Chapter 2
Getting Started

The slides for this course are based on the course textbook: Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 3rd edition, The MIT Press, McGraw-Hill, 2010.

- Many of the slides were provided by the publisher for use with the textbook. They are copyrighted, 2010.
- These slides are for classroom use only, and may be used only by students in this specific course this semester. They are NOT a substitute for reading the textbook!

Formal definition of a problem

We can formally define a problem by specifying an input, an output, and the desired relationship between the two.

Sorting problem

Here is the formal definition of the *sorting problem*:

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

Output: A permutation (reordering) $\langle a_1', a_2', \ldots, a_n' \rangle$ of the input sequence such that $a_1' \le a_2' \le \ldots \le a_n'$

The numbers that we wish to sort are also known as the *keys*.

Insertion sort: method 1

Put all cards on the table; call this the *deck*. Let n be the number of cards in the deck.

The *hand* is empty.

Loop

- Pick the top card from the deck.
- Put it in its correct location in the set of cards in the hand.

until deck is empty.

Insertion sort

Input: Deck

Output: Hand

Entry conditions:

Deck must contain one or more cards.

Hand must be empty.

Exit conditions:

Deck is empty.

Hand consists of sorted sequence of cards.

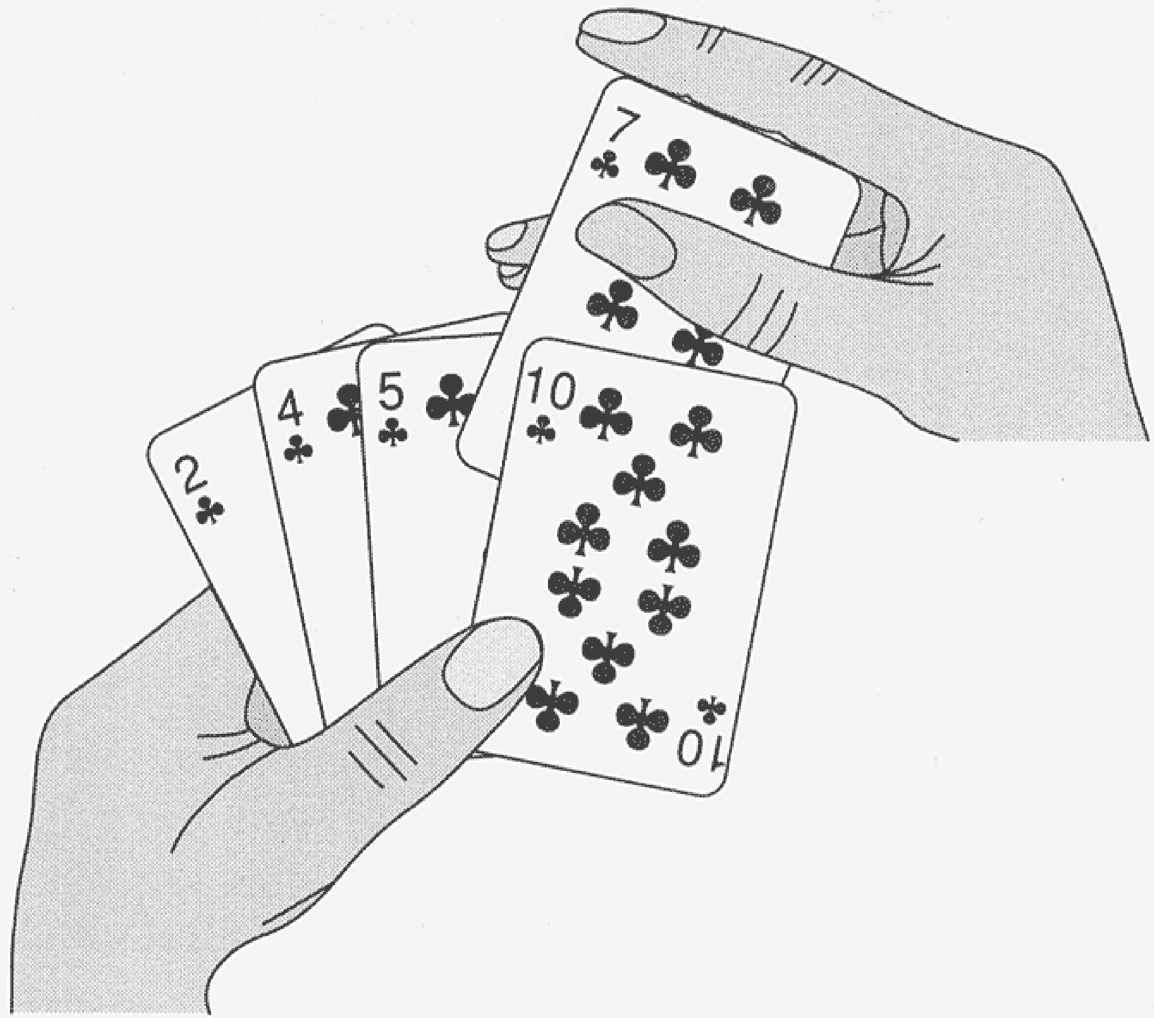


Figure 2.1 Sorting a hand of cards using insertion sort.

Space analysis

What is the space efficiency of this algorithm?

Time efficiency

What is the run-time efficiency of this algorithm?

Loop

- Pick the top card from the deck. (1 step)
- Put it in its correct location in the set of cards in the hand.

until deck is empty.

More detailed analysis

- Put the card from the Deck in its correct location the set of cards in the Hand.

More detailed analysis

So it depends on the # of cards in the Hand.
The worst case is that you might have to search through:

0 HandCards to insert the first DeckCard
1 HandCard to insert the second DeckCard
2 HandCards to insert the third DeckCard
• • •
n -1 HandCards to insert the nth DeckCard

$$TotalSteps =$$

More detailed analysis

In our equation

$$TotalSteps = \frac{n^2}{2} - \frac{n}{2}$$

the n^2 term is the dominant term, so the worst-case performance for our Insertion Sort algorithm is $O(n^2)$

We will talk about the *Big-Oh* notation later.

Insertion sort: method 2

Our first try at doing an insertion sort was inefficient in its use of space, and our analysis was clumsy.

Let's do an *in-place* insertion sort.

Insertion sort: method 2

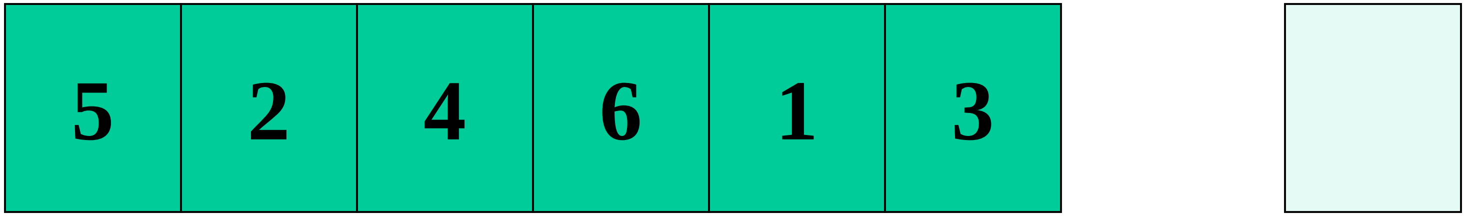
5	**2**	**4**	**6**	**1**	**3**

Here is our original Deck. It is represented by an array of length n, where here $n = 6$.

Insertion-sort(A)

```
for j ← 2 to length(A)
    do key ← A[j]
          // Insert A[j] into the sorted
          // sequence A[1..j - 1]
          i ← j - 1
          while i > 0 and A[i] > key
                do A[i + 1] ← A[i]
                      i ← i - 1
          A[i + 1] ← key
```

Insertion sort: method 2



Set *j* to 0.

A list of length 0 is always sorted.

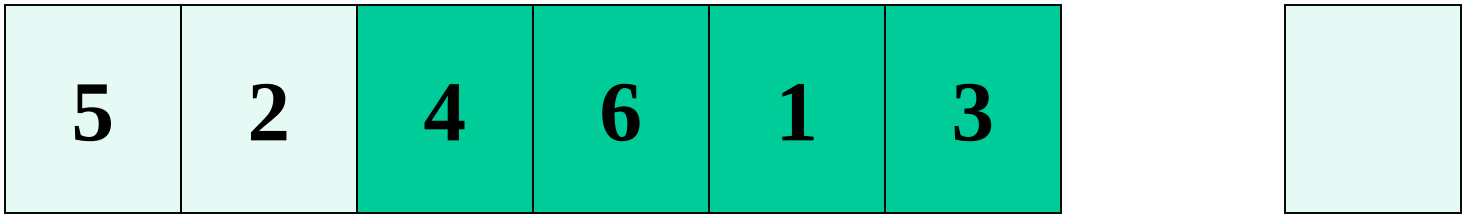
Insertion sort: method 2

5	**2**	**4**	**6**	**1**	**3**

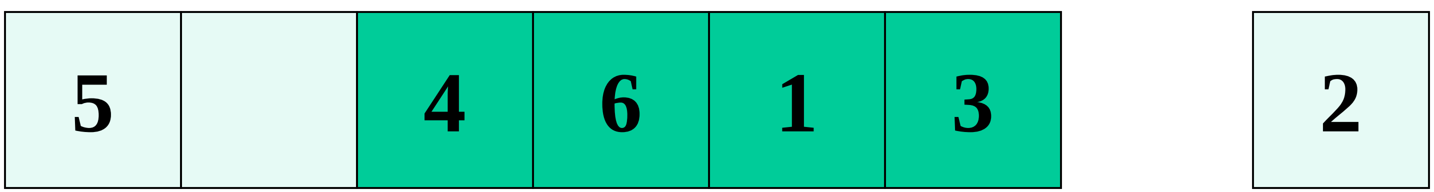
Add 1 to *j*. j is now 1.
A list of length 1 is always sorted.

5	**2**	**4**	**6**	**1**	**3**

Insertion sort: method 2



Add 1 to *j*. *j* is now 2.
Pretend that the array is of length *j*.
Take the *j*th element out of the array and hold it in our temporary storage location.



Insertion sort: method 2

5		**4**	**6**	**1**	**3**		**2**

Insert the temporary element into the list in its correct place.

How do we do this?

We compare the temporary element with each element of the sorted subarray.

5		**4**	**6**	**1**	**3**		**2**

Insertion sort: method 2

5		**4**	**6**	**1**	**3**		**2**

Let’s compare from the right (larger) side of the sorted subarray to the left (smaller) side. The j^{th} position in the sorted subarray is where the 2 was, so let’s set i to j-1 and compare the temporary element with the i^{th} element.

5		**4**	**6**	**1**	**3**		**2**

Insertion sort: method 2

5		**4**	**6**	**1**	**3**		**2**

The i^{th} element is the 1^{st} element (2 - 1 = 1), so we compare the temporary element with 5.
2 is less than 5, so the 5's correct position in the sorted subarray is to the right of the 2.

5		**4**	**6**	**1**	**3**		**2**

Insertion sort: method 2

5		**4**	**6**	**1**	**3**		**2**

Move the first element into the second element of the array (the j^{th} element).

	5	**4**	**6**	**1**	**3**		**2**

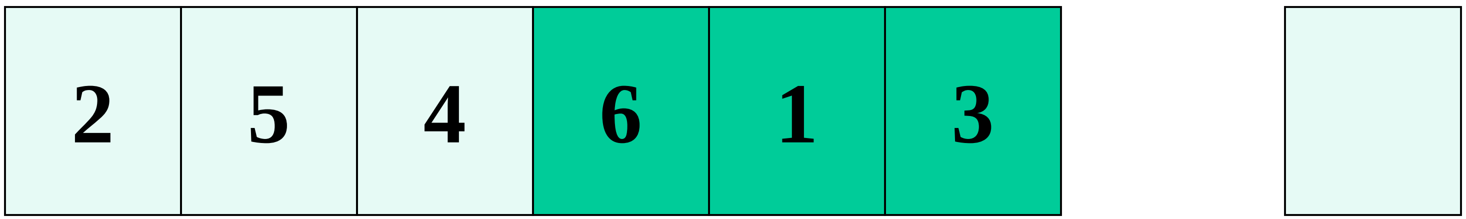
Insertion sort: method 2

	5	**4**	**6**	**1**	**3**		**2**

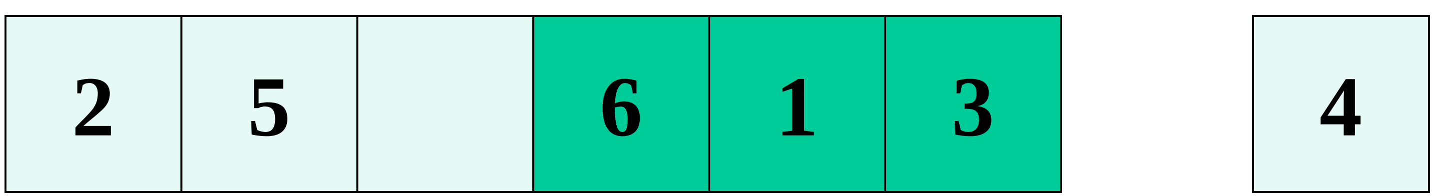
Now decrement *i*. This makes *i* = 0, so stop moving items.
Now insert the 2 in its correct (*i* + 1) place.
We are guaranteed that the array (so far) is sorted.

2	**5**	**4**	**6**	**1**	**3**		

Insertion sort: method 2



Add 1 to *j*. *j* is now 3.
Pretend that the array is of length *j*.
Take the *j*th element out of the array and hold it in our temporary storage location.



Insertion sort: method 2

2	**5**		**6**	**1**	**3**		**4**

Set i to $j - 1$.
Compare the 4 with the i^{th} (second) element in our sorted sublist. 5 is greater than 4, so 4 must come before 5. Move the 5 to the right.
Decrement i. Now $i = 1$.

2		**5**	**6**	**1**	**3**		**4**

Insertion sort: method 2

2		**5**	**6**	**1**	**3**		**4**

Compare the 4 with the i^{th} (first) element in our sorted sublist. 2 is less than 4, so 4 must come after the 2. Stop moving items

2		**5**	**6**	**1**	**3**		**4**

Insertion sort: method 2

2		**5**	**6**	**1**	**3**		**4**

Insert the temporary element into the list in its correct (i + 1) place.
We are guaranteed that the array (so far) is sorted.

2	**4**	**5**	**6**	**1**	**3**		

Insertion sort: method 2

2	**4**	**5**	**6**	**1**	**3**		

Add 1 to *j*. *j* is now 4.
Pretend that the array is of length *j*.
Take the *j*th element out of the array and hold it in our temporary storage location.

2	**4**	**5**		**1**	**3**		**6**

Insertion sort: method 2

2	**4**	**5**		**1**	**3**		**6**

Set i to $j - 1$ (which is 3).
Compare the 6 with the i^{th} (third) element in our sorted sublist. 5 is less than 6, so 6 must come after 5. Don't move anything.

2	**4**	**5**		**1**	**3**		**6**

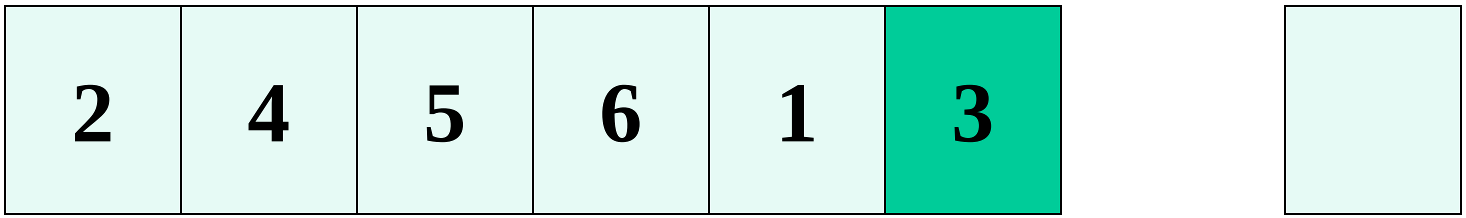
Insertion sort: method 2

2	**4**	**5**		**1**	**3**		**6**

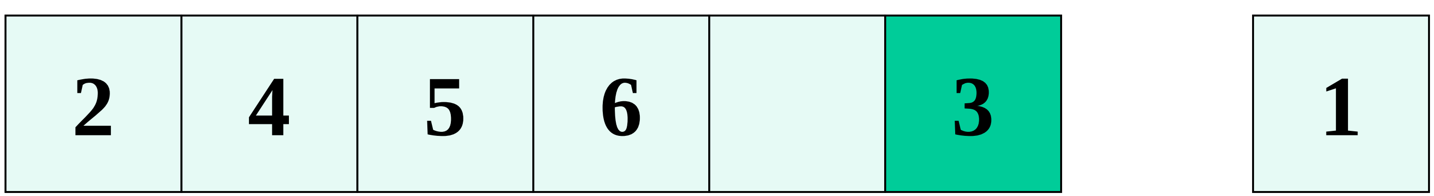
We didn’t have to move anything.
So insert the 6 in its correct (i + 1) place.
We are guaranteed that the array (so far) is sorted.

2	**4**	**5**	**6**	**1**	**3**		

Insertion sort: method 2



Add 1 to j. j is now 5.
Pretend that the array is of length j.
Take the j^{th} element out of the array and hold it in our temporary storage location.



Insertion sort: method 2

2	**4**	**5**	**6**		**3**		**1**

Set i to $j - 1$ (which is 4).
Compare the 1 with the i^{th} (fourth) element in our sorted sublist. 6 is greater than 1, so 1 must come before 6. Move the 6 to the right..
Decrement i. Now $i = 3$.

2	**4**	**5**		**6**	**3**		**1**

Insertion sort: method 2

2	**4**	**5**		**6**	**3**		**1**

Compare the 1 with the i^{th} (third) element in our sorted sublist. 5 is greater than 1, so 1 must come before 5. Move the 5 to the right. Decrement i. Now $i = 2$.

2	**4**		**5**	**6**	**3**		**1**

Insertion sort: method 2

2	**4**		**5**	**6**	**3**		**1**

Compare the 1 with the i^{th} (second) element in our sorted sublist. 4 is greater than 1, so 1 must come before 4. Move the 4 to the right. Decrement i. Now $i = 1$.

2		**4**	**5**	**6**	**3**		**1**

Insertion sort: method 2

2		**4**	**5**	**6**	**3**		**1**

Compare the 1 with the i^{th} (first) element in our sorted sublist. 2 is greater than 1, so 1 must come before 2. Move the 2 to the right. Decrement i. Now $i = 0$.
Since $i = 0$, stop moving things.

	2	**4**	**5**	**6**	**3**		**1**

Insertion sort: method 2

	2	**4**	**5**	**6**	**3**		**1**

The 2, 4, 5, and 6 have been moved.
Now insert the 1 in its correct (*i* + 1) place.
We are guaranteed that the array (so far) is sorted.

1	**2**	**4**	**5**	**6**	**3**		

Insertion sort: method 2

1	**2**	**4**	**5**	**6**	**3**		

Add 1 to *j*. *j* is now 6.
Pretend that the array is of length *j*.
Take the *j*th element out of the array and hold it in our temporary storage location.

1	**2**	**4**	**5**	**6**			**3**

Insertion sort: method 2

1	**2**	**4**	**5**	**6**			**3**

Set i to $j - 1$ (which is 5).
Compare the 3 with the i^{th} (fifth) element in our sorted sublist. 6 is greater than 3, so 3 must come before 6. Move the 6 to the right.
Decrement i. Now $i = 4$.

1	**2**	**4**	**5**		**6**		**3**

Insertion sort: method 2

1	**2**	**4**	**5**		**6**		**3**

Compare the 3 with the i^{th} (fourth) element in our sorted sublist. 5 is greater than 3, so 3 must come before 5. Move the 5 to the right. Decrement i. Now $i = 3$.

1	**2**	**4**		**5**	**6**		**3**

Insertion sort: method 2

1	**2**	**4**		**5**	**6**		**3**

Compare the 3 with the i^{th} (third) element in our sorted sublist. 4 is greater than 3, so 3 must come before 4. Move the 4 to the right. Decrement i. Now $i = 2$.

1	**2**		**4**	**5**	**6**		**3**

Insertion sort: method 2

1	**2**		**4**	**5**	**6**		**3**

Compare the 3 with the i^{th} (second) element in our sorted sublist. 2 is less than 3, so 3 must come after 2. Stop moving things.

1	**2**		**4**	**5**	**6**		**3**

Insertion sort: method 2

1	**2**		**4**	**5**	**6**		**3**

The 4, 5, and 6 have been moved.
Now insert the 3 in its correct (i + 1) place.
We are guaranteed that the array (so far) is sorted.

1	**2**	**3**	**4**	**5**	**6**		

Insertion sort: method 2

1	**2**	**3**	**4**	**5**	**6**		

Add 1 to *j*. *j* is now 7.
Oops! *j* now exceeds *n*. Exit from the outer loop, and return the sorted list.

1	**2**	**3**	**4**	**5**	**6**		

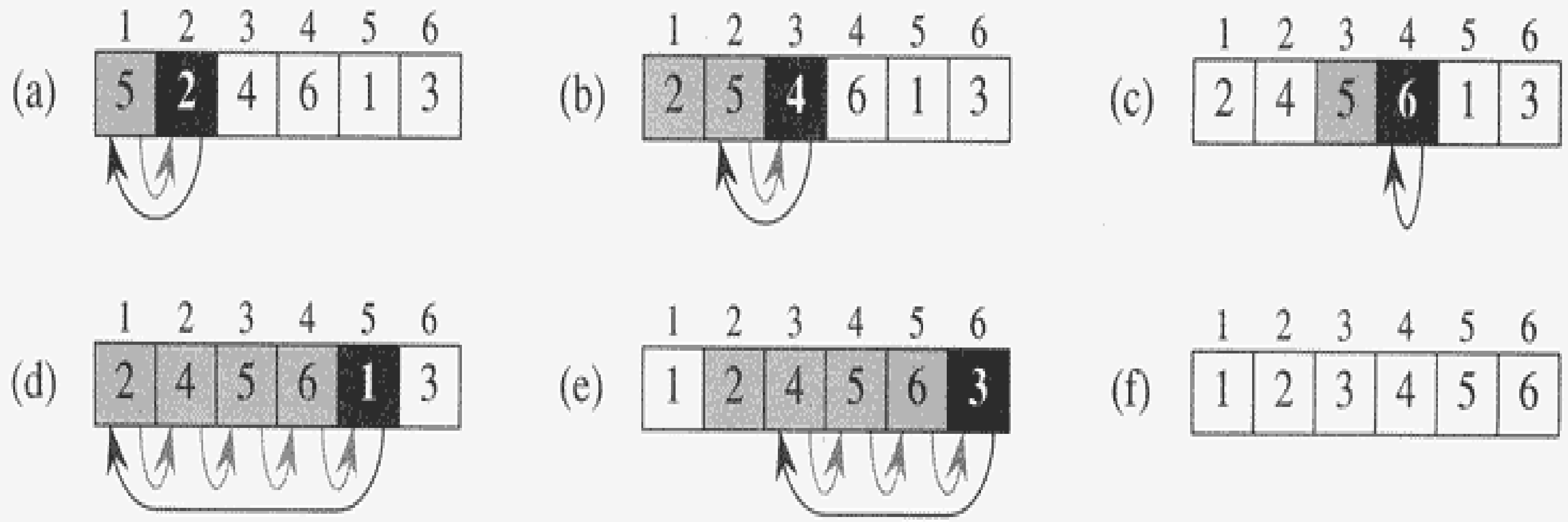


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key is moved to in line 8. **(f)** The final sorted array.

Insertion-sort(A)

```
for j ← 2 to length(A)
  do key ← A[j]
     // Insert A[j] into the sorted
     // sequence A[1..j - 1]
     i ← j - 1
     while i > 0 and A[i] > key
        do A[i + 1] ← A[i]
           i ← i – 1
     A[i + 1] ← key
```

Insertion-sort(A)

As we go through the outer loop (the **`for`** loop), we are guaranteed that the part of the array from element # 1 up through element j – 1 is in sorted order.

We can state this as a formal *loop invariant*:

“At the start of each iteration of the **for** loop of lines 1-8, the subarray A[1..j-1] consists of the elements originally in A[1..j-1] but in sorted order.”

Loop invariants

We must show three things about a loop invariant:

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Loop invariants

Initialization: It is true prior to the first iteration of the loop.

Is the array sorted prior to the first iteration of the loop?

Loop invariants

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

If the array is sorted prior to an iteration of the loop, will that iteration preserve its sorted status ?

Loop invariants

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

What happens when the loop terminates? Does that preserve the sorted order?

Analyzing algorithms

For purposes of analysis, we will assume that all of our algorithms are running on a RAM computer (generic random access machine), with no parallelization, no special instructions, capabilities, etc.

Analyzing algorithms

Running time: Assume that we write our algorithm in such a way that it has i lines (or steps), each of which takes a constant amount of time to execute on our RAM. So, c_1 is the time it takes to execute line 1, c_2 is the time it takes to execute line 2, etc. Obviously, if there are no loops in our algorithm, its running time will be $c_1 + c_2 + \ldots c_i$. A constant plus a constant plus a constant . . . equals a constant. So the running time of any algorithm with no loops is a constant.

Analyzing algorithms

What if the algorithm has loops?

1. Constant number of times:

2. Depends on the size of the *input*:

Analyzing algorithms

The *size* of the input usually means the *number of items* in the input.
However, for some problems the size may best be described in other terms.

For example, to analyze a low-level algorithm for multiplying two integers, the *size* is the number of bits it takes to represent the input.

Another example is an algorithm for manipulating a graph; graphs have both edges and vertices, so the *size* of the input will be two numbers instead of one.

Insertion-sort(A)

```
for j ← 2 to length(A)
   do key ← A[j]
      // Insert A[j] into the sorted
      // sequence A[1..j - 1]
      i ← j - 1
      while i > 0 and A[i] > key do
         A[i + 1] ← A[i]
         i ← i - 1
      A[i + 1] ← key
```

Insertion sort

Let’s look at the details of the running time of this algorithm.

“Cost” is some constant value that indicates the computation cost (e.g., in terms of CPU cycles, etc.) of the operation performed in a line of the algorithm.
“Times” will be the number of times a particular line of the algorithm will be executed.

	INSERTION-SORT(A)	Cost	Times
1	**for** j ← 2 to length(A) **do**	c1	n
2	key ← A[j]	c2	n - 1
3	// Insert A[j] into the sorted sequence A[1..j - 1]	0	n - 1
4	i ← j – 1	c4	n - 1
5	**while** i > 0 and A[i] > key **do**	c5	$\sum_{j=2}^{n} t_j$
6	A[i + 1] ← A[i]	c6	$\sum_{j=2}^{n} (t_j - 1)$
7	i ← i – 1	c7	$\sum_{j=2}^{n} (t_j - 1)$
8	A[i + 1] ← key	c8	n - 1

Insertion sort

• Line 3 is a comment line, and comments are considered not to cost anything, since they will not actually be executed when a program runs.

• Line 1 begins an outer **for** loop. All of the other lines are within this loop.

• Lines 2, 4, and 8 are directly under the **for** loop. The body of this outer loop will execute n–1 times (as j goes from 2 to n, where n = length(A)). So lines 2,4, and 8 will execute n-1 times.

• Line 1 will be checked one extra time for the exit condition from the loop, so it executes n times.

Insertion sort

- Lines 5, 6 and 7 are within the inner **while** loop. The number of times they will be executed (called *t*) depends upon the value of *j* at the time the **while** loop is entered. The value of *j* is determined by the value of the **for** loop in line 1.
- So lines 6 and 7 will be executed $\sum_{j=2}^{n}(t_j - 1)$ times.
- Line 5 will be checked one extra time for the exit condition from the loop every time it is visited, so it executes $\sum_{j=2}^{n} t_j$ times.

Insertion sort

The total cost of insertion sort is:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Insertion sort

The order in which the elements of the array are listed will affect the running time cost of insertion sort.

For this algorithm the *best case* occurs when the array is already sorted.

The *worst case* occurs when the array is in reverse order.

The *average case* occurs – well, most of the time.

These are the three performances of an algorithm we are normally interested in: best case, average case, and worst case.

Insertion sort

Best case:

1	**2**	**3**	**4**	**5**	**6**

The inner loop of our array starts off:

```
while i > 0 and A[i] > key
```

Do we ever have to do the body of this inner loop more than once, for each item in the array?

Insertion sort

So the best case running time for insertion sort is:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

This can be expressed as:

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

which can be reduced to:

an + b

which is a *linear* function of n.

So the best case performance of insertion sort is *linear* time.

Insertion sort

Worst case:

6	**5**	**4**	**3**	**2**	**1**		

The inner loop of our array starts off:

```
while i > 0 and A[i] > key
```

How many times do we have to do the body of this inner loop?

Insertion sort

So the worst case running time for insertion sort is:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

This is equivalent to:

$$T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

which can be reduced to:

$an^2 + bn + c$

which is a *quadratic* function of n

Algorithm analysis

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.
- The average case performance is often about as bad as the worst case.

Algorithm analysis

Average case analysis is especially hard to do:
• What is an “average” input for a problem?
• Can’t just assume that all instances are equally likely.
• We sometimes can use a *randomized algorithm* to allow a *probabilistic analysis*.

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 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

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.

Here is the algorithm for Merge-Sort:

Merge sort

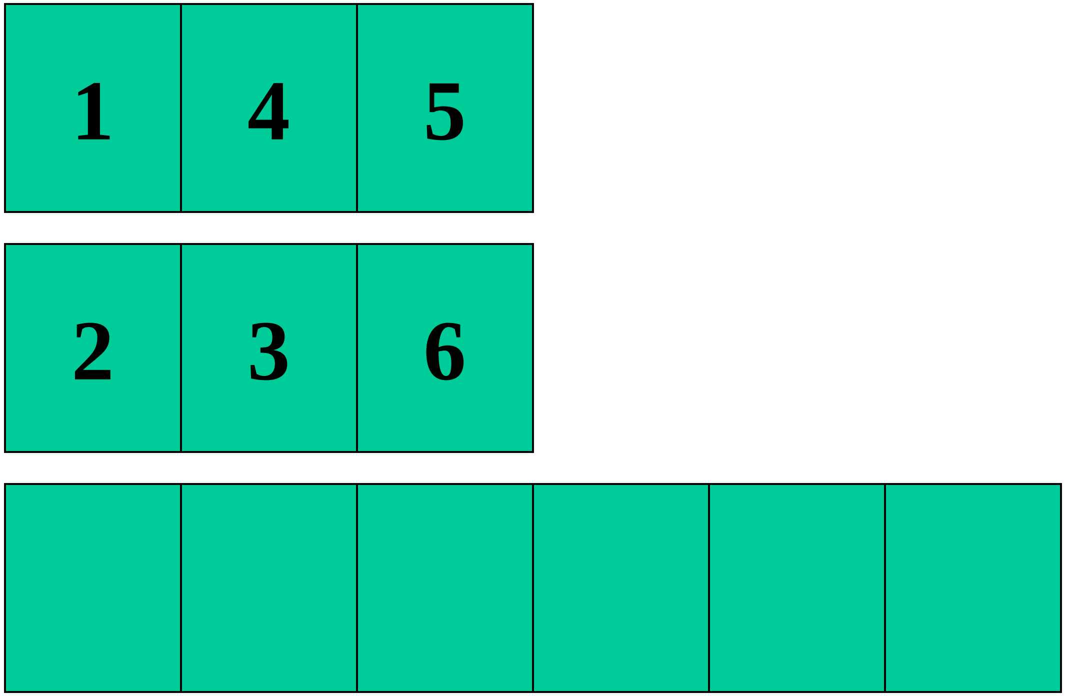
```
Merge-Sort(A, p, r)
if p < r
then {q ← ⌊(p+r)/2⌋
          Merge-Sort(A, p, q)
          Merge-Sort(A, q+1, r)
          Merge(A, p, q, r)}
```

A is the (sub)array *when the procedure is called*.

p, q, and r are indices numbering elements of the array such that $p \leq q \leq r$; p is the lowest index and r is the highest index.

Merge

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

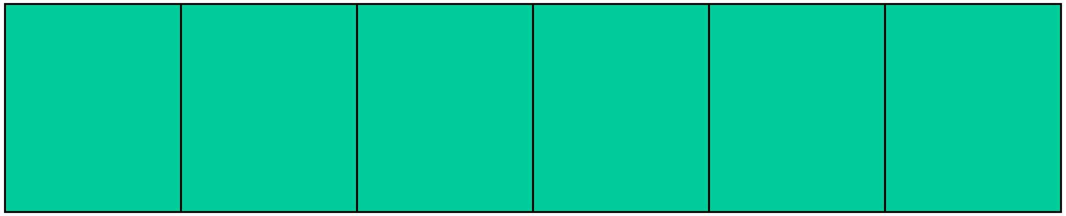


Merge

1	**4**	**7**	**∞**

2	**3**	**9**	**∞**

Let’s assume we have a *sentinel* (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**	**∞**
p		q	

2	**3**	**9**	**∞**
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

1	**4**	**7**	**∞**

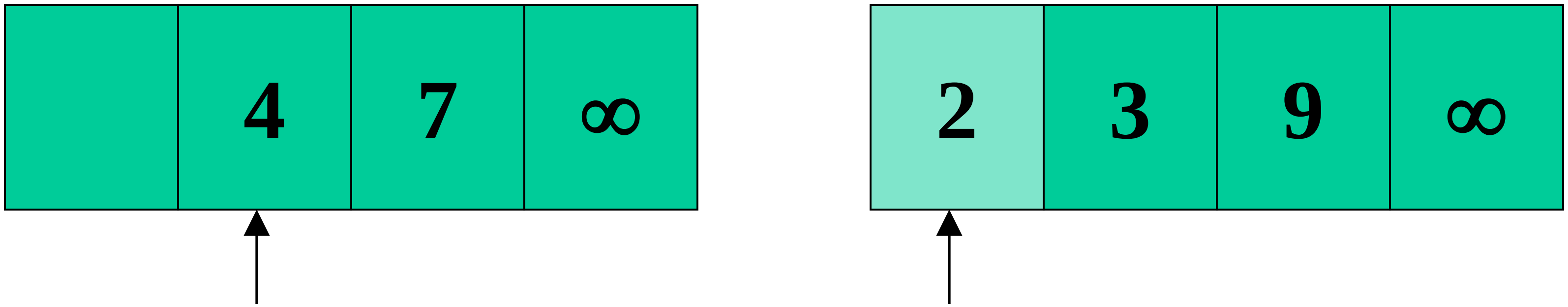
2	**3**	**9**	**∞**

Look at the first item in each subarray. Choose the smallest item.

Move the chosen item to the output array.

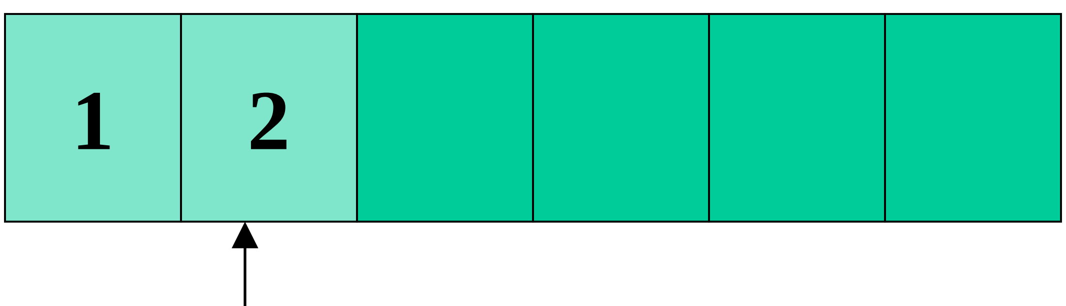
1					

Merge

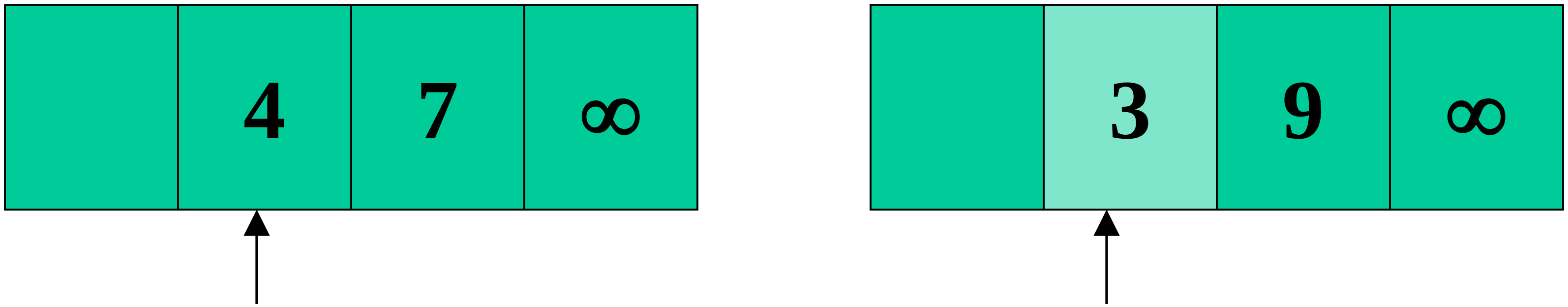


Look at the first item in each subarray. Choose the smallest item.

Move the chosen item to the output array.

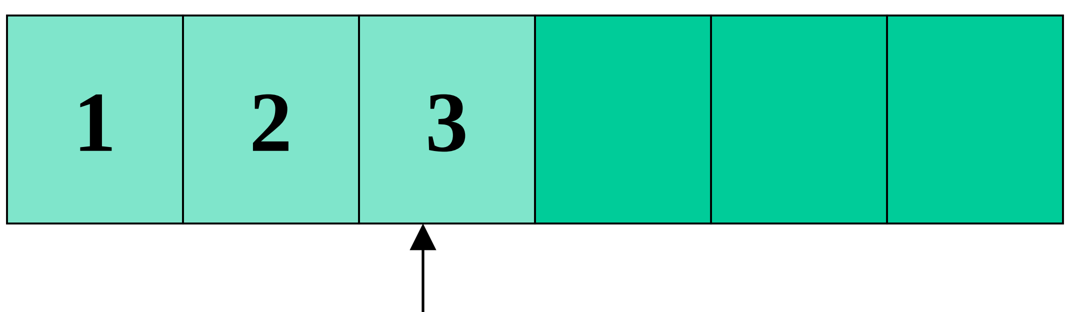


Merge

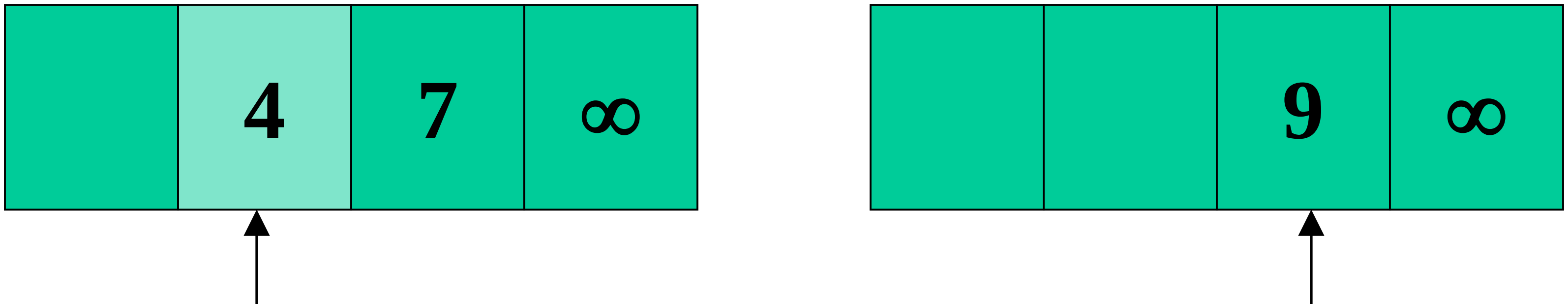


Look at the first item in each subarray. Choose the smallest item.

Move the chosen item to the output array.

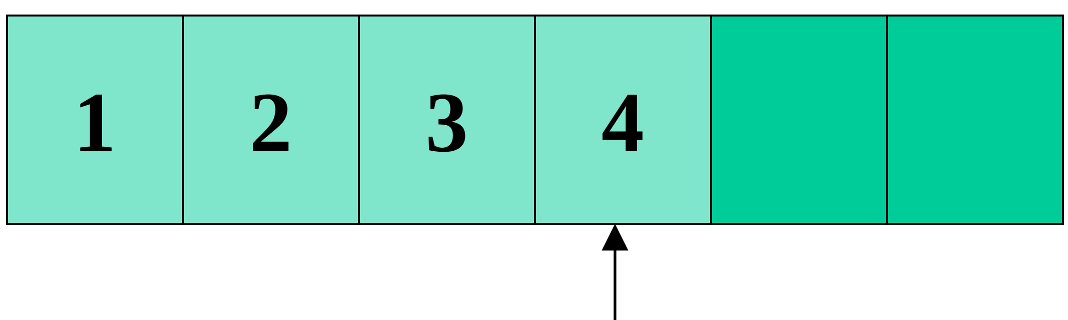


Merge

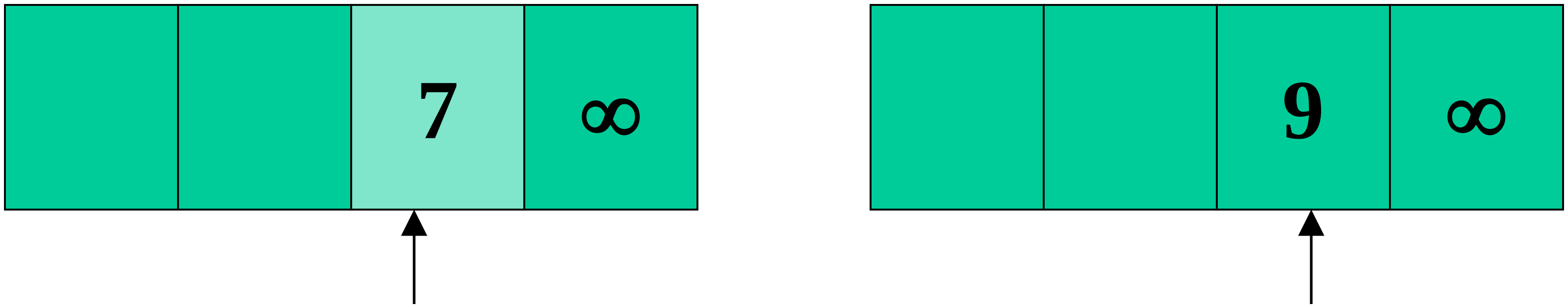


Look at the first item in each subarray. Choose the smallest item.

Move the chosen item to the output array.

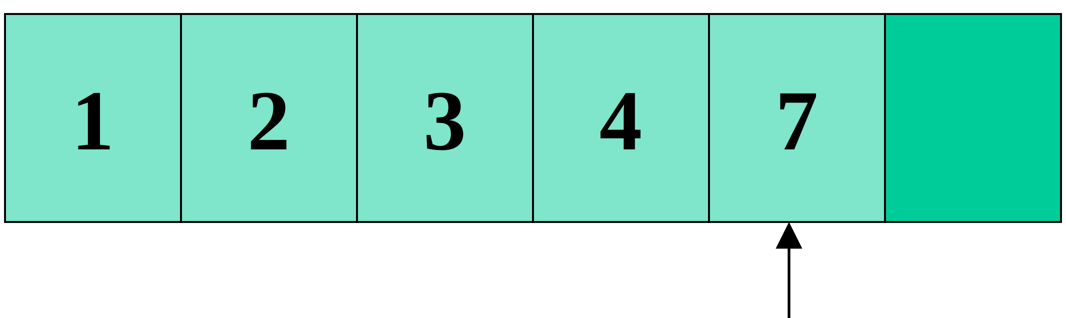


Merge

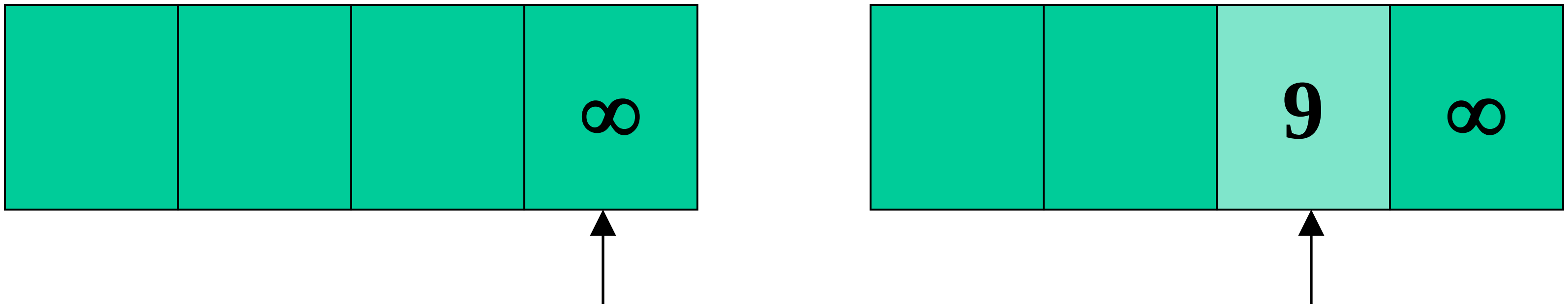


Look at the first item in each subarray. Choose the smallest item.

Move the chosen item to the output array.

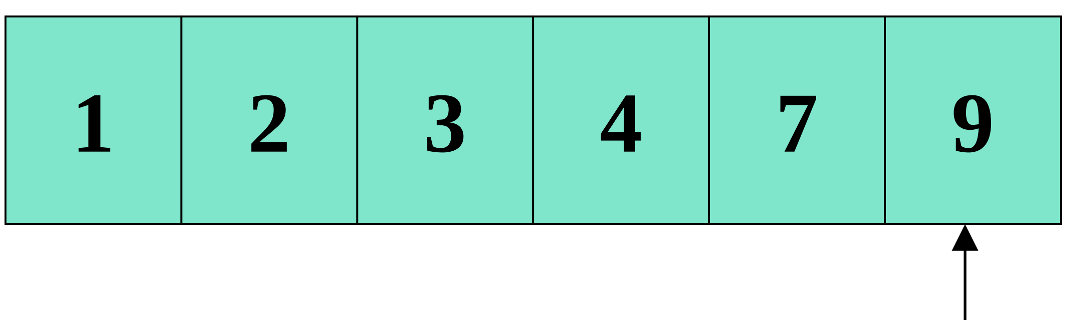


Merge

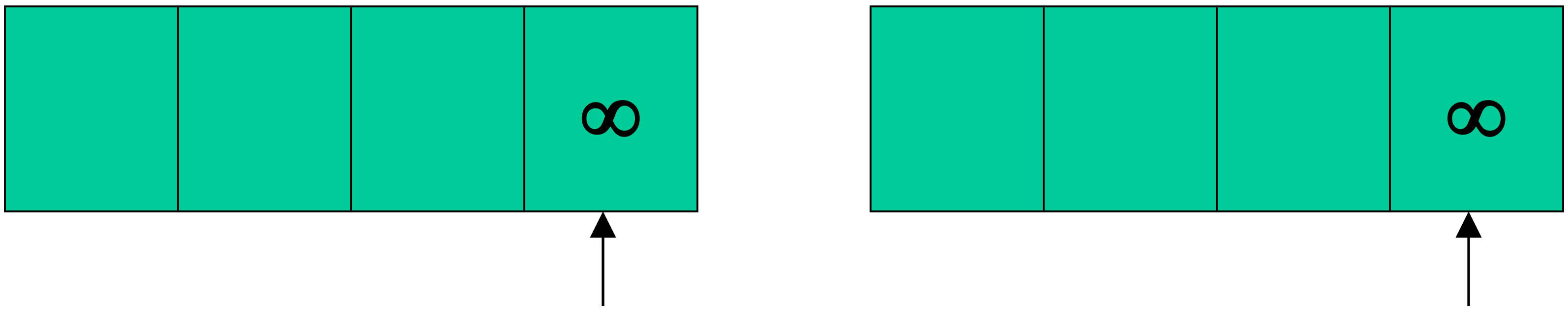


Look at the first item in each subarray. Choose the smallest item.

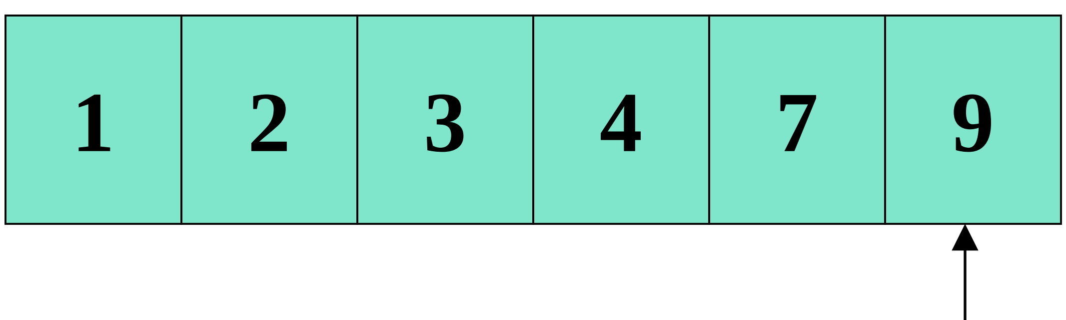
Move the chosen item to the output array.



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 6^{th} move we're through.



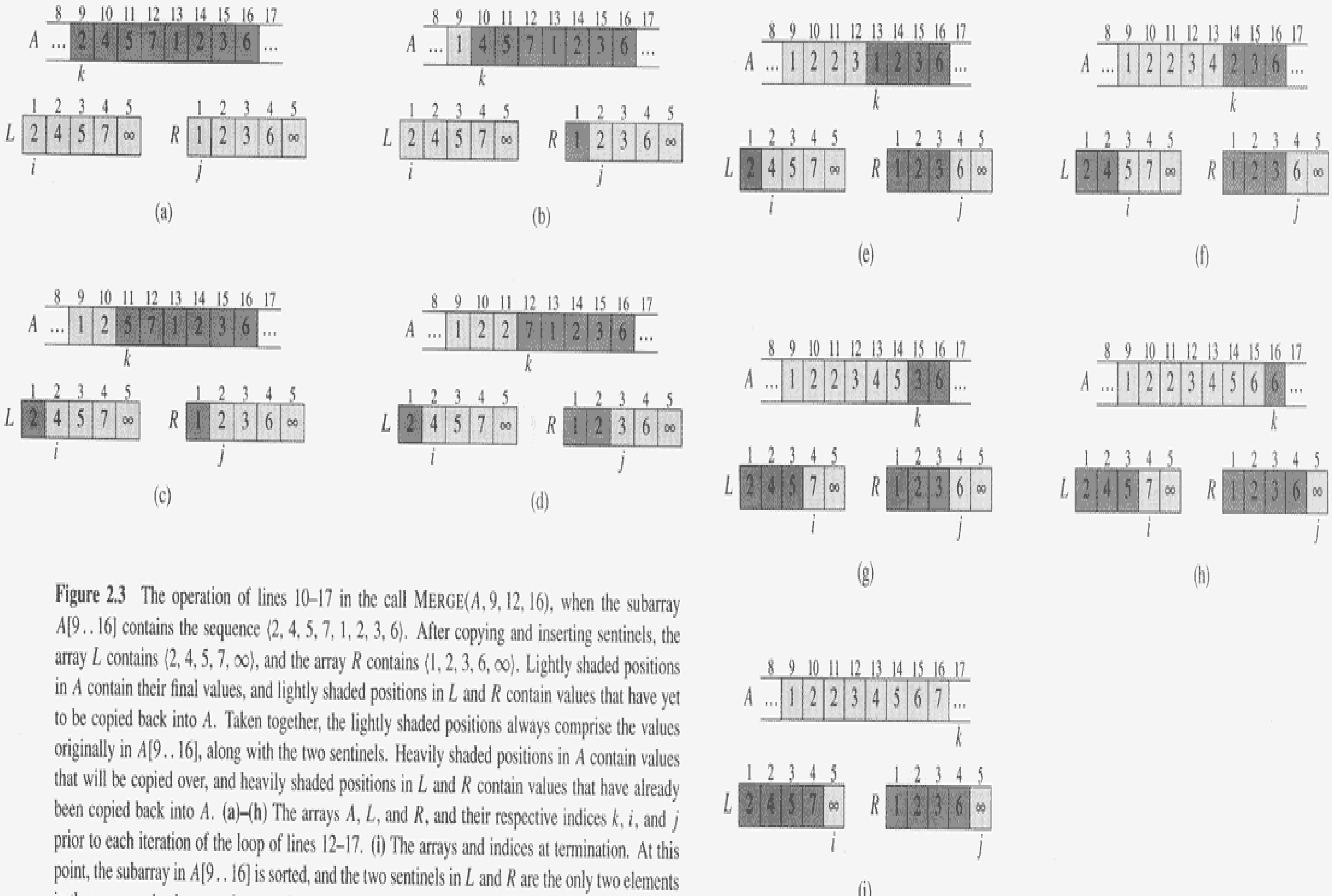


Figure 2.3 The operation of lines 10–17 in the call MERGE(A, 9, 12, 16), when the subarray $A[9..16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$. After copying and inserting sentinels, the array L contains $\langle 2, 4, 5, 7, \infty \rangle$, and the array R contains $\langle 1, 2, 3, 6, \infty \rangle$. Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A. Taken together, the lightly shaded positions always comprise the values originally in $A[9..16]$, along with the two sentinels. Heavily shaded positions in A contain values that will be copied over, and heavily shaded positions in L and R contain values that have already been copied back into A. **(a)–(h)** The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12–17. **(i)** The arrays and indices at termination. At this point, the subarray in $A[9..16]$ is sorted, and the two sentinels in L and R are the only two elements in these arrays that have not been copied into A.

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
```

Merge

Assuming that the two sublists are in sorted order when they are passed to the Merge routine, is Merge guaranteed to output a sorted array?

Analysis of Merge

The loop in lines 12-17 of Merge maintains the loop invariant:

- At the start of each iteration of the **for** loop of lines 12-17, the subarray A[p..k-1]contains the k - p smallest elements of L[1..n_1+1] and R[1..n_2+1], in sorted order. Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

Analysis of Merge

To prove that Merge is a correct algorithm, we must show that:

- **Initialization:** the loop invariant holds prior to the first iteration of the for loop in lines 12-17
- **Maintenance:** each iteration of the loop maintains the invariant
- **Termination:** the invariant provides a useful property to show correctness when the loop terminates

Initialization:

Maintenance:

Termination:

Merge sort

Now let’s look at Merge-Sort again:

```
Merge-Sort(A, p, r)
1  if p < r
2  then {q ← ⌊(p+r)/2⌋
3          Merge-Sort(A, p, q)
4          Merge-Sort(A, q+1, r)
5          Merge(A, p, q, r)}
```

Line 1 is our base case; we drop out of the recursive sequence of calls when p >= r.

Merge sort

Let’s call Merge-Sort with an array of 4 elements:

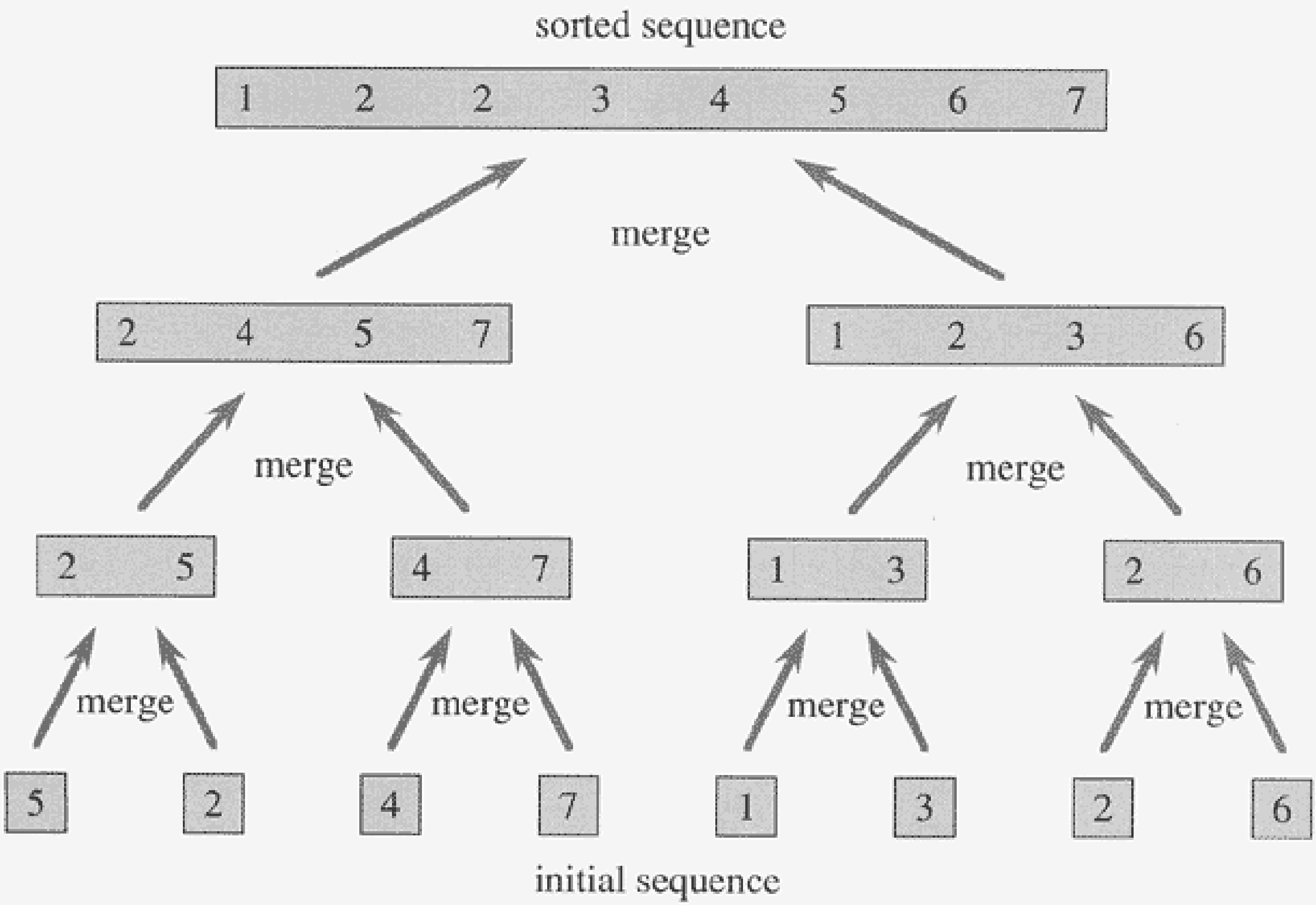


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Analysis of Divide-and-Conquer algorithms

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 algorithms

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 $\Theta(1)$.

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 algorithms

Assume it takes D(n) time to divide the problem into subproblems.

Assume it takes C(n) time to combine the solutions to the subproblem into the solution for the original problem.

We get the recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

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 ½ 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 $\Theta(1)$ factor, as it is irrelevant compared to $\Theta(n)$. We can rewrite this recurrence as:

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

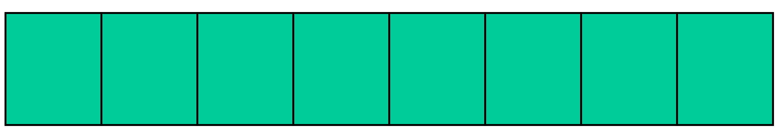
Analysis of Merge-Sort

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.

Example: $8 = 2^3$



Analysis of Merge-Sort

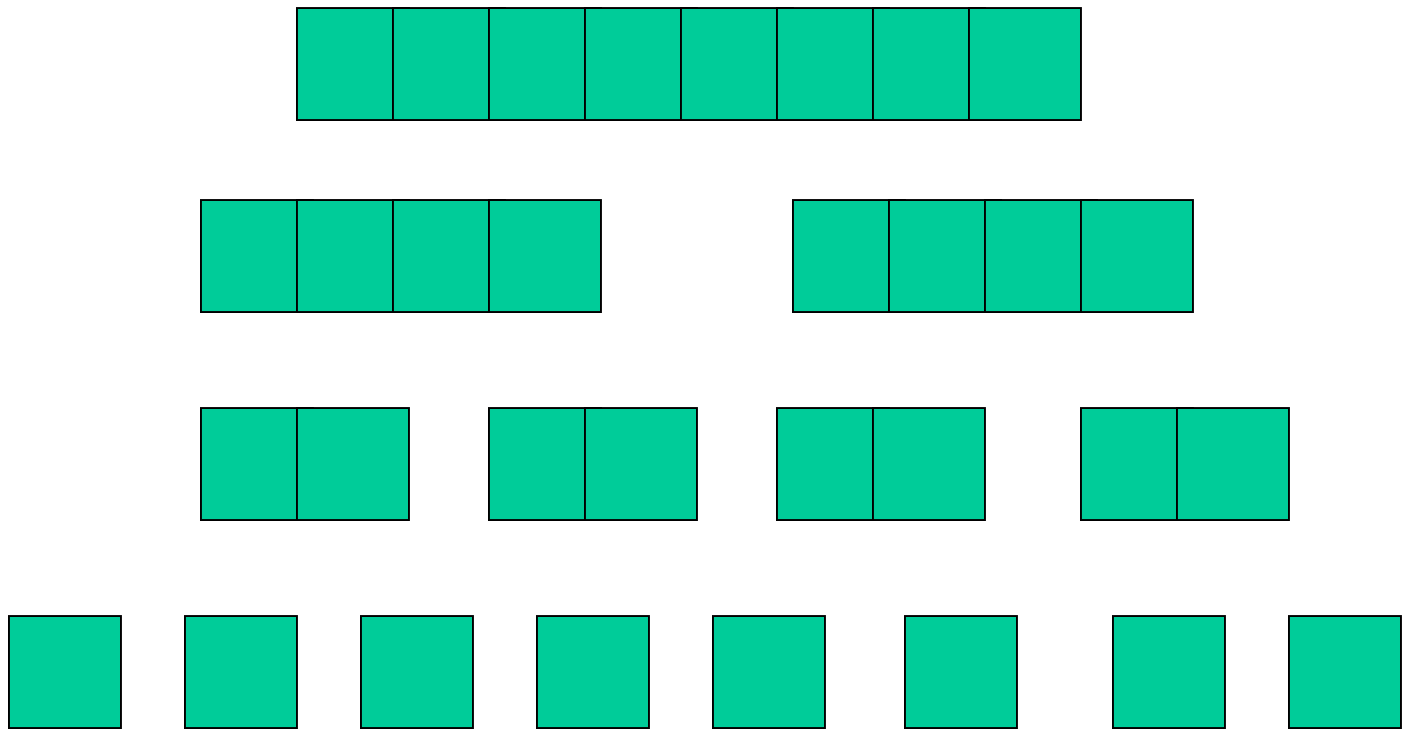
Example: $8 = 2^3$

Step 0:

Step 1:

Step 2:

Step 3:



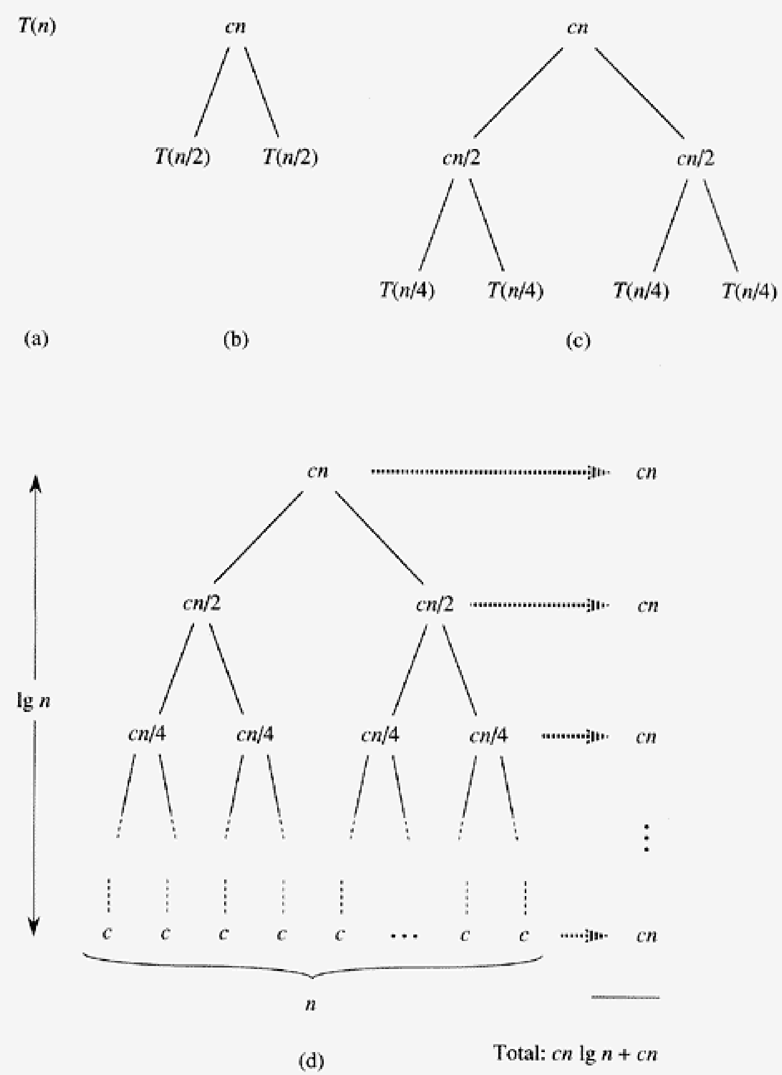


Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each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

$cn(\log_2 n + 1)$

Multiplying this out gives:

$cn(\log_2 n) + cn$

Ignoring the low-order term and the constant c gives:

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

Conclusion

- Insertion Sort
- Merge Sort
- Analysis of Algorithms
- Proof of correctness
- Divide-and-conquer algorithms
- Recurrence relations