Prefix Averages (Method 1)

- The following algorithm computes prefix averages in quadratic time by applying the definition.

<table>
<thead>
<tr>
<th>Algorithm <code>prefixAverages1(X, n)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
</tr>
<tr>
<td><strong>Output</strong></td>
</tr>
<tr>
<td><strong>#operations</strong></td>
</tr>
<tr>
<td>A ← new array of n integers</td>
</tr>
<tr>
<td>for i ← 0 to n - 1 do</td>
</tr>
<tr>
<td>s ← X[0]</td>
</tr>
<tr>
<td>for j ← 1 to i do</td>
</tr>
<tr>
<td>s ← s + X[j]</td>
</tr>
<tr>
<td>A[i] ← s / (i + 1)</td>
</tr>
<tr>
<td>return A</td>
</tr>
</tbody>
</table>

run time $T(n) = (c_1 n^2 + c_2 n)$

Prefix Averages (Method 2)

<table>
<thead>
<tr>
<th>Algorithm <code>prefixAverages2(X, n)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
</tr>
<tr>
<td><strong>Output</strong></td>
</tr>
<tr>
<td><strong>#operations</strong></td>
</tr>
<tr>
<td>A ← new array of n integers</td>
</tr>
<tr>
<td>s ← 0</td>
</tr>
<tr>
<td>for i ← 0 to n - 1 do</td>
</tr>
<tr>
<td>s ← s + X[i]</td>
</tr>
<tr>
<td>A[i] ← s / (i + 1)</td>
</tr>
<tr>
<td>return A</td>
</tr>
</tbody>
</table>

run time $T(n) = (c_3 n + c_4)$
Tools: Asymptotic Notation

- Algorithm 1: \( T(n) = (c_1n^2 + c_2n) \).
  The quadratic growth of the running time \( T(n) \) is an intrinsic property of algorithm \textit{prefixAverages1}
  \[ T(n) = O(n^2) \]

- Algorithm 2: \( T(n) = (c_3n + c_4) \).
  The linear growth of the running time \( T(n) \) is an intrinsic property of algorithm \textit{prefixAverages2}
  \[ T(n) = O(n) \]

Big-Oh Rules

1. Drop lower-order terms
2. Drop constant factors
   - If \( T(n) \) is a polynomial of degree \( d \), then \( T(n) \) is \( O(n^d) \), i.e.,
     - ex: \( 7n-3 \) is \( O(n) \)
     - \( 3\log n + \log \log n \) is \( O(\log n) \)
     - \( 8n^3\log n + 5n^2 + n \) is \( O(n^3\log n) \)

Asymptotic Notation

- \textbf{Big-Oh}, \( O(f(n)) \): Asymptotic upper bound
  - given functions \( T(n) \) and \( f(n) \), \( T(n) \) is \( O(f(n)) \) if and only if there are positive constants \( c \) and \( n_0 \) such that \( T(n) \leq c f(n) \) for \( n \geq n_0 \)
"Big Oh" Example

Example: $2n+6$ is $O(n)$

For functions $T(n)$ and $f(n)$ (to the right) there are positive constants $c$ and $n_0$ such that:

$T(n) \leq c f(n)$ for $n \geq n_0$

Conclusion:

$2n+6$ is $O(n)$.

Another Example

On the other hand...

$n^2$ is not $O(n)$ because there is no $c$ and $n_0$ such that:

$n^2 \leq cn$ for $n \geq n_0$

(As the graph to the right illustrates, no matter how large a $c$ is chosen there is an $n$ big enough that $n^2 > cn$).

Big-Oh Examples

- $7n-2$
  - $7n-2$ is $O(n)$
  - need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq cn$ for $n \geq n_0$
  - this is true for $c = 7$ and $n_0 = 1$

- $3n^2 + 20n^2 + 5$
  - $3n^2 + 20n^2 + 5$ is $O(n^2)$
  - need $c > 0$ and $n_0 \geq 1$ such that $3n^2 + 20n^2 + 5 \leq cn^2$ for $n \geq n_0$
  - this is true for $c = 4$ and $n_0 = 21$

- $3 \log n + \log \log n$
  - $3 \log n + \log \log n$ is $O(\log n)$
  - need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + \log \log n \leq c \log n$ for $n \geq n_0$
  - this is true for $c = 4$ and $n_0 = 2$
Asymptotic Notation (cont.)

• Caution:
  It is correct to say “2n + 6” is $O(n^2)$.

However, a better statement is “2n + 6” is $O(n)$, that is, one should make the approximation as tight as possible.

Asymptotic Notation (cont.)

• Special classes of algorithms:
  - Constant: $O(1)$
  - Logarithmic: $O(\log n)$
  - Linear: $O(n)$
  - Quadratic: $O(n^2)$
  - Polynomial: $O(n^k), k \geq 1$
  - Exponential: $O(a^n), n > 1$

“Relatives” of the Big-Oh:
- $\Omega(f(n))$: Big Omega—asymptotic lower bound
- $\Theta(f(n))$: Big Theta—asymptotic tight bound
- $o(f(n))$: Little oh—asymptotic less than

Asymptotic Notation (cont.)

• Big Omega, $\Omega(f(n))$: asymptotic lower bound
  - Given functions $T(n)$ and $f(n)$. $T(n)$ is $\Omega(f(n))$ if and only if there are positive constants $c$ and $n_0$ such that $T(n) \geq c f(n)$ for $n \geq n_0$

Example:
  $3\log n + \log \log n$ is $\Omega(\log n)$

Proof:
  $3\log n + \log \log n \geq 3\log n$ for $n \geq 2$
Asymptotic Notation (cont.)

- **Big Theta, \( \Theta(f(n)) \):** asymptotic tight bound
  - given functions \( T(n) \) and \( f(n) \), \( T(n) \) is \( \Theta(f(n)) \) if \( T(n) \) is \( O(f(n)) \) and \( T(n) \) is \( \Omega(f(n)) \).
  - or in other words
  - there exist positive constants \( c_1 \) and \( c_2 \) and \( n_0 \) such that \( c_1 f(n) \leq T(n) \leq c_2 f(n) \)

example:

\[ 3 \log n + \log \log n \text{ is } \Theta(\log n) \]

proof:

\[ 3 \log n + \log \log n \leq 4 \log n \text{ for } n \geq 2 \Rightarrow O(\log n) \]

\[ 3 \log n + \log \log n \geq 3 \log n \text{ for } n \geq 2 \Rightarrow \Omega(\log n) \]

Asymptotic Notation (cont.)

- **little oh, \( o(f(n)) \):** asymptotic less than
  - given functions \( T(n) \) and \( f(n) \), \( T(n) \) is \( o(f(n)) \) if for any constants \( c > 0 \), there exists constant \( n_0 > 0 \) such that \( T(n) < cf(n) \) for \( n \geq n_0 \)

Tools: amortization
An Extendable Array

Algorithm push(a):
input item a / array A
//output array A with a appended
if size() = N then
    new array of length f(N)
for i = 0 to N - 1
    B[i] = A[i]
A = B
N = f(N)
t = t + 1
A[t] = a
//return A

• How large should the new array be?
  - tight strategy (add a constant): f(N) = N + c
  - growth strategy (double up): f(N) = 2N

Analyzing Extendable Array

Tight vs. Growth Strategies

• To compare the two strategies, we use the following cost model:

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>RUN TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular push operation</td>
<td>1</td>
</tr>
<tr>
<td>special push operation</td>
<td>N+1</td>
</tr>
</tbody>
</table>
  - create an array of size f(N),
    copy N elements, and add one element

An Extendable Array

Step I.

Step II.

Step III.
Analyzing Extendable Array Growth Strategy

**growth strategy** (double up): \( f(N) = 2N \)

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>RUN TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>special push operation: create an array of size 2N, copy N elements, and add one element</td>
<td>( N + 1 )</td>
</tr>
<tr>
<td>regular push operation: for the next ( N - 1 ) elements</td>
<td>( N - 1 )</td>
</tr>
<tr>
<td>Total time for ( N ) element addition</td>
<td>( 2N )</td>
</tr>
</tbody>
</table>

\( f(n) \in O(n) \)

---

Performance of the Tight Strategy

tight strategy (constant increment): \( f(N) = N + C \)

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>RUN TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>special push operation: create an array of size ( N + C ), copy N elements, and add one element</td>
<td>( N + 1 )</td>
</tr>
<tr>
<td>regular push operation: for the next ( C - 1 ) elements</td>
<td>( C - 1 )</td>
</tr>
<tr>
<td>Total time for ( C ) element addition</td>
<td>( N + C )</td>
</tr>
</tbody>
</table>

\( f(n) \in O(n^2) \)

Growth strategy wins!!

---

Tools: Proof by Induction

- Proof of Statement \( S(n) \) for all \( n \geq 1 \)
  - Step I: Show the base case.
    - say: \( S(k) \) is true for \( k = 1 \)
  - Step II: Show that if \( S(k) \) is true for \( 1 \leq k \leq n \), then \( S(n+1) \) is true.
Proof by Induction

- Proof by Induction
  • ex: Prove that \( \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \) for all \( n \geq 1 \)

  Proof:
  
  Step I: \( \sum_{i=1}^{1} i^2 = \frac{k(k+1)(2k+1)}{6} \) for \( k = 1 \) is trivially true.
  
  Step II: Assume \( \sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6} \) for all \( 1 \leq k \leq n \).
  
  Show that \( \sum_{i=1}^{k+1} i^2 = \frac{(k+1)(k+2)(2k+3)}{6} \) for \( k + 1 \).

Math Fundamentals (continued)

Step II: if \( \sum_{i=1}^{k} i^2 = \frac{n(n+1)(2n+1)}{6} \), show \( \sum_{i=1}^{k+1} i^2 = \frac{(n+1)(n+2)(2(n+1)+1)}{6} \).

\[
\sum_{i=1}^{k+1} i^2 = \sum_{i=1}^{k} i^2 + (k+1)^2 = \frac{n(n+1)(2n+1)}{6} + (n+1)^2
= (n+1) \left( \frac{2n^2 + 7n + 6}{6} \right)
= \frac{(n+1)(n+2)(2(n+1)+1)}{6}
\]

Proof complete.

Lab Quiz 2

1. Prove by induction that \( \sum_{i=1}^{n} i^2 = \left( \sum_{i=1}^{n} i \right)^2 \) for all \( n \geq 1 \)

2. Compute the asymptotic growth time for an extendable table for which the array size is increased from \( N \) to the following possible values:
   a. \( N + \lceil \sqrt{N} \rceil \)
   b. \( N + \lceil \log N \rceil \)
Math Fundamentals (continued)

- Logarithms:
  - definition: \( \log_b x = y \) iff \( b^y = x \)
  - \( \log_b (xy) = \log_b x + \log_b y \)
  - \( \log_b \frac{x}{y} = \log_b x - \log_b y \)
  - \( \log_b y^c = c \log_b y \)
  - \( \log_b y = \frac{\log_{10} y}{\log_{10} b} \)

Merge Sort

- Merge-sort on an input sequence \( S \) with \( n \) elements consists of three steps:
  - Divide: partition \( S \) into two sequences \( S_1 \) and \( S_2 \) of about \( n/2 \) elements each
  - Recur: recursively sort \( S_1 \) and \( S_2 \)
  - Conquer: merge \( S_1 \) and \( S_2 \) into a unique sorted sequence

Algorithm mergeSort(S, C)
Input sequence \( S \) with \( n \) elements, comparator \( C \)
Output sequence \( S \) sorted according to \( C \)
if \( S.size > 1 \)
  \( (S_1, S_2) \leftarrow partition(S, n/2) \)
  mergeSort(S_1, C)
  mergeSort(S_2, C)
  \( S \leftarrow merge(S_1, S_2) \)
Merging Two Sorted Sequences

- The conquer step of merge-sort consists of merging two sorted sequences \( A \) and \( B \) into a sorted sequence \( S \) containing the union of the elements of \( A \) and \( B \).
- Merging two sorted sequences, each with \( n/2 \) elements, can be implemented by means of a doubly linked list, taking \( O(n) \) time.

Algorithm `merge(A, B)`

- Input: sequences \( A \) and \( B \) with \( n/2 \) elements each.
- Output: sorted sequence of \( A \cup B \).

```
S ← empty sequence
while ¬A.isEmpty ∧ ¬B.isEmpty
  if A.first < B.first
    S.insertLast(A.remove(A.first))
  else
    S.insertLast(B.remove(B.first))

while ¬A.isEmpty
  S.insertLast(A.remove(A.first))

while ¬B.isEmpty
  S.insertLast(B.remove(B.first))

return S
```

Merge-Sort Tree

- An execution of merge-sort is depicted by a binary tree:
  - Each node represents a recursive call of merge-sort and stores
    - the unsorted sequence before the execution and its partition
    - the sorted sequence at the end of the execution
  - The root is the initial call.
  - The leaves are calls on subsequences of size 0 or 1.

Execution Example

- Partition
Execution Example (cont.)

• Recursive call, partition

Execution Example (cont.)

• Recursive call, partition

Execution Example (cont.)

• Recursive call, base case
Execution Example (cont.)

- Recursive call, base case

- Merge

Execution Example (cont.)

- Recursive call, ..., base case, merge
Execution Example (cont.)
• Merge

Execution Example (cont.)
• Recursive call, ..., merge, merge

Execution Example (cont.)
• Merge
Analysis of Merge-Sort

- The height $h$ of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth $i$ is $O(n)$
  - we partition and merge $2^i$ sequences of size $n/2^i$
  - we make $2^i$ recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

<table>
<thead>
<tr>
<th>depth</th>
<th>size</th>
<th>seqs</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$n$</td>
<td>1</td>
<td>$b_0$</td>
</tr>
<tr>
<td>1</td>
<td>$n/2$</td>
<td>2</td>
<td>$b_1$</td>
</tr>
<tr>
<td>$i$</td>
<td>$n/2^i$</td>
<td>$2^i$</td>
<td>$b_i$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Tools: Recurrence Equation Analysis

- The conquer step of merge-sort consists of merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes at most $b$ steps, for some constant $b$.
- Likewise, the basis case ($n < 2$) will take at most steps.
- Therefore, if we let $T(n)$ denote the running time of merge-sort:

$$T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn & \text{if } n \geq 2 
\end{cases}$$

- We can therefore analyze the running time of merge-sort by finding a closed form solution to the above equation.
  - That is, a solution that has $T(n)$ only on the left hand side.

The Recursion Tree

- Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn & \text{if } n \geq 2 
\end{cases}$$

**Total time** = $bn + bn \log n$

(last level plus all previous levels)
Iterative Substitution

- In the iterative substitution, or “plug-and-chug,” technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

\[
T(n) = 2T(n/2) + bn
\]

\[
= 2(2T(n/2^2) + kn/2) + bn
\]

\[
= 2^2T(n/2^2) + 2bn
\]

\[
= 2^2T(n/2^3) + 3bn
\]

\[
= 2^2T(n/2^3) + 4bn
\]

\[
= 2^i T(n/2^i) + ibn
\]

- Note that base, \( T(n) = b \), case occurs when \( 2^i = n \). That is, \( i = \log n \).

- So,

\[
T(n) = bn + bn \log n
\]

- Thus, \( T(n) \) is \( O(n \log n) \).

---

Guess-and-Test Method

- In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:

\[
T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn \log n & \text{if } n \geq 2 
\end{cases}
\]

- Guess: \( T(n) < cn \log n \).

\[
T(n) = 2T(n/2) + bn \log n
\]

\[
= 2c(n/2) \log(n/2) + bn \log n
\]

\[
= cn \log n - \log 2^i + bn \log n
\]

Wrong: we cannot make this last line be less than \( cn \log n \)

---

Guess-and-Test Method, Part 2

- Recall the recurrence equation:

\[
T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn \log n & \text{if } n \geq 2 
\end{cases}
\]

- Guess #2: \( T(n) < cn \log^2 n \).

\[
T(n) = 2T(n/2) + bn \log n
\]

\[
= 2c(n/2) \log^2(n/2) + bn \log n
\]

\[
= cn \log n - \log 2^{i+1} + bn \log n
\]

\[
\leq cn \log^2 n
\]

- if \( c > b \).

- So, \( T(n) \) is \( O(n \log^2 n) \).

In general, to use this method, you need to have a good guess and you need to be good at induction proofs.
Master Method

- Many divide-and-conquer recurrence equations have the form:

$$T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases}$$

- The Master Theorem:
  1. if \( f(n) = O(n^d \log^n n) \), then \( T(n) = \Theta(n^d \log^n n) \)
  2. if \( f(n) = \Theta(n^d \log^n n) \), then \( T(n) = \Theta(n^d \log^{n-\delta} n) \) for some \( \delta < 1 \)
  3. if \( f(n) = \Omega(n^d \log^{n+\epsilon} n) \), then \( T(n) = \Theta(n^d \log^{n-\epsilon} n) \)

Master Method, Example 1

- The form:

$$T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases}$$

- The Master Theorem:
  1. if \( f(n) = O(n^d \log^n n) \), then \( T(n) = \Theta(n^d \log^n n) \)
  2. if \( f(n) = \Theta(n^d \log^n n) \), then \( T(n) = \Theta(n^d \log^{n-\delta} n) \) for some \( \delta < 1 \)
  3. if \( f(n) = \Omega(n^d \log^{n+\epsilon} n) \), then \( T(n) = \Theta(n^d \log^{n-\epsilon} n) \)

- Example:

$$T(n) = 4T(n/2) + n$$

Solution: \( \log_2 a = 2 \), so case 1 says \( T(n) = O(n^2) \).

Master Method, Example 2

- The form:

$$T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases}$$

- The Master Theorem:
  1. if \( f(n) = O(n^d \log^n n) \), then \( T(n) = \Theta(n^d \log^n n) \)
  2. if \( f(n) = \Theta(n^d \log^n n) \), then \( T(n) = \Theta(n^d \log^{n-\delta} n) \) for some \( \delta < 1 \)
  3. if \( f(n) = \Omega(n^d \log^{n+\epsilon} n) \), then \( T(n) = \Theta(n^d \log^{n-\epsilon} n) \)

- Example:

$$T(n) = 2T(n/2) + n \log n$$

Solution: \( \log_2 a = 1 \), so case 2 says \( T(n) = O(n \log^2 n) \).
Master Method, Example 3

- The form: 
  \[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]
- The Master Theorem:
  1. If \( f(n) = \Theta(n^d \cdot \log^k n) \), then \( T(n) = \Theta(n^d \cdot \log^k n) \)
  2. If \( f(n) = \Theta(n^d \cdot \log^k n) \), then \( T(n) = \Theta(n^d \cdot \log^{k-1} n) \)
  3. If \( f(n) = \Theta(n^d \cdot \log^k n) \), then \( T(n) = \Theta(f(n)) \), provided \( a < \Theta(n^d) \) for some \( \delta < 1 \).
- Example:
  \[ T(n) = 2T(n/3) + n \log n \]
  Solution: \( \log_2 a = 0 \), so case 3 says \( T(n) \) is \( O(n \log n) \).

---

Master Method, Example 4

- The form: 
  \[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]
- The Master Theorem:
  1. If \( f(n) = \Theta(n^d \cdot \log^k n) \), then \( T(n) = \Theta(n^d \cdot \log^k n) \)
  2. If \( f(n) = \Theta(n^d \cdot \log^k n) \), then \( T(n) = \Theta(n^d \cdot \log^{k-1} n) \)
  3. If \( f(n) = \Theta(n^d \cdot \log^k n) \), then \( T(n) = \Theta(f(n)) \), provided \( a \geq \Theta(n^d) \) for some \( \delta < 1 \).
- Example:
  \[ T(n) = 2T(n/2) + n^2 \]
  Solution: \( \log_2 a = 3 \), so case 1 says \( T(n) \) is \( O(n^2) \).

---

Master Method, Example 5

- The form: 
  \[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]
- The Master Theorem:
  1. If \( f(n) = \Theta(n^d \cdot \log^k n) \), then \( T(n) = \Theta(n^d \cdot \log^k n) \)
  2. If \( f(n) = \Theta(n^d \cdot \log^k n) \), then \( T(n) = \Theta(n^d \cdot \log^{k-1} n) \)
  3. If \( f(n) = \Theta(n^d \cdot \log^k n) \), then \( T(n) = \Theta(f(n)) \), provided \( a \geq \Theta(n^d) \) for some \( \delta < 1 \).
- Example:
  \[ T(n) = 2T(n/3) + n^3 \]
  Solution: \( \log_3 a = 2 \), so case 3 says \( T(n) \) is \( O(n^3) \).
Master Method, Example 6

- The form: \[ T(n) = \begin{cases} \frac{c}{d} & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases} \]

- The Master Theorem:
  1. If \( f(n) = \Theta(n^d \log^k n) \) then \( T(n) = \Theta(n^d \log^k n) \)
  2. If \( f(n) = \Theta(n^{d-1} \log^k n) \) then \( T(n) = \Theta(n^d \log^{k-1} n) \)
  3. If \( f(n) = \Theta(n^{d+\epsilon}) \) then \( T(n) = \Theta(n^d \log^{k-\frac{\epsilon}{1-\delta}}} \) for some \( \delta < 1 \).

- Example:
  \[ T(n) = T(n/2) + 1 \] (binary search)
  Solution: \( \log a = 0 \), so case 2 says \( T(n) \) is \( \Theta(n \log n) \).

Master Method, Example 7

- The form: \[ T(n) = \begin{cases} \frac{c}{d} & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases} \]

- The Master Theorem:
  1. If \( f(n) = \Theta(n^{d-1} \log^k n) \) then \( T(n) = \Theta(n^{d-1} \log^k n) \)
  2. If \( f(n) = \Theta(n^{d-1} \log^k n) \) then \( T(n) = \Theta(n^{d-1} \log^{k-1} n) \)
  3. If \( f(n) = \Theta(n^{d+\epsilon}) \) then \( T(n) = \Theta(n^{d} \log^{k-\frac{\epsilon}{1-\delta}}} \) for some \( \delta < 1 \).

- Example:
  \[ T(n) = 2T(n/2) + \log n \] (heap construction)
  Solution: \( \log a = 1 \), so case 1 says \( T(n) \) is \( \Theta(n) \).

Iterative “Proof” of the Master Theorem

- Using iterative substitution, let us see if we can find a pattern:
  \[ T(n) = aT(n/b) + f(n) \]
  \[ = a^2T(n/b^2) + af(n/b) + f(n) \]
  \[ = a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \]
  \[ = \cdots \]
  \[ = a^{\log_b n}T(1) + \sum_{i=1}^{\log_b n} a^if(n/b^i) \]
  \[ = nT(1) + \sum_{i=1}^{\log_b n} \frac{a^i}{b^i} \]

- We then distinguish the three cases as
  - The first term is dominant
  - Each part of the summation is equally dominant
  - The summation is a geometric series