \]

Recursive level | How many nodes at each level? | How much work done by each node? | How much work across each level?
--- | --- | --- | ---
0 | 1 | n | n
1 | 2 | \( \frac{n}{2} \) | n
2 | 4 | \( \frac{n}{4} \) | n
3 | 8 | \( \frac{n}{8} \) | n

\( i = \frac{n}{2}, \quad w = \frac{n}{2} \)

\( i = \frac{n}{4}, \quad w = \frac{n}{4} \)

\( i = \frac{n}{8}, \quad w = \frac{n}{8} \)

\( i = \frac{n}{16}, \quad w = \frac{n}{16} \)

...
Tree Method Practice

1. What is the size of the input on level $i$?
   \[ \frac{n}{2^i} \]

2. What is the work done by each node on the $i^{th}$ recursive level?
   \[ \frac{n}{2^i} \]

3. What is the number of nodes at level $i$?
   \[ 2^i \]

4. What is the total work done at the $i^{th}$ recursive level?
   \[ \text{numNodes} \times \text{workPerNode} = 2^i \left( \frac{n}{2^i} \right) = n \]

5. What value of $i$ does the last level occur?
   \[ \frac{n}{2^i} = 1 \Rightarrow n = 2^i \Rightarrow i = \log_2 n \]

6. What is the total work across the base case level?
   \[ \text{numNodes} \times \text{workPerNode} = 2^{\log_2 n} (1) = n \]

<table>
<thead>
<tr>
<th>Level ($i$)</th>
<th>Number of Nodes</th>
<th>Work per Node</th>
<th>Work per Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$\frac{n}{2}$</td>
<td>$n$</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>$\frac{n}{4}$</td>
<td>$n$</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>$\frac{n}{8}$</td>
<td>$n$</td>
</tr>
<tr>
<td>$\log_2 n$</td>
<td>$n$</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Combining it all together...

\[ T(n) = \begin{cases} 
1 & \text{when } n \leq 1 \\
2T\left(\frac{n}{2}\right) + n & \text{otherwise}
\end{cases} \]

\[ T(n) = \sum_{i=0}^{\log_2 n - 1} n + n = n\log_2 n + n = \Theta(n\log n) \]
Recurrence to Big-Theta: Our Toolbox

Master Theorem

**MASTER THEOREM**

\[ T(n) = \begin{cases} 
  d & \text{if } n \text{ is at most some constant} \\
  aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise}
\end{cases} \]

**PROS:** Convenient to plug ‘n’ chug

**CONS:** Only works for certain format of recurrences

Unrolling the Recurrence

\[ T(1) = d \]
\[ T(2) = 2T(2-1) + c = 2(d) + c \]
\[ T(3) = 2T(3-1) + c = 2(2(d) + c) + c = 4d + 3c \]

**PROS:** Least complicated setup

**CONS:** Requires intuitive pattern matching, no formal technique

Tree Method

(followed by Asymptotic Analysis)

**PROS:** Convenient to plug ‘n’ chug

**CONS:** Complicated to set up for a given recurrence
Questions
Summations
Modeling Complex Loops

Write a mathematical model of the following code

```java
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        System.out.println("Hello!");
    }
}
```

Keep an eye on loop bounds!

\[ f(n) = n^2 \]
Modeling Complex Loops

```java
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        System.out.print("Hello!");
    }
    System.out.println();
}
```

\[ T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 \]

**How do we model this part?**

**Summations!**

\[ 1 + 2 + 3 + 4 + \ldots + n = \sum_{i=1}^{n} i \]

**Definition: Summation**

\[ \sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \ldots + f(b-2) + f(b-1) + f(b) \]

\[ T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 \]

**What is the Big O?**
Simplifying Summations

\[ T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} 1 \cdot i = 1 \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \frac{1}{2} n^2 - \frac{1}{2} n = O(n^2) \]

Summation of a constant: \[ \sum_{i=0}^{n-1} c = cn \]

Factoring out a constant: \[ \sum_{i=a}^{b} cf(i) = c \sum_{i=a}^{b} f(i) \]

Gauss's Identity: \[ \sum_{i=0}^{n} i = \frac{n(n+1)}{2} \]

for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        System.out.println("Hello!");
    }
}

Find closed form using summation identities (given on exams)

Closed form: \[ T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 \]

Simplified: \[ \frac{n(n-1)}{2} \]

Tight big O: \[ O(n^2) \]
Questions
Tree Method Practice

\[ T(n) = \begin{cases} 
4 & \text{when } n \leq 1 \\
3T\left(\frac{n}{4}\right) + n^2 & \text{otherwise}
\end{cases} \]

Answer the following questions:
1. What is the size of the input on level \( i \)?
2. What is the work done by each node on the \( i^{th} \) recursive level?
3. What is the number of nodes at level \( i \)?
4. What is the total work done at the \( i^{th} \) recursive level?
5. What value of \( i \) does the last level occur?
6. What is the total work across the base case level?
1. What is the size of the input on level $i$? \( \frac{n}{4^i} \)

2. What is the work done by each node on the $i^{th}$ recursive level? \( \left( \frac{n}{4^i} \right)^2 \)

3. What is the number of nodes at level $i$? \( 3^i \)

4. What is the total work done at the $i^{th}$ recursive level? 
\[
3^i \left[ \left( \frac{n}{4^i} \right)^2 \right] = \left( \frac{3}{16} \right)^i n^2
\]

5. What value of $i$ does the last level occur?
\[
\frac{n}{4^i} = 1 \Rightarrow n = 4^i \Rightarrow i = \log_4 n
\]

6. What is the total work across the base case level? 
\[3^{\log_4 n} \cdot 4 \]

Combining it all together...
\[
T(n) = \sum_{i=0}^{\log_4 n - 1} \left( \frac{3}{16} \right)^i n^2 + 4n^{\log_4 3}
\]
Tree Method Practice

\[ T(n) = \sum_{i=0}^{\log_4 n - 1} \left( \frac{3}{16} \right)^i n^2 + 4n^{\log_4 3} \]

Identities are on the webpage. You don’t need to memorize them.

So what’s the big-\(\Theta\)...\[
T(n) = n^2 \left( -\frac{16}{13} \right) \left( \frac{3}{16} \right)^{\log_4 n} + \left( \frac{16}{13} \right) n^2 + 4n^{\log_4 3}
\]

Closed form:

\[
T(n) = n^2 \left( \frac{3}{16} \right)^{\log_4 n} \left( \frac{3}{16} - 1 \right) + 4n^{\log_4 3}
\]
More Tree Method

\[ T(n) = \begin{cases} 
6T\left(\frac{n}{2}\right) + 2n & \text{if } n > 8 \\
3 & \text{otherwise}
\end{cases} \]
$$T(n) = \begin{cases} 6T\left(\frac{n}{2}\right) + 2n & \text{if } n > 8 \\ 3 & \text{otherwise} \end{cases}$$

Answer the following questions:
1. What is the size of the input on level \(i\)?
2. What is the work done by each node on the \(i^{th}\) recursive level?
3. What is the number of nodes at level \(i\)?
4. What is the total work done at the \(i^{th}\) recursive level?
5. What value of \(i\) does the last level occur?
6. What is the total work across the base case level?
1. What is the size of the input on level $i$? \[ \frac{n}{2^i} \]

2. What is the work done by each node on the $i^{th}$ recursive level? \[ 2 \cdot \frac{n}{2^i} \]

3. What is the number of nodes at level $i$? \[ 6^i \]

4. What is the total work done at the $i^{th}$ recursive level? \[ 6^i \left[ 2 \cdot \frac{n}{2^i} \right] = 2 \cdot 3^i \cdot n \]

5. What value of $i$ does the last level occur? \[ \frac{n}{2^i} = 2 \Rightarrow n = 2^{i+1} \Rightarrow i = \log_2(n) - 1 \]

6. What is the total work across the base case level? \[ 6^{\log_2(n)-1} \cdot 3 \]

Combining it all together...

\[ T(n) = \sum_{i=0}^{\log_2(n)-2} 2 \cdot 3^i n + \frac{1}{2} n^{\log_2 6} \]

\[ T(n) = \begin{cases} 6T \left( \frac{n}{2} \right) + 2n & \text{if } n > 2 \\ 3 & \text{otherwise} \end{cases} \]
\[
T(n) = \sum_{i=0}^{\log_2(n) - 2} 2 \cdot 3^i n + \frac{1}{2} n^{\log_2 6}
\]

\[
= 2n \sum_{i=0}^{\log_2(n)-1} 3^i + \frac{1}{2} n^{\log_2 6}
\]

\[
= 2n \frac{3^{\log_2(n)-1}}{3 - 1} + \frac{1}{2} n^{\log_2 6}
\]

\[
= n \cdot \frac{n^{\log_2(3)}}{3} + \frac{1}{2} n^{\log_2 6}
\]

\[
= \frac{n^{\log_2(3)+1}}{3} + \frac{1}{2} n^{\log_2 6}
\]

\[
= \frac{n^{\log_2(6)}}{3} + \frac{1}{2} n^{\log_2 6} = \frac{5}{6} n^{\log_2 6}
\]

finite geometric series
\[
\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}
\]

power of a log
\[
x^{\log_b y} = y^{\log_b x}
\]

\[
1 = \log_2 2
\]

\[
\log_a b + \log_a c = \log_a (bc)
\]
Summation Practice

```java
public static void primesUpToN(int n) {
    System.out.print("1 2 ");
    for (int i = 3; i <= n; i++) {
        for (int j = 2; j < i; j++) {
            if (j != i && j % i == 0) {
                System.out.print(i + " ");
                break;
            }
        }
    }
    System.out.println();
}
```

\[ T(n) = 1 + \sum_{i=3}^{n} \sum_{j=2}^{i-1} 5 = 1 + \sum_{i=0}^{n-3} \sum_{j=0}^{i-3} 5 = 1 + \sum_{i=0}^{n-3} 5(i - 2) = 1 + 5(\sum_{i=0}^{n-3} i - \sum_{i=0}^{n-3} 2) = 1 + 5\left(\frac{(n-2)(n-3)}{2}\right) - (n-2)(2) \]

- Adjusting summation bounds
- Summation of a constant
- Factoring out a constant
- Gauss's identity