ICSI 403 DESIGN AND ANALYSIS OF ALGORITHMS

Lecture 05 – Chapter 2 (Cormen), Getting Started

J Marques de Carvalho

Introduction

- This chapter gives a framework that will be followed in subsequent chapters
- We do some algorithm analysis on two sorting algorithms: Insertion Sort and Merge Sort
- We introduce the authors' pseudocode conventions, and see how to describe algorithms in pseudocode
- Begin using asymptotic notation to express running analysis
- Learn the technique of "divide and conquer" in the context of Merge Sort

The Sorting Problem

- The *sorting problem* (introduced in Chapter 1):
- Input: A sequence of n 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 $a_1' \leq a_2' \leq \ldots \leq a_n'$
- The numbers we wish to sort are called *keys*
- The input and output key sequences are typically stored in arrays

The Sorting Problem (2)

- There may also be data associated with the keys that we don't sort by (satellite data)
  - Perhaps the keys are student ID numbers, but each student ID number has an associated name, address, GPA, transcript, etc. (the satellite data)
  - We sort the keys themselves, but the satellite data associated WITH each key must stay associated with the right key

Expressing Algorithms (1)

- We express algorithms in whatever way is clearest and most concise
- Sometimes, English is the best way
- When issues of control need to be made perfectly clear, we use <u>*pseudocode*</u>
- Pseudocode is similar to C++, Java, C#, Pascal
- Pseudocode is for expressing the algorithms to humans; not to machines

Expressing Algorithms (2)

- Software Engineering and Best Programming Practices like data abstraction, modularity, and error handling are typically ignored in pesudocode
- Occasionally, we will embed English statements in pseudocode

Insertion Sort

- Good for sorting small lists of keys
- Works much like you would sort a hand of playing cards:
  - You start with your hand empty, and all of the cards still sitting (face-down) on the table
  - One-by-one, you take a card from the table, and put it in its proper place in your hand
  - To find the "proper place" for a new card, you compare it, left-to-right (or right-to-left) with the existing cards
  - At all times, the cards in your hand are sorted

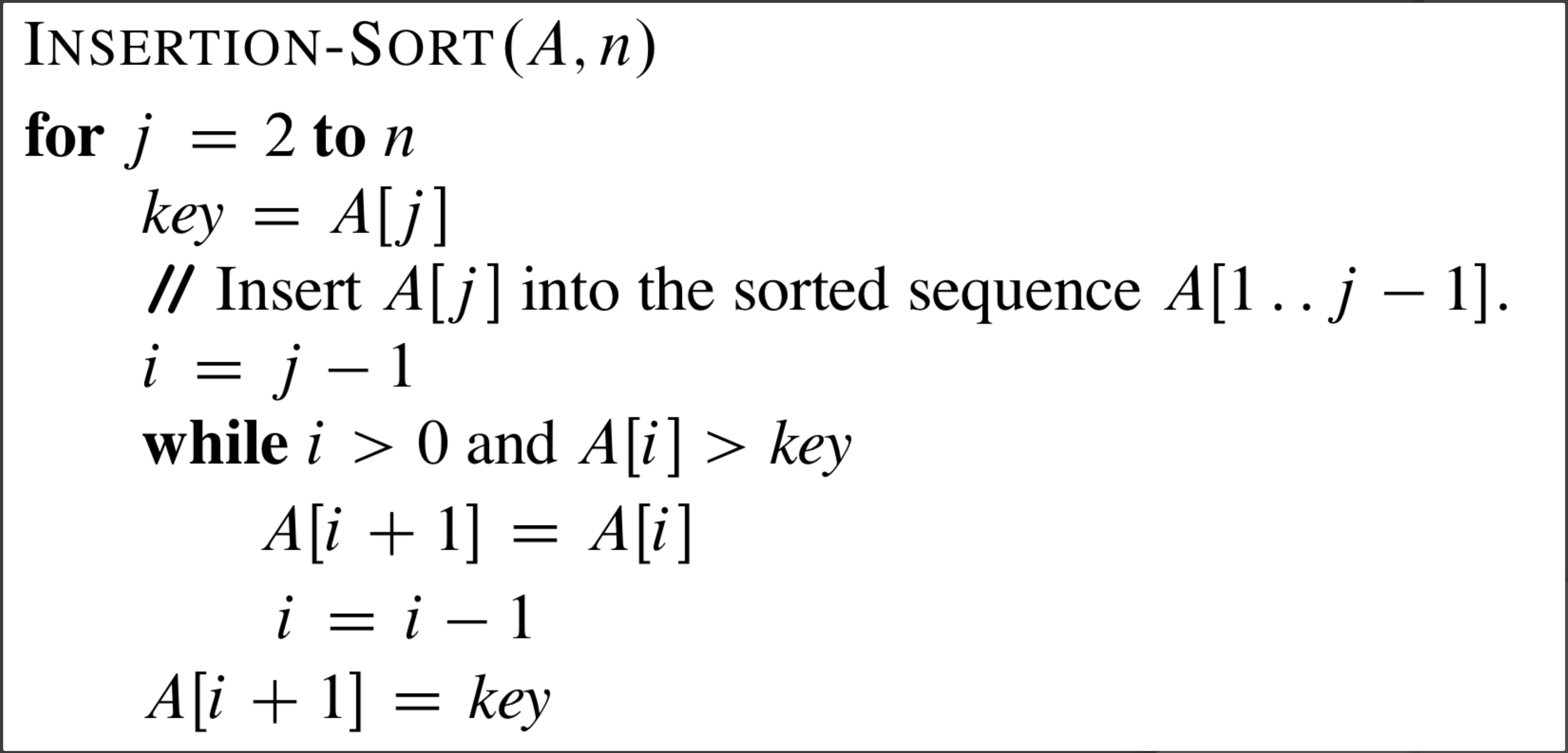
Pseudocode for INSERTION-SORT

- Takes an array $A[1..n]$ and the length n of the array as parameters
- As in Pascal, we use ".." to denote a range in an array
  - The text uses *1-based* arrays (array elements are numbered from 1 to n), rather than *zero-based* arrays, (which are numbered from 0 to n-1).
  - This may require some adjustments in source code
  - The easiest way around this is to set up the array to go from element 0 to element n, and just not use element 0.

Pseudocode for INSERTION-SORT

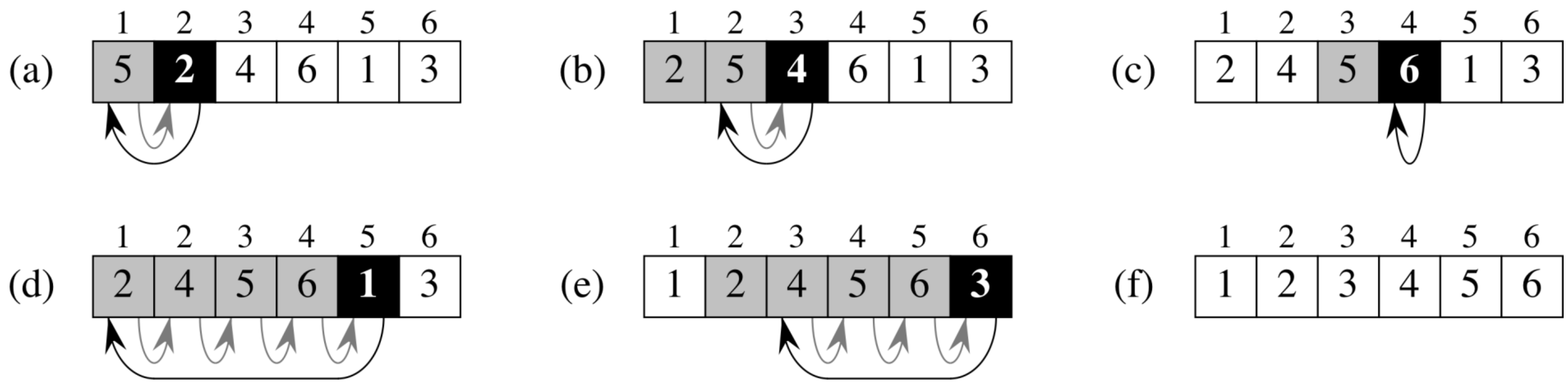
- The array A is sorted in-place: the numbers are rearranged within the existing array.
  - There is, at most, a constant amount of storage required beyond the array A itself

Pseudocode for INSERTION-SORT



```
INSERTION-SORT$(A, n)$
for $j = 2$ to $n$
    $key = A[j]$
    // Insert $A[j]$ into the sorted sequence $A[1 \, . \, . \, j - 1]$.
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i + 1] = A[i]$
        $i = i - 1$
    $A[i + 1] = key$
```

Insertion Sort in Action



Insertion Sort in Action

```
for j = 2 to n
  insert A[j] into list A[1]…A[j-1]
```

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]	A[13]
17	3	9	5	4	8	6	-4	11	13	2	1	16

Step 1: Insert `A[2]` into list `A[1]...A[1]` (move `A[1]` down 1 to make room)

Insertion Sort in Action

```
for j = 2 to n
  insert A[j] into list A[1]…A[j-1]
```

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]	A[13]
3	17	9	5	4	8	6	-4	11	13	2	1	16

Step 1: Insert `A[2]` into list `A[1]...A[1]` (move `A[1]` down 1 to make room)
Step 2: Insert `A[3]` into list `A[1]...A[2]` (move `A[2]` down 1 to make room)

Insertion Sort in Action

```
for j = 2 to n
  insert A[j] into list A[1]…A[j-1]
```

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]	A[13]
3	9	17	5	4	8	6	-4	11	13	2	1	16

Step 1: Insert `A[2]` into list `A[1]...A[1]`	(move `A[1]` down 1 to make room)
Step 2: Insert `A[3]` into list `A[1]...A[2]`	(move `A[2]` down 1 to make room)
Step 3: Insert `A[4]` into list `A[1]...A[3]`	(move `A[2]` and `A[3]` down 1)

The Correctness of INSERTION-SORT

- We often use a *loop invariant* to help understand why an algorithm gives the correct answer
  - **Loop Invariant**: At the start of each iteration of the "outer" for loop (the loop indexed by j) the subarray $A[1 \,..\, j-1]$ consists of the elements originally in $A[1 \,..\, j-1]$, but in sorted order

The Correctness of INSERTION-SORT

- To use a loop invariant to prove correctness, we must show three things about it:
  - *Initialization*: It is true prior to the loop's first iteration
  - *Maintenance*: If it is true before an iteration of the loop, then it remains true before the next iteration
  - *Termination*: When the loop terminates, the invariant (usually along with the reason the loop terminated) gives us a useful property that helps show that the algorithm is correct

The Correctness of INSERTION-SORT

- Loop invariants are like mathematical induction:
  - To prove that a property holds, prove a base case and an inductive step
  - Showing that the invariant holds before the first iteration is like the base case
  - Showing that it holds from iteration to iteration is like the inductive step
  - The termination part differs from the usual use of induction, in which the induction step continues indefinitely. We stop the "induction" when the loop terminates
  - We can show the three parts in any order

The Correctness of INSERTION-SORT

- The loop invariant components of Insertion Sort:
  - Initialization: Just before the first iteration, $j = 2$. The subarray $A[1 .. j - 1]$ is the single element $A[1]$, which is the element originally in $A[1]$, and is trivially sorted
  - Maintenance: To be precise, we would need to state and prove a loop invariant for the "inner" **while** loop. Rather than getting bogged down in another loop invariant, note that the body of the inner **while** loop works by moving $A[j - 1]$, $A[j - 2]$, $A[j - 3]$, and so on, by one position to the right until the proper position for *key* (which started in position $A[j]$) is found. At that point, the value of *key* is placed into this position

The Correctness of INSERTION-SORT

- The loop invariant components of Insertion Sort:
  - Termination: The outer **for** loop ends when $j > n$, which occurs when $j = n + 1$. Therefore, $j - 1 = n$. Plugging n in for $j - 1$ in the loop invariant, the subarray $A[1 .. n]$ consists of the elements originally in $A[1 .. n]$, but in sorted order. In other words, the entire array is sorted.

Pseudocode Conventions (1)

- There is no single, "proper" way to write pseudocode. The authors of this text have their own style, and other texts may use another style
- Once a style is established, however, it is usually a good idea to stick with it.
- The complete list of the text's pseudocode conventions can be found on pp. 20-22, but here are some of the highlights:

Pseudocode Conventions (2)

- Indentation (not braces) indicate block structure
- Looping constructs are like C++/Java
- The looping variable is still available after the loop terminates (the loop variable is not local to the loop)
- Comments are indicated with //
- Variables are local, unless otherwise specified
- The dot notation is used between objects and their attributes (**object.attribute**)

Pseudocode Conventions (3)

- Objects are treated like references. If x and y are references, then $x = y$ makes them both refer to the same object; it does not copy one object to the other
- Parameters are passed by value, as in Java/C
- Objects are passed by reference (pointer). The called function can change the object's attributes
- The Boolean operators "and" and "or" are short-circuiting
  - Finding "false" in an "and" or "true" in an "or" stops evaluation of the "and" or the "or" without evaluating the other operands

Analyzing Algorithms

- We want to predict the resources that the algorithm requires.
  - Usually, the resource we're most concerned about is running time
- In order to predict resource requirements, we need a computational model

Random-Access Machine (RAM) Model

- Instructions are executed sequentially (no concurrent operations)
- It's too tedious to define each of the instructions and their associated time costs
- Instead, we recognize that we'll use instructions commonly found in real computers:
  - Arithmetic: +, -, *, /, %, floor, ceiling, left/right shift
  - Data movement: load, store, copy
  - Control: conditional / unconditional branch, call, return
- Each of these instructions takes a constant amount of time

Random-Access Machine (RAM) Model

- The RAM model uses integer and floating-point data types
- We won't worry about precision, although it is crucial in certain numerical applications
- There is a limit on the word size: when working with inputs of size n, assume that integers are represented by $c \lg n$ bits for some constant $c \geq 1$
  - $\lg n$ is shorthand for $\log_2 n$
  - c is a constant, so the word size can't grow arbitrarily
  - $c \geq 1$, so we can hold the value of n, and we can index the individual elements

Algorithm Run Times

- How do we analyze an algorithm's run time?
- The time taken by an algorithm depends on the input
  - Sorting 1000 numbers takes longer than sorting 3
  - A given sorting algorithm may even take differing amounts of time on two inputs of the same size
    - For example, we'll see that insertion sort takes less time to sort n elements when they are already sorted than it does when they are in reverse sorted order

"Size" of the Input

- Depends on the problem being studied.
  - Usually, the number of items in the input. Like the size n of the array being sorted.
  - But could be something else. If multiplying two integers, could be the total number of bits in the two integers.
  - Could be described by more than one number. For example, graph algorithm running times are usually expressed in terms of the number of vertices and the number of edges in the input graph.

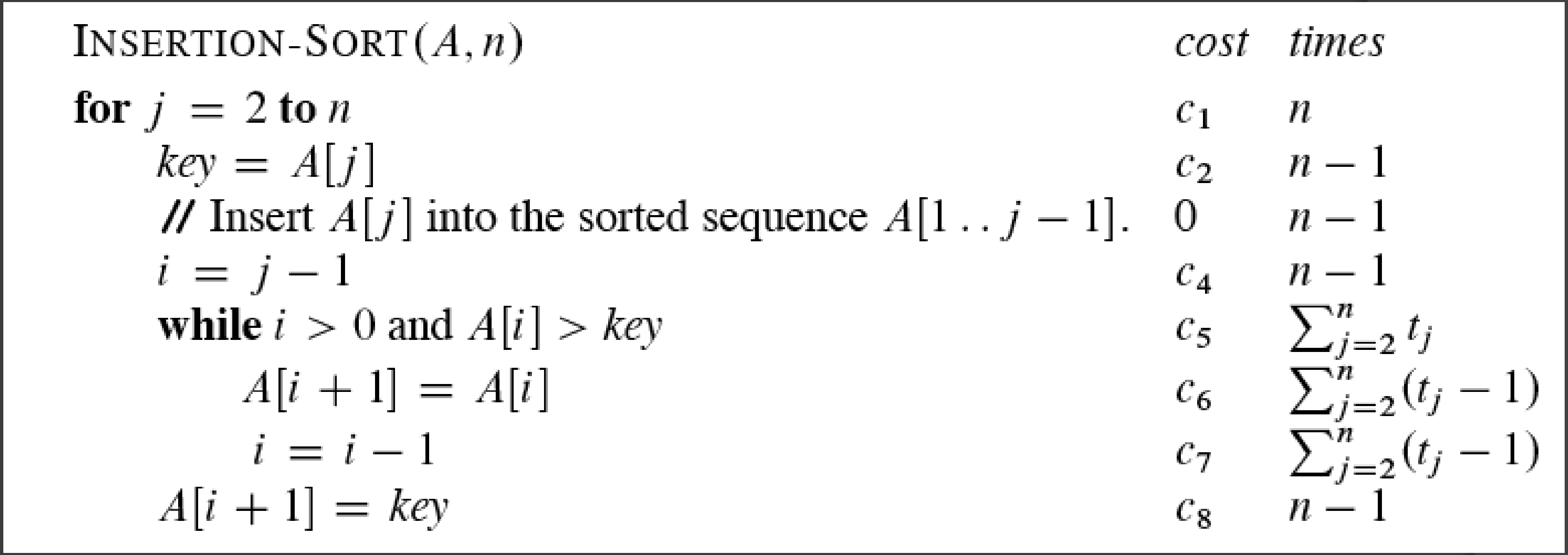
Running Time (1)

- On a particular input, it is the number of primitive operations (steps) executed.
  - We want to define steps to be machine-independent.
  - Figure that each line of pseudocode requires a constant amount of time.
  - One line may take a different amount of time than another, but each execution of line *i* takes the same amount of time c_i .

Running Time (2)

- One line may take a different amount of time than another, but each execution of line *i* takes the same amount of time c_i .
- This is assuming that the line consists only of primitive operations.
  - If the line is a subroutine call, then the actual call takes constant time, but the execution of the subroutine being called might not.
  - If the line specifies operations other than primitive ones, then it might take more than constant time. Example: “sort the points by *x-coordinate*.”

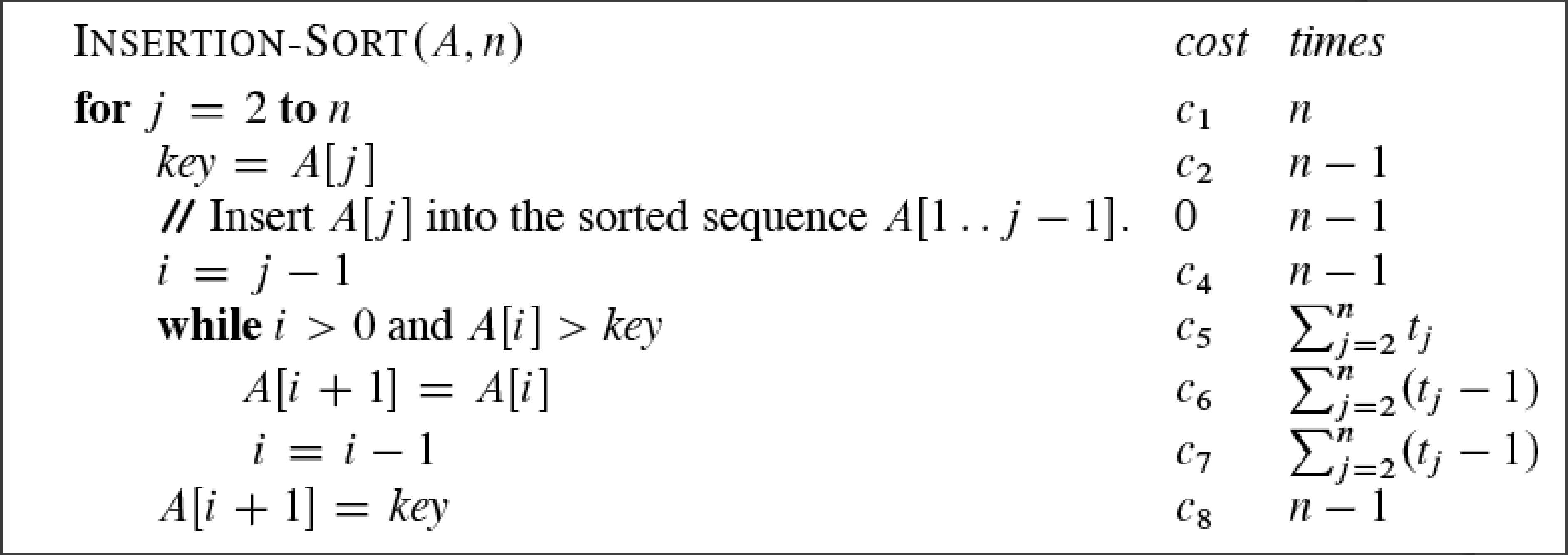
Analysis of INSERTION-SORT



```
INSERTION-SORT($A, n$)                                              cost    times
for $j = 2$ to $n$                                                  $c_1$   $n$
    $key = A[j]$                                                    $c_2$   $n - 1$
    // Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.    0       $n - 1$
    $i = j - 1$                                                     $c_4$   $n - 1$
    while $i > 0$ and $A[i] > key$                                  $c_5$   $\sum_{j=2}^{n} t_j$
        $A[i + 1] = A[i]$                                           $c_6$   $\sum_{j=2}^{n} (t_j - 1)$
        $i = i - 1$                                                 $c_7$   $\sum_{j=2}^{n} (t_j - 1)$
    $A[i + 1] = key$                                                $c_8$   $n - 1$
```

- Assume that the i^{th} line takes time c_i, which is a constant (comment lines take no time)

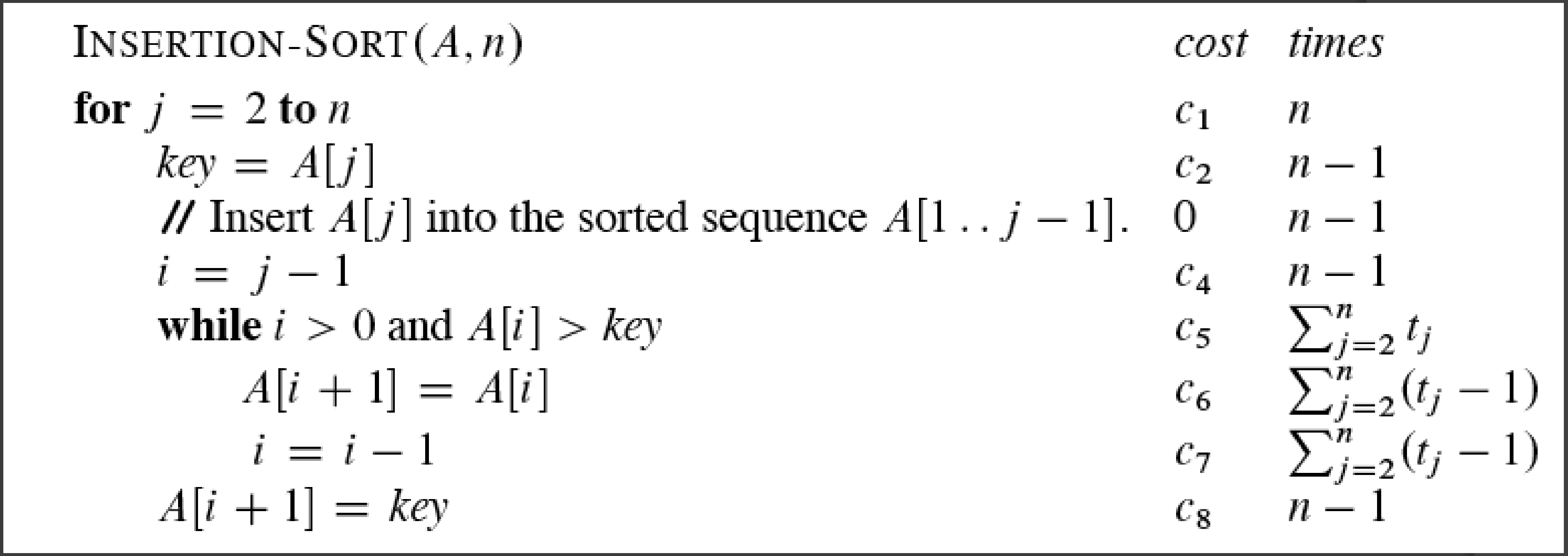
Analysis of INSERTION-SORT



```
INSERTION-SORT($A, n$)                                          cost    times
for $j = 2$ to $n$                                              $c_1$   $n$
    $key = A[j]$                                                $c_2$   $n-1$
    // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.      $0$     $n-1$
    $i = j-1$                                                   $c_4$   $n-1$
    while $i > 0$ and $A[i] > key$                              $c_5$   $\sum_{j=2}^{n} t_j$
        $A[i+1] = A[i]$                                         $c_6$   $\sum_{j=2}^{n} (t_j - 1)$
        $i = i-1$                                               $c_7$   $\sum_{j=2}^{n} (t_j - 1)$
    $A[i+1] = key$                                              $c_8$   $n-1$
```

- For $j = 2, 3, \ldots, n$, let t_j be the number of times that the while loop test is executed for that value of j

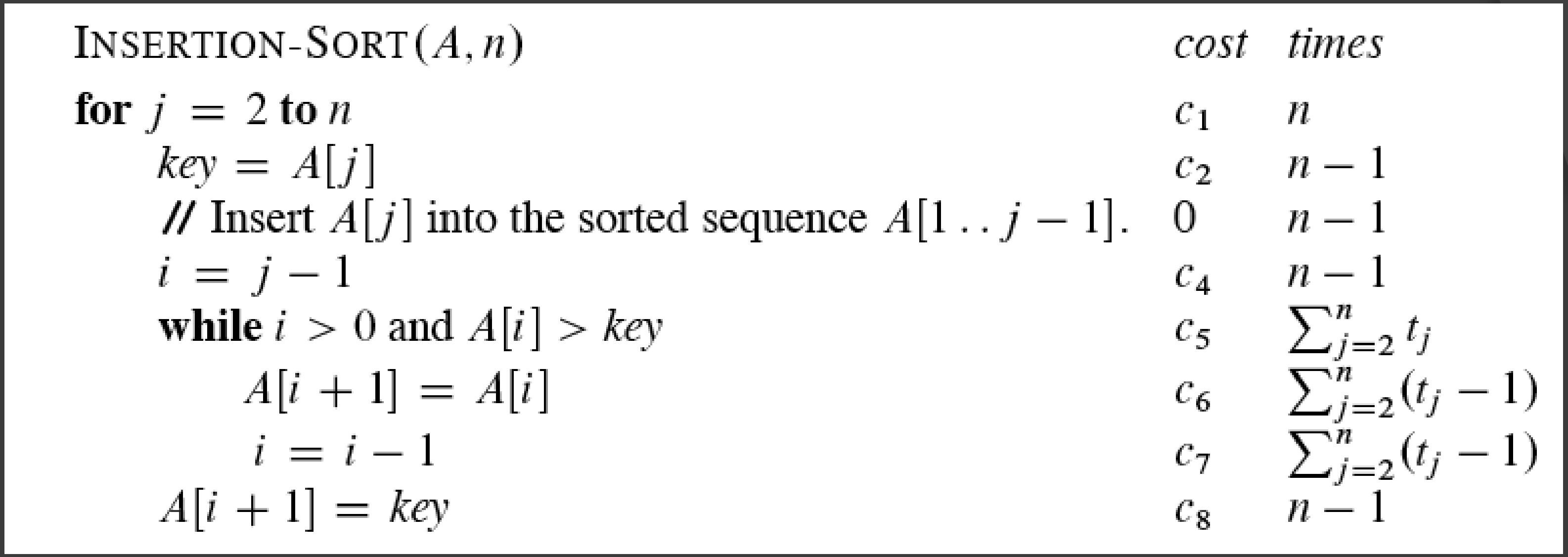
Analysis of INSERTION-SORT



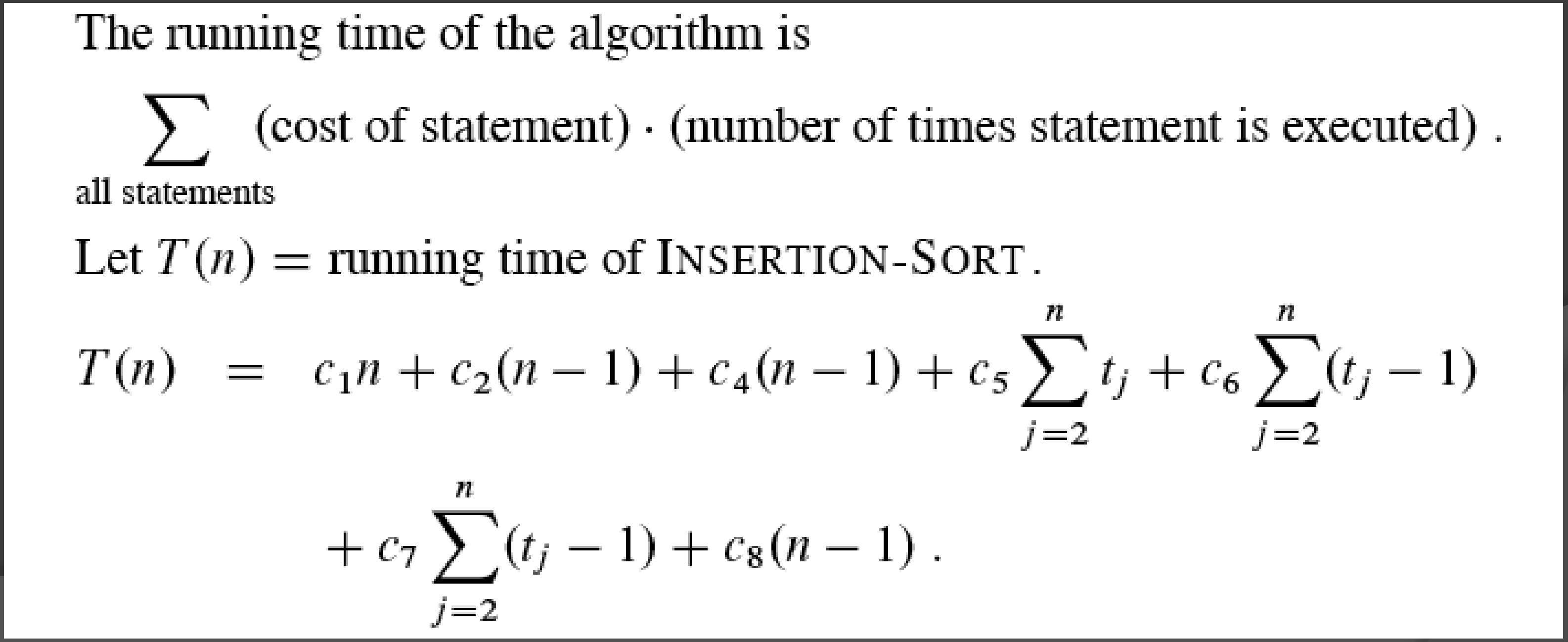
```
INSERTION-SORT(A, n)                                                cost    times
for $j = 2$ to $n$                                                  $c_1$   $n$
    $key = A[j]$                                                    $c_2$   $n - 1$
    // Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.      0       $n - 1$
    $i = j - 1$                                                     $c_4$   $n - 1$
    while $i > 0$ and $A[i] > key$                                  $c_5$   $\sum_{j=2}^{n} t_j$
        $A[i+1] = A[i]$                                             $c_6$   $\sum_{j=2}^{n} (t_j - 1)$
        $i = i - 1$                                                 $c_7$   $\sum_{j=2}^{n} (t_j - 1)$
    $A[i+1] = key$                                                  $c_8$   $n - 1$
```

- Note that when a for or while loop exits in the usual way (due to the test in the loop header), the test is executed one time more than the loop body

Analysis of INSERTION-SORT



```
INSERTION-SORT($A, n$)                                         cost    times
for $j = 2$ to $n$                                             $c_1$   $n$
    $key = A[j]$                                               $c_2$   $n-1$
    // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.     0       $n-1$
    $i = j - 1$                                                $c_4$   $n-1$
    while $i > 0$ and $A[i] > key$                             $c_5$   $\sum_{j=2}^{n} t_j$
        $A[i+1] = A[i]$                                        $c_6$   $\sum_{j=2}^{n} (t_j - 1)$
        $i = i - 1$                                            $c_7$   $\sum_{j=2}^{n} (t_j - 1)$
    $A[i+1] = key$                                             $c_8$   $n-1$
```



The running time of the algorithm is

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed}) .$$

Let $T(n)$ = running time of INSERTION-SORT.

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

Analysis of INSERTION-SORT

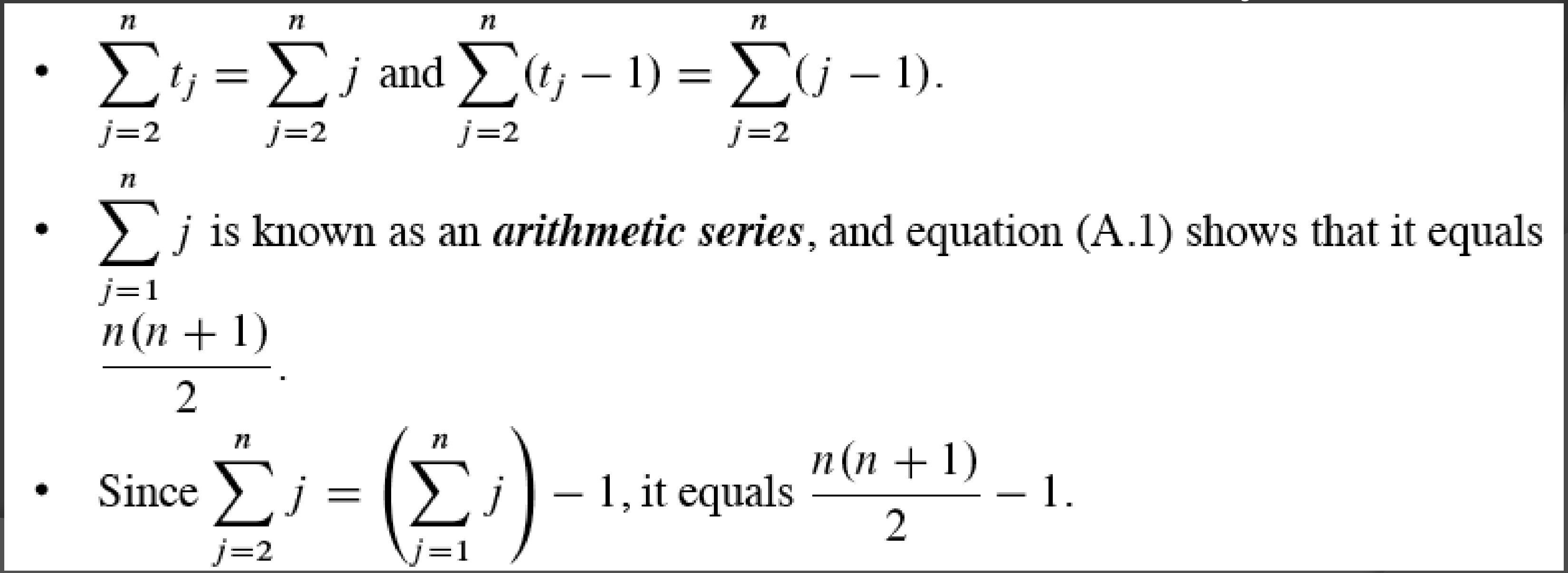
- The running time depends on the values of t_j These vary according to the input.
- ***Best case:*** the array is already sorted.
  - We always find that $A[i] \le key$ upon the first time the **while** loop test is run (when $i = j - 1$).
  - All t_j are 1
  - Running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

  - We can express $T(n)$ as $an+b$ for some constants a and b (that depend on the statement costs c_i) $\Rightarrow T(n)$ is a *linear function* of n.

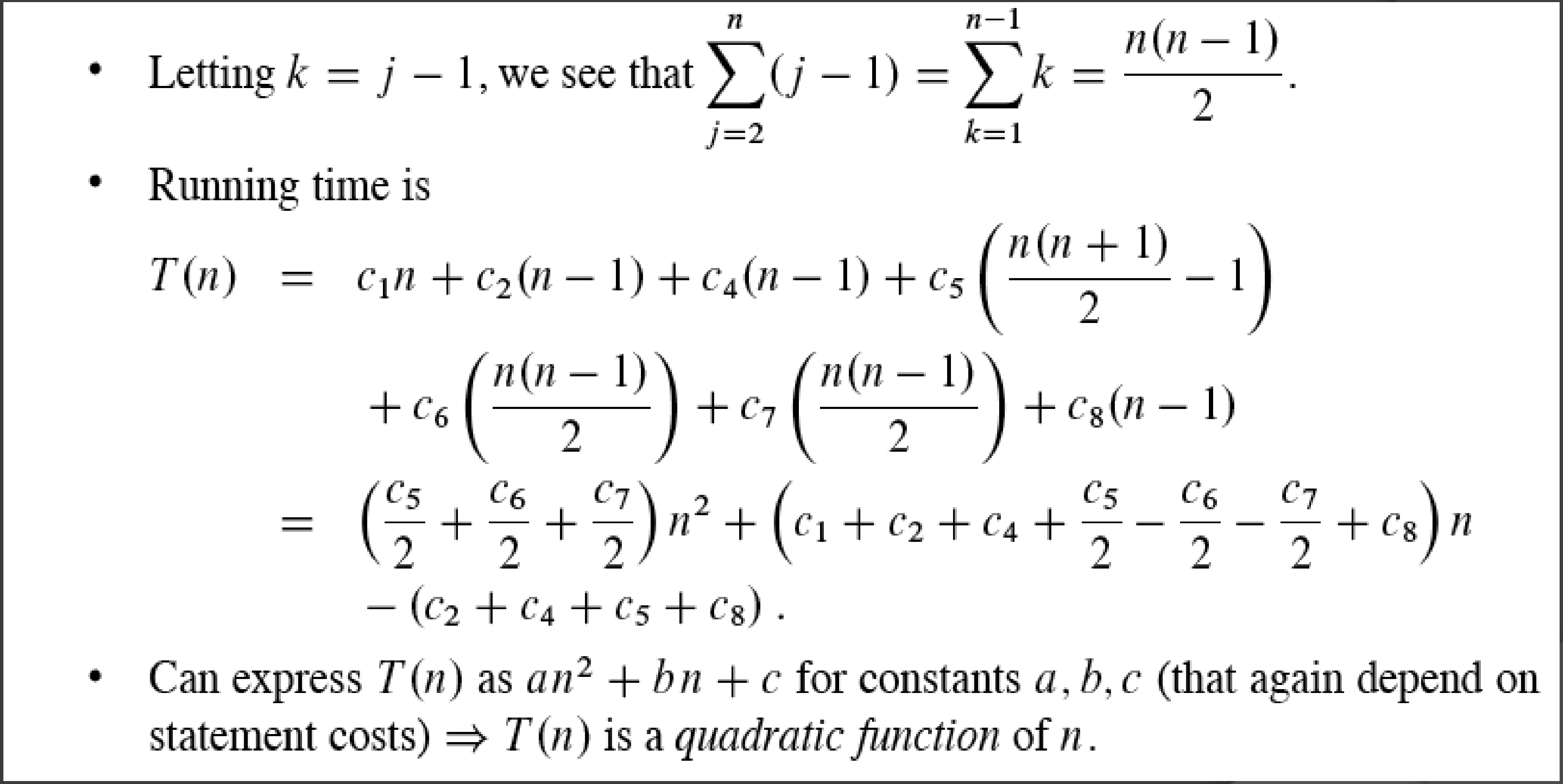
Analysis of INSERTION-SORT

- ***Worst case:*** the array is reverse sorted.
  - We always find that $A[i] > key$ in the **while** loop test
  - We have to compare key with all elements to the left of the j^{th} position) ⇒ compare with $j - 1$ elements
  - Since the **while** loop exits because i reaches 0, there's one additional test after the $j - 1$ tests ⇒ $t_j = j$



- $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ and $\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1)$.
- $\sum_{j=1}^{n} j$ is known as an ***arithmetic series***, and equation (A.1) shows that it equals $\frac{n(n+1)}{2}$.
- Since $\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\frac{n(n+1)}{2} - 1$.

Analysis of INSERTION-SORT



- Letting $k = j - 1$, we see that $\sum_{j=2}^{n}(j-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$.
- Running time is

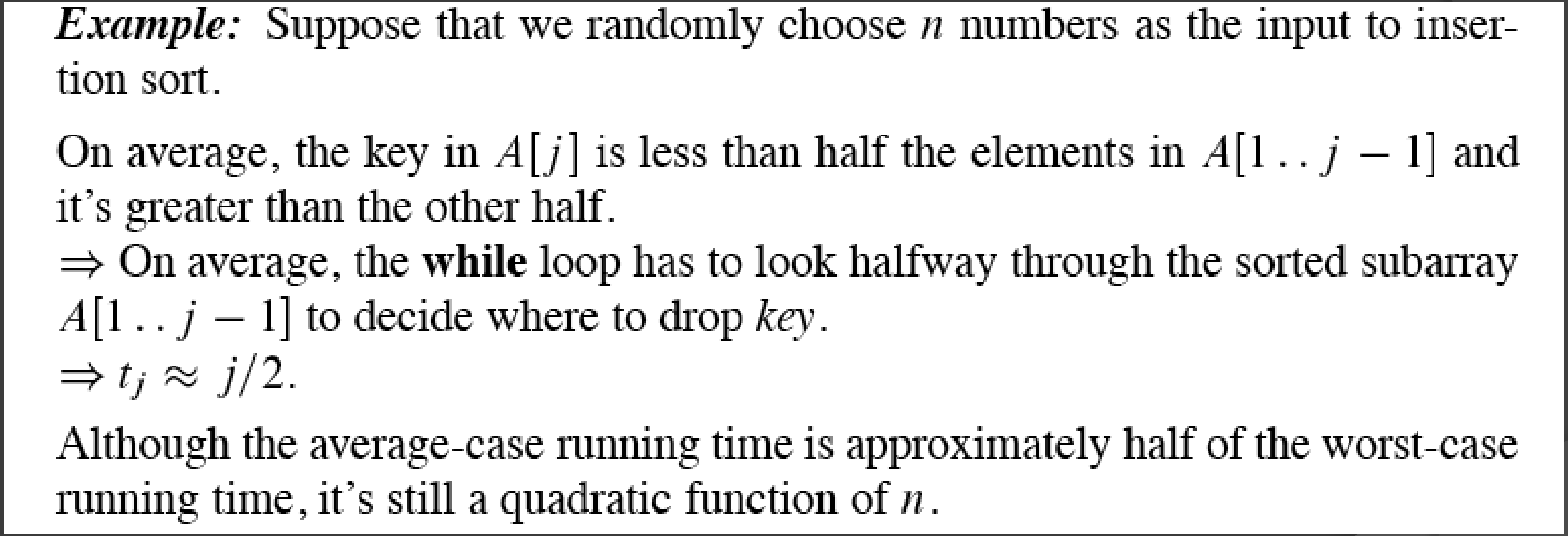
$$\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\
&\quad - (c_2 + c_4 + c_5 + c_8) .
\end{aligned}$$

- Can express $T(n)$ as $an^2 + bn + c$ for constants a, b, c (that again depend on statement costs) $\Rightarrow T(n)$ is a *quadratic function* of n.

Worst Case and Average Case Analysis

- We usually concentrate on finding the *worst-case running time:* the longest running time for any input of size n. Why look at the worst case?
  - The worst-case running time gives a guaranteed upper bound on the running time for any input.
  - For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.
  - Why not analyze the *average case*? Because it's often about as bad as the worst case.

Worst Case and Average Case Analysis

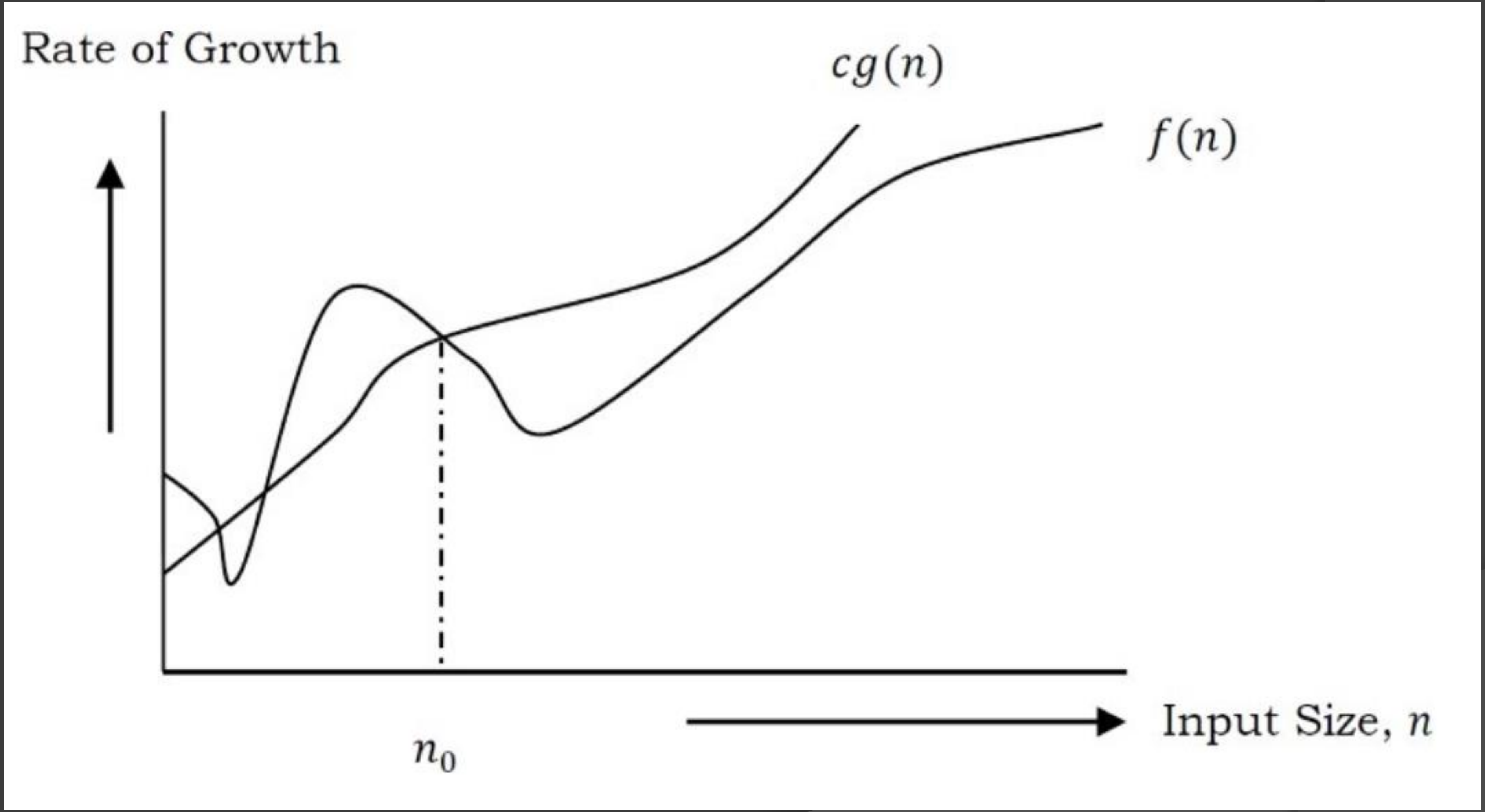


Example: Suppose that we randomly choose n numbers as the input to insertion sort.

On average, the key in $A[j]$ is less than half the elements in $A[1\,..\,j-1]$ and it's greater than the other half.
$\Rightarrow$ On average, the **while** loop has to look halfway through the sorted subarray $A[1\,..\,j-1]$ to decide where to drop *key*.
$\Rightarrow t_j \approx j/2$.

Although the average-case running time is approximately half of the worst-case running time, it's still a quadratic function of n.

Order of Growth



Order of Growth

- O-notation is defined as
  - $O(g(n)) = \{ f(n)$: there exists positive constants c and n_o such that $0 <= f(n) <= cg(n)$ for all $n > n_0 \}$.
  - g(n) is an asymptotic tight upper bound for f(n).
  - O(g(n)) is the set of functions with smaller or the same order of growth as g(n).
    - $O(n^2)$ includes O(1), O(n), O(nlogn), etc.
    - We analyze the algorithms at larger values of n only.
    - What this means is, below n_0 we do not care about the rate of growth.

Order of Growth

- No Uniqueness.
- There no unique set of values for n_o and c in proving the asymptotic bounds. Consider, for instance, $f(n) = n^2 + 1$.
  - $n^2 + 1 <= 2n^2 => n^2 >= 1$; f(n) = O(n2) for all n >= 1, $n_o = 1$ and c = 2 is a solution.

Order of Growth

- Another abstraction to easy analysis and focus on the important features of the performance.
- We look only at the leading term of the formula for the running time.
- Drop lower-order terms.
- Ignore the constant coefficient in the leading term.

Order of Growth

- *Ex*: For insertion sort, we already "abstracted away" the actual statement costs to conclude that the worst-case running time is $an^2 + bn + c$
  - Drop lower-order terms $\Rightarrow an^2$
  - Ignore constant coefficient $\Rightarrow n^2$
  - But we cannot say that the worst-case running time $T(n)$ <u>equals</u> n^2 – it <u>*grows like*</u> n^2; but it doesn't <u>*equal*</u> n^2
  - We say the running time is $\Theta(n^2)$ to capture the notion that the *order of growth* is n^2
  - We usually consider one algorithm to be "more efficient" than another if its worst-case running time has a smaller order of growth

Designing Algorithms

- There are many ways to design algorithms.
  - For example, insertion sort is *incremental:* having sorted $A[1 \mathinner{..} j-1]$, place $A[j]$ correctly, so that $A[1 \mathinner{..} j]$ is sorted.
- *Divide and conquer*
  - Another common approach
    - *Divide* the problem into a number of subproblems that are smaller instances of the same problem
    - *Conquer* the subproblems by solving them recursively
    - *Combine* the subproblem solutions to give a solution to the original problem.

Binary Search

- A search algorithm based on divide and conquer
- Its worst-case running time has a lower order of growth than linear search.
- Because we are dealing with subproblems, each subproblem is stated as searching a subarray $A[p\,..\,r]$
- Initially, $p = 1$ and $r = n$, but these values change as we recurse through subproblems.

Binary Search

Search a sorted array for a given item, x

- If x == middle array element, return true
- Else, BinarySearch lower (x<mid) or upper (x>mid) sub-array
- 1 subproblem, half as large

```
BinarySearch(A, x)
   if (A.size == 1) return (x == A[0])
   mid = A.size / 2
   if (x == A[mid]) return true
   else if (x < A[mid]) return BinarySearch( A_LowerHalf, x)
   else if (x > A[mid]) return BinarySearch( A_UpperHalf, x)
```

Equation:

Base case

Subproblem size

$$T(1) \le b$$

$$T(n) \le 1\,T(n/2) + c \quad \text{for } n>1$$

Work dividing and combining

# subproblems

Binary Search Solution

Equation:

$T(1) \leq b$

$T(n) \leq T(n/2) + c \quad \text{for } n>1$

Solution: (finding the **closed form solution**)

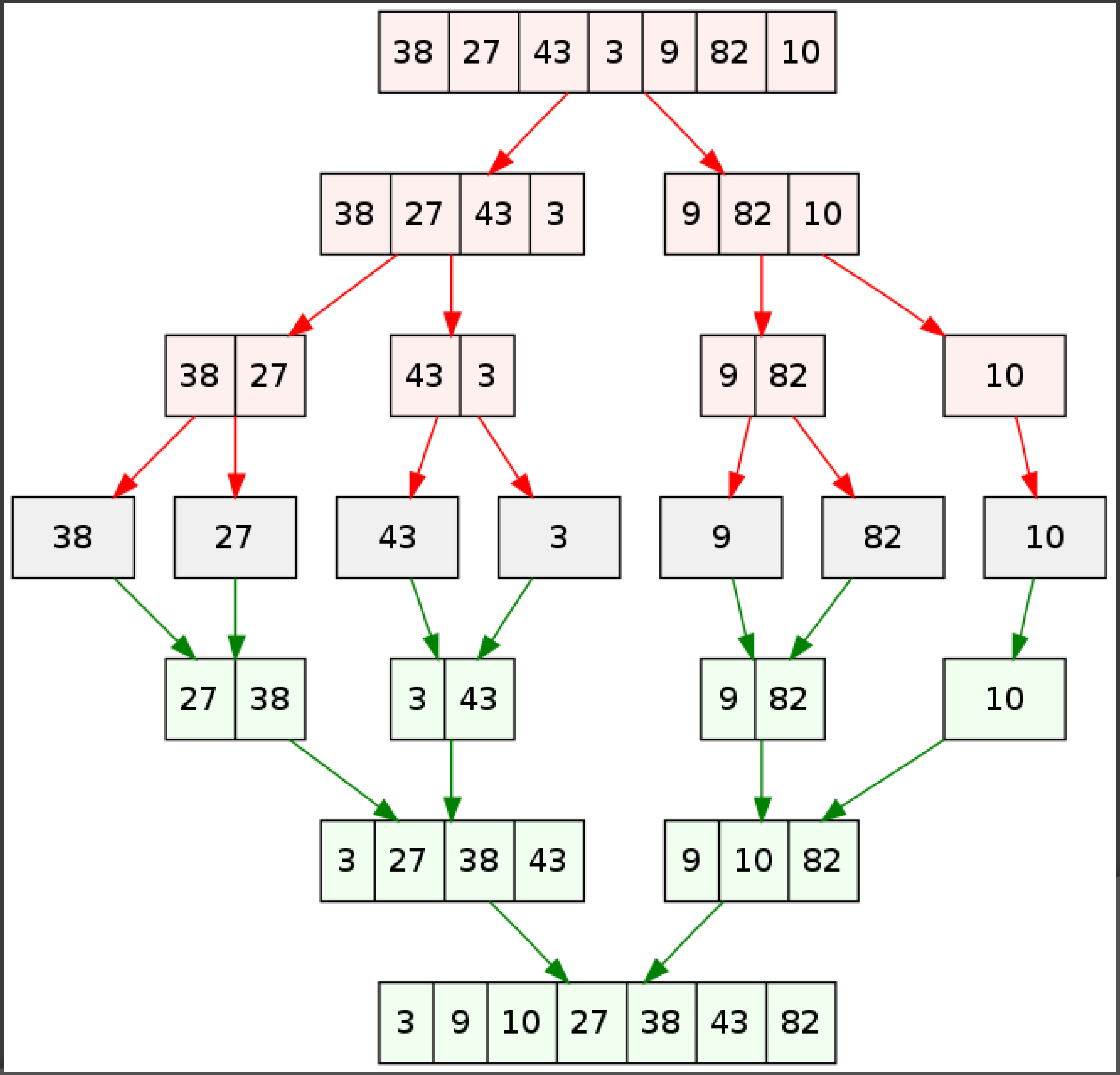
$$\left.\begin{aligned}
T(n) &\leq T(n/2) + c \\
&\leq T(n/4) + c + c \\
&\leq T(n/8) + c + c + c \\
&\leq T(n/2^k) + kc \\
&\leq T(1) + c \log n \quad \text{where } k = \log n \\
&\leq b + c \log n \quad = \quad O(\log n)
\end{aligned}\right\}$$

Iterative substitution method

Merge Sort

- A sorting algorithm based on divide and conquer
- Its worst-case running time has a lower order of growth than insertion sort.
- Because we are dealing with subproblems, each subproblem is stated as sorting a subarray $A[p \mathinner{..} r]$
- Initially, $p = 1$ and $r = n$, but these values change as we recurse through subproblems.

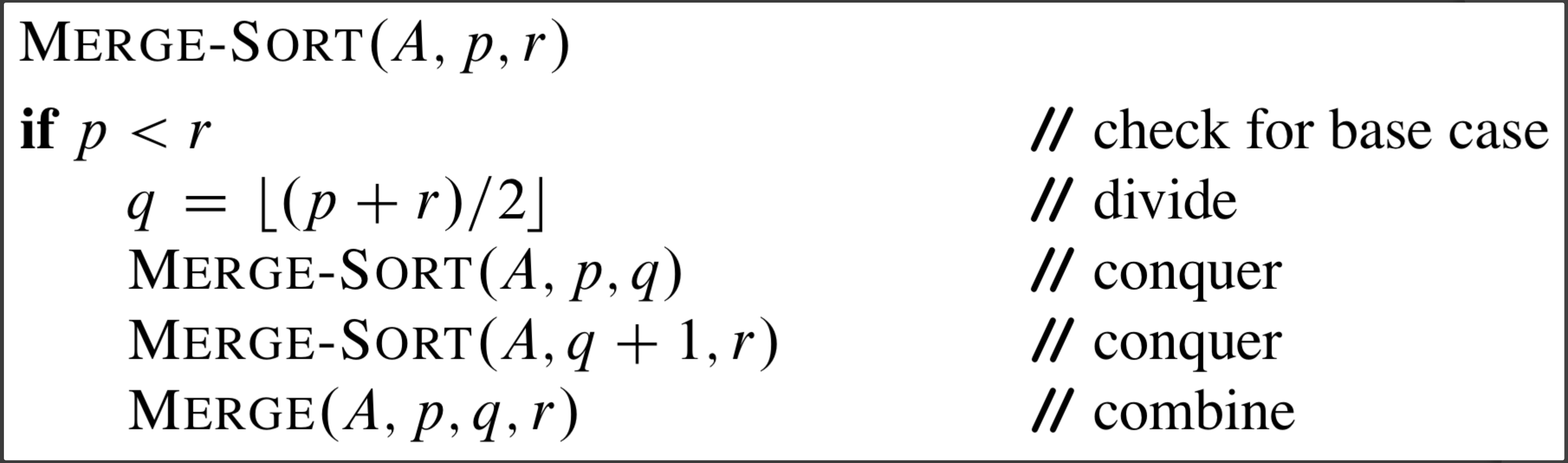
Merge Sort


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Merge Sort

- To sort $A[p \mathinner{..} r]$:
  - _Divide_ by splitting into two subarrays:
    $A[p \mathinner{..} q]$ and $A[q + 1 \mathinner{..} r]$, where q is the halfway point of $A[p \mathinner{..} r]$
  - _Conquer_ by recursively sorting the two subarrays:
    $A[p \mathinner{..} q]$ and $A[q + 1 \mathinner{..} r]$
  - _Combine_ by merging the two sorted subarrays:
    $A[p \mathinner{..} q]$ and $A[q + 1 \mathinner{..} r]$
    to produce a single sorted subarray $A[p \mathinner{..} r]$
    - For this step, we'll define a procedure $\text{MERGE}(A, p, q, r)$
    - The recursion bottoms out when the subarray has just 1 element, so that it's trivially sorted.

Pseudocode – MERGE-SORT

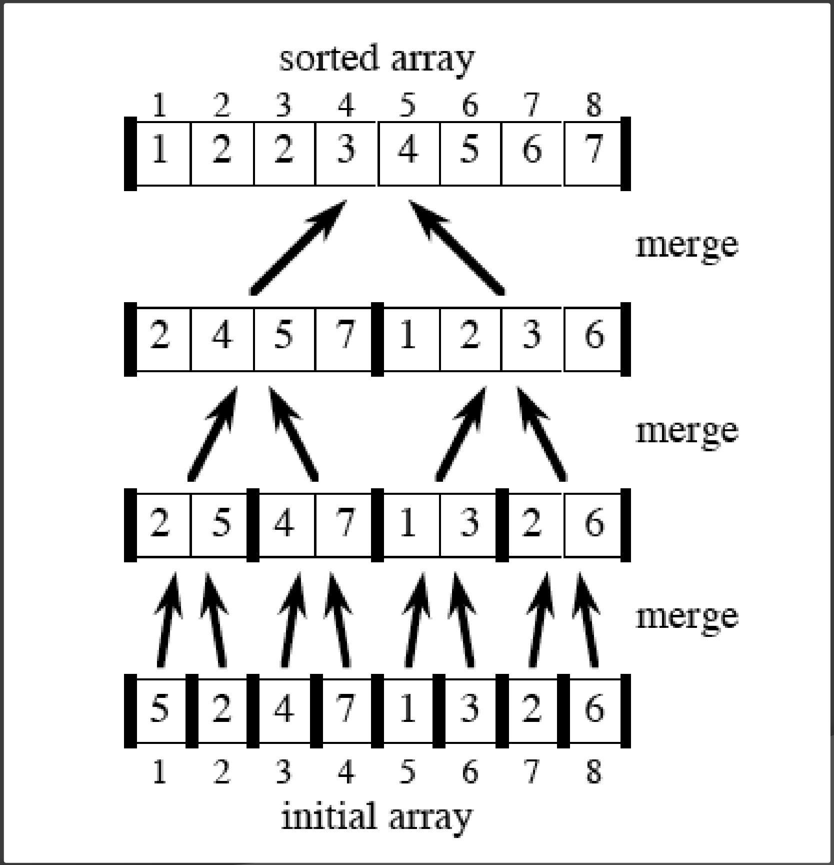


```
MERGE-SORT($A, p, r$)
if $p < r$                              // check for base case
    $q = \lfloor (p + r)/2 \rfloor$            // divide
    MERGE-SORT($A, p, q$)               // conquer
    MERGE-SORT($A, q + 1, r$)           // conquer
    MERGE($A, p, q, r$)                 // combine
```

Initial call: MERGE-SORT(A, 1, n)

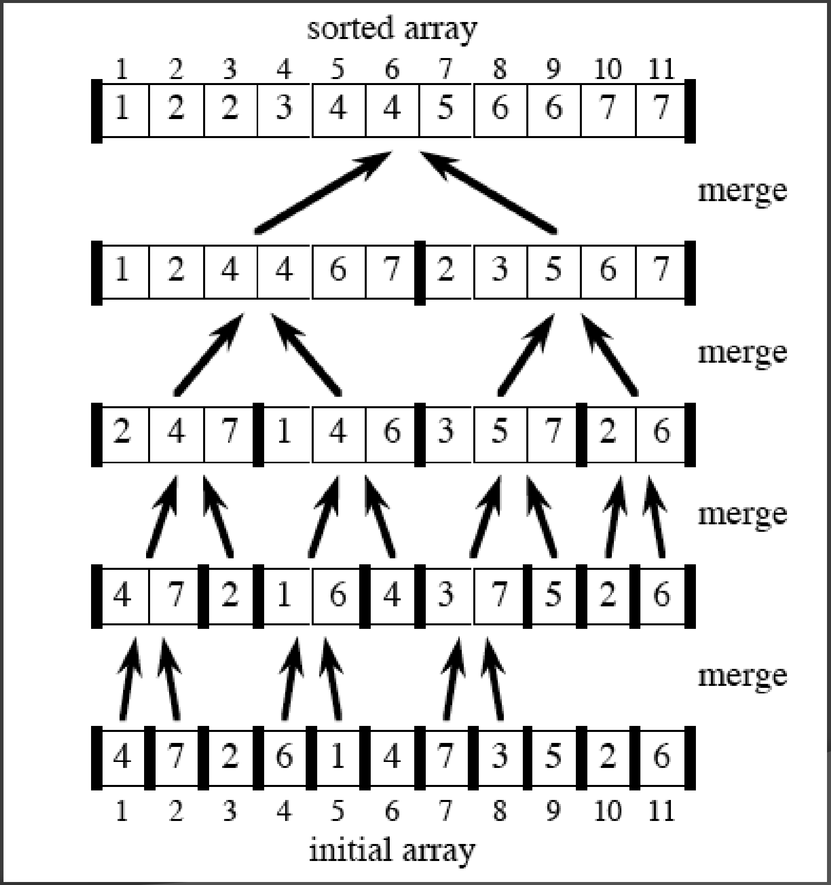
MERGE-SORT – Example

Example: Bottom-up view for $n = 8$:



MERGE-SORT – Example

The bottom-up view is pretty straightforward when n is a power of 2. Here's an example for $n = 11$:



Merging (1)

- What remains is the MERGE procedure.
- *Input*: Array A and indices p, q, r such that
  - $p \leq q < r$
  - Subarray $A[p \,..\, q]$ is sorted and subarray $A[q+1 \,..\, r]$ is sorted. By the restrictions on p, q, and r, neither subarray is empty
- *Output*: The two subarrays are merged into a single sorted subarray in $A[p \,..\, r]$
- We implement it so that it takes $\Theta(n)$ time, where $n = r - p + 1$ = the number of elements being merged. We'll examine the linear MERGE shortly

Merging (2)

- What is n?
  - Until now, n has stood for the original size of the problem.
  - Now, however, we're using it for the size of a subproblem.
  - We will use this technique when we analyze recursive problems.
  - Although we may denote the original problem size by n, in general n will be the size of a given subproblem

Pseudocode for MERGE

```
MERGE($A$, $p$, $q$, $r$)
  $n_1 = q - p + 1$
  $n_2 = r - q$
  let $L[1 .. n_1 + 1]$ and
      $R[1 .. n_2 + 1]$ be new
      arrays
  for $i = 1$ to $n_1$
      $L[i] = A[p + i - 1]$
  for $j = 1$ to $n_2$
      $R[j] = A[q + j]$
  $L[n_1 + 1] = \infty$
  $R[n_2 + 1] = \infty$
  $i = 1$
  $j = 1$
  for $k = p$ to $r$
      if $L[i] \leq R[j]$
          $A[k] = L[i]$
          $i = i + 1$
      else
          $A[k] = R[j]$
          $j = j + 1$
```

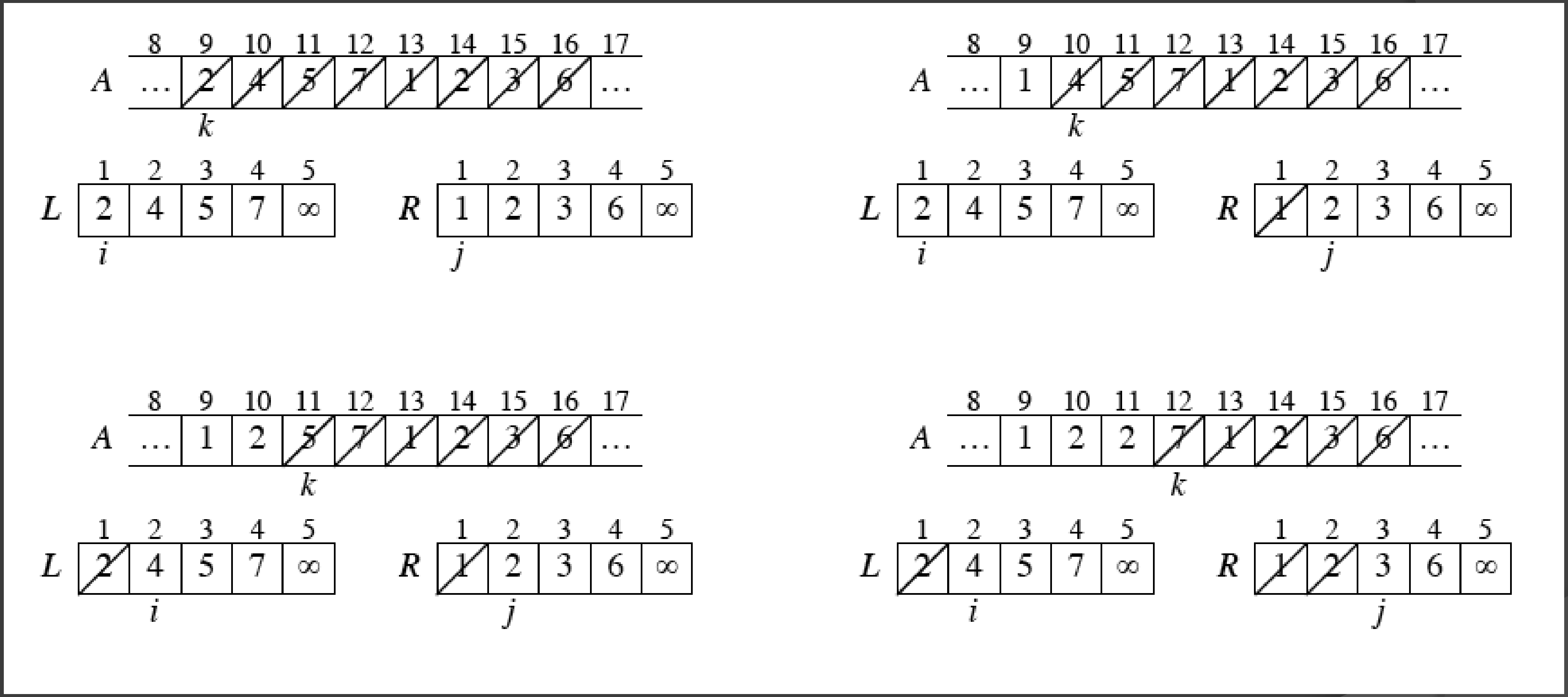
The Idea Behind Linear Time Merging

- Think of two piles of playing cards
  - Each pile is sorted and placed face-up on a table with the smallest cards on top
  - We will merge these into a single sorted pile, face-down on the table
  - Repeat this basic step until one pile is empty:
    - Choose the smaller of the two top cards
    - Remove it from its pile (exposing a new top card)
    - Place the chosen card face-down on the output pile
    - Once one pile is empty, take the remaining pile and place it face-down on the output pile
  - Each basic step takes constant time (look at 2 cards)
  - There are $\leq n$ basic steps (started with n cards) $\Rightarrow \Theta(n)$

The Idea Behind Linear Time Merging (2)

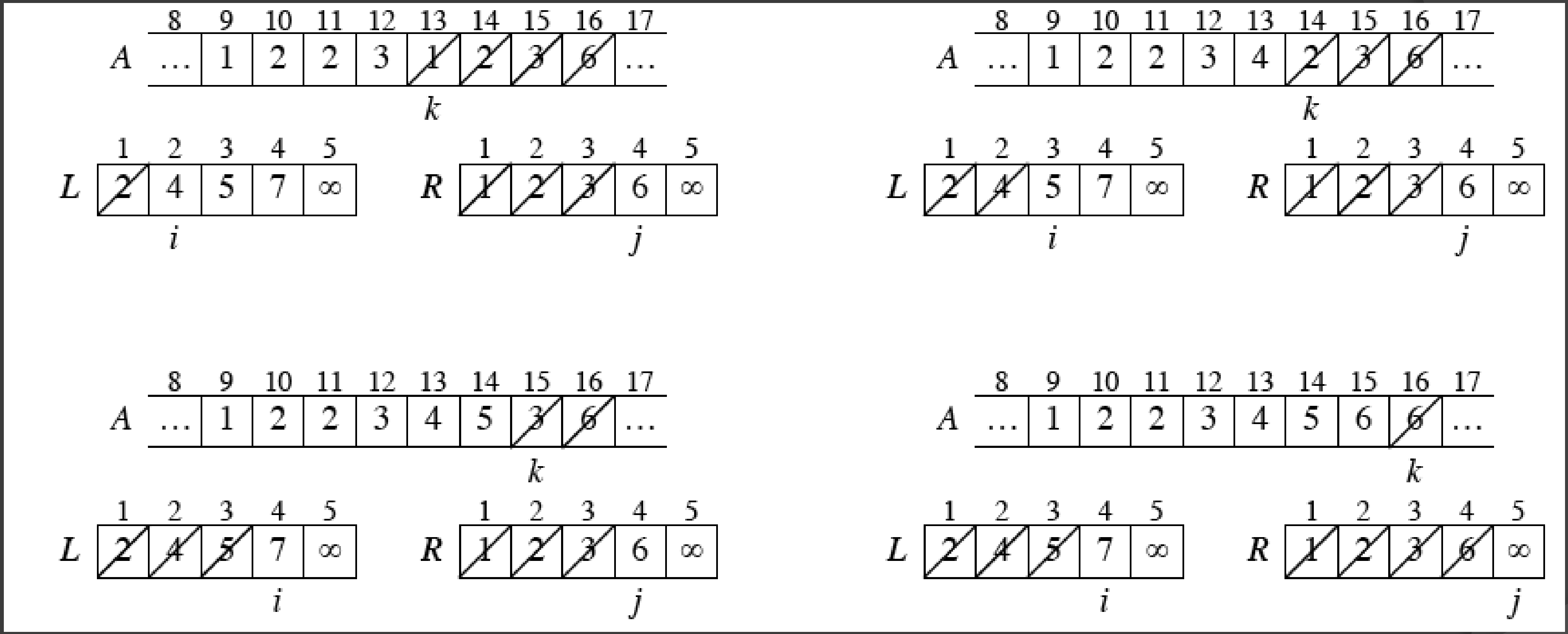
- We don't actually need to check whether a pile is empty before each basic step
  - Add to the bottom of each pile a *sentinel* card
  - It contains a special value (∞) to simplify the code
  - The sentinel value is guaranteed to lose to any other value. The only way it doesn't lose is when *both* piles are showing the sentinel value
  - But by then, all non-sentinel cards have been handled
  - We know in advance there are exactly $r - p + 1$ non-sentinel cards, after which we stop
  - Rather than counting $r - p + 1$ individual steps, just fill the output array from index p through index r

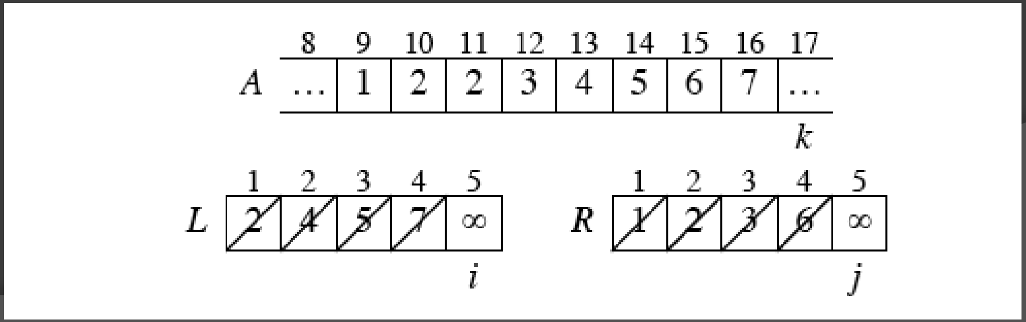
Example: MERGE(9, 12, 16)



Concludes on next slide…

Example: MERGE(9, 12, 16)





Running Time for MERGE

```
MERGE($A, p, q, r$)
    $n_1 = q - p + 1$
    $n_2 = r - q$
    let $L[1 .. n_1 + 1]$ and
        $R[1 .. n_2 + 1]$ be new
        arrays
    for $i = 1$ to $n_1$
        $L[i] = A[p + i - 1]$
    for $j = 1$ to $n_2$
        $L[j] = A[q + j]$
    $L[n_1 + 1] = \infty$
    $R[n_2 + 1] = \infty$
    $i = 1$
    $j = 1$
    for $k = p$ to $r$
        if $L[i] \leq R[j]$
            $A[k] = L[i]$
            $i = i + 1$
        else
            $A[k] = R[j]$
            $j = j + 1$
```

Constant Time

$\Theta(n_1)$

$\Theta(n_2)$

$\Theta(r - p + 1)$

But $n_1 + n_2 = n$, and $(r - p + 1) = n$, so $\Theta(n_1 + n_2) + \Theta(r - p + 1) \Rightarrow \Theta(n)$

Analyzing Divide-and-Conquer Algorithms

- We use a *recurrence equation* *(more commonly, a recurrence)* to describe the running time of a divide-and-conquer algorithm.
- Let $T(n)$ =running time on a problem of size n
  - If the problem size is small enough (say, $n \leq c$ for some constant c), we have a base case. The brute-force solution takes constant time: $\Theta(1)$
  - Otherwise, suppose that we divide into a subproblems, each $1/b$ the size of the original.
    - Merge Sort divided the list into two pieces ($a = 2$) of equal size (i.e., halves, so $b = 2$), so for Merge Sort, $a = b = 2$)
    - Continued…

Analyzing Divide-and-Conquer Algorithms

- Let the time to divide a size-n problem be $D(n)$
- We have a subproblems to solve, each of size n/b, so
- Each subproblem takes $T(n/b)$ time to solve, so
- We spend $aT(n/b)$ time solving subproblems
- Let the time to combine solutions be $C(n)$
- We get the recurrence

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

Analyzing MERGE-SORT

- For simplicity, assume that n is a power of 2
    - Each divide step yields two subproblems, both of size exactly $n/2$
- The base case occurs when $n = 1$
- When $n \geq 2$, time for merge sort steps:
    - *Divide*: Just compute q as the average of p and $r \Rightarrow D(n) = \Theta(1)$
    - *Conquer*: Recursively solve 2 subproblems, each of size $n/2 \Rightarrow 2T(n/2)$
    - *Combine*: MERGE on an n-element subarray takes $\Theta(n)$ time $\Rightarrow C(n) = \Theta(n)$

Analyzing MERGE-SORT (2)

- Since $D(n) = \Theta(1)$ and $C(n) = \Theta(n)$, summed together they give a function that is linear in n: $\Theta(n) \Rightarrow$ recurrence for merge sort running time is

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

Solving the MERGE-SORT Recurrence

- By the master theorem (which we'll get to in Chapter 4), we can show that this recurrence has the solution $T(n) = \Theta(n \lg n)$
- Compared to INSERTION-SORT (with its worst-case time of $\Theta(n^2)$), MERGE-SORT is faster
  - Trading a factor of n for a factor of $\lg n$ is a good deal
- On *small* inputs, INSERTION-SORT may be faster
  - But for large enough inputs, MERGE-SORT will always be faster, because its running time grows more slowly than INSERTION-SORT's

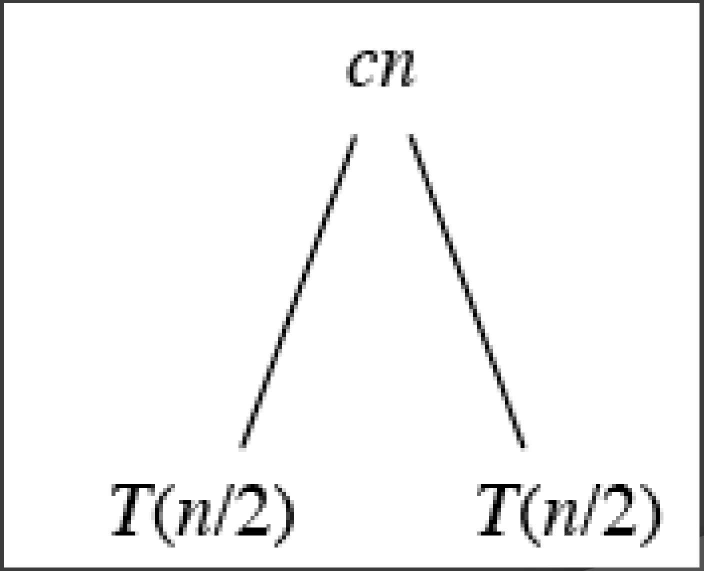
Solving the MERGE-SORT Recurrence

- We can understand how to solve the merge-sort recurrence without the master theorem
- Let c be a constant that describes the running time for the base case and also is the time per array element for the divide and conquer steps
- We rewrite the recurrence as

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

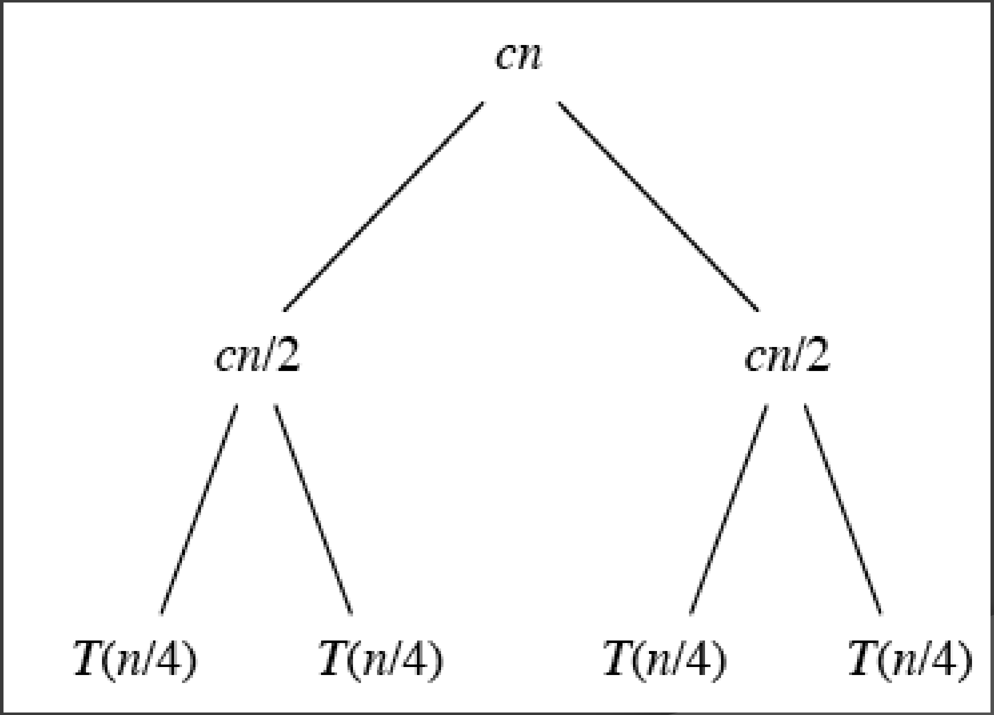
Solving the MERGE-SORT Recurrence

- Draw a *recursion tree*, which shows successive expansions of the recurrence.
- For the original problem, we have a cost of cn, plus the two subproblems, each costing $T(n/2)$:



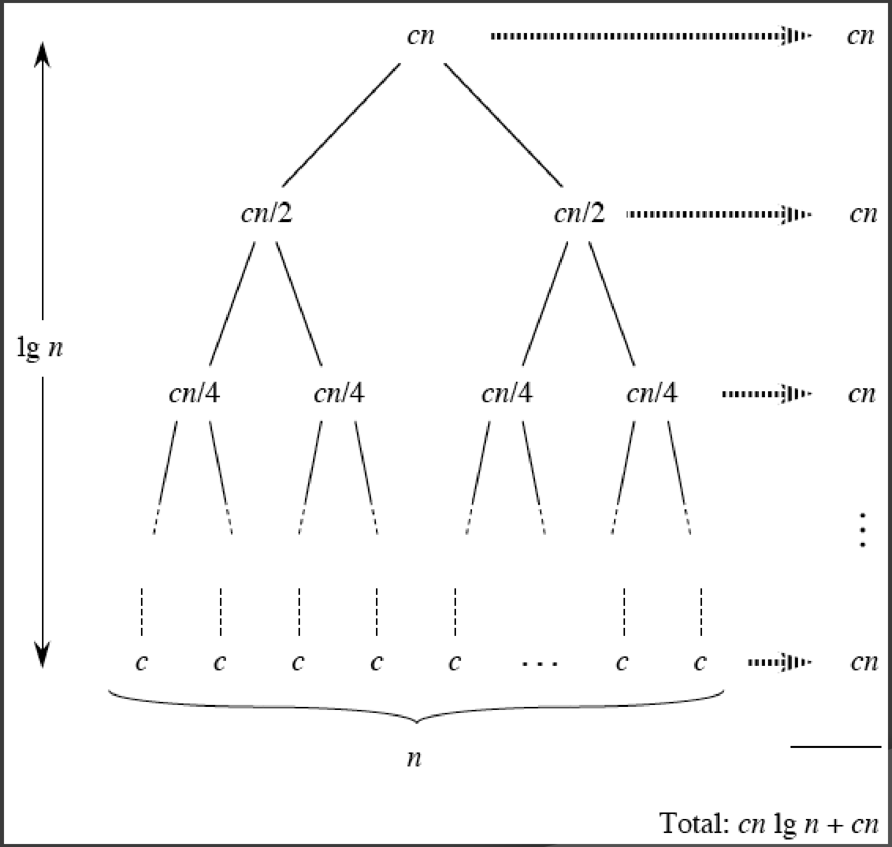
Solving the MERGE-SORT Recurrence

- For each of the size-$n/2$ subproblems, we have a cost of $cn/2$, plus two subproblems, each costing $T(n/4)$:



Solving the MERGE-SORT Recurrence

- Continue expanding until the problem sizes get down to 1:



Solving the MERGE-SORT Recurrence

- Each level has cost cn
  - The top level has cost cn
  - The next level down has 2 subproblems, each contributing cost $cn/2$
  - …
- There are $\lg n + 1$ levels (height is $\lg n$)
  - Use induction
  - Base case: $n = 1 \Rightarrow$ 1 level, and $\lg 1 + 1 = 0 + 1 = 1$
  - Inductive hypothesis is that a tree for a problem size of 2^i has $\lg 2^i + 1 = i + 1$ levels

Solving the MERGE-SORT Recurrence

- Because we assume that the problem size is a power of 2, the next problem size up after 2^i is 2^{i+1}
- A tree for a problem size of 2^{i+1} has has one more level than the size-2^i tree $\Rightarrow i + 2$ levels
- Since $\lg 2^{i+1} + 1 = i + 2$, we're done with the inductive argument
- The total cost is the sum of the costs at each level
- We have $\lg n + 1$ levels, each costing $cn \Rightarrow$ total cost is $cn \lg n + cn$
- Ignore low-order term of cn and constant coefficient c
- $cn \lg n + cn \Rightarrow \Theta(n \lg n)$

End of Chapter 2

Questions?