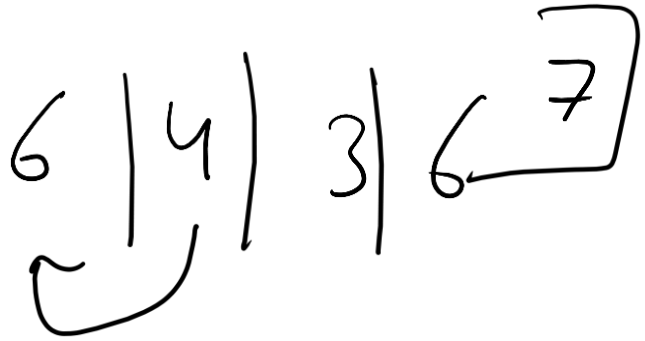
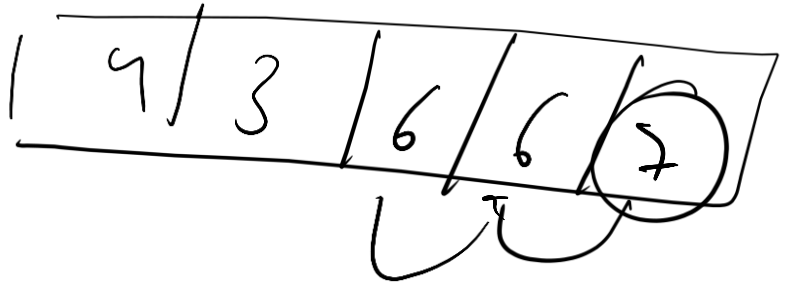
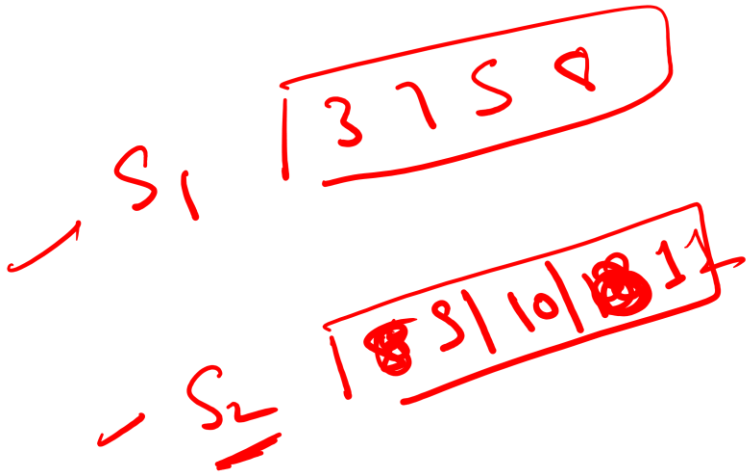
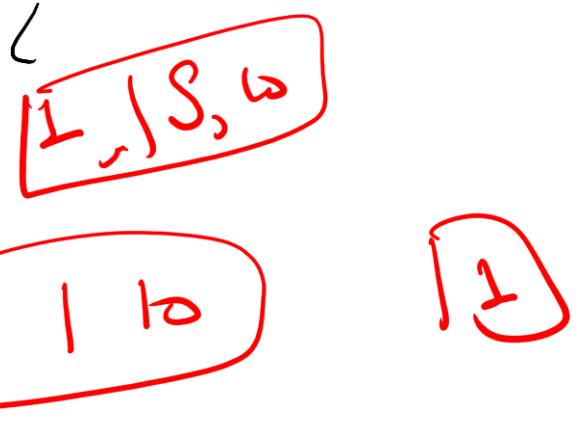
```
InsertionSort (array Arr):
    1.  for i in range(1, len(Arr)):
    2.      current_element = Arr[i]
    3.      j = i - 1
    4.      # Move elements of arr[0..i-1], that are greater than current_element,
    5.      # one position ahead of their current position
    6.      while j >= 0 and current_element < Arr[j]:
    7.          Arr[j + 1] = Arr[j]
    8.          j = j-1
    9.      Arr[j + 1] = current_element
```


6 4 3 6 7


4 3 6 6 7





	InsertionSort (array Arr)	Cost	Times
1	**for** i **in range** $n(1, len(Arr))$:	$c1$	n
2	$current\_element = Arr[i]$	$c2$	$n-1$
3	$j = i - 1$	$c3$	$n-1$
4	# *Move elements …current_element,*	0	$n-1$
5	# *one position ahead*	0	$n-1$
6	**while** $j > 0$ *and* $current\_element < Arr[j]$:	$c4$	$\sum_{j=2}^{n} iter_j$
7	$Arr[j+1] = Arr[j]$	$c5$	$\sum_{j=2}^{n} iter_j - 1$
8	$j = j - 1$	$c6$	$\sum_{j=2}^{n} iter_j - 1$
9	$Arr[j+1] = current\_element$	$c7$	$n-1$

1+2+3+4... n(n+1)/2

= summation

(n-1)(n-1+1)/2 = n(n-1)/2

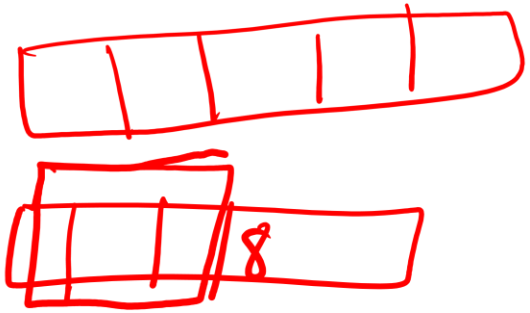
$$T(n) = c1(n) + c2(n-1) + c3(n-1) + c4\left(\frac{n(n+1)}{2} - 1\right)$$
$$+ c5\left(\frac{n(n-1)}{2}\right) + c6\left(\frac{n(n-1)}{2}\right) + c7(n-1),$$

$$T(n) = 0.5(c4 + c5 + c6)n^2 + 0.5(2(c1 + c2 + c3 + c7)$$
$$+ (c4 - c5 - c6))n - (c2 + c3 + c4 + c7).$$

Theta(n^2)

- The employment of **loop invariants aids in comprehending the validity of an algorithm**. It necessitates demonstrating three key conditions of loop invariant:

Condition 1 (Initialization):
This condition holds before the initial iteration of the loop.



Condition 2 (Maintenance):
If the condition is true before a loop iteration, it will persist before the subsequent iteration.

Condition 3 (Termination):
Upon the conclusion of the for loop, such invariant furnishes an important feature that aids in demonstrating the correctness of the algorithm.

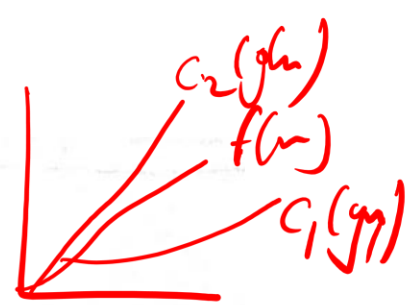
Now we have seen how the insertion sort operates, let us look at how these characteristics relate to the insertion sort algorithm. Let us utilize the example array Arr[2,4,3,6,8,9] to apply the loop invariant to the Insertion sort algorithm.

Condition 1: We initiate insertion sort by demonstrating the validity of the loop invariant at onset of initial iteration, where i=2. In this case, the subarray Arr[1:i-1] comprises only the element Arr[1], which is essentially the original element in Arr[1]. Furthermore, it is evident that this subarray is inherently sorted, affirming the preservation of loop invariant preceding initial iteration.

Condition 2: Next, we address 2nd condition, where iterations loop invariant should be maintained. In simpler terms, the operations of the for loop involves shifting elements, and so forth, one position to the right, until it identifies the correct position for Arr[i]. Subsequently, it inserts the value of Arr[i] at that position. The subarray now contains elements Arr[1:i-1], following sorted order. The increment of i subsequent iteration of the loop ensures preservation of the loop invariant.

Condition 3: Lastly, let us analyze what occurs upon the termination of the loop. The condition that leads to the termination of the for loop is that j>A:length equals n. Since each iteration increments j by 1, it follows that j=n+1 when the loop terminates. By substituting n+1 for j in the context of the loop invariant, we affirm that the subarray Arr[1:n] comprises the elements originally in Arr[1:n], now arranged in a sorted order.

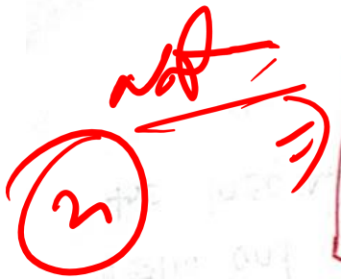
Asymptotic Analysis: $O-notation$



* The Θ-notation asymptotically bounds a function from above and below.

* When, we have only an asymptotic upper bound, we use O-notation

* For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of g of n" or sometimes just "oh of g of n") the set of functions.



$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$

$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \Rightarrow f(n) = O(g(n))$

Precise definition: "Schaum's outlines"

Suppose $f(n)$ and $g(n)$ are functions defined on positive integers with the property that $f(n)$ is bounded by some multiplier of $g(n)$ for almost all n. That is, suppose there exist a positive integer n_0 and a positive number c such that, for all $n \ge n_0$, we have

O

$f(n) = 3n^2 - 5$ $\quad g(n) = n^2$

$0 \le f(n) \le c\,g(n)$

$3n^2 - 5 \le c\,n^2$

$3 - \frac{5}{n^2} \le c$

$\lim_{n \to \infty} \Rightarrow 3 \le c$

$c = 3$

$n_0 = 1$

$3 - 5 = -2$

$n_0 = 2$

(7)

$n_0 = 2$

$c = 3$

$3n^2 - 5 \le 3(n^2)$

$\Downarrow$

$f(n) = 3n^2 - 5$

$\Downarrow$

$O(n^2)$

Asymptotic Analysis: $O-notation$

$$|f(n)| \leq c\,|g(n)|$$

This is also written as: $f(n) = O(g(n))$

It is read as "$f(n)$ is of order $g(n)$." For any polynomial $P(n)$ of degree m, we shall be solved that $P(n) = O(n^m)$; e.g.;

$$8n^3 - 576n^2 + 832n - 28 = O(n^3)$$

n_0

$O(n^3)$

Recursive analysis

Master theorem

We can also write

$$f(n) = h(n) + O(g(n)) \text{ when } f(n) - h(n) = O(g(n))$$

The complexity of certain well-known searching and sorting algorithms:

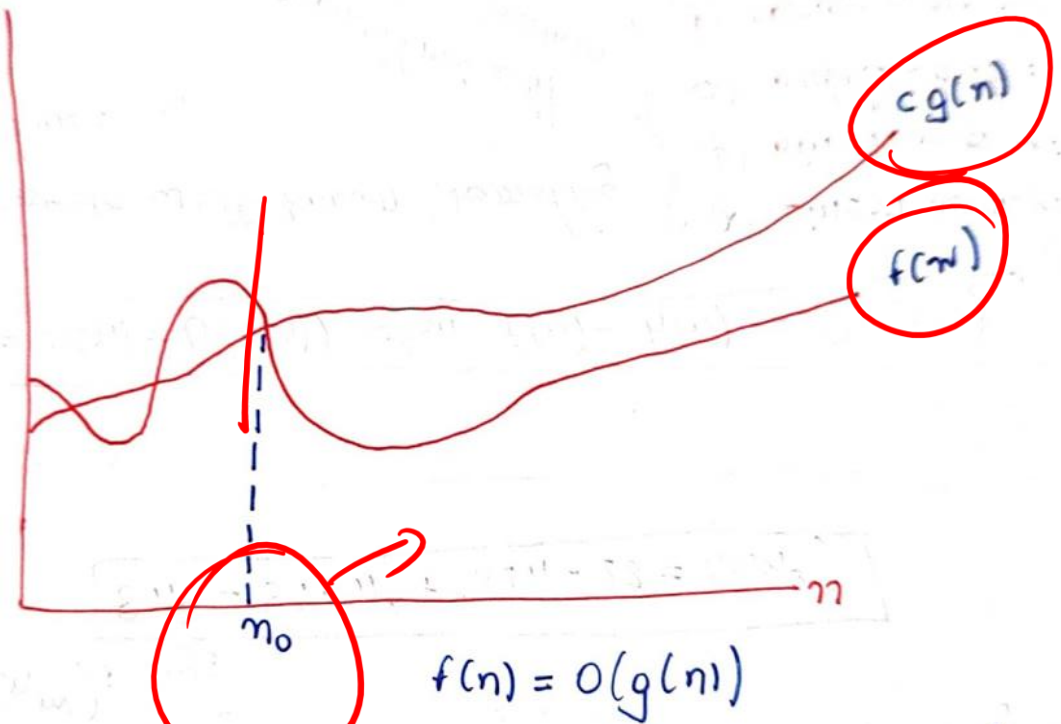
(a) Linear search: $O(n)$
(b) Binary search: $O(\log n)$
(c) Bubble sort: $O(n^2)$
(d) Merge sort: $O(n \log n)$

the list should be shortened

These algorithms will be discussed in further lectures.

Asymptotic Analysis: $O - notation$

* We use O-notation to give an upper bound on a function, to within a constant factor.



$f(n) = O(g(n))$

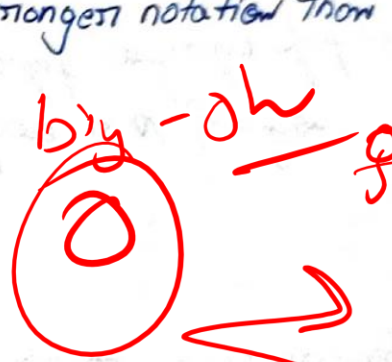
* For all values n at and to the right of n_0, the value of the function $f(n)$ is on or below $c\,g(n)$.

Asymptotic Analysis: $O-notation$

* We write $f(n) = O(g(n))$ to indicate that a function $f(n)$ is a member of the set $O(g(n))$.
* $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$ since Θ-notation is a stronger notation than O-notation.
* Written set-theoretically, we have $\Theta(g(n)) \subseteq O(g(n))$.
* Our proof that any quadratic function an^2+bn+c where $a>0$ is in $\Theta(n^2)$ also shows that any such quadratic function is in $O(n^2)$.
* A surprizing aspect:

  when $a>0$, any linear function $\{an+b\}$ is $O(n^2)$

* It is easily verified by taking $c = a+|b|$ and $n_0 = \max(1, -b/a)$.



linear
~~quadratic~~ expression
~~an^2+bn+c~~
$an+b$

$c = a+|b|$

$\Downarrow$

$n_0 = \max(1, -b/a)$

Asymptotic Analysis: $O - notation$

* Using O-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure.
* For example, the doubly nested loop structure of insertion sort algorithm immediately yields an $O(n^2)$ upper bound on the worst-case running time:
* Since O-notation describes an upper bound, when we use it to bound the worst-case running time of an algorithm, we have a bound on the running time of the algorithm on every input.
* $O(n^2)$ bound on worst-case running time of insertion sort also applies to its running time on every input.
* When we say "the running time is $O(n^2)$", we mean that there is a function $f(n)$ that is $O(n^2)$ such that for any value of n, no matter what particular input of Size n

Asymptotic Analysis: Ω – *notation*

Big-omega notation

Average → ① Big theta (Θ)

Worst case → ② Big oh (O)

Best-case → ③ Big omega (Ω)

* Just as O-notation provides an asymptotic upper bound on a function, Ω-notation provides an asymptotic lower bound.

For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions.

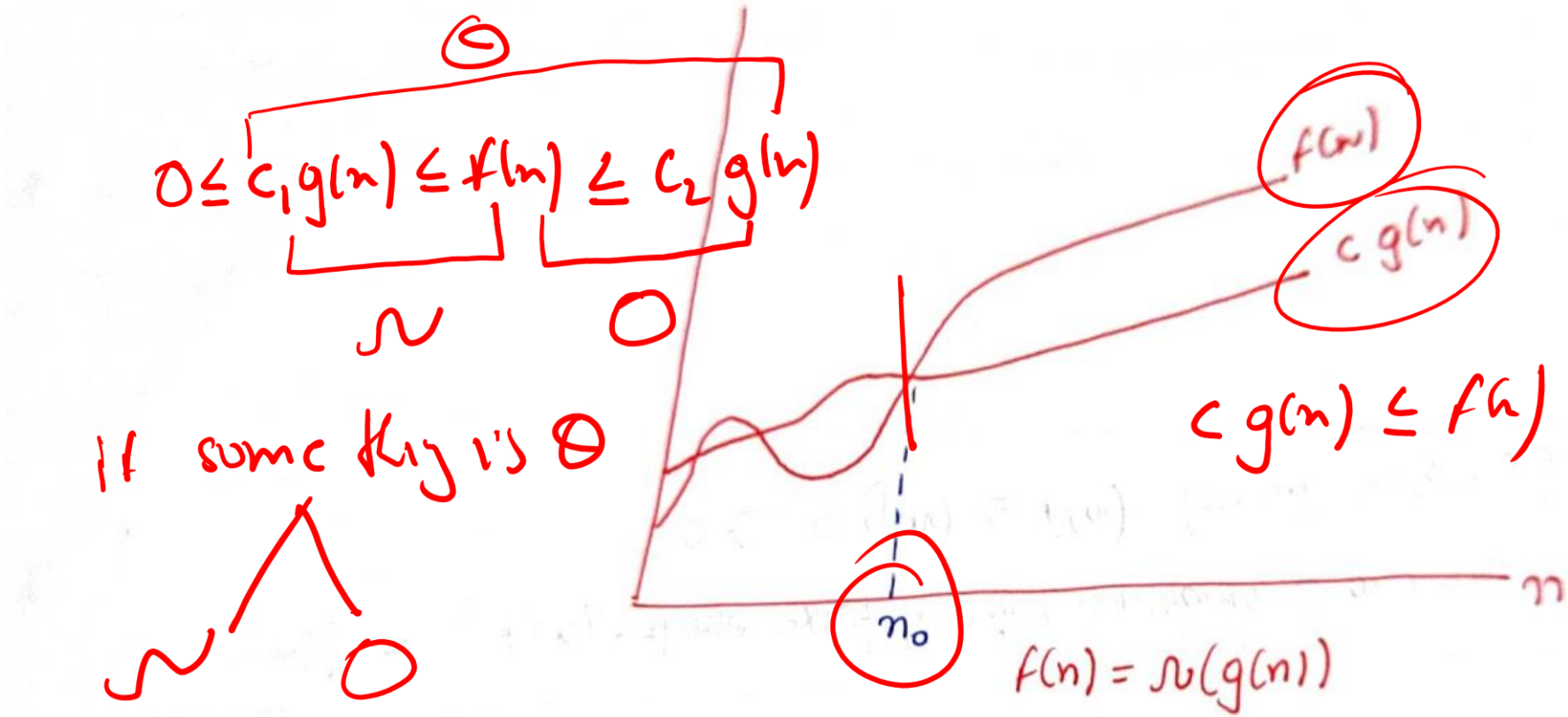
$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \text{ for all } n \ge n_0 \}$$

$$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$$

Θ ~ O

$$f(n) = \Omega(g(n))$$

Asymptotic Analysis: Ω – *notation*



For all values n at or to the right of n_0, the value of f(n) is on or above cg(n).

Asymptotic Analysis: Ω – *notation*

Definition:

$f(n) = \Omega(g(n))$ (read as f of n is omega of g of n), iff there exists a positive number n_0 and a positive number M such that $|f(n)| \geq M|g(n)|$, for all $n \geq n_0$.

For $f(n) = 18n + 9$, $f(n) > 18n$ for all n, hence $f(n) = \Omega(n)$. Also, for $f(n) = 90n^2 + 18n + 6$, $f(n) > 90n^2$ for $n \geq 0$ and therefore $f(n) = \Omega(n^2)$

* For $f(n) = \Omega(g(n))$, $g(n)$ is a lower bound function and there may be several such functions.

* However, it is appropriate that the function which is almost as large a function of (n) as possible such that the definition of Ω is satisfied, is chosen as $g(n)$.

Asymptotic Analysis: Ω – *notation*

Theorem 3.1

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Poof will cover afterwards

⇒ The earlier proof that $an^2 + bn + c = \Theta(n^2)$ for any constant a, b, and c, where $a > 0$, immediately implies that $an^2 + bn + c = \Omega(n^2)$ and $an^2 + bn + c = O(n^2)$.

In practice, rather than using the theorem to obtain asymptotic upper and lower bounds we usually use it to prove asymptotically tight bounds.

" When we say that the running time of an algorithm is $\Omega(g(n))$, we mean that no matter what particular input of size n is chosen for each value of n, the running time on that input is at least a constant time $g(n)$.

$f(n) = 6n^2 - 9$ $\quad g(n) = n^2$ $\quad f \sim g$

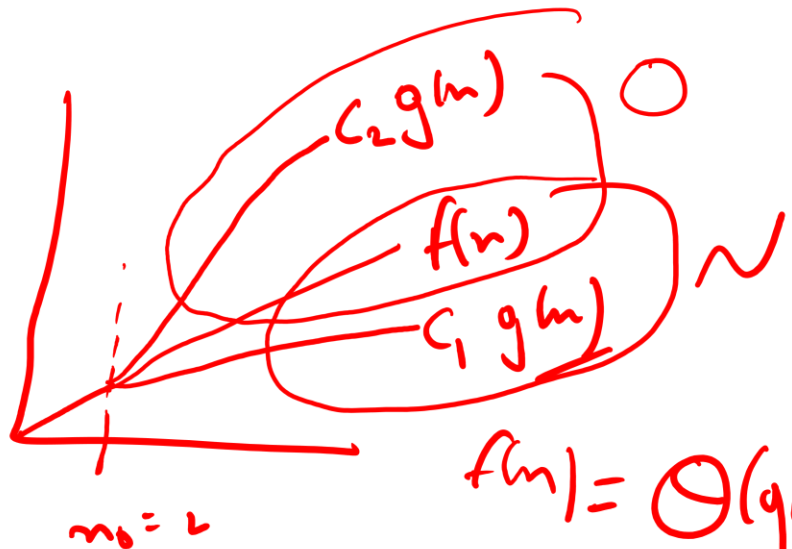
$\hookrightarrow \Theta \rightarrow \quad 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \Rightarrow \Theta$

$c_1 n^2 \le 6n^2 - 9 \le c_2 n^2$

$c_1 \le 6 - \frac{9}{n^2}$ $\quad \forall \; 6 \le c_2$

$n \to \infty \Rightarrow c_1 \le 6$

$c_1 = 5 \;\forall\; c_2 = 7$



$f(n) = \Theta(g(n))$

$f(n) \le c_2 (g(n)) \quad - \bigcirc$

$f(n) = O(g(n))$

$f(n) = \Omega(g(n))$

$\hookrightarrow O \Rightarrow$

$f(n) = O(g(n))$

Asymptotic Analysis: o – notation

o-notation "little-oh of g of n":

o-notation

$n^2+2 \Rightarrow n^2$

$n+2 \Rightarrow n^2$

* The asymptotic upper bound provided by O-notation may or may not be asymptotically tight.
* The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not.

$f(n) 2n \Leftarrow g(n) = n^2$

$\frac{2n}{n^2} = \frac{1}{n}$

$2n^2+3 \Rightarrow n^2$

$n \to \infty$

$\frac{f(n)}{g(n)} = 1$

* we use o-notation as the set

$o(g(n)) = \{ f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0 \}$

$0 \le f(n) < cg(n)$ → Strict upper bound

$(2n) = o(n^2)$

* The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \le f(n) \le c(g(n))$ holds for some constants $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \le f(n) < cg(n)$ holds for all constants $c > 0$.

* In o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

$f(n) = 2n + 6$

$g(n) = n^2$

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$ ✗

(0)

$f(n) \leq c\, g(n)$

$2n + 6 \leq c\, n^2$

$\frac{2n+6}{n^2} \leq c$

$\rightarrow f(n) < c \cdot g(n)$

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

$\frac{2}{n} + \frac{6}{n^2} \leq c$

$0 \leq c$

$f(n) = o(g(n))$

Definition:

$f(n) = o(g(n))$ if for every positive constant c, there exists an n_0 such that for all $n \geq n_0$, $0 \leq f(n) < c \cdot g(n)$

Example:

Consider $f(n) = 2n$ and $g(n) = n^2$. We claim $f(n) = o(g(n))$.

- To prove this, for any $c > 0$, we need $2n < c \cdot n^2$.
- Dividing both sides by n (assuming $n > 0$ gives
  $2 < c \cdot n$
- Choose $n_0 = \frac{2}{c}$. For $n \geq n_0$, $2 < c \cdot n$ holds. Therefore,
  $\lim_{n\to\infty} \frac{2n}{n^2} = \lim_{n\to\infty} \frac{2}{n} = 0$

This shows $f(n) = o(g(n))$.

Asymptotic Analysis: ω – notation

$c\,g(n) < f(n)$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

ω-notation:

By analogy, ω-notation is to Ω notation as o-notation is to O-notation. We use ω-notation to denote a lower bound that is not asymptotically ~~tight~~.

$f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$.

$f(n) = 3n^2 + 5$

$g(n) \in n$

We define $\omega(g(n))$ ("little-omega of g of n") as the set

$$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le c\,g(n) < f(n) \text{ for all } n \ge n_0\}$$

ω

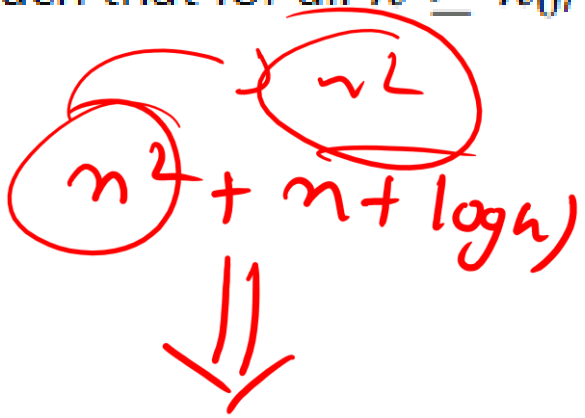
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

The relation $f(n) = \omega(g(n))$ implies that if the limit exists. That is $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity

Little Omega Notation (ω)

Definition:

$f(n) = \omega(g(n))$ if for every positive constant c, there exists an n_0 such that for all $n \geq n_0$, $0 \leq c \cdot g(n) < f(n)$



Example:

Consider $f(n) = n^2$ and $g(n) = n$. We claim $f(n) = \omega(g(n))$.

- To prove this, for any $c > 0$, we need $c \cdot n < n^2$.
- Dividing both sides by n (assuming $n > 0$ gives $c < n$
- Choose $n_0 = c$. For $n \geq n_0$, $c < n$ holds. Therefore, $\lim_{n\to\infty} \frac{n^2}{n} = \lim_{n\to\infty} n = \infty$

This shows $f(n) = \omega(g(n))$.

$c_1, c_2, n_0, c \rightarrow$ positive constant

1) Theta (Θ) $\Rightarrow$ $0 \le c_1 g(n) \le f(n) \le c_2 g(n) \Rightarrow f(n) = \Theta(g(n))$

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$

2) Big oh (O) $\Rightarrow$ $0 \le f(n) \le c g(n) \Rightarrow f(n) = O(g(n))$

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$

3) Big-Omega (Ω) $\Rightarrow$ $0 \le c g(n) \le f(n) \Rightarrow f(n) = \Omega(g(n))$ $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$

4) Little o-oh (o) $\Rightarrow$ $0 \le f(n) < c g(n) \Rightarrow f(n) = o(g(n))$ $\lim_{n \to \infty} f(n)/g(n) = 0$

5) Little omega (ω) $\Rightarrow$ $0 \le c g(n) < f(n) \Rightarrow f(n) = \omega(g(n))$ $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$

Asymptotic Analysis: *numerical example*

Big O notation: Consider the function f(n) = n^3 + 2n^2 + 5. The big O notation for f(n) is O(n^3) because the highest-order term in the polynomial is n^3. As n gets larger, the contribution of the other terms becomes relatively insignificant compared to the dominant n^3 term.

Big Omega notation: Consider the function f(n) = n^2 + 10n. The big Omega notation for f(n) is Ω(n^2) because the lowest order term in the polynomial is n^2. As n gets larger, the contribution of the other term (10n) becomes relatively insignificant compared to the dominant n^2 term.

Big Theta notation: Consider the function f(n) = 3n^2 + 4n + 2. The big Theta notation for f(n) is Θ(n^2) because the highest and lowest order terms in the polynomial are n^2. As n gets larger, the contribution of the other term (4n + 2) becomes relatively insignificant compared to the dominant n^2 term.

Asymptotic Analysis: *numerical example*

4. Little o notation: Consider the function f(n) = n^2 + 2log(n). The little O notation for f(n) is o(log n).

$$\lim_{n\to\infty}\frac{\log(n)}{n}=\lim_{n\to\infty}\frac{\frac{d}{dn}(\log(n))}{\frac{d}{dn}(n)}=\lim_{n\to\infty}\frac{1/n}{1}=\lim_{n\to\infty}\frac{1}{n}=$$

5. Little Omega notation:
Consider the function f(n) = n^2 + nlog(n).
The little Omega notation for f(n) is ω(nlogn)

Since n^2 is the dominant term in $f(n)$, a natural choice for $g(n)$ is $n\log(n)$.

1. **Compute the Ratio:**

$$\frac{f(n)}{g(n)}=\frac{n^2+n\log(n)}{n\log(n)}$$

Simplify the expression:

$$=\frac{n^2}{n\log(n)}+\frac{n\log(n)}{n\log(n)}=\frac{n}{\log(n)}+1$$

2. **Take the Limit:**

$$\lim_{n\to\infty}\left(\frac{n}{\log(n)}+1\right)=\lim_{n\to\infty}\frac{n}{\log(n)}+\lim_{n\to\infty}1$$

- As $n\to\infty$, the term $\frac{n}{\log(n)}\to\infty$ because n grows faster than $\log(n)$.
- The constant 1 does not affect the limit.

3. **Conclude:**

Since the limit is infinity, we conclude that:

$$n^2+n\log(n)=\omega(n\log(n))$$

Properties of Asymptotic Notations

Asymptotic notation, which includes ***Big O ("O"), Big Omega ("Ω"), Big Theta ("θ"), Little o ("o"), and Little Omega ("ω"),*** have several properties that make them useful in analyzing and comparing the computational complexity of algorithms and functions.

1.Reflexivity: Any function is asymptotically equivalent to itself. This means that $f(n) = \Theta(f(n))$, $f(n) = O(f(n))$, and $f(n) = \Omega(f(n))$ are always true.

2.Transitivity: If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$. Similarly, if $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, then $f(n) = \Omega(h(n))$. This property allows us to compare the asymptotic behavior of different functions.

3.Symmetry: If $f(n) = \Theta(g(n))$, then $g(n) = \Theta(f(n))$. This property allows us to interchange the roles of f(n) and g(n) when analyzing the complexity of algorithms.

Properties of Asymptotic Notations

4. Addition: If f(n) = Θ(h(n)) and g(n) = Θ(k(n)), then f(n) + g(n) = Θ(max(h(n), k(n))). This property allows us to analyze the complexity of an algorithm that involves multiple functions.

5. Multiplication: If f(n) = O(h(n)) and g(n) = O(k(n)), then f(n)g(n) = O(h(n)k(n)). This property allows us to analyze the complexity of an algorithm that involves multiple functions.

6. Transpose: If f(n) is a non-negative monotonic increasing function, then $f(n)=\Theta(g(n))$, then $f^{-1}(n)=\Theta(g^{-1}(n))$ where f^-1 is the inverse function of f. This property allows us to analyze the complexity of the inverse of a function.

These properties of asymptotic notation allow us to analyze the complexity of algorithms and functions and compare their efficiency in terms of time and space complexity.

Questions on Asymptotic Notations

1. What is the Big O notation for the function ***f(n) = 2n^2 + 3n + 1?***

2. What is the Big Omega notation for the function ***g(n) = 4n + 2log(n)?***

3. If ***f(n) = 3n^2 + 2nlog(n) and g(n) = 5nlog(n)***, which function has a higher order of growth as n approaches infinity?

4. If ***f(n) = 2^n and g(n) = n!***, which function has a higher order of growth as n approaches infinity?

5. What is the Big Theta notation for the function ***h(n) = n^3 + 5n^2 + 3n + 1?***

Questions and Answers on Asymptotic Notations

1. What is the Big O notation for the function f(n) = 2n^2 + 3n + 1? **Answer:** The Big O notation for f(n) is O(n^2) because the highest order term in the polynomial is n^2.

1. What is the Big Omega notation for the function g(n) = 4n + 2log(n)? **Answer:** The Big Omega notation for g(n) is Ω(n) because the lowest order term in the polynomial is n.

1. If f(n) = 3n^2 + 2nlog(n) and g(n) = 5nlog(n), which function has a higher order of growth as n approaches infinity? **Answer:** To compare the growth rate of the two functions, we need to compare their dominant terms. The dominant term of f(n) is n^2 and the dominant term of g(n) is nlog(n). Therefore, as n approaches infinity, f(n) has a higher order of growth than g(n).

Questions and Answers on Asymptotic Notations

4. If f(n) = 2^n and g(n) = n!, which function has a higher order of growth as n approaches infinity? **Answer:** To compare the growth rate of the two functions, we need to compare their dominant terms. The dominant term of f(n) is 2^n and the dominant term of g(n) is n^n. Therefore, as n approaches infinity, g(n) has a higher order of growth than f(n).

5. What is the Big Theta notation for the function h(n) = n^3 + 5n^2 + 3n + 1? **Answer:** The Big Theta notation for h(n) is Θ(n^3) because both the highest and lowest order terms in the polynomial are n^3.

Summary

Asymptotic notation is a significant idea in computer science and algorithm analysis because it gives a mechanism to evaluate and compare the efficiency of algorithms in a way that is independent of the exact hardware or software platform on which they are implemented. The following considerations clarify the significance of the asymptotic notation:

- Platform independence
- Simplicity and abstraction
- Algorithm selection
- Predicting scalability

Summary

Let us describe the mechanism of using asymptotic notation:

- Analyzing time complexity
- Understanding space complexity
- Comparing algorithms
- Identifying dominant terms

Summary

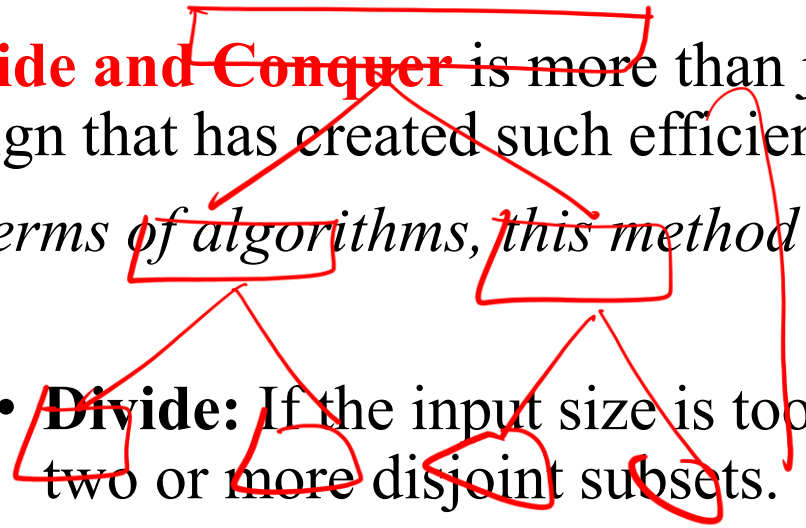
There are several commonly used asymptotic notations, including:

- Big O notation (O)
- Omega notation (Ω)
- Theta notation (Θ)
- Little o notation (o): This indicates a non-asymptotically tight upper bound. It describes functions that grow strictly faster than a given function.
- Little omega notation (): This denotes a non-asymptotically tight lower bound. It describes functions that grow strictly slower than a given function.

Divide-and-Conquer

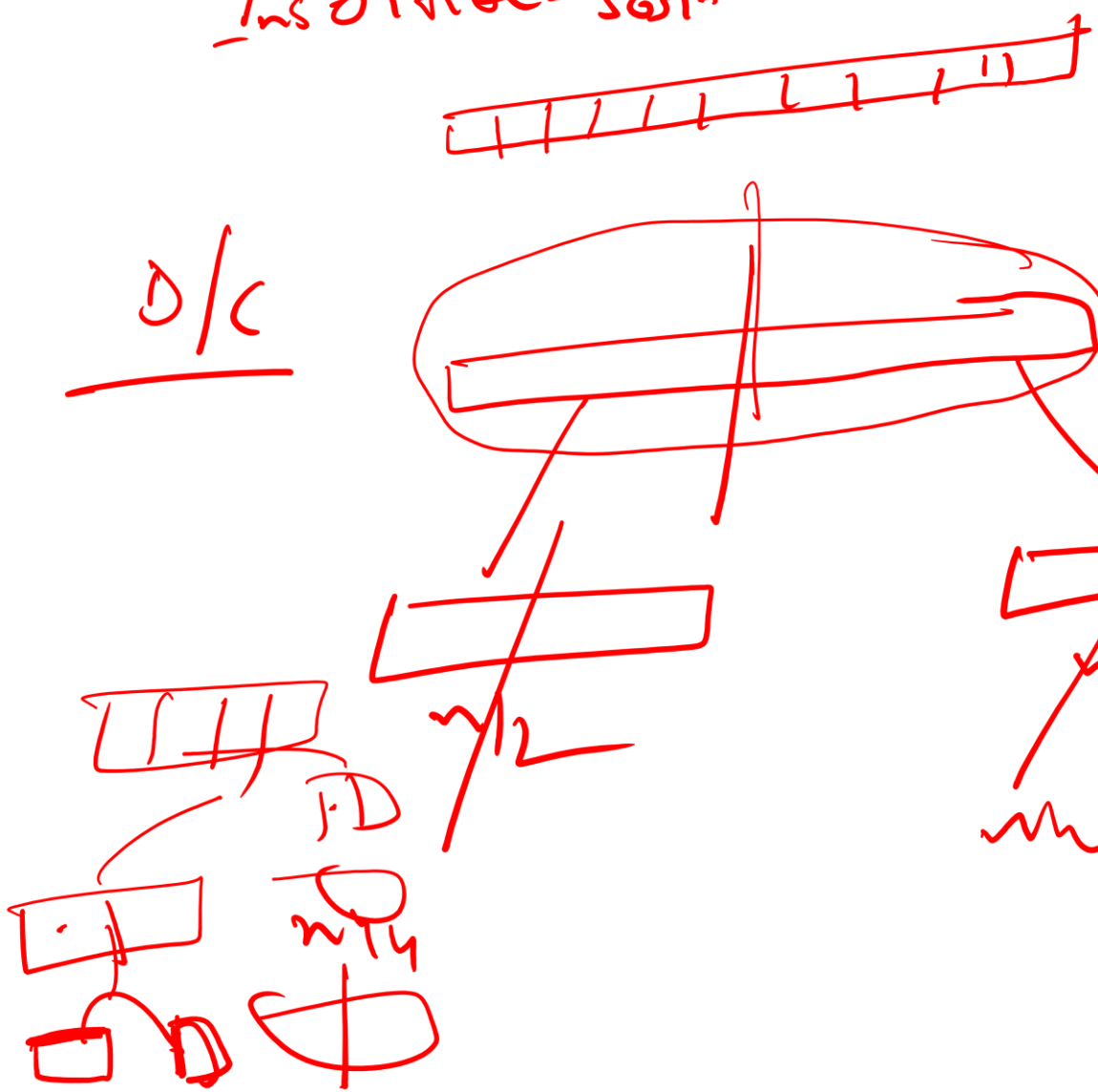
Divide and Conquer is more than just a military strategy; it is also a method of algorithm design that has created such efficient algorithms as Merge Sort.

In terms of algorithms, this method has three distinct steps:



- **Divide:** If the input size is too large to deal with straightforwardly, divide the data into two or more disjoint subsets.
- **Conquer:** Use divide and conquer to solve the sub-problems associated with the data subsets.
- **Combine:** Take the solutions to the sub-problems and "merge" these solutions into a solution for the original problem.

Insertion-sort



D/C

Divide

Conquer

Combine

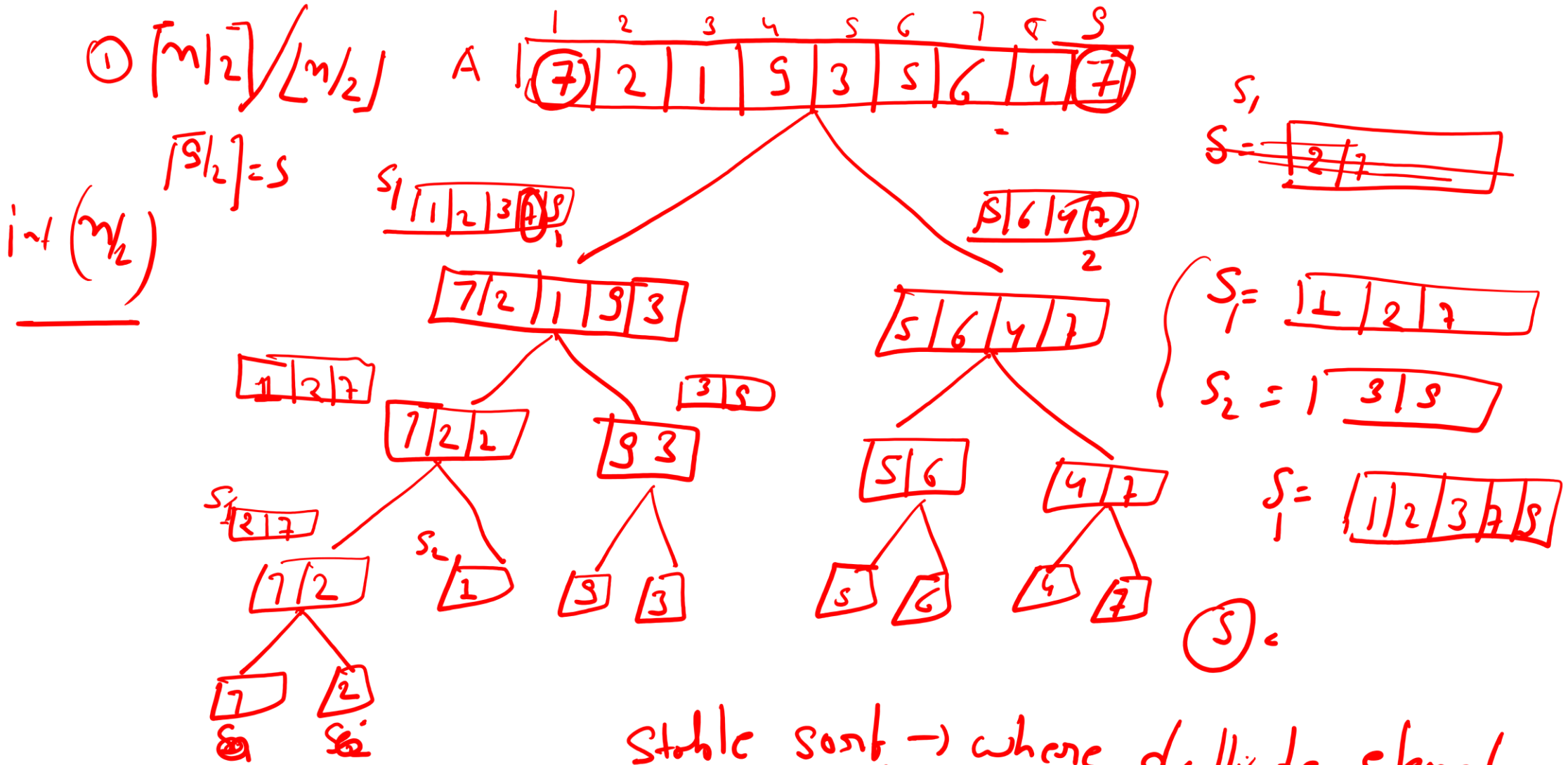
① Divide the problem

② Conquer

③ Combine

Merge sort

- Merge Sort is a **divide-and-conquer algorithm** (divide and conquer will be detailed later) that efficiently sorts an array or a list by recursively dividing it into smaller sub-problems, sorting those sub-problems, and then merging the results. Merge Sort has a time complexity of **O(n log n) in the worst, average, and best cases, (we will see this)** making it a stable and consistent sorting algorithm. The steps of the Merge sort algorithm are defined as follows:
  - **Divide:** Divide the unsorted list into **n** sublists, each containing one element. This is the base case of the recursion.
  - **Conquer:** Recursively sort each sub-list.
  - **Combine** (Merge)**:** Merge the sorted sublists back together to produce a single sorted list.
- The algorithm starts by **dividing** the unsorted list into **n** sublists, each with one element. This is the **base case**, and the recursion stops when each sub-list contains one element. The algorithm recursively sorts each sub-list.
- The recursion continues until all sublists are sorted. The **divide-and-conquer** strategy ensures that sorting smaller sublists is simpler and more efficient.
- The final step involves **merging** the sorted sublists to create a single sorted list.
- This is achieved by repeatedly comparing the smallest unprocessed element from each sub-list and selecting the smaller. The selected element is then added to the merged list.
- This process continues until all elements from the sublists are merged into the final sorted list.


① ⌈n/2⌉ / ⌊n/2⌋
A
1 2 3 4 5 6 7 8 9
7 2 1 5 3 5 6 4 7
⌈5/2⌉ = 3
int (n/2)
S1
S2
S1 = 1 2 7
S2 = 1 3 5
S = 1 2 3 4 5
stable sort → where duplicate element
maintain the position.

Merge sort

The advantages of the Merge sort are as follows:

1. **Stable Sorting:** Merge Sort is a stable sorting algorithm, meaning that it maintains the relative order of equal elements in the sorted output.
2. **Predictable Performance:** Its time complexity is consistently O(n log n) in the worst, average, and best cases.
3. **Adaptability:** Merge Sort is well-suited for linked lists and ***external sorting*** where random access is expensive.

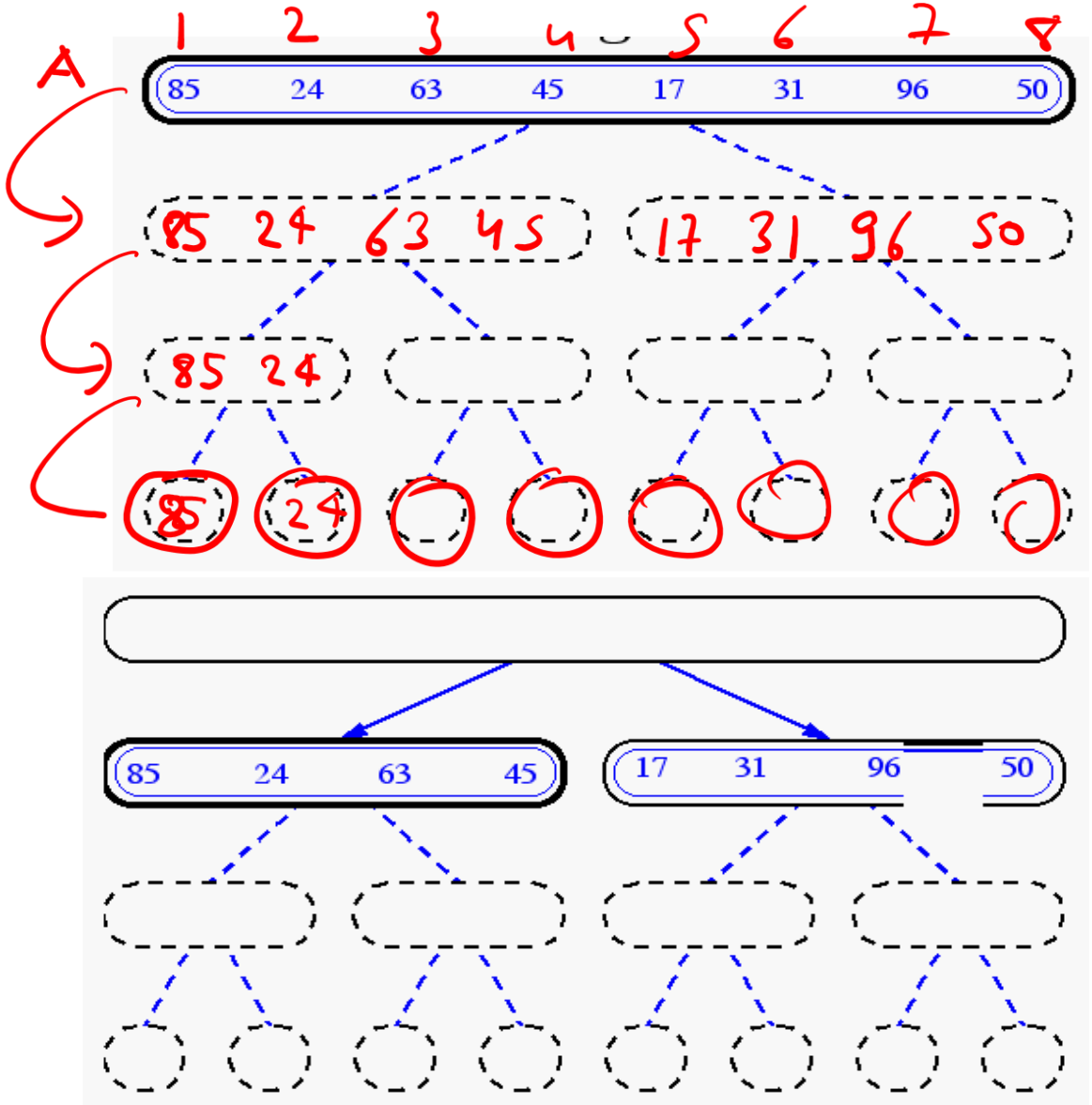
The primary drawbacks of their merge sort are as follows:

1. **Space Complexity:** Merge Sort typically requires additional space for merging, making it less memory-efficient compared to in-place algorithms.
2. **Not In-Place:** *It is not an in-place sorting algorithm, which means it may not be suitable for scenarios with limited memory*.

Merge-Sort

- Algorithm:
  - **Divide**: If **S** has at least two elements (nothing needs to be done if S has zero or one element), remove all the elements from S and put them into two sequences, S_1 and S_2, each containing about half of the elements of S. (i.e. S_1 contains the **first** $\lceil n/2 \rceil$ elements and S_2 contains the remaining $\lfloor n/2 \rfloor$ elements.
  - **Conquer** Recursive sort sequences S_1 and S_2.
  - **Combine:** Put back the elements into S by merging the sorted sequences S_1 and S_2 into a unique sorted sequence.

Merge-Sort

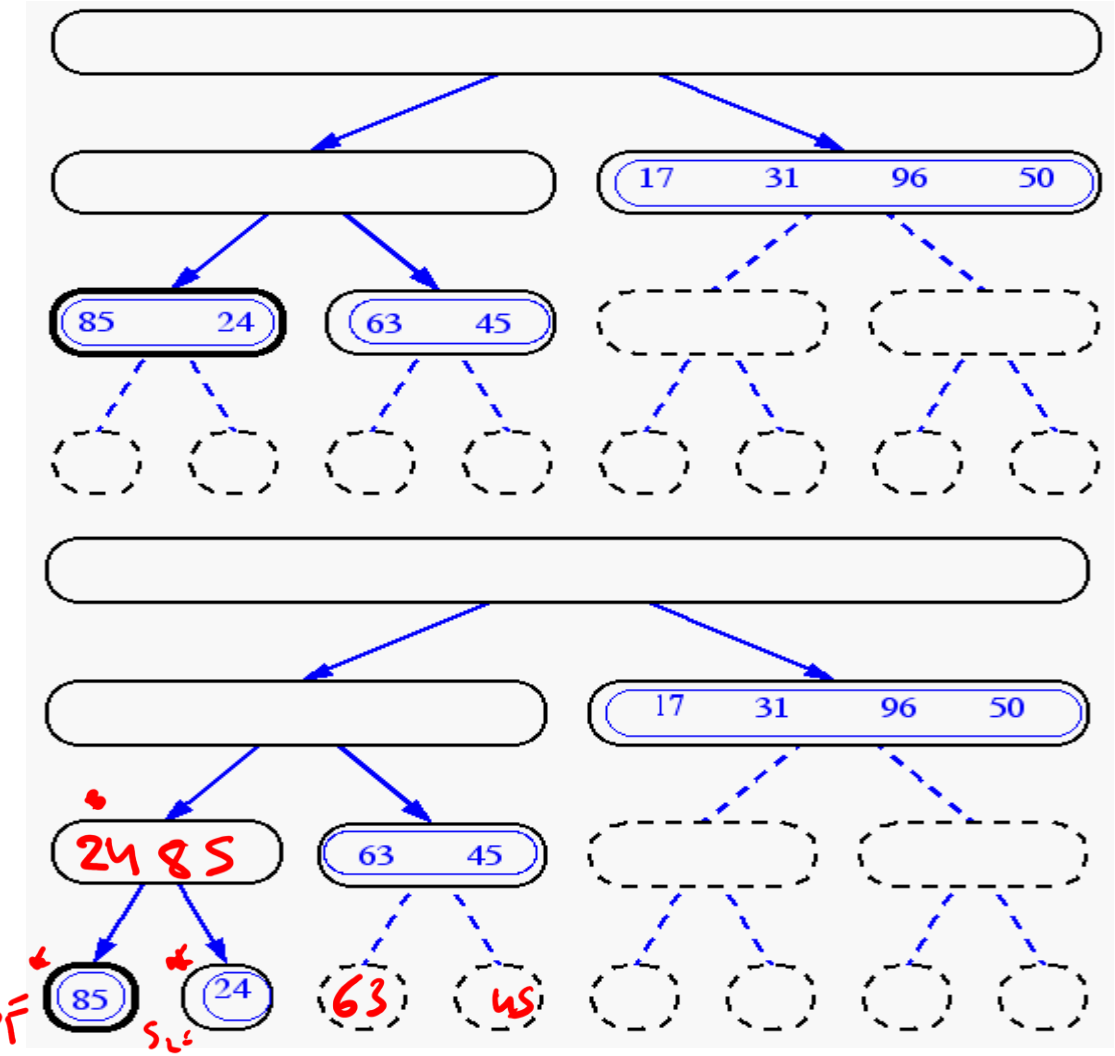


$\lceil n/2 \rceil = 8 = [4]$

$\lceil 4/2 \rceil = 2$

$\lceil 2/2 \rceil = 1$

Merge-Sort(cont.)



$S_1 = [85]$

$S_2 = [24]$

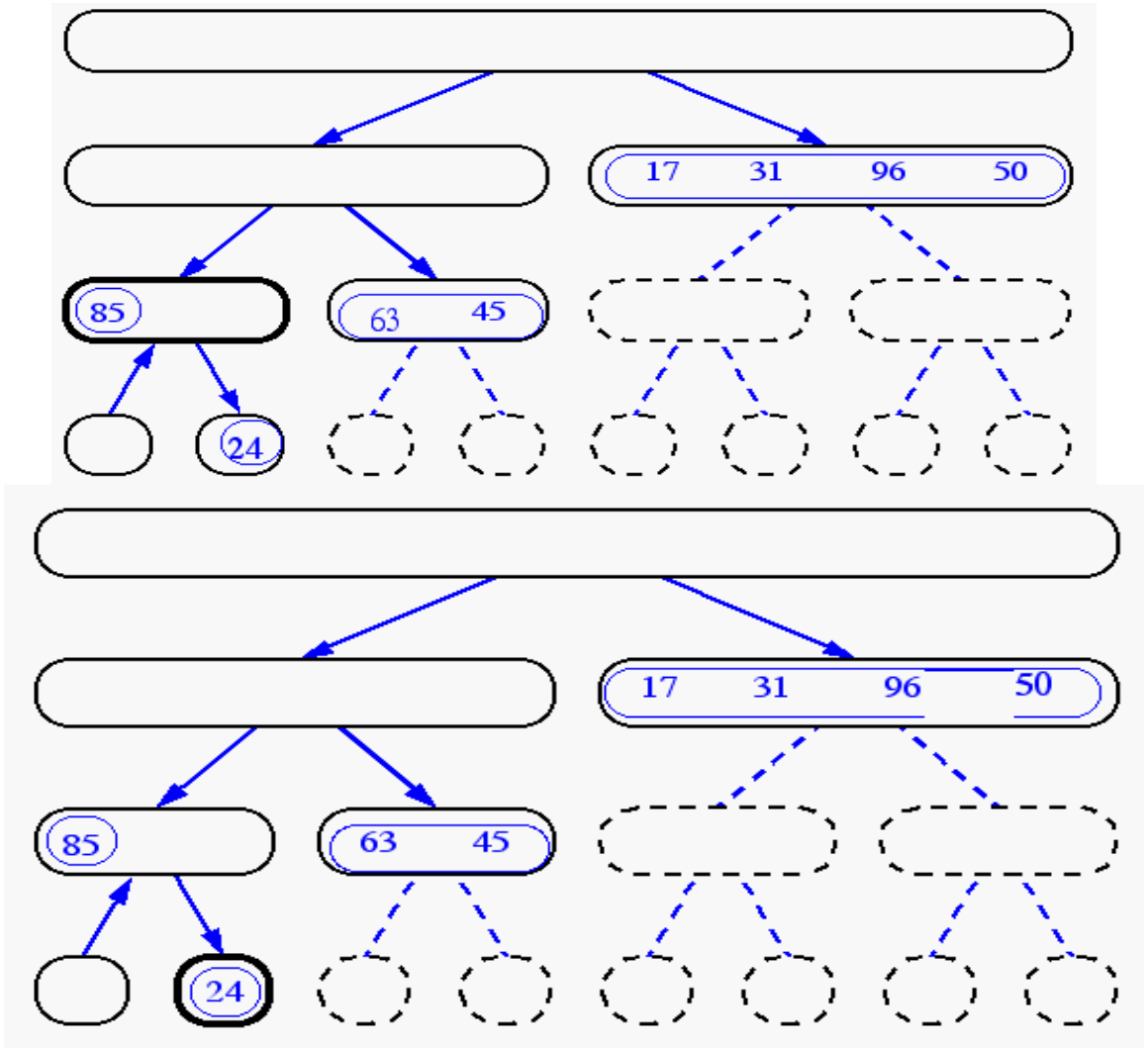
$S = [24\ 85]$

$S_1 = [63]$

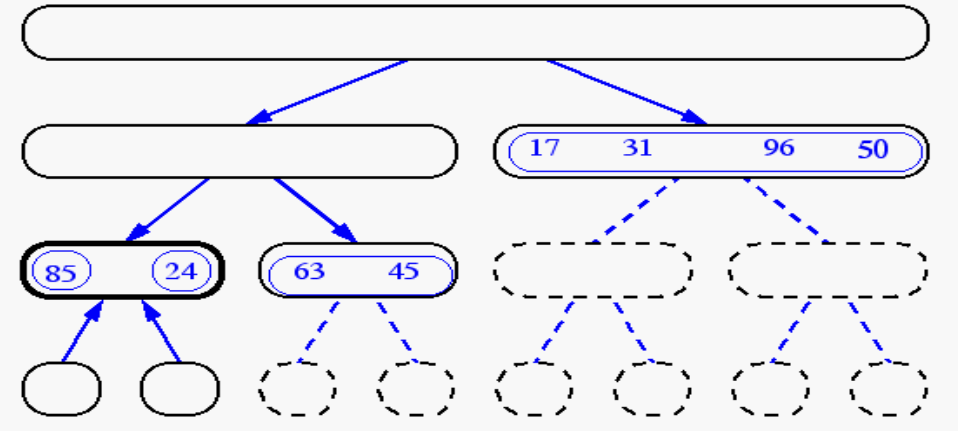
$S_2 = [45]$

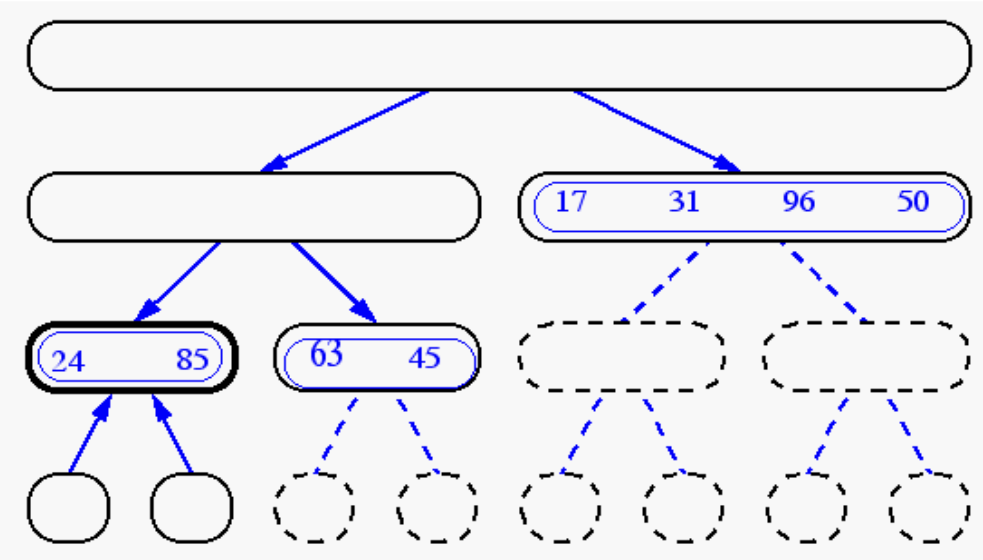
$S = [45, 63]$

Merge-Sort (cont.)

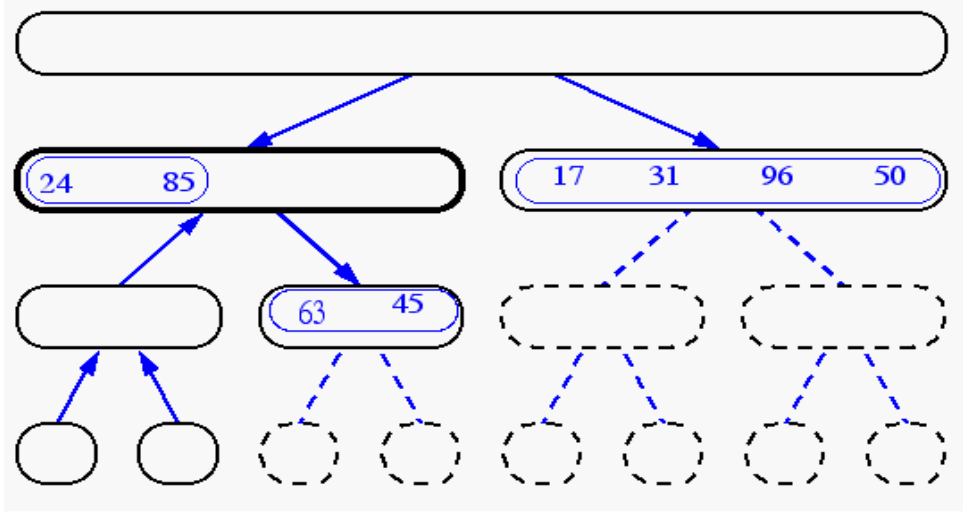


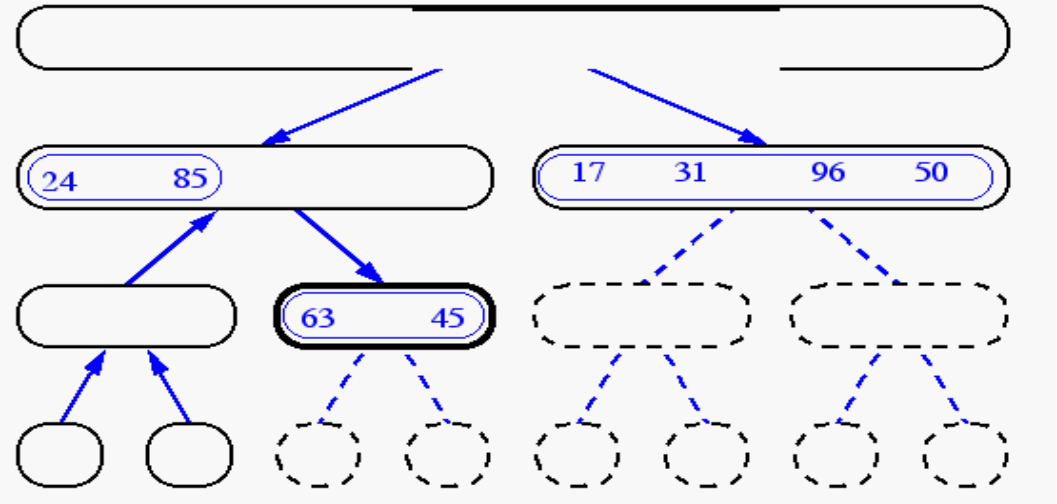
Merge-Sort (cont.)



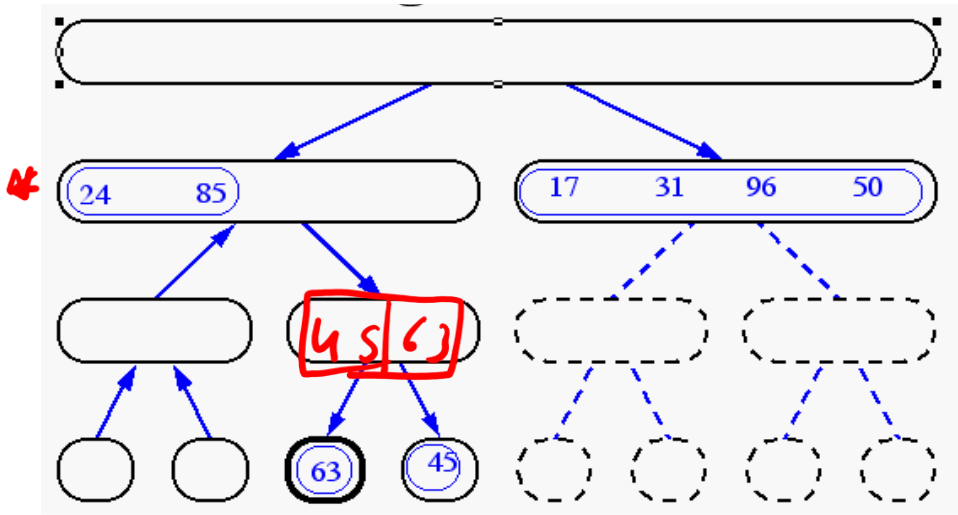


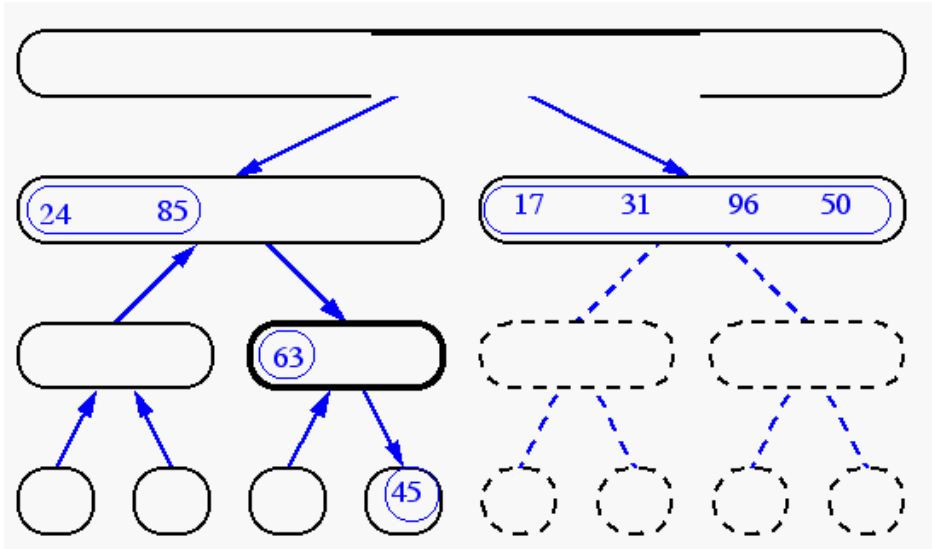
Merge-Sort (cont.)



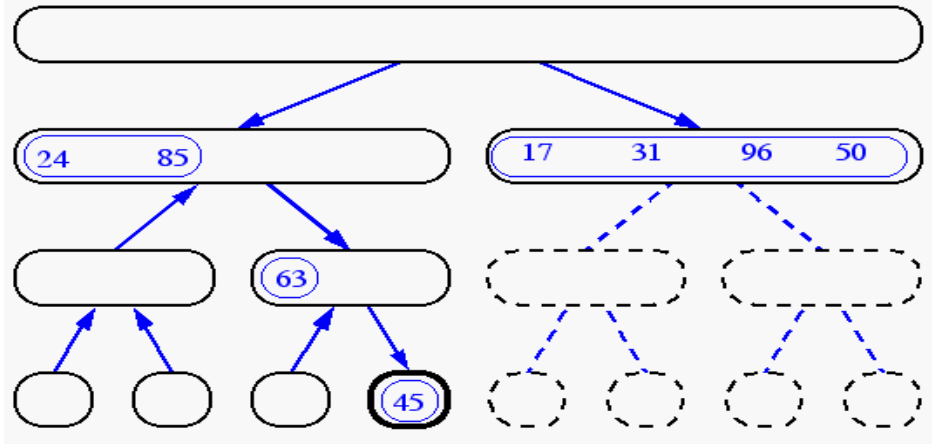


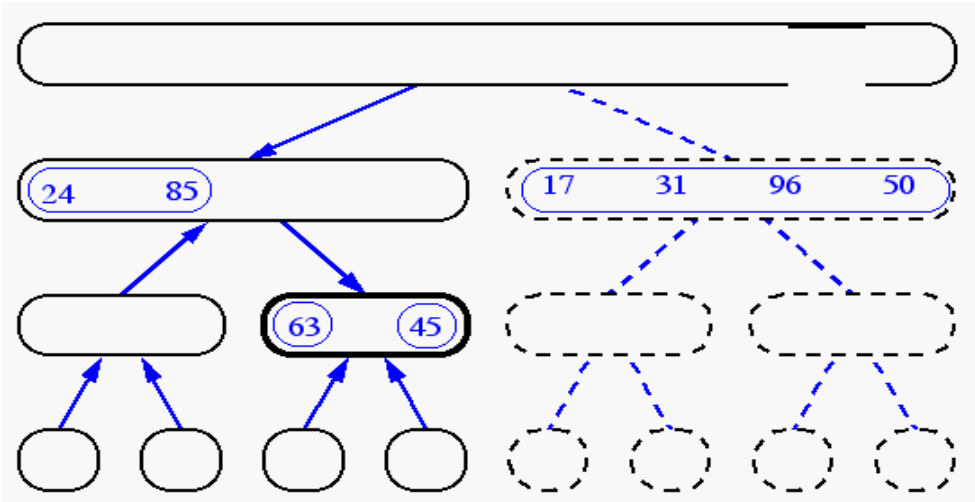
Merge-Sort (cont.)



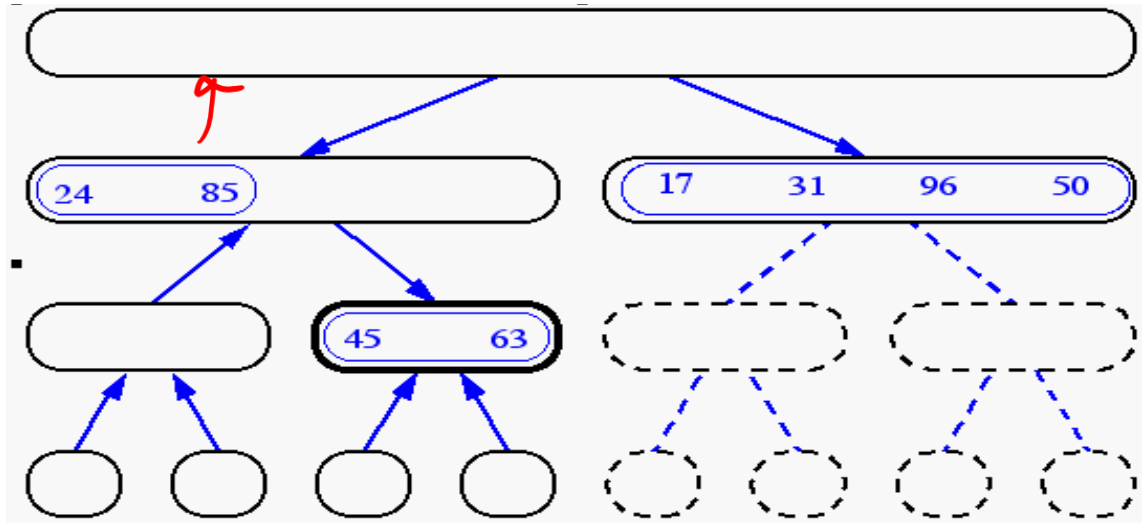


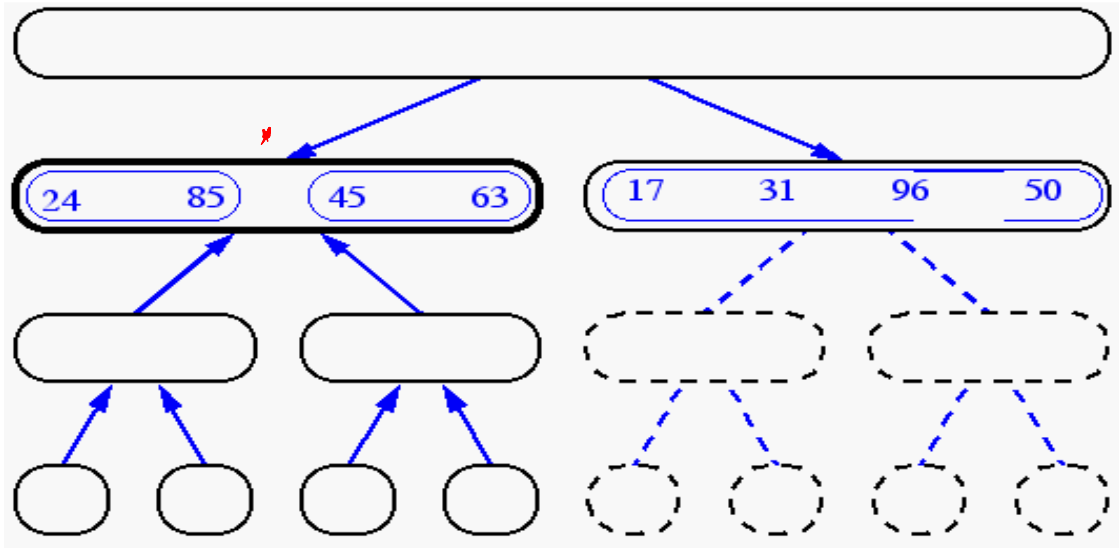
Merge-Sort (cont.)



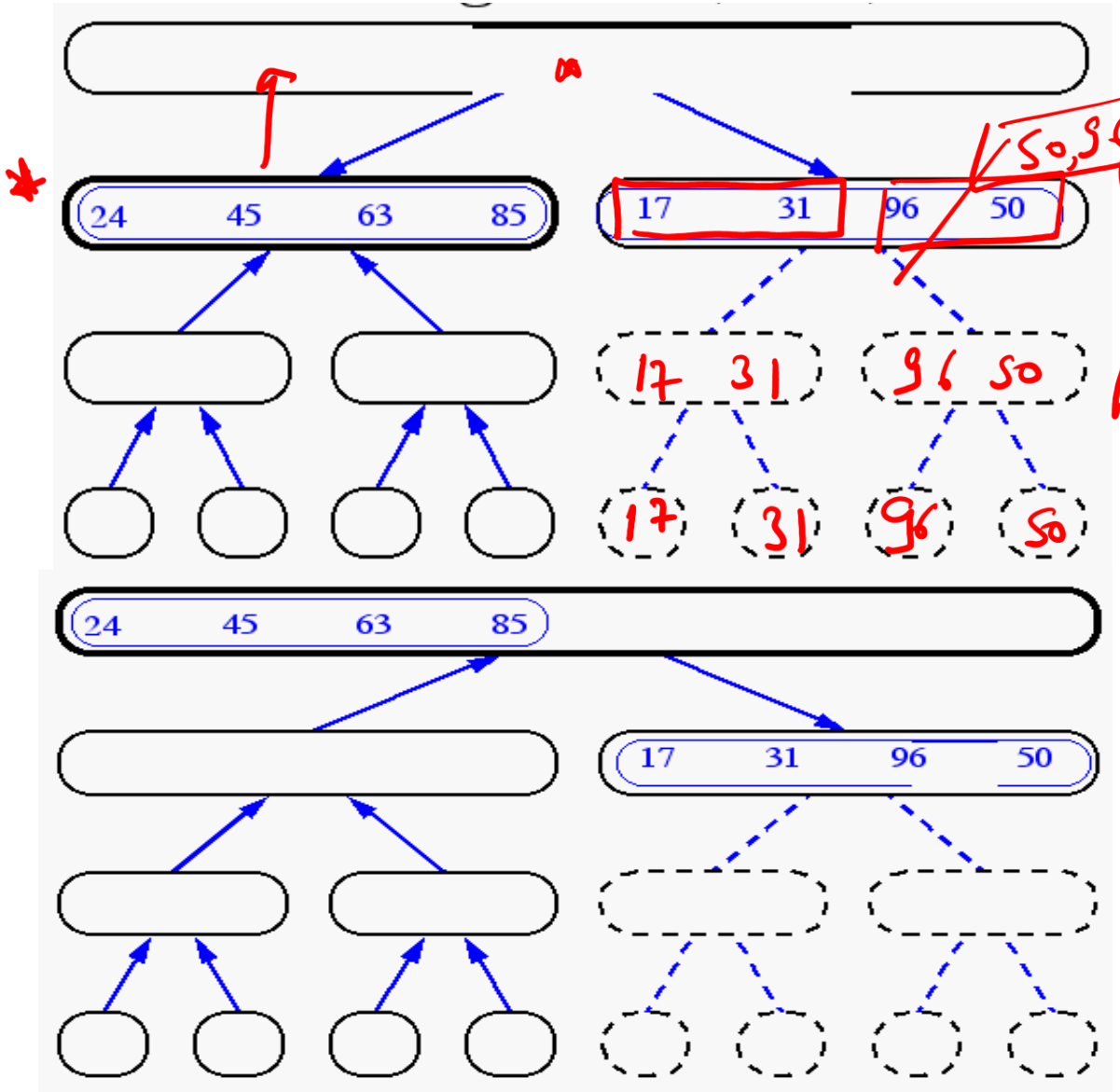


Merge-Sort(cont.)

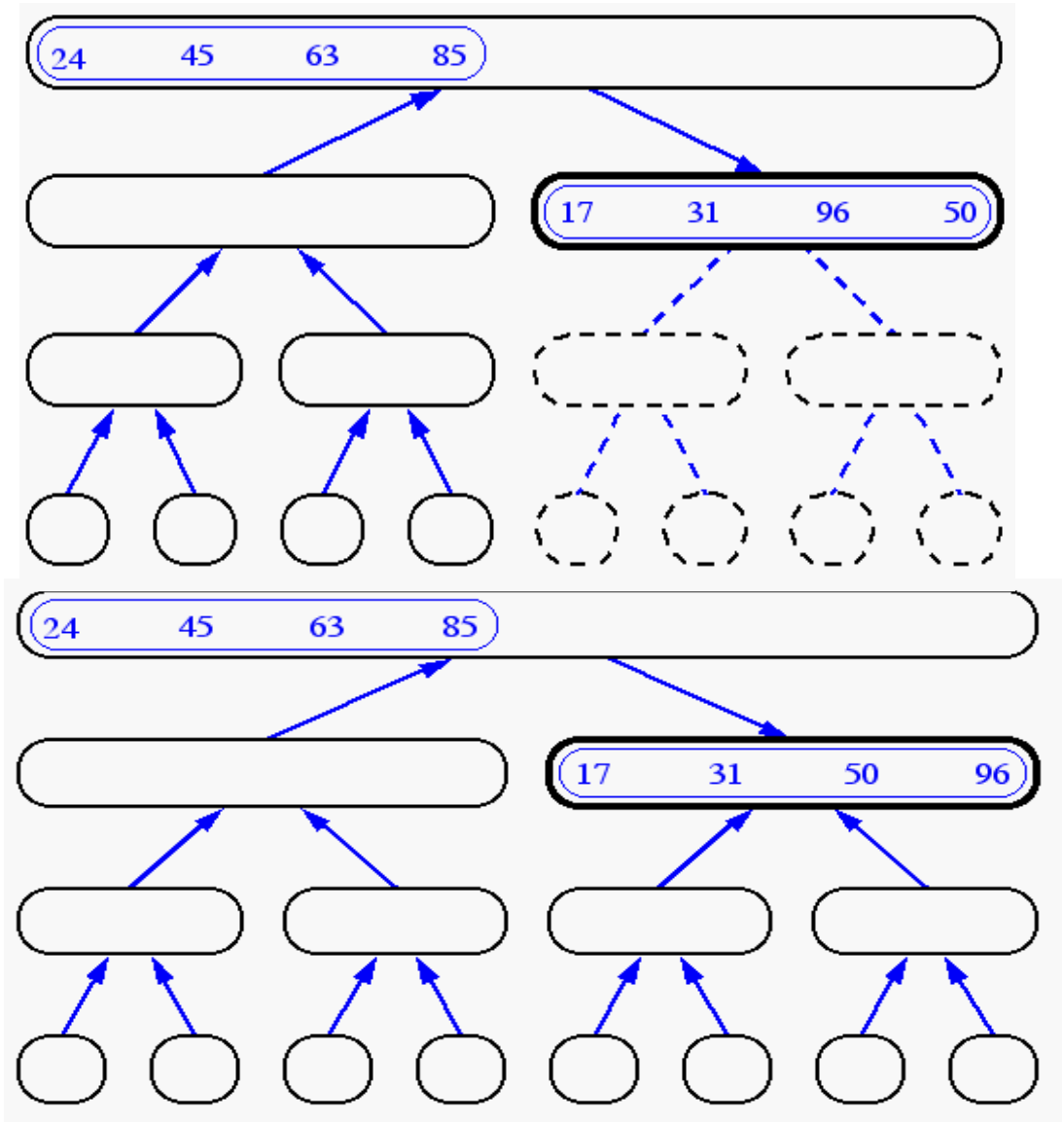




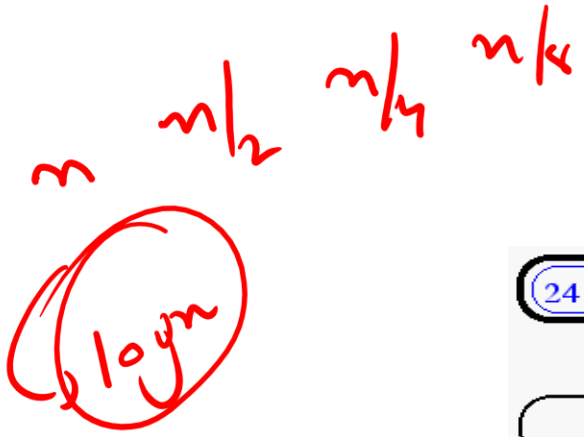
Merge-Sort (cont.)



Merge-Sort (cont.)

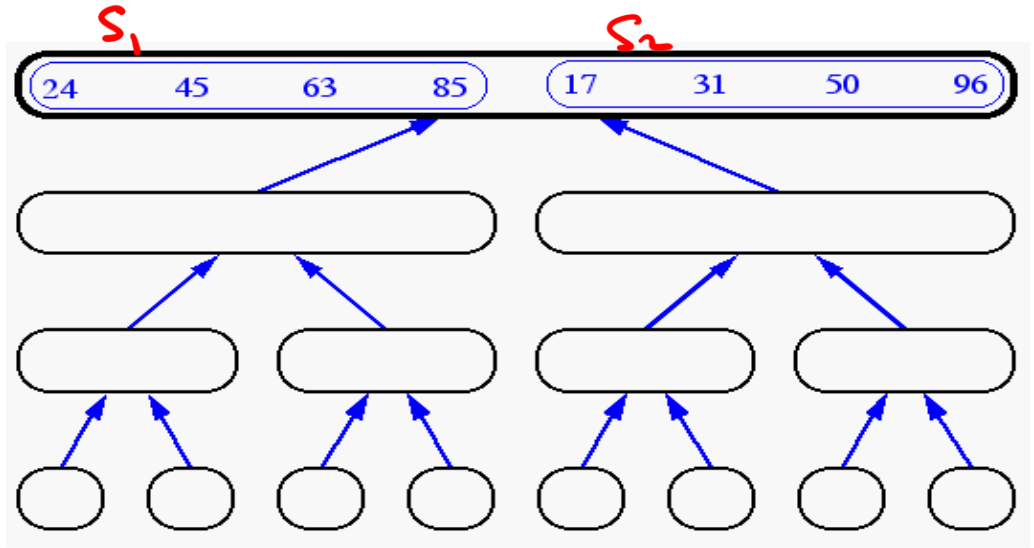


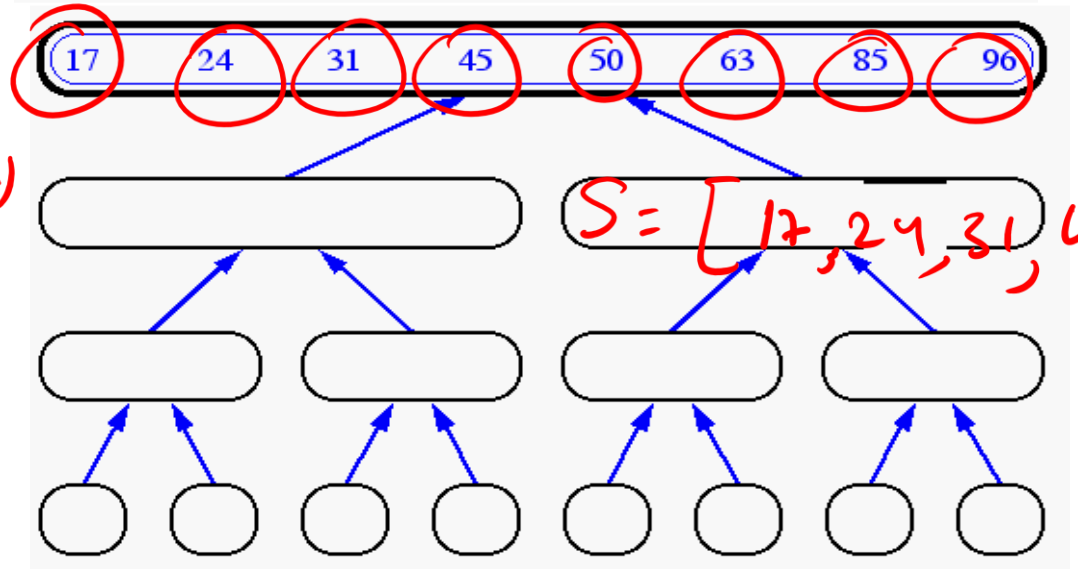
Merge-Sort (cont.)

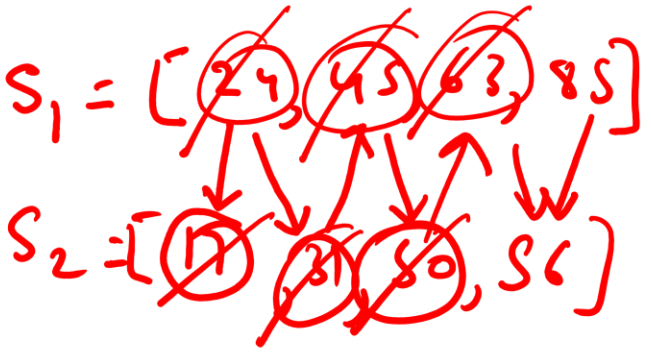


$T(n) = 2T(n/2) + O(n)$

Recurrence form







One-dimensional.

Contd.

Merging Two Sequences

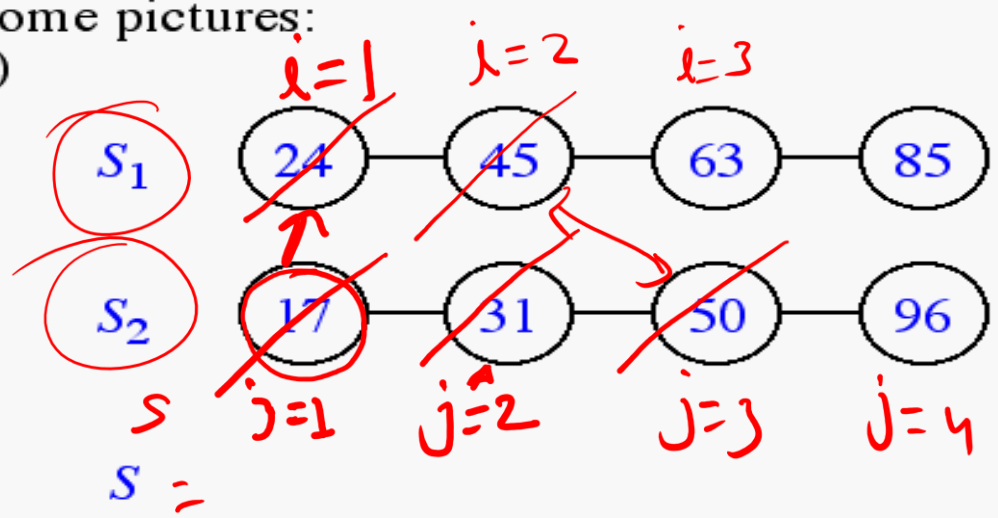
Pseudo-code for merging two sorted sequences into a unique sorted sequence

```
Algorithm merge (S1, S2, S):
    Input: Sequence S1 and S2 (on whose elements a
    total order relation is defined) sorted in nondecreas
    ing order, and an empty sequence S.
    Ouput: Sequence S containing the union of the ele
    ments from S1 and S2 sorted in nondecreasing order;
    sequence S1 and S2 become empty at the end of the
    execution
    while S1 is not empty and S2 is not empty do
      if S1.first().element() ≤ S2.first().element() then
         {move the first element of S1 at the end of S}
         S.insertLast(S1.remove(S1.first()))
      else
         { move the first element of S2 at the end of S}
         S.insertLast(S2.remove(S2.first()))
    while S1 is not empty do
      S.insertLast(S1.remove(S1.first()))
      {move the remaining elements of S2 to S}
    while S2 is not empty do
      S.insertLast(S2.remove(S2.first()))
```

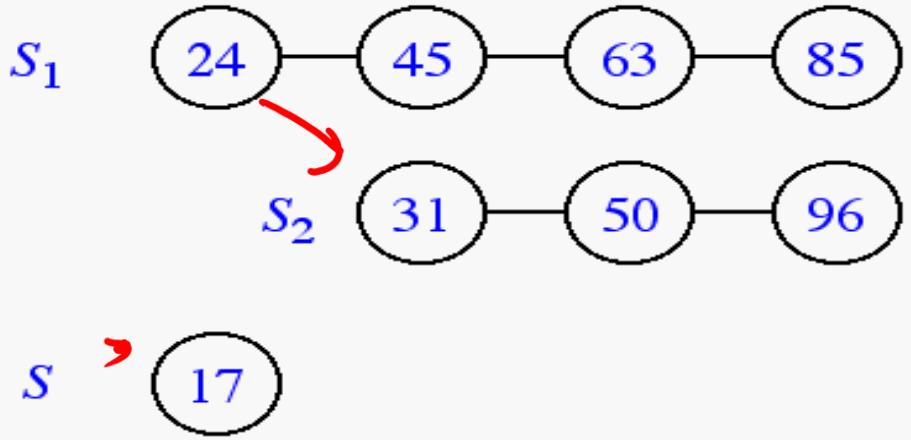
Merging Two Sequences (cont.)

- Some pictures:

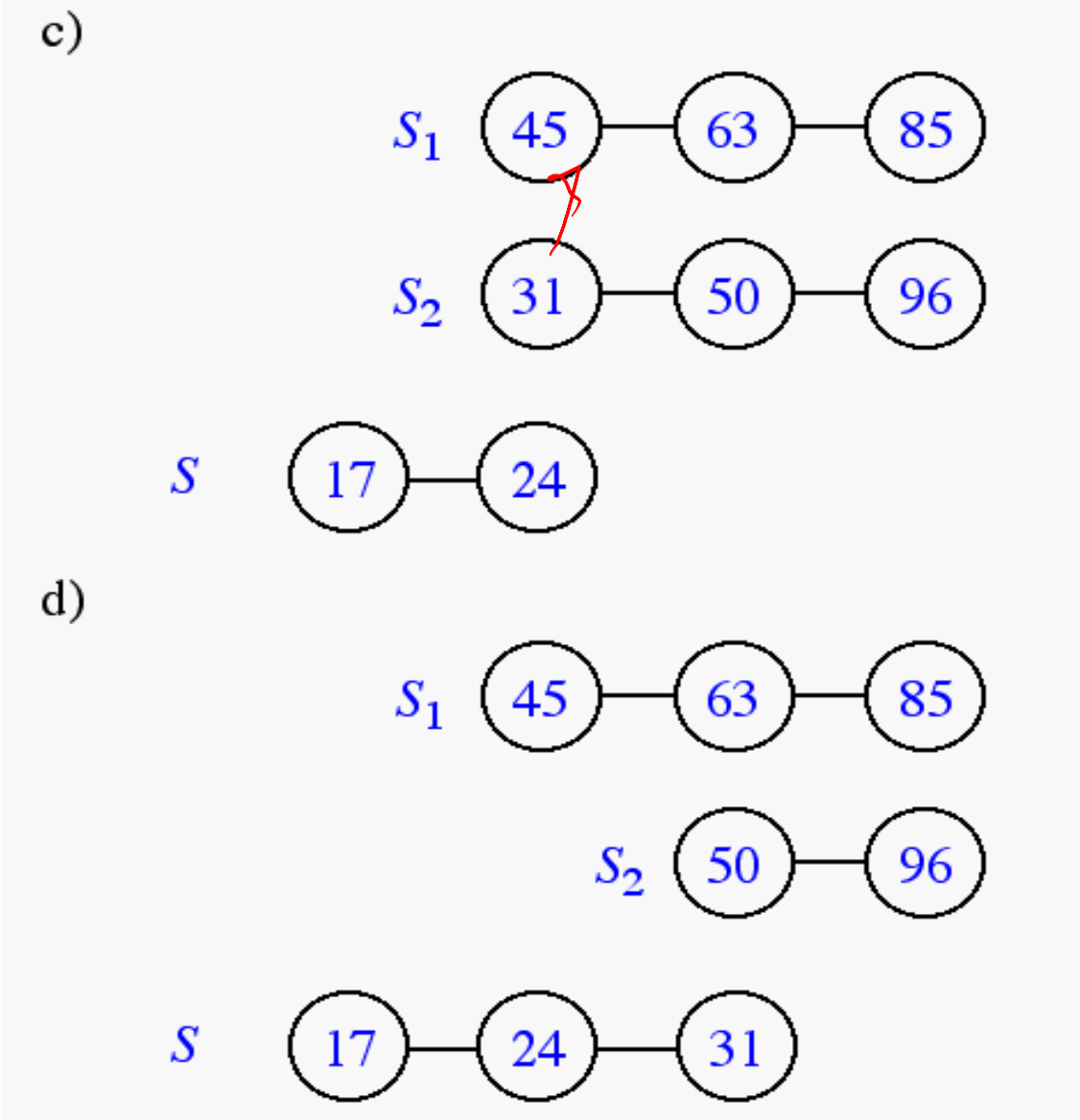
a)



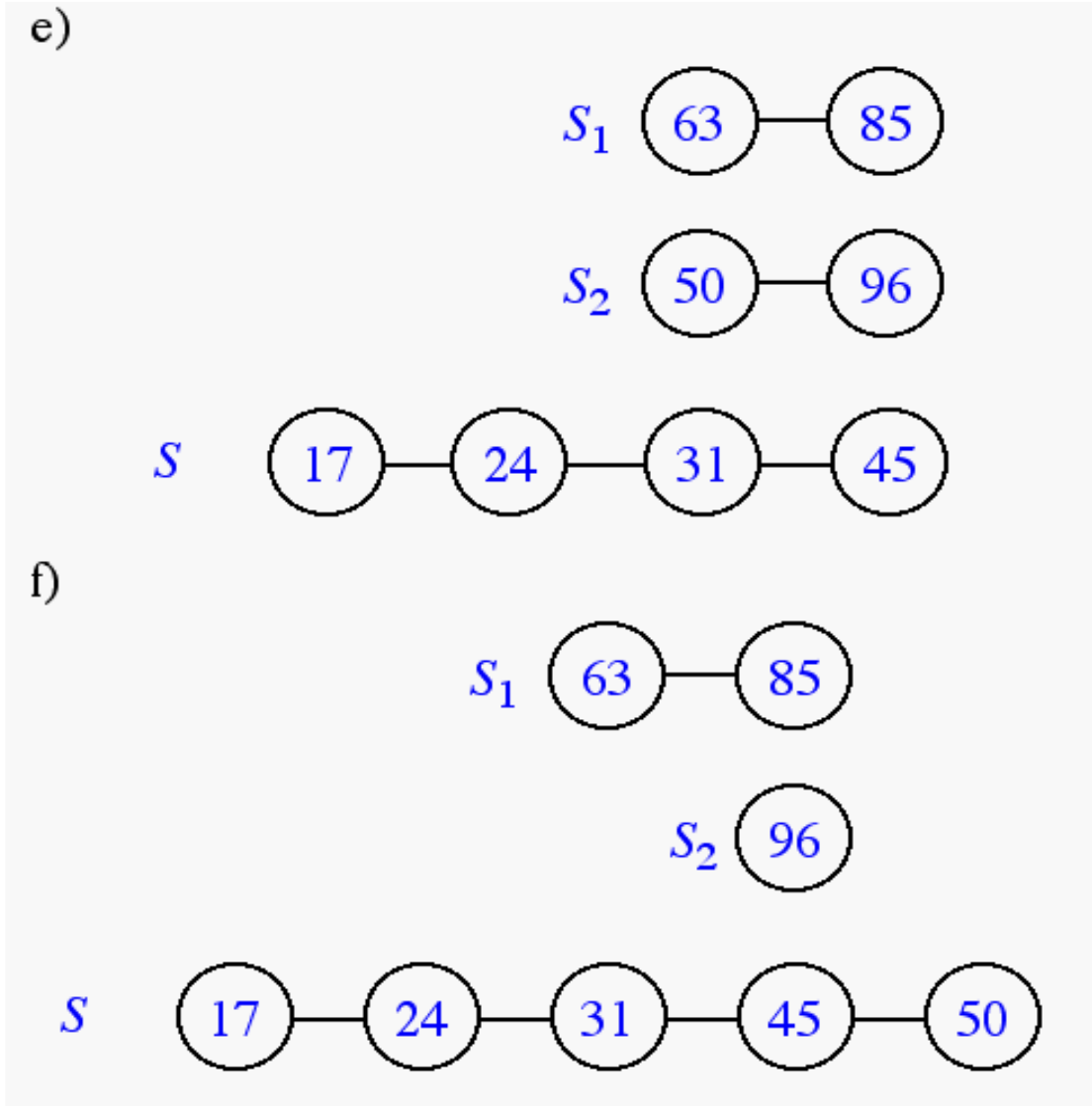
b)



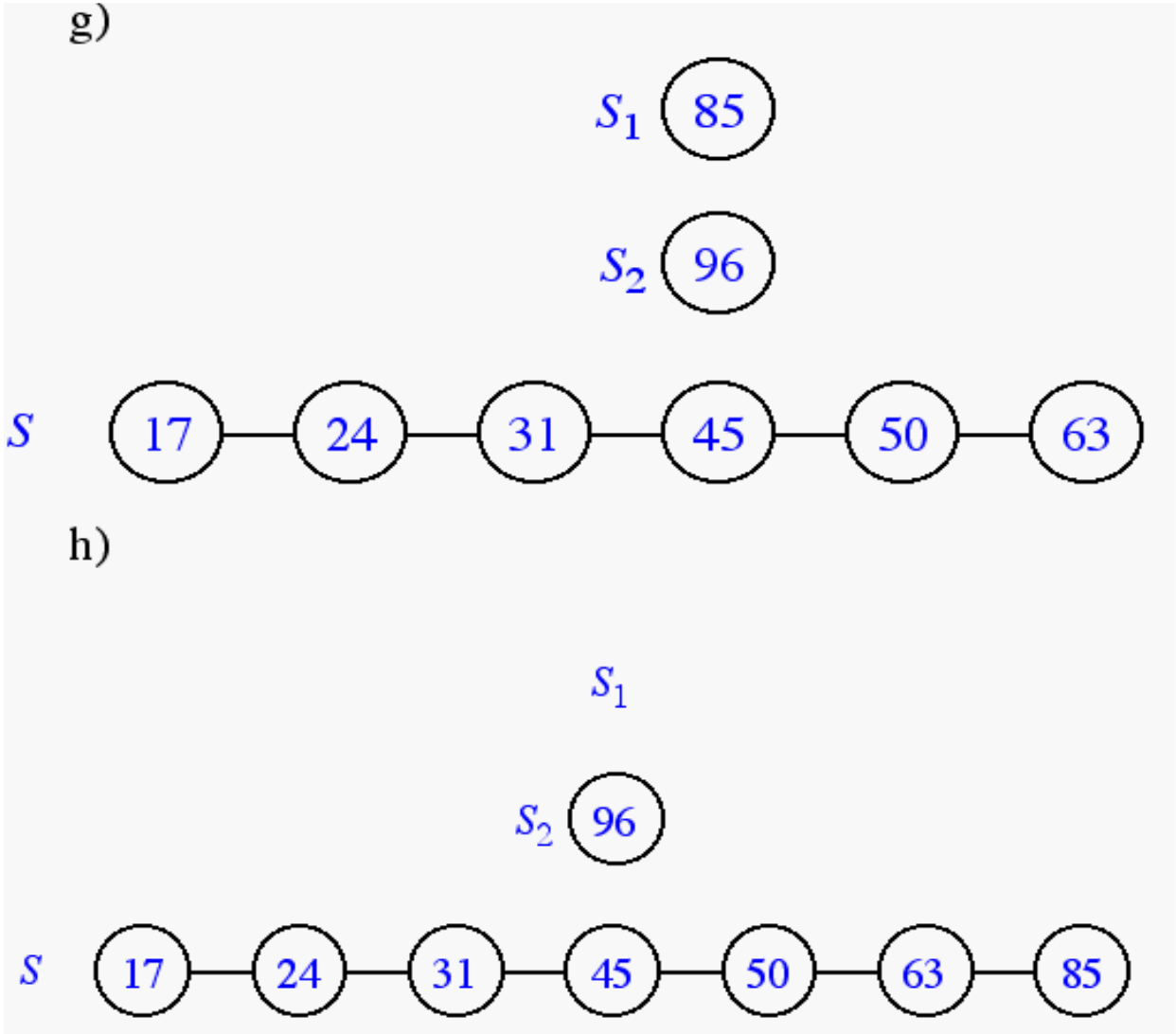
Merging Two Sequences (cont.)



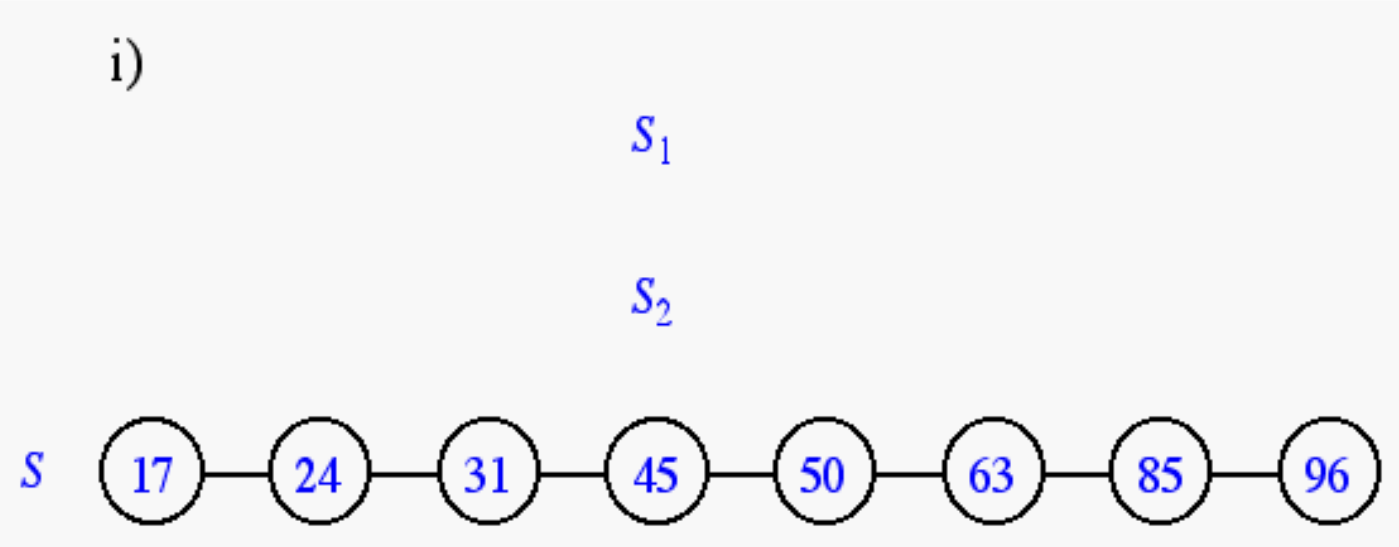
Merging Two Sequences (cont.)



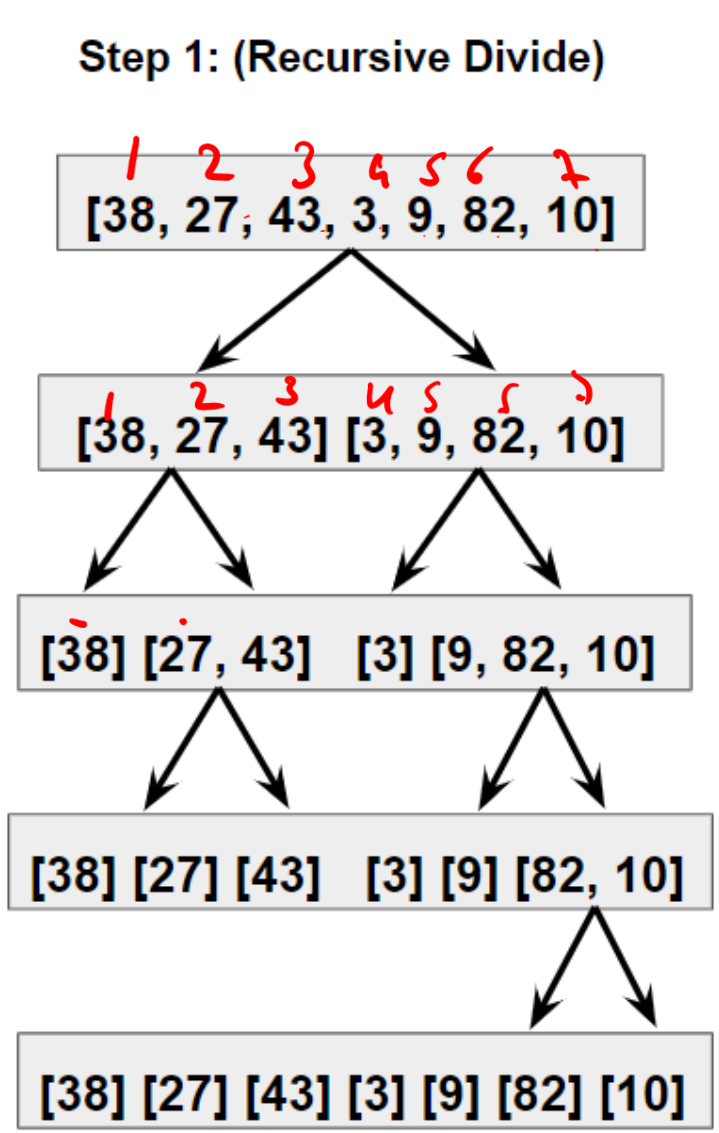
Merging Two Sequences (cont.)



Merging Two Sequences (cont.)



Merge sort



Step 2: (Conquer)

[27] [38] [3] [43] [9] [10] [82]

Step 3: Combine (Merge)

[27, 38] [3, 43] [9] [10, 82]

Final Sorted Array

[3, 9, 10, 27, 38, 43, 82]

Merge sort

```
#Procedure for MergeSort
MergeSort(arr):
    if length(arr) <= 1:
        return arr
    middle = length(arr) / 2
    left_half = MergeSort(arr[:middle])
    right_half = MergeSort(arr[middle:])
    return Merge(left_half, right_half)
```

```
#Procedure for Merge
Merge(left, right):
    result = []
    left_index = right_index = 0
    while left_index < length(left) and right_index < length(right):
        if left[left_index] < right[right_index]:
            result.append(left[left_index])
            left_index += 1
        else:
            result.append(right[right_index])
            right_index += 1
    result.extend(left[left_index:])
result.extend(right[right_index:])
return result
```

Asymptotic analysis of Merge Sort

It involves understanding its time complexity, which is consistently O(n log n) in the worst, average, and best cases. Let's break down the analysis step by step.

Time Complexity Analysis:

- **Divide:** Dividing the array of size n takes $O(1)$ time.
- **Conquer:** The recursive calls on subproblems occur until each sublist contains only one element, resulting in $O(\log n)$ levels of recursion.
- **Combine (Merge):** Merging two sorted sublists of size $n/2$ takes $O(n)$ time.

Overall Time Complexity of Merge Sort can be expressed using following **recurrence relation**

$$T(n) = 2T(n/2) + O(n).$$