Algorithms and Data Structures

Module 2

Lecture 11
‘Divide-and-Conquer’ strategy. MergeSort.

Greedy algorithms (reminder)

Key characteristics of a greedy algorithm:

- Can solve an optimization problem.
- Builds solution iteratively, adding one element after another.
- At each step, adds the element which is the best at the current situation.
- Does not revise the decisions (one-pass algorithm).

Divide-and-Conquer strategy

A ‘Divide-and-Conquer’ strategy:

1. **Divide** the given instance of the problem into several *independent smaller* instances of the *same* problem. Division does not necessarily means dividing the dataset into disjoint datasets; it can mean a more general sort of *reduction*.
2. **Solve** all smaller instances. Usually we ***recursively*** use the same algorithm for solving smaller instances.
3. **Combine** the solutions of the smaller instances into the solution of the initial problem instance.

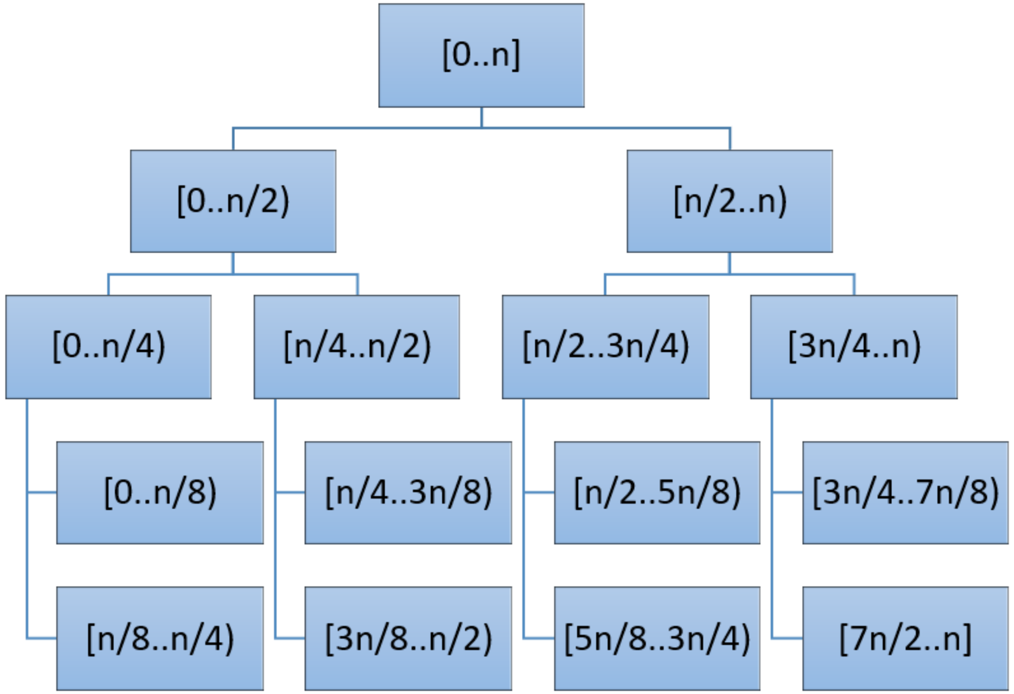
Divide-and-Conquer strategy

A ‘Divide-and-Conquer’ algorithm is usually implemented as a recursive function.

The run of a recursive algorithm

can be represented

as a *recursion tree*.



MergeSort

Task: given an array $A[0..n-1]$, sort it in ascending order.

MergeSort:

1. [**Divide**] Divide the array into two subarrays each of size approximately $n/2$.
2. [**Solve**] Recursively sort both subarrays, using MergeSort.
3. [**Combine**] Merge the sorted subarrays into the resulting sorted array.

MergeSort: steps' implementation

Divide:

Just divide the array $A[0..n-1]$ into two subarrays.

This can be made in an 'in-place' manner if we let the subarrays to be the segments $A\left[0..\frac{n}{2}-1\right]$ and $A\left[\frac{n}{2}..n-1\right]$.

Time complexity: $O(1)$.

MergeSort: steps' implementation

`Solve`:

Recursive call the `MergeSort` procedure.

!! For any recursive procedure we must provide a non-recursive branch. For `MergeSort`, we process short arrays non-recursively. Options:

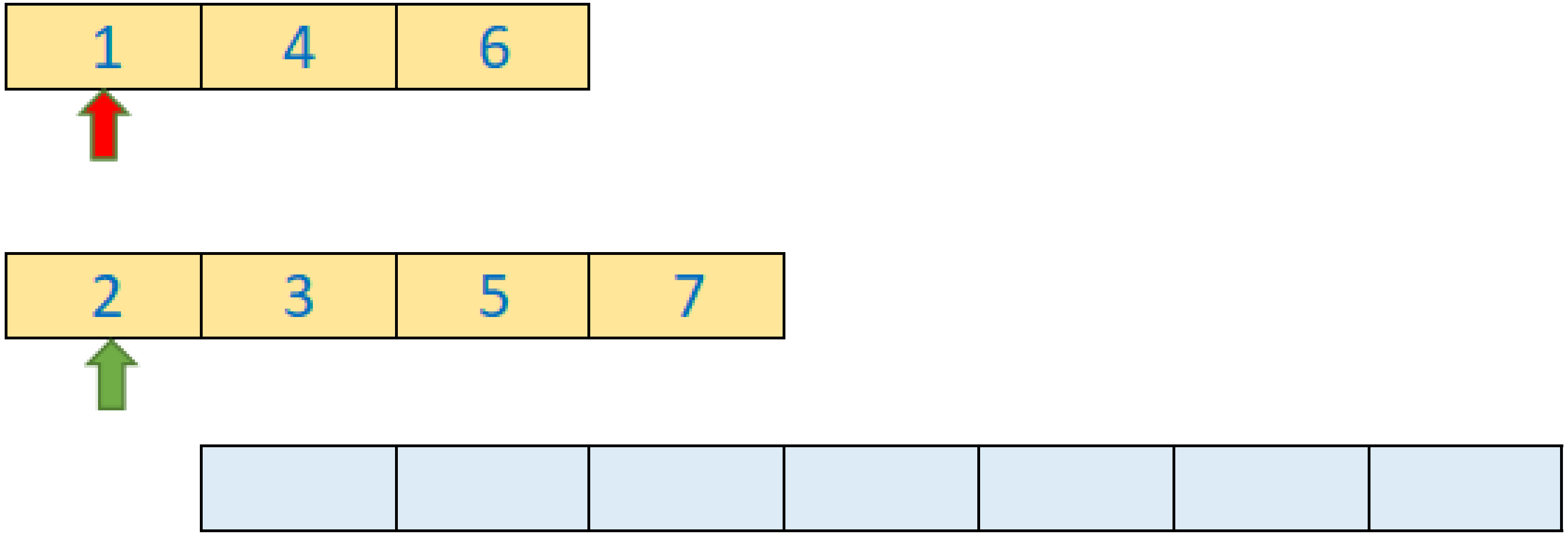
- For cases $n = 0$ and $n = 1$ no sorting needed.
- For small arrays (say, $n \leq 100$) running non-recursive sorting procedure (e.g. bubble sort) is more efficient.

Time complexity: depends on complexity of the 'Combine' step.

MergeSort: steps’ implementation

<u>`Combine`</u>:

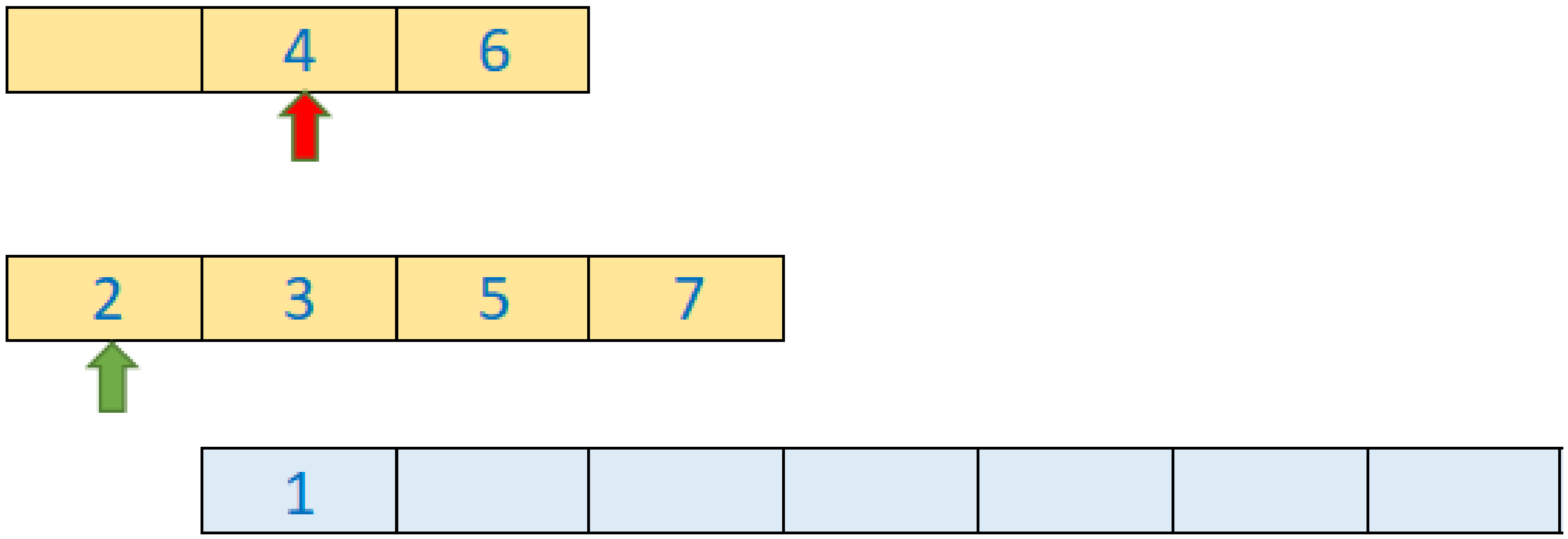
Given two sorted subarrays, how can we get one single sorted array? We compare the first items of subarrays.



MergeSort: steps' implementation

Combine:

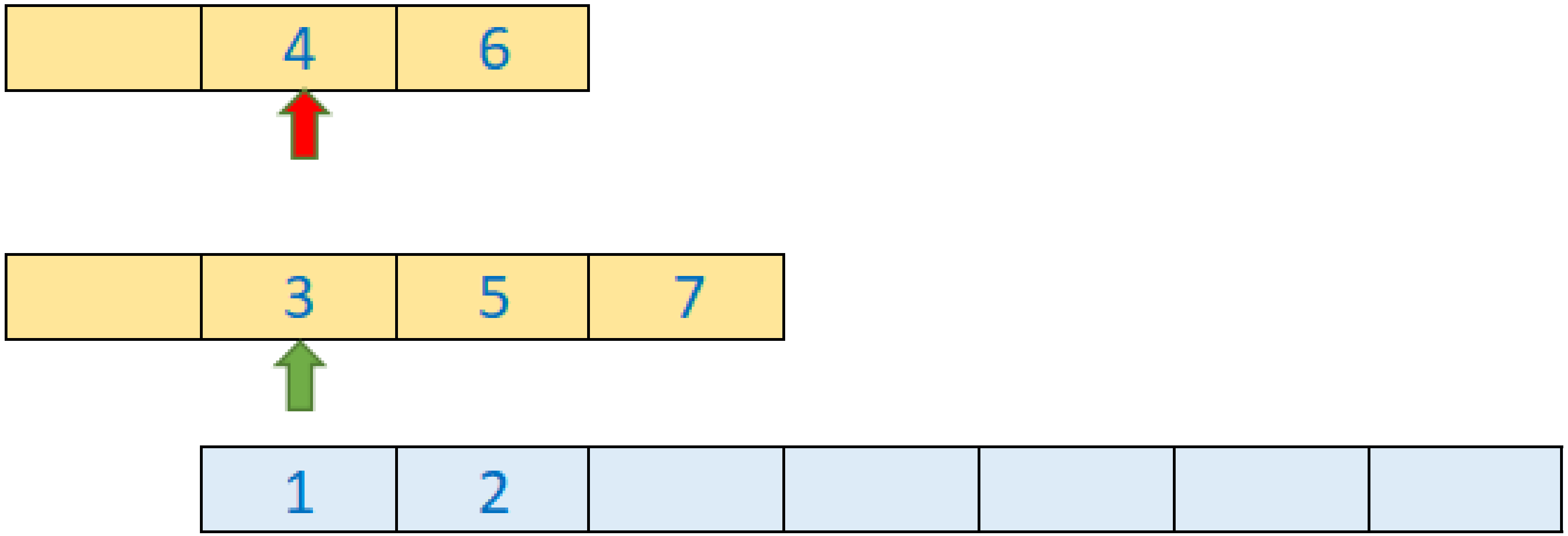
We put the least item to the target array and move the pointer in it's subarray to the next position.



MergeSort: steps' implementation

`Combine`:

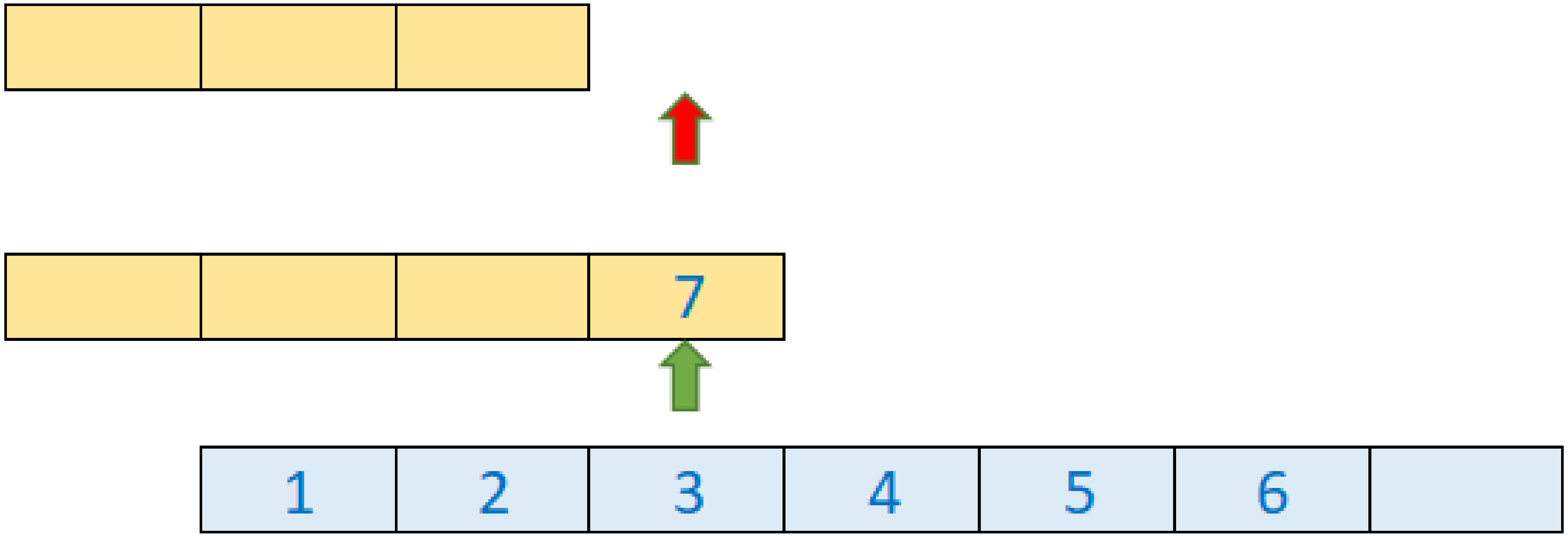
Then we repeat this procedure…



MergeSort: steps' implementation

<u>`Combine`</u>:

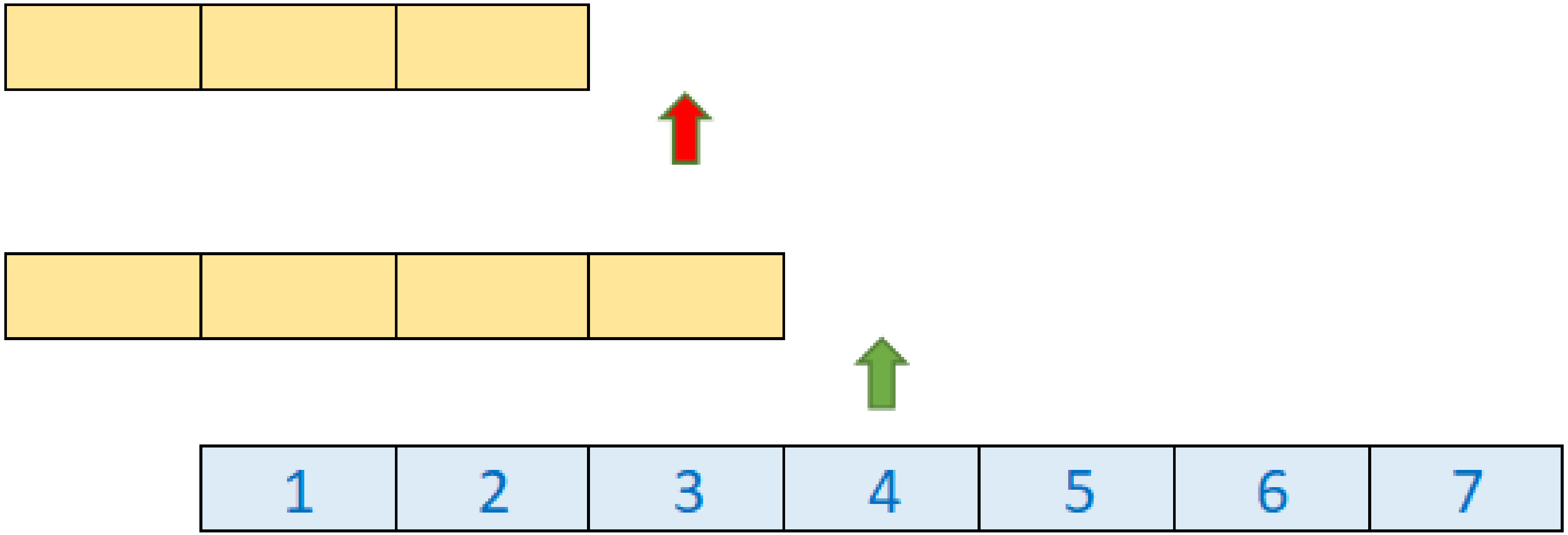
… until one of the subarrays becomes empty.



MergeSort: steps' implementation

<u>`Combine`</u>`:`

As a result we get the sorted array!



MergeSort: time complexity

Let’s evaluate the total time complexity of the `MergeSort` procedure.

`Divide`: takes $O(1)$ time.

`Solve`: `???`

`Combine`: takes $O(n)$ comparisons and assignments.

Let $T(n)$ be the time complexity of merge sorting for an array of size n.

MergeSort: time complexity

Let $T(n)$ be the time complexity of merge sorting for an array of size n.

$$T(n) = \begin{cases} c, & for \quad n = 1 \\ 2T\left(\frac{n}{2}\right) + dn, & for \quad n > 1 \end{cases}$$

After solving this *recurrence*, we get $T(n) = O(n \cdot \log_2 n)$.

Divide-and-Conquer: Master theorem

Consider a recursive algorithm of this form:

```
procedure p(input x of size n):
    if n < some constant k:
        Solve x directly without recursion
    else:
        Create a subproblems of x, each having size n/b
        Call procedure p recursively on each subproblem
        Combine the results from the subproblems
```

$f(n)$

https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)

Divide-and-Conquer: Master Theorem

Time complexity of this procedure is

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

The **Master Theorem**: there are 3 cases:

1) If $f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right)$ for some constant $\varepsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$.

2) If $f(n) = \Omega\left(n^{\log_b a} (\log n)^k\right)$ for $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} (\log n)^{k+1})$.

3) If $f(n) = \Omega\left(n^{\log_b a-\varepsilon}\right)$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

The regularity condition: $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n.

Divide-and-Conquer: Master Theorem

There is a simplified version of the Master Theorem for the case $f(n) = O(n)$.

The **Master Theorem (simplified version)**: there are 3 cases:

1) If $a < b$, then $T(n) = \Theta(n)$.

2) If $a = b$, then $T(n) = \Theta(n \log n)$.

3) If $a > b$, then $T(n) = \Theta(n^{\log_b a})$.

Divide-and-Conquer: Master Theorem

Let's consider the simplified case. For simplicity let $n = b^k$.

$$T(n) = \begin{cases} c, & for \quad n = 1 \\ aT\left(\frac{n}{b}\right) + dn, & for \quad n > 1 \end{cases}$$

Hence, $T(n) = aT\left(\frac{n}{b}\right) + dn = a\left[aT\left(\frac{n}{b^2}\right) + d \cdot \frac{\mathrm{n}}{b}\right] + dn = a^2 \cdot T\left(\frac{n}{b^2}\right) + dn \cdot \left(\frac{a}{b} + 1\right) = \cdots = a^k \cdot T\left(\frac{n}{b^k}\right) + dn \cdot \sum_{i=0}^{k-1}\left(\frac{a}{b}\right)^i = a^k c + dn \cdot \sum_{i=0}^{k-1}\left(\frac{a}{b}\right)^i.$

Divide-and-Conquer: Master Theorem

$T(n) = \cdots = a^k c + dn \cdot \sum_{i=0}^{k-1} \left(\frac{a}{b}\right)^i.$

<u>Case 1: $a < b$</u>

In this case $\sum_{i=0}^{k-1} \left(\frac{a}{b}\right)^i \longrightarrow const$, hence $T(n) = \Theta(a^k) + \Theta(n) =$

$\Theta(b^k) + \Theta(n) = \Theta(n).$

Divide-and-Conquer: Master Theorem

$T(n) = \cdots = a^k c + dn \cdot \sum_{i=0}^{k-1} \left(\frac{a}{b}\right)^i.$

<u>Case 2: $a = b$</u>

In this case $\frac{a}{b} = 1$ and $\sum_{i=0}^{k-1} \left(\frac{a}{b}\right)^i = k$, hence $T(n) = \Theta(a^k) + \Theta(n \cdot k) = \Theta(n) + \Theta(n \cdot \log n) = \Theta(n \log n)$.

Divide-and-Conquer: Master Theorem

$T(n) = \cdots = a^k c + dn \cdot \sum_{i=0}^{k-1} \left(\frac{a}{b}\right)^i.$

<u>Case 3: $a > b$</u>

Treat $\sum_{i=0}^{k-1} \left(\frac{a}{b}\right)^i$ as the partial sum of the geometric sequence: $\sum_{i=0}^{k-1} \left(\frac{a}{b}\right)^i = \frac{\left(\frac{a}{b}\right)^k - 1}{\frac{a}{b} - 1} = \cdot$

$\frac{a^k - b^k}{a - b} \frac{1}{b^k} = \Theta\left(\frac{a^k}{b^k}\right)$. Recall that $b^k = n$, hence $T(n) = a^k c + dn \cdot \sum_{i=0}^{k-1} \left(\frac{a}{b}\right)^i = a^k c + dn \cdot$

$\Theta\left(\frac{a^k}{n}\right) = a^k c + \Theta(a^k) = \Theta(a^k) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a}).$

Divide-and-Conquer: Master Theorem

Algorithm	Recurrence relationship	Run time
Binary search	$T(n) = T\left(\frac{n}{2}\right) + O(1)$	$O(\log n)$
Merge sort	$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$	$O(n \log n)$