Outline

1. Insertion Sort
2. Divide & Conquer Strategy and Merge Sort
3. Master Theorem
Binary Search

- Input: Sorted array $A[1..n]$ and a number $x$
- Output: Find $i$ such that $A[i] = x$, if no such $i$ exists, output “no”.

We use a function $\text{BinarySearch}(A, p, r, x)$ that searches $x$ in $A[p..r]$.

1: if $p = r$
2: if $A[p] = x$ return $p$
3: if $A[p] \neq x$ return “no”
4: else
5: $q = (p + r) / 2$
6: if $A[q] = x$ return $q$
7: if $A[q] > x$ call $\text{BinarySearch}(A, p, q - 1, x)$
8: if $A[q] < x$ call $\text{BinarySearch}(A, q + 1, r, x)$
9: end if
Binary Search

- Input: Sorted array $A[1..n]$ and a number $x$
- Output: Find $i$ such that $A[i] = x$, if no such $i$ exists, output “no”.

We use a function $\text{BinarySearch}(A, p, r, x)$ that searches $x$ in $A[p..r]$. 
Binary Search

- **Input:** Sorted array $A[1..n]$ and a number $x$
- **Output:** Find $i$ such that $A[i] = x$, if no such $i$ exists, output “no”.

We use a function $\text{BinarySearch}(A, p, r, x)$ that searches $x$ in $A[p..r]$.

**BinarySearch**($A, p, r, x$)

1: **if** $p = r$ **then**
2: **if** $A[p] = x$ **return** $p$
3: **if** $A[p] \neq x$ **return** “no”
4: **else**
5: $q = (p + r)/2$
6: **if** $A[q] = x$ **return** $q$
7: **if** $A[q] > x$ call **BinarySearch**($A, p, q - 1, x$)
8: **if** $A[q] < x$ call **BinarySearch**($A, q + 1, r, x$)
9: **end if**
Binary Search

- **Input:** Sorted array $A[1..n]$ and a number $x$
- **Output:** Find $i$ such that $A[i] = x$, if no such $i$ exists, output “no”.

We use a function $\text{BinarySearch}(A, p, r, x)$ that searches $x$ in $A[p..r]$.  

**BinarySearch**($A, p, r, x$)

1. if $p = r$ then
2. if $A[p] = x$ return $p$
3. if $A[p] \neq x$ return “no”
4. else
5. $q = (p + r)/2$
6. if $A[q] = x$ return $q$
7. if $A[q] > x$ call $\text{BinarySearch}(A, p, q - 1, x)$
8. if $A[q] < x$ call $\text{BinarySearch}(A, q + 1, r, x)$
9. end if
Insertion Sort

Sorting

- Input: An array $A[1..n]$
- Output: Re-ordered $A$ such that $A[i] \leq A[i + 1]$ for any $1 \leq i \leq n - 1$. 

© Hu Ding (Michigan State University)
Insertion Sort

InsertionSort(A[1..n])

1: For i = 1 to n – 1
2: if A[i] > A[i + 1] then
3:   if i = 1 then
4:     swap A[i] and A[i + 1]
5:   else
8:     else
9:       Find A[i′] such that 1 ≤ i′ < i, A[i′] ≤ A[i + 1] < A[i′ + 1],
11:   end if
12: end if
13: end if
Analysis for Insertion Sort

At most $\Theta\left(\sum_{i=1}^{n} i\right) = \Theta(n^2)$ movements and comparisons $\Rightarrow T(n) = \Theta(n^2)$.

Using binary search, we can reduce the number of comparisons to $\Theta\left(\sum_{i=1}^{n} \log i\right) = \Theta(n \log n)$.

Sometimes comparison is more expensive than movement.

Only $O(1)$ extra space.
Analysis for Insertion Sort

- At most $\Theta(\sum_{i=1}^{n} i) = \Theta(n^2)$ movements and comparisons
  $\Rightarrow T(n) = \Theta(n^2)$. 

- Using binary search, we can reduce the number of comparisons to $\Theta(\sum_{i=1}^{n} i \log i) = \Theta(n \log n)$. Sometimes comparison is more expensive than movement.

- Only $O(1)$ extra space.
Analysis for Insertion Sort

- At most $\Theta(\sum_{i=1}^{n} i) = \Theta(n^2)$ movements and comparisons $\implies T(n) = \Theta(n^2)$.

- Using binary search, we can reduce the number of comparisons to $\Theta(\sum_{i=1}^{n} \log i) = \Theta(n \log n)$. Sometimes comparison is more expensive than movement.
Analysis for Insertion Sort

- At most $\Theta(\sum_{i=1}^{n} i) = \Theta(n^2)$ movements and comparisons $\implies T(n) = \Theta(n^2)$.
- Using binary search, we can reduce the number of comparisons to $\Theta(\sum_{i=1}^{n} \log i) = \Theta(n \log n)$. Sometimes comparison is more expensive than movement.
- Only $O(1)$ extra space.
Outline

1. Insertion Sort
2. Divide & Conquer Strategy and Merge Sort
3. Master Theorem
Divide and Conquer Strategy

- Algorithm design is more an art, less so a science.
Divide and Conquer Strategy

- Algorithm design is more an art, less so a science.
- There are a few useful strategies, but no guarantee to succeed.
Divide and Conquer Strategy

- Algorithm design is more an art, less so a science.
- There are a few useful strategies, but no guarantee to succeed.
- We will discuss: Divide and Conquer, Greedy, Dynamic Programming.
Divide and Conquer Strategy

- Algorithm design is more an art, less so a science.
- There are a few useful strategies, but no guarantee to succeed.
- We will discuss: Divide and Conquer, Greedy, Dynamic Programming.
- For each of them, we will discuss a few examples, and try to identify common schemes.
Divide and Conquer Strategy

- Algorithm design is more an art, less so a science.
- There are a few useful strategies, but no guarantee to succeed.
- We will discuss: **Divide and Conquer, Greedy, Dynamic Programming**.
- For each of them, we will discuss a few examples, and try to identify common schemes.

**Divide and Conquer**
Divide and Conquer Strategy

- Algorithm design is more an art, less so a science.
- There are a few useful strategies, but no guarantee to succeed.
- We will discuss: Divide and Conquer, Greedy, Dynamic Programming.
- For each of them, we will discuss a few examples, and try to identify common schemes.

Divide and Conquer
- Divide the problem into smaller subproblems (of the same type).
Divide and Conquer Strategy

- Algorithm design is more an art, less so a science.
- There are a few useful strategies, but no guarantee to succeed.
- We will discuss: Divide and Conquer, Greedy, Dynamic Programming.
- For each of them, we will discuss a few examples, and try to identify common schemes.

Divide and Conquer

- Divide the problem into smaller subproblems (of the same type).
- Solve each subproblem (usually by recursive calls).
Divide and Conquer Strategy

- Algorithm design is more an art, less so a science.
- There are a few useful strategies, but no guarantee to succeed.
- We will discuss: **Divide and Conquer, Greedy, Dynamic Programming**.
- For each of them, we will discuss a few examples, and try to identify common schemes.

**Divide and Conquer**

- Divide the problem into smaller subproblems (**of the same type**).
- Solve each subproblem (**usually by recursive calls**).
- Combine the solutions of the subproblems into the solution of the original problem.
Merge Sort

Input: an array $A[1..n]$
Output: Sort $A$ into increasing order.
Merge Sort

**MergeSort**

Input: an array $A[1..n]$
Output: Sort $A$ into increasing order.

- Use a recursive function `MergeSort(A, p, r)`. 
Merge Sort

**MergeSort**

**Input:** an array $A[1..n]$

**Output:** Sort $A$ into increasing order.

- Use a recursive function $\text{MergeSort}(A, p, r)$.
MergeSort

Input: an array $A[1..n]$
Output: Sort $A$ into increasing order.

- Use a recursive function $\text{MergeSort}(A, p, r)$.
- In main program, we call $\text{MergeSort}(A, 1, n)$. 

© Hu Ding (Michigan State University)  CSE 331 Algorithm and Data Structures
Merge Sort

**Merge**($A, p, q, r$)

1: $i = p, j = q + 1, B = zeros(r - p + 1), flag = 0.$
2: **While**($i \leq q$ or $j \leq r$)
3: flag=flag+1
4: if $i \leq q$ and $j \leq r$ then
5: if $A[i] \leq A[j]$ then
6: $B[flag] = A[i], i = i + 1$
7: else
8: $B[flag] = A[j], j = j + 1$
9: end if
10: else if $i = q + 1$ then
11: $B[flag : r - p + 1] = A[j : r], j = r + 1$
12: else
13: $B[flag : r - p + 1] = A[i : q], i = q + 1$
14: end if
15: End While
Merge Sort

\textbf{MergeSort}(A, p, r)

1: \textbf{if} \ (p < r) \ \textbf{then}
2: \quad q = (p + r)/2
3: \quad \text{MergeSort}(A, p, q)
4: \quad \text{MergeSort}(A, q + 1, r)
5: \quad \text{Merge}(A, p, q, r)
6: \ \textbf{else}
7: \quad \text{do nothing}
8: \ \textbf{end if}
Merge Sort

MergeSort(A, p, r)

1: if (p < r) then
2: \[ q = (p + r)/2 \]
3: MergeSort(A, p, q)
4: MergeSort(A, q + 1, r)
5: Merge(A, p, q, r)
6: else
7: do nothing
8: end if

- Divide A[p..r] into two sub-arrays of equal size.
Merge Sort

**MergeSort**\((A, p, r)\)

1: \textbf{if} \ (p < r) \ 	extbf{then}
2: \hspace{1em} q = (p + r)/2
3: \hspace{1em} MergeSort\((A, p, q)\)
4: \hspace{1em} MergeSort\((A, q + 1, r)\)
5: \hspace{1em} Merge\((A, p, q, r)\)
6: \textbf{else}
7: \hspace{1em} do nothing
8: \textbf{end if}

- Divide \(A[p..r]\) into two sub-arrays of equal size.
- Sort each sub-array by recursive call.
Merge Sort

**MergeSort**($A, p, r$)

1: if ($p < r$) then
2:   $q = (p + r)/2$
3:   MergeSort($A, p, q$)
4:   MergeSort($A, q + 1, r$)
5:   Merge($A, p, q, r$)
6: else
7:   do nothing
8: end if

- Divide $A[p..r]$ into two sub-arrays of equal size.
- Sort each sub-array by recursive call.
- **Merge**($A, p, q, r$) is a procedure that, assuming $A[p..q]$ and $A[q+1..r]$ are sorted, merge them into sorted $A[p..r]$
- It can be done in $\Theta(k)$ time where $k = r - p$ is the number of elements to be sorted.
Let \( T(n) \) be the runtime function of MergeSort\((A[1..n])\). Then:

\[
T(n) = \begin{cases} 
O(1) & \text{if } n = 1 \\
2T(n/2) + \Theta(n) & \text{if } n > 1 
\end{cases}
\]
Analysis of MergeSort

Let $T(n)$ be the runtime function of MergeSort($A[1..n]$). Then:

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

- If $n = 1$, MergeSort does nothing, hence $O(1)$ time.
Analysis of MergeSort

Let $T(n)$ be the runtime function of MergeSort($A[1..n]$). Then:

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

- If $n = 1$, MergeSort does nothing, hence $O(1)$ time.
- Otherwise, we make 2 recursive calls. The input size of each is $n/2$. Hence the runtime $2T(n/2)$.
Analysis of MergeSort

Let $T(n)$ be the runtime function of MergeSort($A[1..n]$). Then:

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

- If $n = 1$, MergeSort does nothing, hence $O(1)$ time.
- Otherwise, we make 2 recursive calls. The input size of each is $n/2$. Hence the runtime $2T(n/2)$.
- $\Theta(n)$ is the time needed by $\text{Merge}(A, p, q, r)$ and all other processing.
Outline

1. Insertion Sort
2. Divide & Conquer Strategy and Merge Sort
3. Master Theorem
Master Theorem

For DaC algorithms, the runtime function often satisfies:

\[
T(n) = \begin{cases} 
O(1) & \text{if } n \leq n_0 \\
 aT(n/b) + \Theta(f(n)) & \text{if } n > n_0 
\end{cases}
\]
Master Theorem

For DaC algorithms, the runtime function often satisfies:

\[ T(n) = \begin{cases} 
  O(1) & \text{if } n \leq n_0 \\
  aT(n/b) + \Theta(f(n)) & \text{if } n > n_0 
\end{cases} \]

- If \( n \leq n_0 \) (\( n_0 \) is a small constant), we solve the problem directly without recursive calls. Since the input size is fixed (bounded by \( n_0 \)), it takes \( O(1) \) time.
Master Theorem

For DaC algorithms, the runtime function often satisfies:

\[ T(n) = \begin{cases} 
O(1) & \text{if } n \leq n_0 \\
 aT\left(\frac{n}{b}\right) + \Theta(f(n)) & \text{if } n > n_0
\end{cases} \]

- If \( n \leq n_0 \) (\( n_0 \) is a small constant), we solve the problem directly without recursive calls. Since the input size is fixed (bounded by \( n_0 \)), it takes \( O(1) \) time.
- We make \( a \) recursive calls. The input size of each is \( \frac{n}{b} \). Hence the runtime \( T(n/b) \).
Master Theorem

For DaC algorithms, the runtime function often satisfies:

\[
T(n) = \begin{cases} 
O(1) & \text{if } n \leq n_0 \\
 aT(n/b) + \Theta(f(n)) & \text{if } n > n_0 
\end{cases}
\]

- If \( n \leq n_0 \) (\( n_0 \) is a small constant), we solve the problem directly without recursive calls. Since the input size is fixed (bounded by \( n_0 \)), it takes \( O(1) \) time.
- We make \( a \) recursive calls. The input size of each is \( n/b \). Hence the runtime \( T(n/b) \).
- \( \Theta(f(n)) \) is the time needed by all other processing.
Master Theorem

For DaC algorithms, the runtime function often satisfies:

\[ T(n) = \begin{cases} 
O(1) & \text{if } n \leq n_0 \\
\alpha T\left(\frac{n}{b}\right) + \Theta(f(n)) & \text{if } n > n_0 
\end{cases} \]

- If \( n \leq n_0 \) (\( n_0 \) is a small constant), we solve the problem directly without recursive calls. Since the input size is fixed (bounded by \( n_0 \)), it takes \( O(1) \) time.
- We make \( \alpha \) recursive calls. The input size of each is \( n/b \). Hence the runtime \( T(n/b) \).
- \( \Theta(f(n)) \) is the time needed by all other processing.
- \( T(n) =? \)
Master Theorem

Master Theorem (Theorem 4.1, Cormen’s book.)

1. If \( f(n) = O(n^\log_b a - \epsilon) \) for some constant \( \epsilon > 0 \), then \( T(n) = \Theta(n^\log_b a) \).

2. If \( f(n) = \Theta(n^\log_b a) \), then \( T(n) = \Theta(n^\log_b a \log n) \).

3. If \( f(n) = \Omega(n^\log_b a + \epsilon) \) for some constant \( \epsilon > 0 \), and \( af(n/b) \leq cf(n) \) for some \( c < 1 \) for sufficiently large \( n \), then \( T(n) = \Theta(f(n)) \).

Example: MergeSort

We have \( a = 2, b = 2 \), hence \( \log_b a = \log_2 2 = 1 \). So \( f(n) = \Theta(n^1) = \Theta(n^\log_b a) \).

By statement (2), \( T(n) = \Theta(n^1 \log n) = \Theta(n \log n) \).
Master Theorem

Master Theorem (Theorem 4.1, Cormen’s book.)

1. If \( f(n) = O(n^{\log_b a - \epsilon}) \) for some constant \( \epsilon > 0 \), then \( T(n) = \Theta(n^{\log_b a}) \).
2. If \( f(n) = \Theta(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a} \log n) \).
3. If \( f(n) = \Omega(n^{\log_b a + \epsilon}) \) for some constant \( \epsilon > 0 \), and \( af(n/b) \leq cf(n) \) for some \( c < 1 \) for sufficiently large \( n \), then \( T(n) = \Theta(f(n)) \).

Example: MergeSort
We have \( a = 2, b = 2 \), hence \( \log_b a = \log_2 2 = 1 \). So \( f(n) = \Theta(n^1) = \Theta(n^{\log_b a}) \).
Master Theorem

Master Theorem (Theorem 4.1, Cormen’s book.)

1. If \( f(n) = O(n^{\log_b a - \epsilon}) \) for some constant \( \epsilon > 0 \), then \( T(n) = \Theta(n^{\log_b a}) \).
2. If \( f(n) = \Theta(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a \log n}) \).
3. If \( f(n) = \Omega(n^{\log_b a + \epsilon}) \) for some constant \( \epsilon > 0 \), and \( af(n/b) \leq cf(n) \) for some \( c < 1 \) for sufficiently large \( n \), then \( T(n) = \Theta(f(n)) \).

Example: MergeSort
We have \( a = 2, b = 2 \), hence \( \log_b a = \log_2 2 = 1 \). So \( f(n) = \Theta(n^1) = \Theta(n^{\log_b a}) \).

By statement (2), \( T(n) = \Theta(n^{\log_2 2 \log n}) = \Theta(n \log n) \).
Binary Search

- Input: Sorted array $A[1..n]$ and a number $x$
- Output: Find $i$ such that $A[i] = x$, if no such $i$ exists, output “no”.

```plaintext
We use a function BinarySearch($A$, $p$, $r$, $x$) that searches $x$ in $A[p..r]$.

1: if $p = r$ then
2: if $A[p] = x$ return $p$
3: if $A[p] \neq x$ return “no”
4: else
5: $q = (p + r) / 2$
6: if $A[q] = x$ return $q$
7: if $A[q] > x$ call BinarySearch($A$, $p$, $q - 1$, $x$)
8: if $A[q] < x$ call BinarySearch($A$, $q + 1$, $r$, $x$)
9: end if
```
Binary Search

- Input: Sorted array $A[1..n]$ and a number $x$
- Output: Find $i$ such that $A[i] = x$, if no such $i$ exists, output “no”.

We use a function $\text{BinarySearch}(A, p, r, x)$ that searches $x$ in $A[p..r]$. 
Binary Search

- Input: Sorted array $A[1..n]$ and a number $x$
- Output: Find $i$ such that $A[i] = x$, if no such $i$ exists, output “no”.

We use a function $\text{BinarySearch}(A, p, r, x)$ that searches $x$ in $A[p..r]$.

$\text{BinarySearch}(A, p, r, x)$

1: if $p = r$ then
2: if $A[p] = x$ return $p$
3: if $A[p] \neq x$ return “no”
4: else
5: $q = (p + r)/2$
6: if $A[q] = x$ return $q$
7: if $A[q] > x$ call $\text{BinarySearch}(A, p, q - 1, x)$
8: if $A[q] < x$ call $\text{BinarySearch}(A, q + 1, r, x)$
9: end if
Binary Search

Input: Sorted array $A[1..n]$ and a number $x$
Output: Find $i$ such that $A[i] = x$, if no such $i$ exists, output “no”.

We use a function $\text{BinarySearch}(A, p, r, x)$ that searches $x$ in $A[p..r]$.

\begin{algorithm}
\begin{algorithmic}
\State $\text{BinarySearch}(A, p, r, x)$
\State 1: if $p = r$ then
\State 2: \hspace{1em} if $A[p] = x$ return $p$
\State 3: \hspace{1em} if $A[p] \neq x$ return “no”
\State 4: \hspace{1em} else
\State 5: \hspace{2em} $q = (p + r)/2$
\State 6: \hspace{2em} if $A[q] = x$ return $q$
\State 7: \hspace{2em} if $A[q] > x$ call $\text{BinarySearch}(A, p, q - 1, x)$
\State 8: \hspace{2em} if $A[q] < x$ call $\text{BinarySearch}(A, q + 1, r, x)$
\State 9: end if
\end{algorithmic}
\end{algorithm}
Analysis of Binary Search

- If $n = p - r + 1 = 1$, it takes $O(1)$ time.
Analysis of Binary Search

- If \( n = p - r + 1 = 1 \), it takes \( O(1) \) time.
- If not, we make at most one recursive call, with size \( n/2 \).
Analysis of Binary Search

- If $n = p - r + 1 = 1$, it takes $O(1)$ time.
- If not, we make at most one recursive call, with size $n/2$.
- All other processing take $f(n) = \Theta(1)$ time.
Analysis of Binary Search

- If \( n = p - r + 1 = 1 \), it takes \( O(1) \) time.
- If not, we make at most one recursive call, with size \( n/2 \).
- All other processing take \( f(n) = \Theta(1) \) time.
- So \( a = 1, b = 2 \) and \( f(n) = \Theta(n^0) \) time.
  Since \( \log_b a = \log_2 1 = 0 \), \( f(n) = \Theta(n^{\log_b a}) \).
If $n = p - r + 1 = 1$, it takes $O(1)$ time.
If not, we make at most one recursive call, with size $n/2$.
All other processing take $f(n) = \Theta(1)$ time.
So $a = 1$, $b = 2$ and $f(n) = \Theta(n^0)$ time.
Since $\log_b a = \log_2 1 = 0$, $f(n) = \Theta(n^{\log_b a})$.
Hence $T(n) = \Theta(n^{\log_b a \log n}) = \Theta(\log n)$. 
Example

A function makes 4 recursive calls, each with size $n/2$. Other processing takes $f(n) = \Theta(n^3)$ time.
Example

A function makes 4 recursive calls, each with size $n/2$. Other processing takes $f(n) = \Theta(n^3)$ time.

$$T(n) = 4T(n/2) + \Theta(n^3)$$
A function makes 4 recursive calls, each with size $n/2$. Other processing takes $f(n) = \Theta(n^3)$ time.

$$T(n) = 4T(n/2) + \Theta(n^3)$$

We have $a = 4$, $b = 2$. So $\log_b a = \log_2 4 = 2$. 
A function makes 4 recursive calls, each with size $n/2$. Other processing takes $f(n) = \Theta(n^3)$ time.

$$T(n) = 4T(n/2) + \Theta(n^3)$$

We have $a = 4$, $b = 2$. So $\log_b a = \log_2 4 = 2$. $f(n) = n^3 = \Theta(n^{\log_b a+1}) = \Omega(n^{\log_b a+0.5})$.

This is the case 3 of Master Theorem. We need to check the 2\textsuperscript{nd} condition:
A function makes 4 recursive calls, each with size \( n/2 \). Other processing takes \( f(n) = \Theta(n^3) \) time.

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

We have \( a = 4, \ b = 2 \). So \( \log_b a = \log_2 4 = 2 \).

\[
f(n) = n^3 = \Theta(n^{\log_b a+1}) = \Omega(n^{\log_b a+0.5}).
\]

This is the case 3 of Master Theorem. We need to check the 2nd condition:

\[
a \cdot f(n/b) = 4 \left(\frac{n}{2}\right)^3 = \frac{4}{8} n^3 = \frac{1}{2} \cdot f(n)
\]

If we let \( c = 1/2 < 1 \), we have: \( a \cdot f(n/b) \leq c \cdot f(n) \).

Hence by case 3, \( T(n) = \Theta(f(n)) = \Theta(n^3) \).
Master Theorem

If \( f(n) \) has the form \( f(n) = \Theta(n^k) \) for some \( k \geq 0 \), We have the following:

A simpler version of Master Theorem

\[
T(n) = \begin{cases} 
O(1) & \text{if } n \leq n_0 \\
aT(n/b) + \Theta(n^k) & \text{if } n > n_0
\end{cases}
\]

1. If \( k < \log_b a \), then \( T(n) = \Theta(n^{\log_b a}) \).
2. If \( k = \log_b a \), then \( T(n) = \Theta(n^k \log n) \).
3. If \( k > \log_b a \), then \( T(n) = \Theta(n^k) \).
Master Theorem

If \( f(n) \) has the form \( f(n) = \Theta(n^k) \) for some \( k \geq 0 \), We have the following:

A simpler version of Master Theorem

\[
T(n) = \begin{cases} 
O(1) & \text{if } n \leq n_0 \\
\ aT(n/b) + \Theta(n^k) & \text{if } n > n_0 
\end{cases}
\]

1. If \( k < \log_b a \), then \( T(n) = \Theta(n^{\log_b a}) \).
2. If \( k = \log_b a \), then \( T(n) = \Theta(n^k \log n) \).
3. If \( k > \log_b a \), then \( T(n) = \Theta(n^k) \).

Only the case 3 is different. In this case, we need to check the 2\(^{nd}\) condition. Because \( k > \log_b a \), \( b^k > a \) and \( a/b^k < 1 \):

\[
a \cdot f(n/b) = a \cdot \left( \frac{n}{b} \right)^k = \frac{a}{b^k} \cdot f(n) = c \cdot f(n)
\]

where \( c = \frac{a}{b^k} < 1 \), as needed.
Master Theorem

- How to understand/memorize Master Theorem?
How to understand/memorize Master Theorem?

The cost of a DaC algorithm can be divided into two parts:

1. The total cost of all recursive calls is $\Theta(n \log b^a)$.
2. The total cost of all other processing is $\Theta(f(n))$.

If (1) > (2), (1) dominates the total cost: $T(n) = \Theta(n \log b^a)$.

If (1) < (2), (2) dominates the total cost: $T(n) = \Theta(f(n))$.

If (1) = (2), the cost of two parts are about the same, somehow we have an extra factor $\log n$. The proof of Master Theorem is given in textbook. We'll illustrate two examples in class.
How to understand/memorize Master Theorem?

The cost of a DaC algorithm can be divided into two parts:

1. The total cost of all recursive calls is $\Theta(n^{\log_b a})$.

If (1) > (2), (1) dominates the total cost: $T(n) = \Theta(n^{\log_b a})$.

If (1) < (2), (2) dominates the total cost: $T(n) = \Theta(f(n))$.

If (1) = (2), the cost of two parts are about the same, somehow we have an extra factor $\log n$.

The proof of Master Theorem is given in textbook.

We’ll illustrate two examples in class.
How to understand/memorize Master Theorem?

The cost of a DaC algorithm can be divided into two parts:

1. The total cost of all recursive calls is $\Theta(n^{\log_b a})$.
2. The total cost of all other processing is $\Theta(f(n))$.
How to understand/memorize Master Theorem?
The cost of a DaC algorithm can be divided into two parts:

1. The total cost of all recursive calls is $\Theta(n^{\log_b a})$.
2. The total cost of all other processing is $\Theta(f(n))$.

If (1) $>$ (2), (1) dominates the total cost: $T(n) = \Theta(n^{\log_b a})$. If (1) = (2), the cost of two parts are about the same, somehow we have an extra factor $\log n$. The proof of Master Theorem is given in the textbook. We’ll illustrate two examples in class.
How to understand/memorize Master Theorem?

The cost of a DaC algorithm can be divided into two parts:

1. The total cost of all recursive calls is $\Theta(n^{\log_b a})$.
2. The total cost of all other processing is $\Theta(f(n))$.

If (1) > (2), (1) dominates the total cost: $T(n) = \Theta(n^{\log_b a})$.
If (1) < (2), (2) dominates the total cost: $T(n) = \Theta(f(n))$. 

The proof of Master Theorem is given in textbook.
We'll illustrate two examples in class.
How to understand/memorize Master Theorem?

The cost of a DaC algorithm can be divided into two parts:

1. The total cost of all recursive calls is $\Theta(n^{\log_b a})$.
2. The total cost of all other processing is $\Theta(f(n))$.

If (1) > (2), (1) dominates the total cost: $T(n) = \Theta(n^{\log_b a})$.

If (1) < (2), (2) dominates the total cost: $T(n) = \Theta(f(n))$.

If (1) = (2), the cost of two parts are about the same, somehow we have an extra factor $\log n$. 

The proof of Master Theorem is given in textbook. 
We'll illustrate two examples in class.
Master Theorem

How to understand/memorize Master Theorem?
The cost of a DaC algorithm can be divided into two parts:

1. The total cost of all recursive calls is $\Theta(n^{\log_b a})$.
2. The total cost of all other processing is $\Theta(f(n))$.

If (1) > (2), (1) dominates the total cost: $T(n) = \Theta(n^{\log_b a})$.
If (1) < (2), (2) dominates the total cost: $T(n) = \Theta(f(n))$.
If (1) = (2), the cost of two parts are about the same, somehow we have an extra factor $\log n$.

The proof of Master Theorem is given in textbook.
We’ll illustrate two examples in class.
Example

For some simple cases, Master Theorem does not work.
Example

For some simple cases, Master Theorem does not work.

Example

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

For some simple cases, Master Theorem does not work.

Example

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

- \( a = 2, b = 2, \log_a b = \log_2 2 = 1.\)
- \( f(n) = n^1 \log n = n^{\log_b a} \log n \)
Example

For some simple cases, Master Theorem does not work.

Example

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

- \( a = 2, b = 2, \log_a b = \log_2 2 = 1 \).
- \( f(n) = n^1 \log n = n^{\log_b a} \log n \)
- \( f(n) = \Omega(n), \text{ but } f(n) \neq \Omega(n^{1+\epsilon}) \) for any \( \epsilon > 0 \).
Example

For some simple cases, Master Theorem does not work.

Example

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

- \( a = 2, b = 2, \log_a b = \log_2 2 = 1 \). \( f(n) = n^1 \log n = n^{\log_b a} \log n \)
- \( f(n) = \Omega(n) \), but \( f(n) \neq \Omega(n^{1+\epsilon}) \) for any \( \epsilon > 0 \).
- Master Theorem does not apply.
Example

For some simple cases, Master Theorem does not work.

Example

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

- \( a = 2, \ b = 2, \ \log_a b = \log_2 2 = 1. \ f(n) = n^1 \log n = n^{\log_b a} \log n \)
- \( f(n) = \Omega(n), \ \text{but} \ f(n) \neq \Omega(n^{1+\epsilon}) \ \text{for any} \ \epsilon > 0. \)
- Master Theorem does not apply.

Theorem

If \( T(n) = aT(n/b) + f(n) \), where \( f(n) = \Theta(n^{\log_b a} (\log n)^k) \), then
\[ T(n) = \Theta(n^{\log_b a} (\log n)^{k+1}). \]
For some simple cases, Master Theorem does not work.

Example

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

- \( a = 2, b = 2, \log_a b = \log_2 2 = 1 \).
- \( f(n) = n^1 \log n = n^{\log_b a} \log n \)
- \( f(n) = \Omega(n), \text{ but } f(n) \neq \Omega(n^{1+\epsilon}) \text{ for any } \epsilon > 0. \)
- Master Theorem does not apply.

Theorem

If \( T(n) = aT(n/b) + f(n) \), where \( f(n) = \Theta(n^{\log_b a}(\log n)^k) \), then
\[ T(n) = \Theta(n^{\log_b a}(\log n)^{k+1}). \]

In the above example, \( T(n) = \Theta(n \log^2 n) \)