# DESIGN AND ANALYSIS OF ALGORITHMS (DAA)

MASTER THEOREM

MERGE SORT

Course Instructor: Dr. Shreya Ghosh

# MASTER THEOREM

- When analyzing algorithms, recall that we only care about the **asymptotic behavior**.

- Recursive algorithms are no different. Rather than solve exactly the recurrence relation associated with the cost of an algorithm, it is enough to give an asymptotic characterization.

- The main tool for doing this is the master theorem



"Asymptotic" refers to the behavior of a function as its argument or input heads towards a particular value or infinity.

# MASTER THEOREM

**Theorem (Master Theorem)**

*Let* $T(n)$ *be a monotonically increasing function that satisfies*

$$\begin{aligned} T(n) &= aT(\tfrac{n}{b}) + f(n) \\ T(1) &= c \end{aligned}$$

*where* $a \geq 1, b \geq 2, c > 0$. *If* $f(n) \in \Theta(n^d)$ *where* $d \geq 0$, *then*

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

# MASTER THEOREM - PITFALLS

You *cannot* use the Master Theorem if

- $T(n)$ is not monotone, ex: $T(n) = \sin n$
- $f(n)$ is not a polynomial, ex: $T(n) = 2T(\frac{n}{2}) + 2^n$
- $b$ cannot be expressed as a constant, ex: $T(n) = T(\sqrt{n})$

Note here, that the Master Theorem does *not* solve a recurrence relation.

# MASTER THEOREM - EXAMPLES

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

$$\begin{aligned} a &= 1 \\ b &= 2 \\ d &= 2 \end{aligned}$$

Therefore which condition?

Since $1 < 2^2$, case 1 applies.

Thus we conclude that

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

# MASTER THEOREM - EXAMPLES

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

$$\begin{aligned} a &= 2 \\ b &= 4 \\ d &= \frac{1}{2} \end{aligned}$$

Therefore which condition?

Since $2 = 4^{\frac{1}{2}}$, case 2 applies.

Thus we conclude that

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

# MASTER THEOREM - EXAMPLES

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

$$\begin{aligned} a &= 3 \\ b &= 2 \\ d &= 1 \end{aligned}$$

Therefore which condition?

Since $3 > 2^1$, case 3 applies. Thus we conclude that

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

# FOURTH CONDITION

Recall that we cannot use the Master Theorem if f(n) (the non-recursive cost) is not polynomial.

There is a limited 4-th condition of the Master Theorem that allows us to consider polylogarithmic functions.

**Corollary**

*If* $f(n) \in \Theta(n^{\log_b a} \log^k n)$ *for some* $k \geq 0$ *then*

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



This final condition is fairly limited and we present it merely for completeness

# "FOURTH" CONDITION EXAMPLE

Say that we have the following recurrence relation:

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

Clearly, $a = 2, b = 2$ but $f(n)$ is not a polynomial. However,

$$f(n) \in \Theta(n\log n)$$

for $k = 1$, therefore, by the 4-th case of the Master Theorem we can say that

$$T(n) \in \Theta(n\log^2 n)$$

# *ASIDE* UNDERSTANDING THE MASTER THEOREM

**Theorem (Master Theorem)**

*Let $T(n)$ be a monotonically increasing function that satisfies*

$$\begin{aligned} T(n) &= aT(\tfrac{n}{b}) + f(n) \\ T(1) &= c \end{aligned}$$

*where $a \geq 1, b \geq 2, c > 0$. If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then*

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

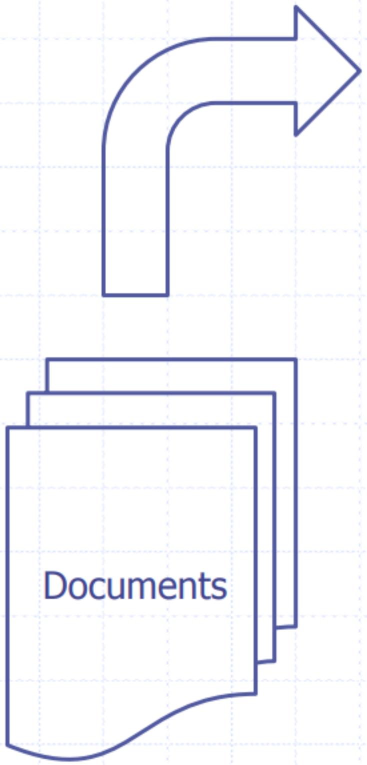
- a measures how many recursive calls are triggered by each method instance
- b measures the rate of change for input
- d measures the dominating term of the non recursive work within the recursive method
- c measures the work done in the base case

- The $\log_b a <$ d 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, $n^d$ term
- The $\log_b a = d$ 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 $\log_b a > d$ case
  - Recursive case breaks inputs apart quickly and doesn't do much non recursive work
  - Most work happens near bottom of call stack

# Application: Internet Search Engines

- Sorting has a lot of applications, including uses in Internet search engines.

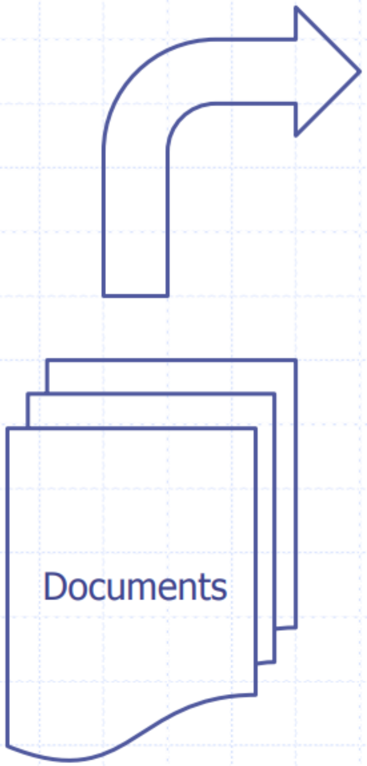
- Sorting arises in the steps needed to build a data structure, known as the **inverted file** or **inverted index**, that allows a search engine to quickly return a list of the documents that contain a given keyword.




| Word | Document Number & word location |
|---|---|
| banana | 1:3, 2:45 |
| butterfly | 2:15, 3:12 |
| camel | 4:40 |
| dog | 1:60, 1:70, 2:22, 3:20, 4:11 |
| horse | 4:21 |
| pig | 2:55 |
| pizza | 1:56, 3:33 |

# Application: How Sorting Builds an Internet Search Engine

- To build an inverted file we need to identify, for each keyword, k, the documents containing k.
- Bringing all such documents together can be done simply by sorting the set of keyword document pairs by keywords.
- This places all the (k, d) pairs with the same keyword, k, right next to one another.
- From this sorted list, it is then a simple computation to scan the list and build a lookup table of documents for each keyword that appears in this sorted list.



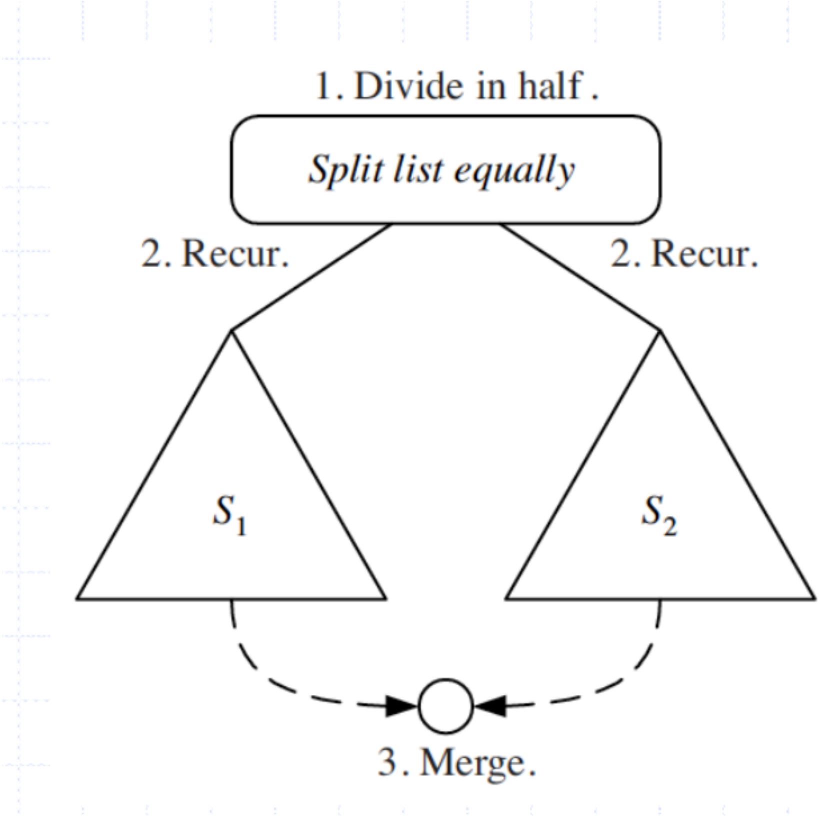

| Word | Document Number & word location |
|---|---|
| banana | 1:3, 2:45 |
| butterfly | 2:15, 3:12 |
| camel | 4:40 |
| dog | 1:60, 1:70, 2:22, 3:20, 4:11 |
| horse | 4:21 |
| pig | 2:55 |
| pizza | 1:56, 3:33 |

# Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:

  - Divide: divide the input data S in two disjoint subsets S1 and S2
  - Recur: solve the subproblems associated with S 1 and S2
  - Combine the solutions for S 1 and S2 into a solution for S

- The base case for the recursion are subproblems of size 0 or 1



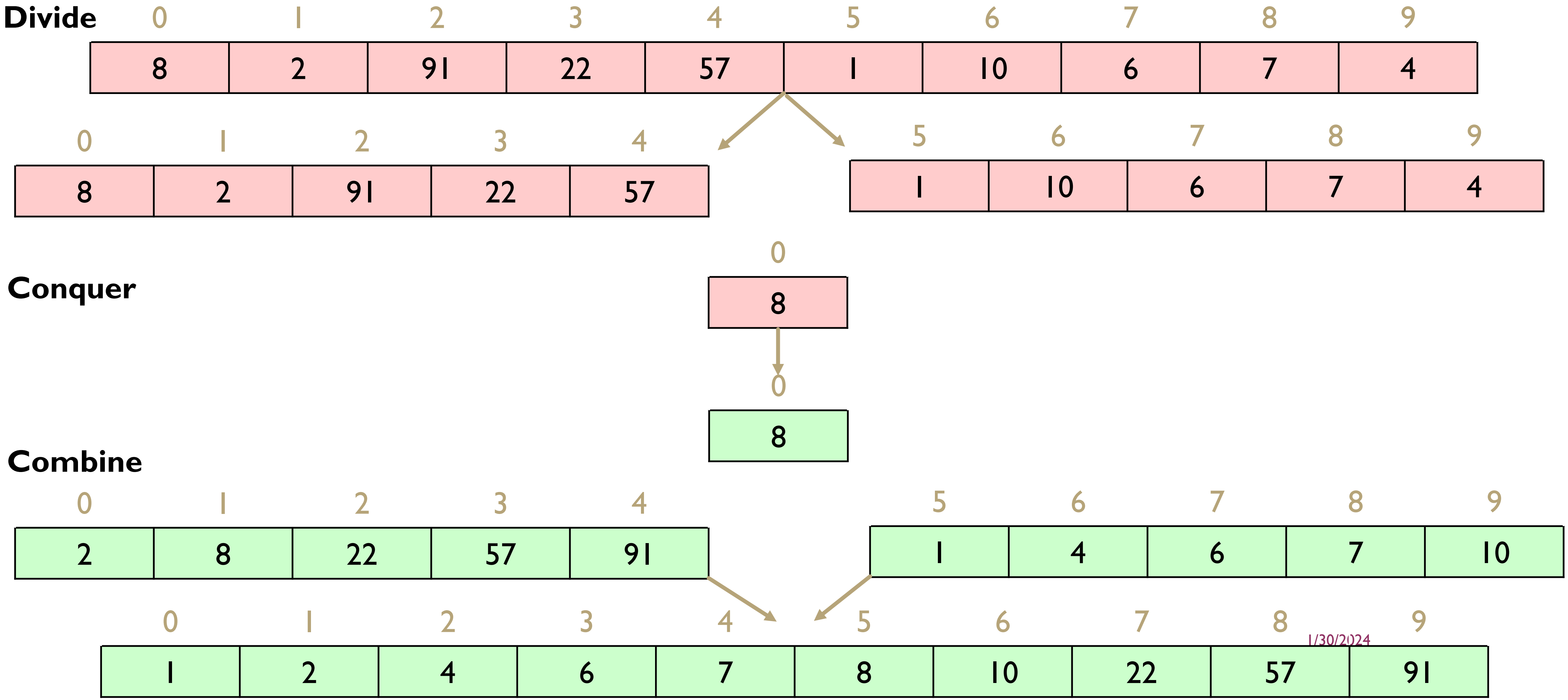
# The MERGE-SORT Algorithm

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*($S$)
    **Input** sequence $S$ with $n$ elements
    **Output** sequence $S$ sorted according to $C$
    **if** $S.size() > 1$
        $(S_1, S_2) \leftarrow partition(S, n/2)$
        $mergeSort(S_1)$
        $mergeSort(S_2)$
        $S \leftarrow merge(S_1, S_2)$

# Merge sort


Divide
0 1 2 3 4 5 6 7 8 9
8 2 91 22 57 1 10 6 7 4
0 1 2 3 4
8 2 91 22 57
5 6 7 8 9
1 10 6 7 4
Conquer
0
8
0
8
Combine
0 1 2 3 4
2 8 22 57 91
5 6 7 8 9
1 4 6 7 10
0 1 2 3 4 5 6 7 8 9
1 2 4 6 7 8 10 22 57 91

**Algorithm** ***mergeSort(S)***

**Input** sequence $S$ with $n$ elements

**Output** sequence $S$ sorted according to $C$

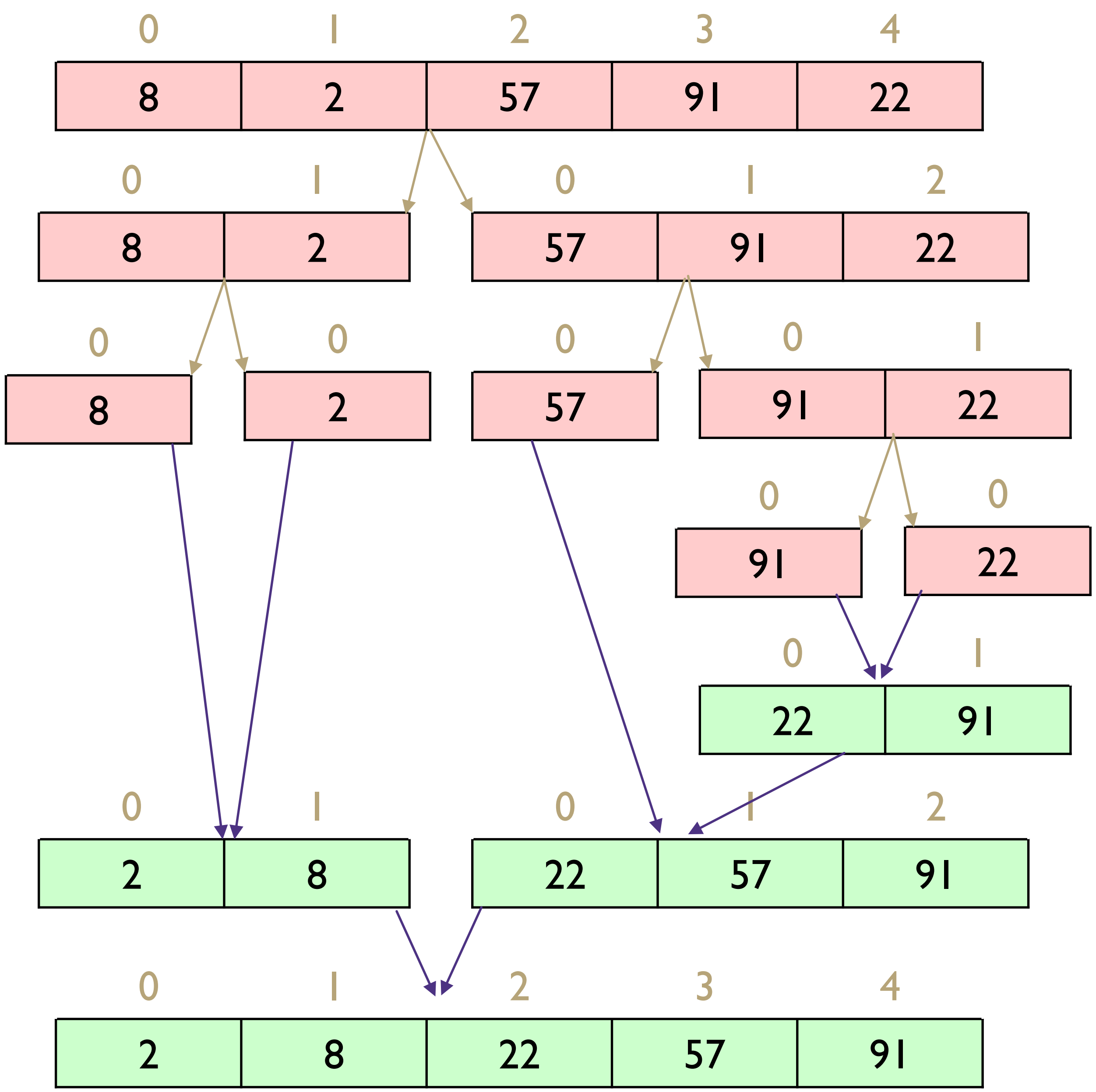
**if** $S.size() > 1$

$(S_1, S_2) \leftarrow partition(S, n/2)$

$mergeSort(S_1)$

$mergeSort(S_2)$

$S \leftarrow merge(S_1, S_2)$




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

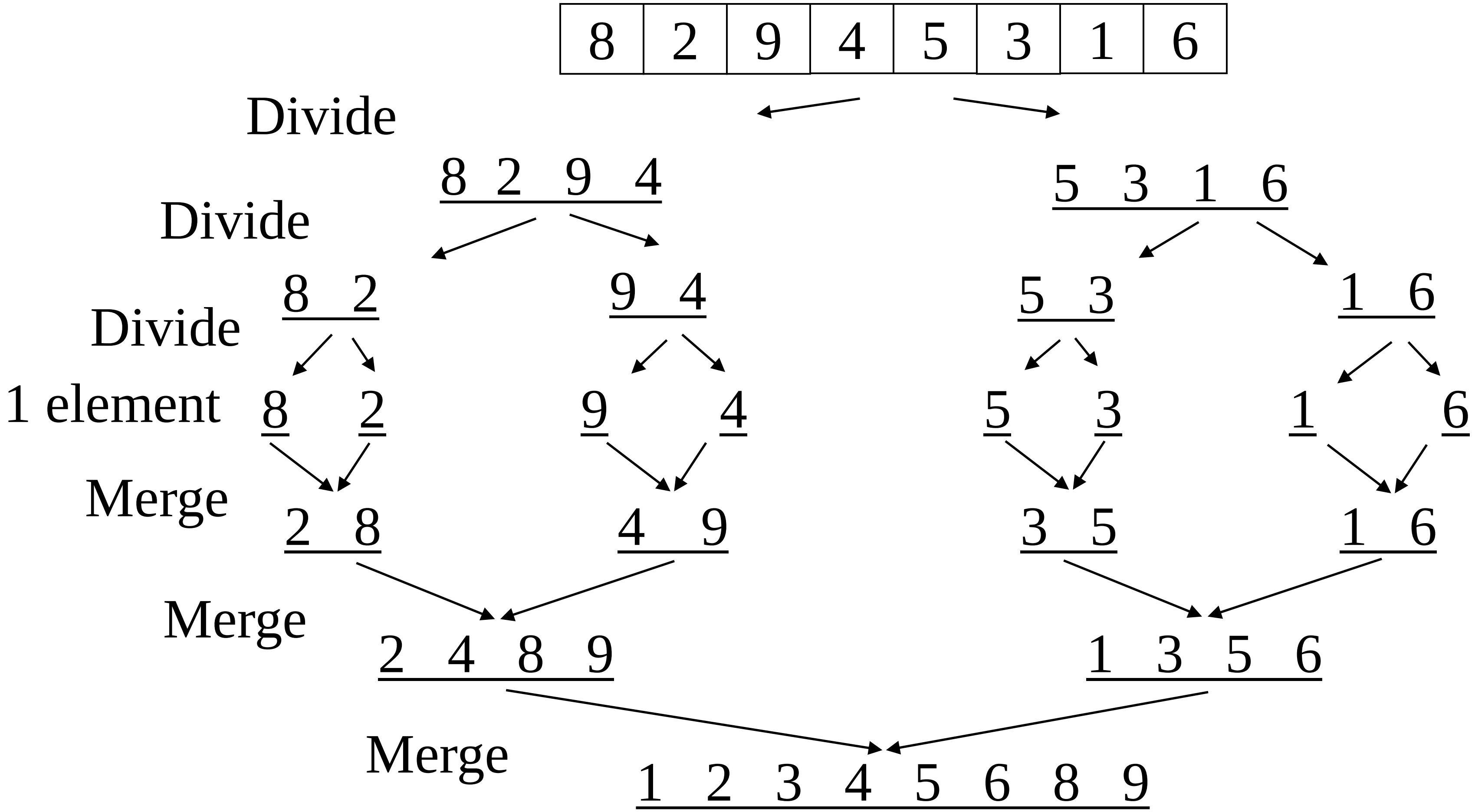
# The MERGE-SORT Algorithm

Merge-sort on an input sequence $S$ with $n$ elements consists of three steps:

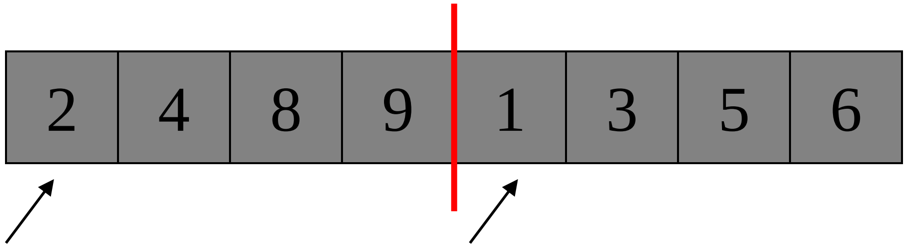
- **Divide**: partition $S$ into two sequences $S_1$ and $S_2$ of about $n/2$ elements each
- **Recur**: recursively sort $S_1$ and $S_2$
- **Conquer**: merge $S_1$ and $S_2$ into a unique sorted sequence

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

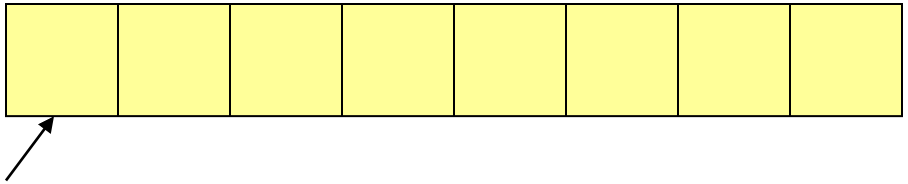
# MERGE SORT EXAMPLE



# AUXILIARY ARRAY

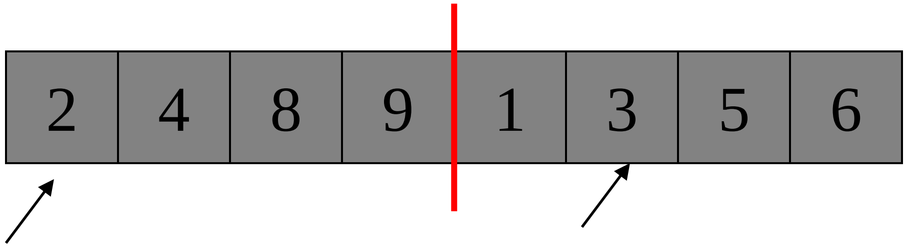



- The merging requires an auxiliary array.

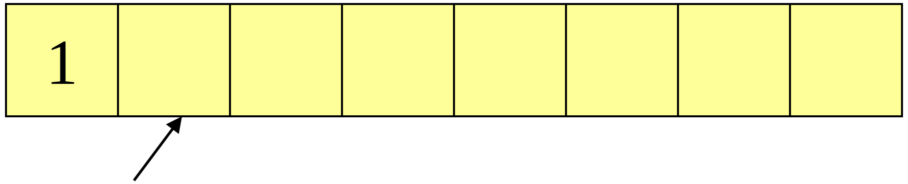


Auxiliary array

# AUXILIARY ARRAY

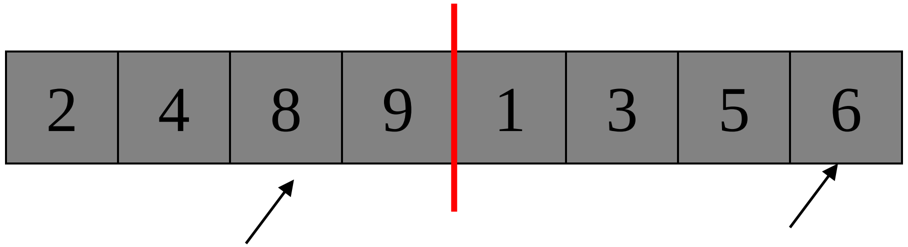



- The merging requires an auxiliary array.

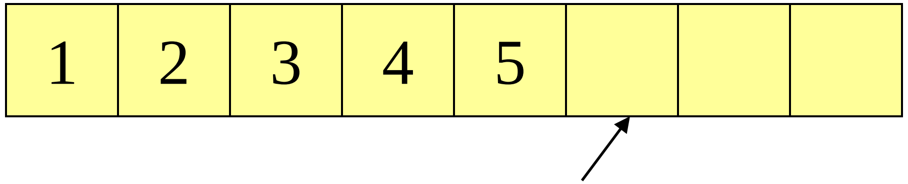



Auxiliary array

# AUXILIARY ARRAY

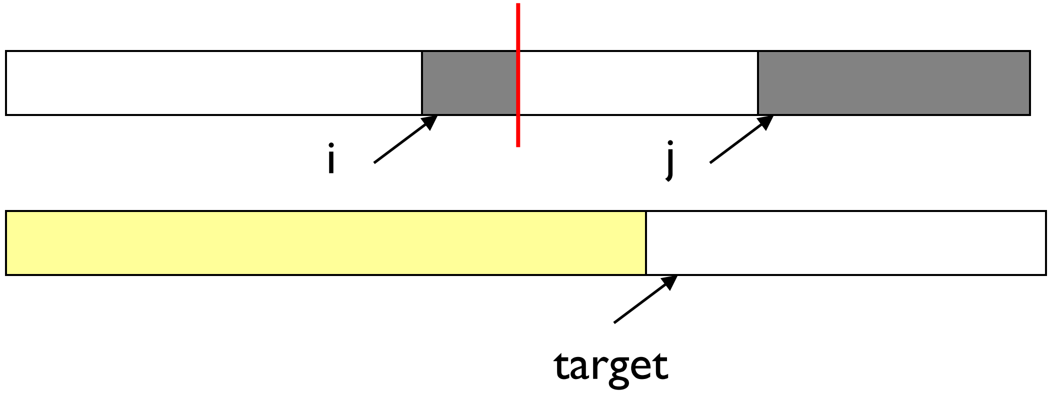



- The merging requires an auxiliary array.

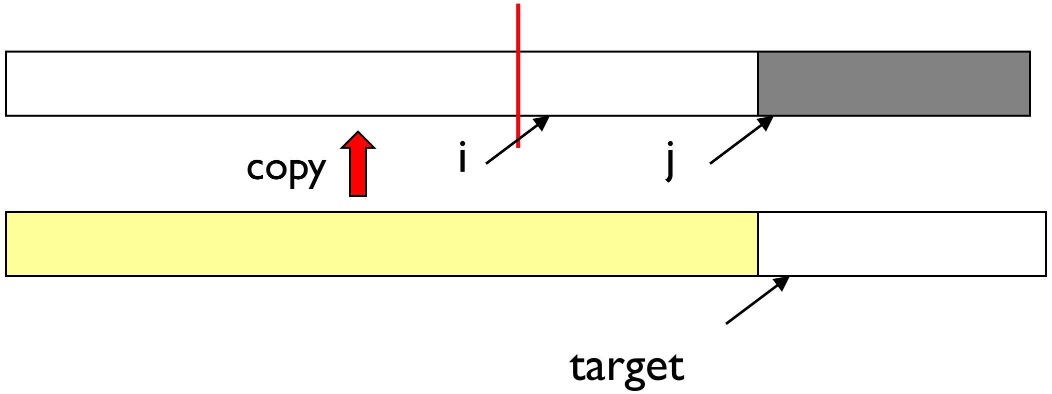



Auxiliary array

# MERGING

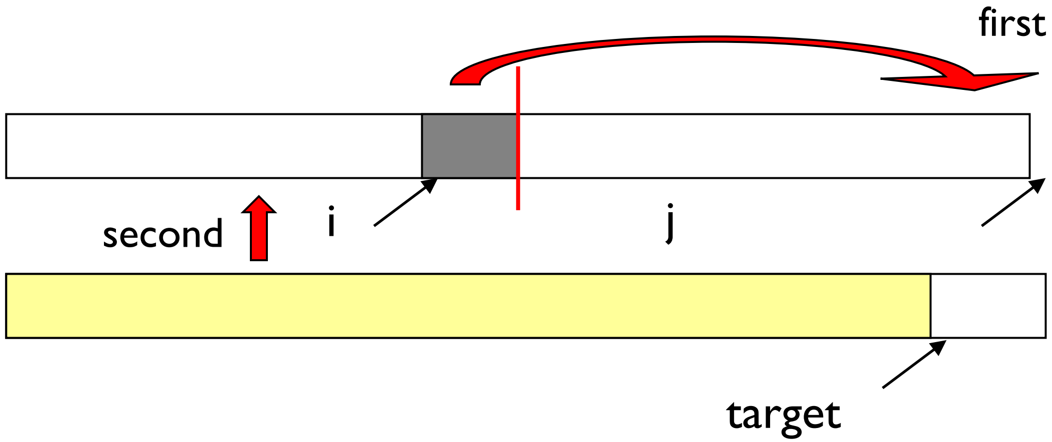



normal




Left completed first

# MERGING




Right completed first

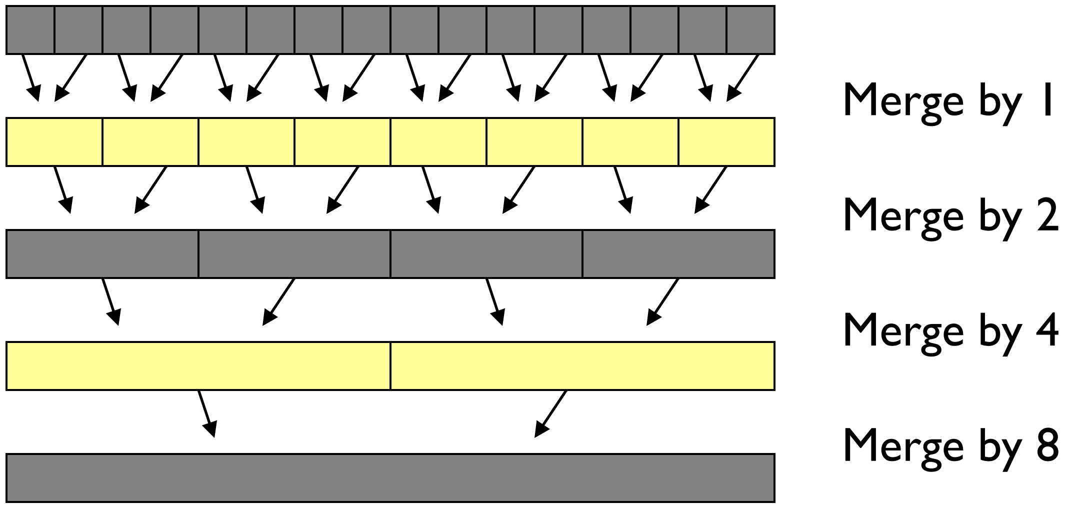
# MERGING

```
Merge(A[], T[] : integer array, left, right : integer) : {
  mid, i, j, k, l, target : integer;
  mid := (right + left)/2;

  i := left; j := mid + 1; target := left;

  while i ≤ mid and j ≤ right do
    if A[i] ≤ A[j] then T[target] := A[i] ; i:= i + 1;
      else T[target] := A[j]; j := j + 1;
    target := target + 1;

  if i > mid then //left completed//
    for k := left to target-1 do A[k] := T[k];
  if j > right then //right completed//
    k : = mid; l := right;
    while k ≥ i do A[l] := A[k]; k := k-1; l := l-1;
    for k := left to target-1 do A[k] := T[k];
}
```

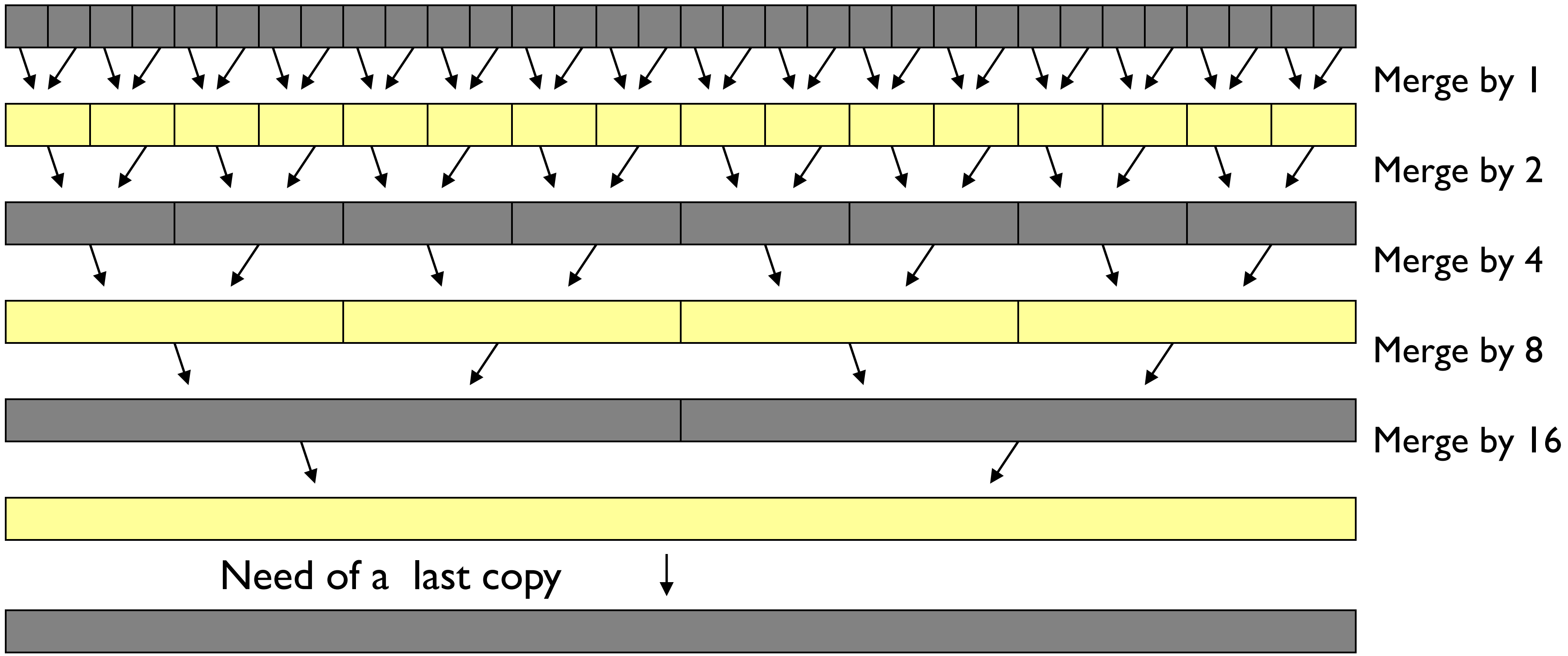
# RECURSIVE MERGESORT

```
Mergesort(A[], T[] : integer array, left, right : integer) : {
  if left < right then
    mid := (left + right)/2;
    Mergesort(A,T,left,mid);
    Mergesort(A,T,mid+1,right);
    Merge(A,T,left,right);
}

MainMergesort(A[1..n]: integer array, n : integer) : {
  T[1..n]: integer array;
  Mergesort[A,T,1,n];
}
```

# ITERATIVE MERGESORT



# ITERATIVE MERGESORT



## What is the Big-Theta of worst-case Merge Sort?

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

### MASTER THEOREM

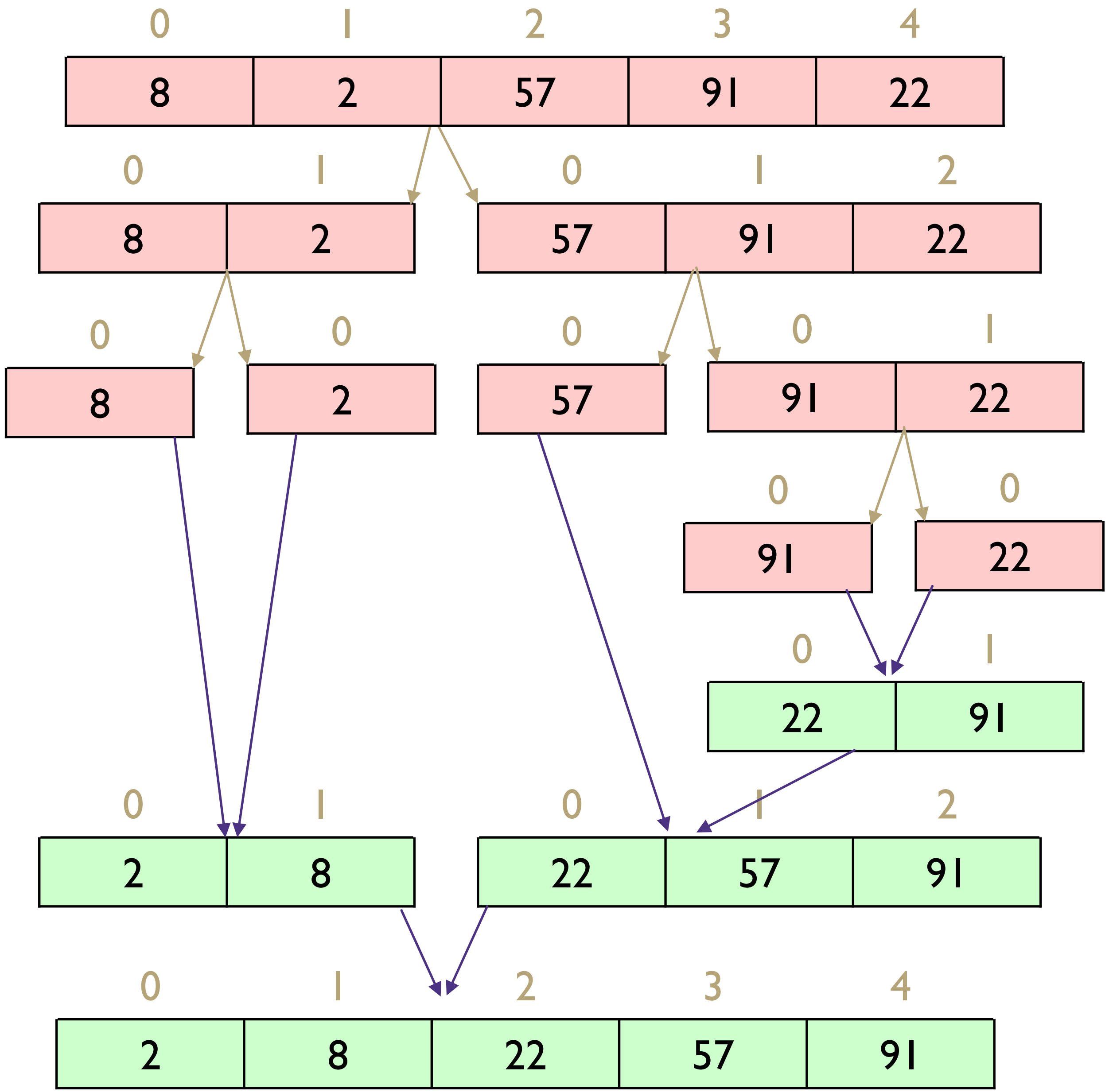
$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

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 $T(n) \in \Theta\left(n^{\log_b a}\right)$



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

**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 $\Theta(n^c)$

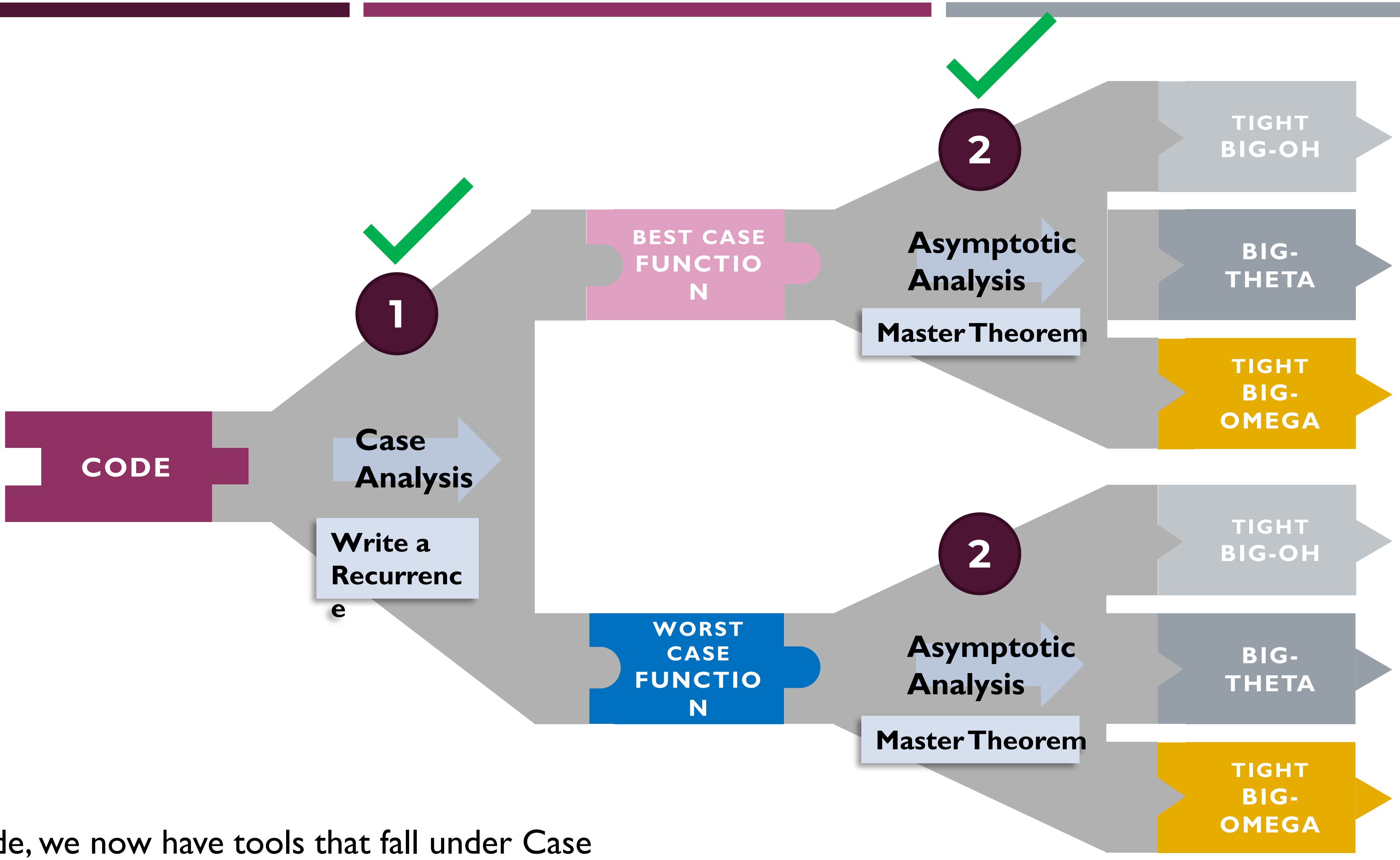
| | | | |
|---|---|---|---|
| If | $\log_b a < c$ | then | $T(n) \in \Theta(n^c)$ |
| If | $\log_b a = c$ | then | $T(n) \in \Theta(n^c \log n)$ |
| If | $\log_b a > c$ | then | $T(n) \in \Theta\left(n^{\log_b a}\right)$ |

*a=2 b=2 and c=1*

$\log_2 2 = 1$

**We're in case 2**

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




For recursive code, we now have tools that fall under Case Analysis (Writing Recurrences) and Asymptotic Analysis (The Master Theorem).