Chap.2 Getting Started

- Here we study two sorting algorithms
- **Insertion sort** sorts by inserting into a sorted list the elements of the input array one after the other
- **Merge sort** sorts by recursively the input array into halves, sorting the halves separately, and then merging them into a full sorted list

The problem of sorting

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

Example:

Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Example of insertion sort

8 2 4 9 3 6

Example of insertion sort

8 2 4 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6

Example of insertion sort

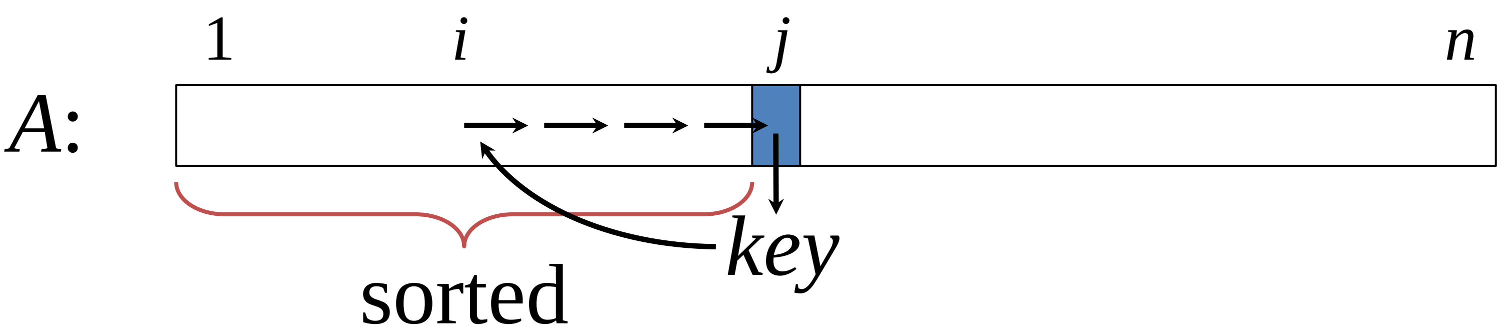
8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6

Example of insertion sort

8	2	4	9	3	6	
2	8	4	9	3	6	
2	4	8	9	3	6	
2	4	8	9	3	6	
2	3	4	8	9	6	
2	3	4	6	8	9	*Done!*

Example of insertion sort

8	2	4	9	3	6	
2	8	4	9	3	6	
2	4	8	9	3	6	
2	4	8	9	3	6	
2	3	4	8	9	6	
2	3	4	6	8	9	*Done!*



Insertion Sort

```
INSERTION-SORT(A)
for j = 2 to length[A]
    do key ← A[j]
        //insert A[j] to sorted sequence A[1..j-1]
        i ← j-1
        while i >0 and A[i]>key
            do A[i+1] ← A[i]  //move A[i] one position right
               i ← i-1
        A[i+1] ← key
```

Running time

- The running time depends on the input: an already sorted sequence is easier to sort.

- <u>Major Simplifying Convention</u>: Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.

  $T_A(n) =$ time of A on length n inputs
- Generally, we seek upper bounds on the running time, to have a guarantee of performance.

Kinds of analysis

Worst-case:

$T(n)$ = maximum time of algorithm on any input of size n.

Average-case:

$T(n)$ = expected time of algorithm over all inputs of size n.

Need assumption of statistical distribution of inputs.

Best-case:

works fast on *some* input.

Analysis of Algorithm

- When we analyze an algorithm, we are often primarily interested in its *worst-case* performance.
- Why?
  - The worst-case is an *upper bound on the running time of an algorithm*. (We know its performance can't be any worse than that.)
  - For some algorithms, the worse case occurs fairly often.

Machine-independent time

What is insertion sort's worst-case time?

Basic Idea:

Ignore machine dependent constants,
otherwise impossible to verify and to compare algorithms
Look at ***growth*** of $T(n)$ as $n \to \infty$.

Asymptotic Analysis

Insertion sort analysis

- Worst case
  - Reverse sorted list
  - $O(n^2)$
- Best Case
  - Sorted input
  - $O(n)$
- Is insertion sort a fast sorting algorithm?
  - Moderately so, for small *n*.
  - Not at all, for large *n*.

Merge Sort

- Insertion sort used an *incremental approach* to sorting: sort the smallest subarray (1 item), add one more item to the subarray, sort it, add one more item, sort it, etc.
- Merge sort uses a *divide-and-conquer approach,* based on the concept of *recursion.*

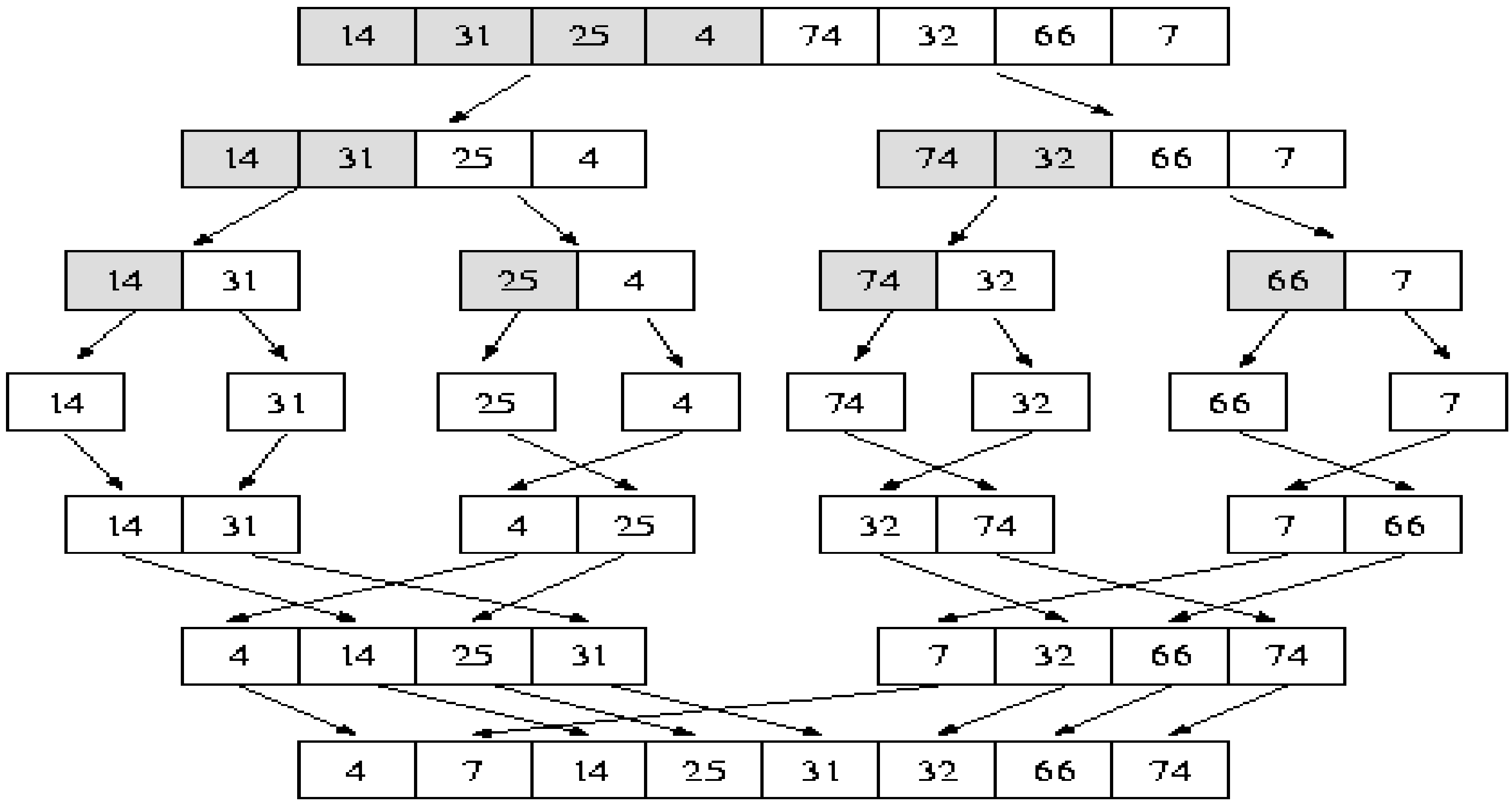
Merge Sort

- *Divide-and-conquer:*
  - *Divide* the problem into several subproblems.
  - *Conquer* the subproblems by solving them recursively. If the subproblems are small enough, solve them directly.
  - *Combine* the solutions to the subproblems to get the solution for the original problem.

Merge Sort (divide & conquer)

- *Divide the n-element sequence to be sorted into* two subsequences of n/2 each.
- *Conquer by sorting the subsequences recursively* by calling merge sort again. If the subsequences are small enough (of length 1), solve them directly.(Arrays of length 1 are already sorted.)
- *Combine the two sorted subsequences by merging* them to get a sorted sequence.

Merge Sort Example



Merge sort

- Merge sort basically consists of recursive calls to itself.
- The base case (which stops the recursion) occurs when a subsequence has a size of 1.
- The combine step is accomplished by a call to an algorithm called Merge.

Merge Sort(A,p,r)

1. if $p < r$
2. then $q \leftarrow \lfloor (p+r)/2 \rfloor$
3. MERGE-SORT(A, p, q)
4. MERGE-SORT(A, q+1, r)
5. MERGE(A, p, q, r)

Call to MERGE-SORT(A,1,n) (suppose n=length(A))

Merge

- Merge works by assuming you have two already sorted sublists and an empty array

1	4	7

2	3	9

Merge

1	4	7	∞		2	3	9	∞

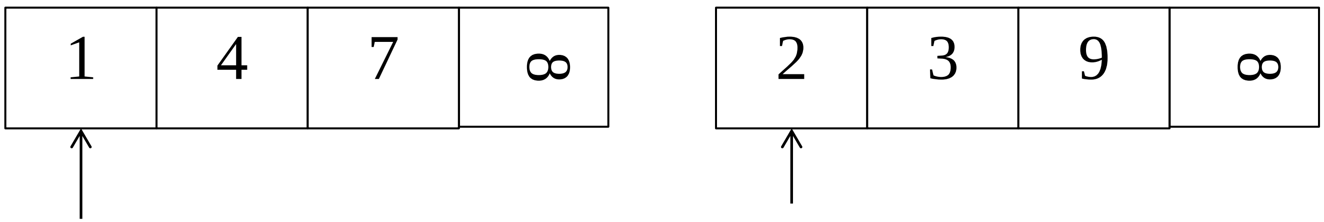
- Let's assume we have an *infinity, which* is guaranteed to be larger than the last item at the end of each sublist which lets us know when we have hit the end of the sublist.

Merge

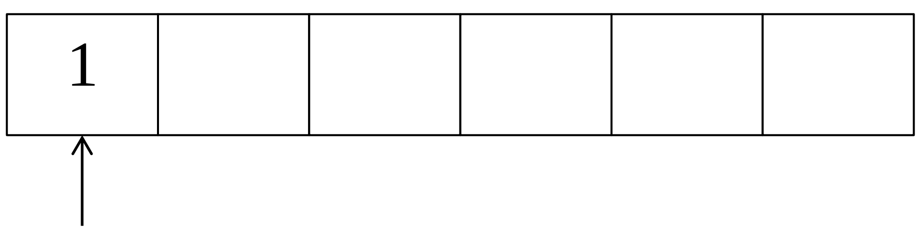
1	4	7	∞	2	3	9	∞
p		q		q+1		r	

- The two sublists are indexed from p to q (for the first sublist) and from q+1 to r for the second sublist. There are (r – p) + 1 items in the two sublists combined, so we will need an output array of that size.

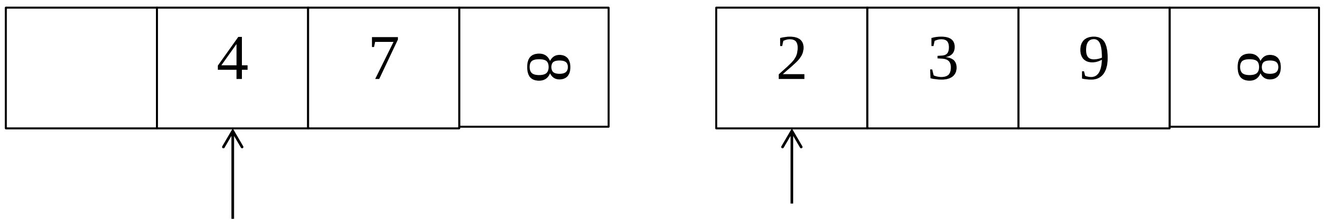
Merge



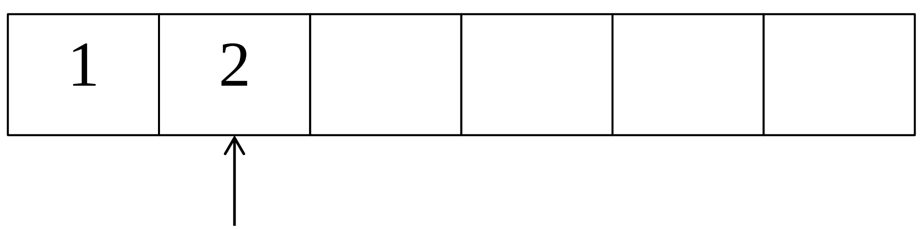
- Look at the first item in each subarray. Choose the smallest item.
- Move the chosen item to the output array



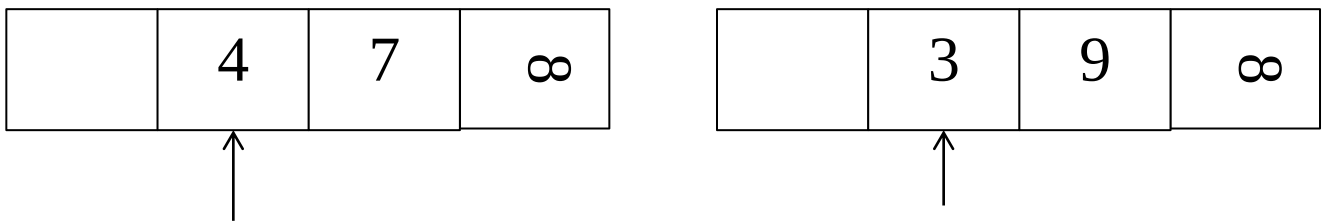
Merge



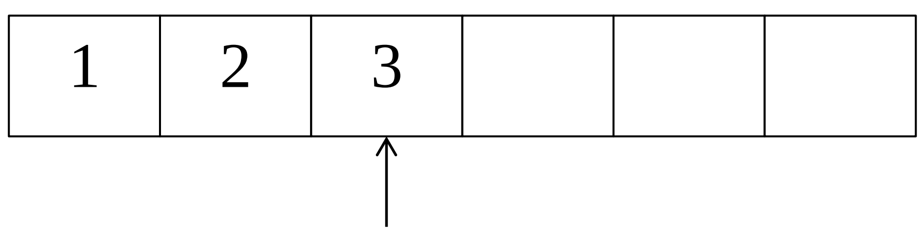
- Look at the first item in each subarray. Choose the smallest item.
- Move the chosen item to the output array



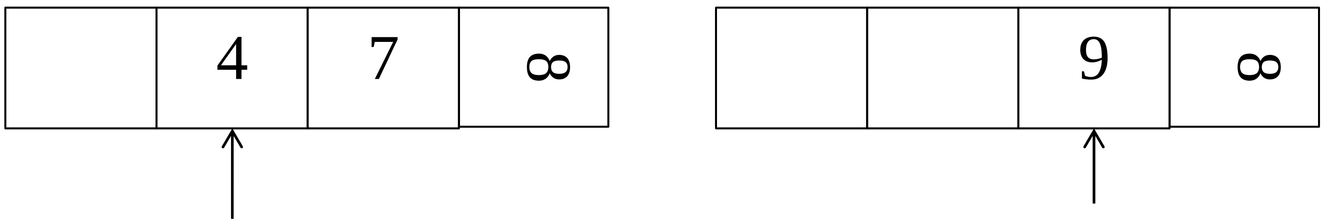
Merge



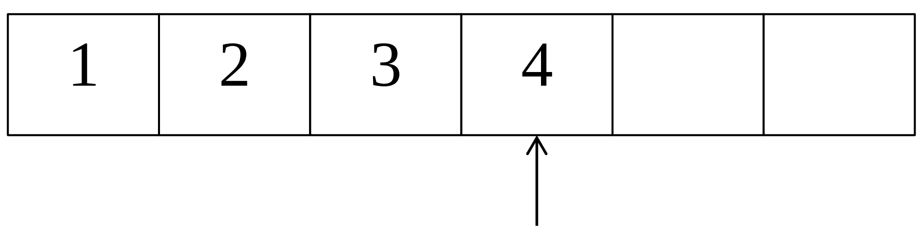
- Look at the first item in each subarray. Choose the smallest item.
- Move the chosen item to the output array



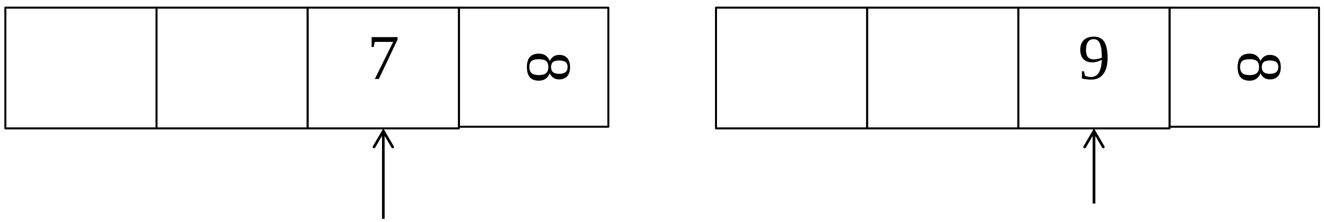
Merge



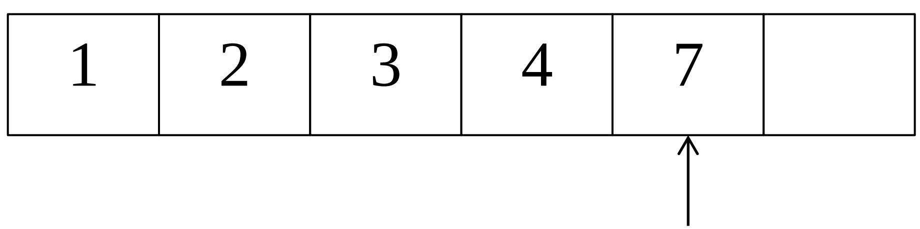
- Look at the first item in each subarray. Choose the smallest item.
- Move the chosen item to the output array



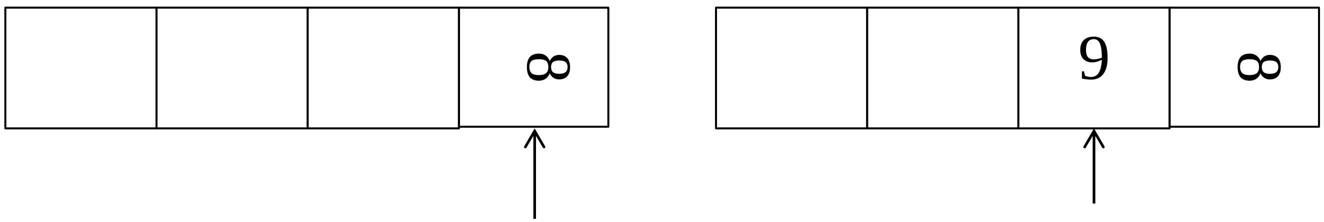
Merge



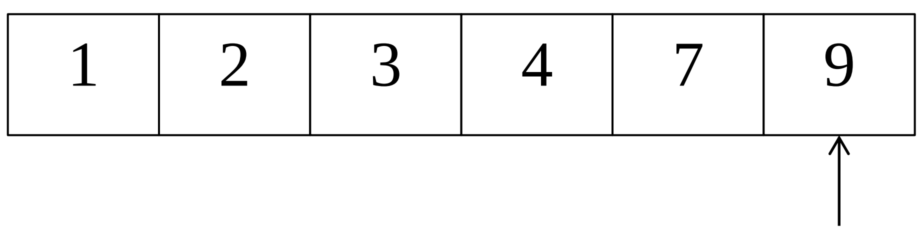
- Look at the first item in each subarray. Choose the smallest item.
- Move the chosen item to the output array



Merge



- Look at the first item in each subarray. Choose the smallest item.
- Move the chosen item to the output array



Merge

			∞

			∞

- We know that we have only n = (r – p) + 1 items. So, we will make only (r – p) + 1 moves.
- Here r = 1 and p = 6, and (6 – 1) + 1 = 6, so when we have made our 6th move we're through.

1	2	3	4	7	9

Merge(A,p,q,r)

```
MERGE(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
for i ← 1 to n1
        do L[i] ← A[p + i − 1]
for j ← 1 to n2
        do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
        do if L[i] ≤ R[j]
              then A[k] ← L[i]
                   i ← i + 1
              else A[k] ← R[j]
                   j ← j + 1
```

Analysis of Divide-and-Conquer

- The Merge-Sort algorithm contains a recursive call to itself. When an algorithm contains a recursive call to itself, its running time often can be described by a *recurrence equation, or recurrence.*
- The recurrence equation describes the running time on a problem of size n in terms of the running time on smaller inputs.
- We can use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

Analysis of Divide-and-Conquer

- A recurrence of a divide-and-conquer algorithm is based on its 3 parts: divide, conquer, and combine.
- Let T(n) be the running time on a problem of size n.
- If the problem is small enough, say n <= c, we can solve it in a straightforward manner, which takes constant time, which we write as Θ(1).
- If the problem is bigger, we solve it by dividing the problem to get ***a*** *subproblems, each of which is 1/****b*** *the size of the original.*
  For Merge-Sort, both *a and b are 2.*

Analysis of Divide-and-Conquer

- Described by recursive equation
- Suppose $T(n)$ is the running time on a problem of size n.
- $T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$

Where a: number of subproblems

n/b: size of each subproblem

$D(n)$: cost of divide operation

$C(n)$: cost of combination operation

Analysis of MERGE-SORT

- **Base case:** $n = 1$. Merge sort on an array of size 1 takes constant time, $\Theta(1)$.
- **Divide:** The Divide step of Merge-Sort just calculates the middle of the subarray. This takes constant time. So $D(n) = \Theta(1)$.
- **Conquer:** We make 2 calls to Merge-Sort. Each call handles 1/2 of the subarray that we pass as a parameter to the call. The total time required is $2T(n/2)$.
- **Combine:** Running Merge on an n-element subarray takes $\Theta(n)$, so $C(n) = \Theta(n)$.

Analysis of MERGE-SORT

- Here is what we get:

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

- By inspection, we can see that we can ignore the Θ(1) factor, as it is irrelevant compared to Θ(n). We can rewrite this recurrence as:

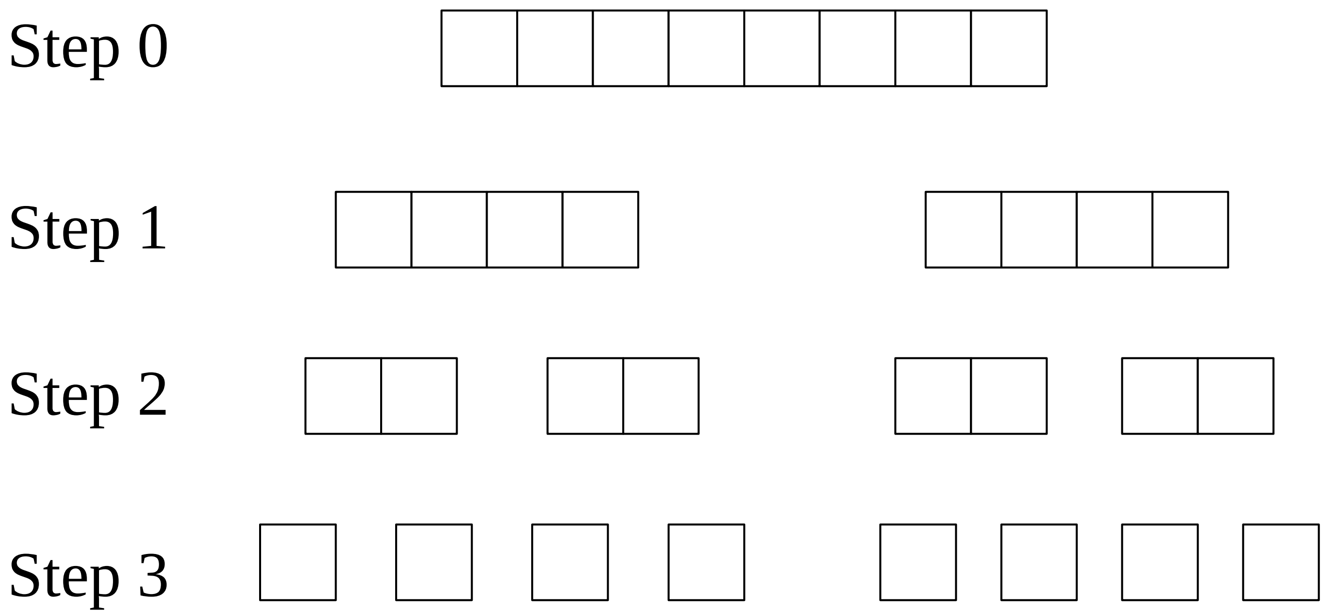
$$T(n) = \begin{cases} c & \text{if } n=1 \\ 2T(n/2) + c(n) & \text{if } n>1 \end{cases}$$

Analysis of MERGE-SORT

- How many Divide steps?
- Let's assume that n is some power of 2.
- Then for an array of size n, it will take us $\log_2 n$ steps to recursively subdivide the array into subarrays of size 1.

Analysis of MERGE-SORT

Example: $8 = 2^3$



Analysis of MERGE-SORT

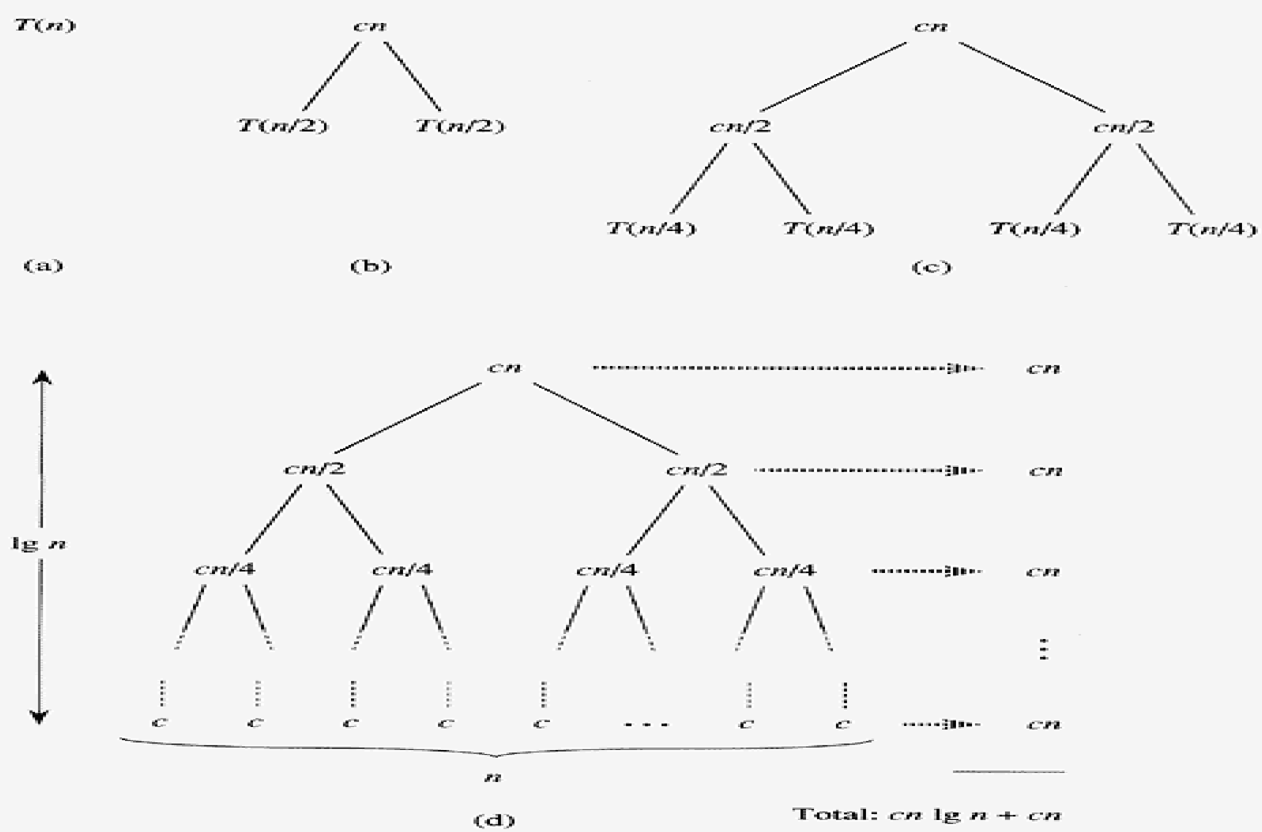


Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

Analysis of MERGE-SORT

- So, it took us $\log_2 n$ steps to divide the array all the way down into subarrays of size 1.
- As a result, we will have $\log_2 n + 1$ layers in the recurrence tree.
- Every layer of the recurrence tree it takes us n steps, since we have to put each array item into its proper position within each array.

Analysis of MERGE-SORT

- Consequently, the total cost can be expressed as: $cn(\log_2 n + 1)$
  $= cn(\log_2 n) + cn$
- Ignoring the low-order term and the constant c gives:

$$\Theta(n \cdot \log_2 n)$$