CS4349
Ch 1: Introduction

Algorithms

- **Algorithm:** a well-defined computational procedure that takes some value, or a set of values, as input and produces some value, or a set of values, as output.

- **Data Structure:** a way to store and organize data to facilitate access and modifications.

- **Instance of a problem:** consists of the input - satisfying the constraints – needed to compute a solution to the problem.

Types of Problems Solved by Algorithms

- Analyzing the human DNA to determine the sequences of the 3 billion pairs, to store this information in databases, and to develop tools for data analysis.
- In Internet, finding good routes for data transfer, using a search engine to retrieve pages on which particular information resides.
- Integrating public-key cryptography and digital signatures, which are based on numerical algorithms and number theory, to achieve secure e-commerce.
- Finding out ways to allocate scarce resources most effectively (linear programming).
- etc.

Example: The sorting problem

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

Output: a permutation of the input such that $\langle a_{i1} \leq \ldots \leq a_{in} \rangle$

- The input is typically stored in arrays
- There are several solutions to this problem
  - insertion sort, merge sort, etc.

Efficiency- An Example

- Comparing a faster computer (computer A) running insertion sort against a slower computer (computer B) running merge sort.
- To sort n items:
  - Insertion sort: Takes time c_1n^2
  - Merge sort: Takes time $c_2 n\lg n$

where c_1 and c_2 are constants that do not depend on n.

Efficiency – An Example

- Each must sort an array of 10 million numbers.
- Computer A executes 10 billion instructions per second and computer B executes 10 million instructions per second.
- Suppose insertion sort is coded in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort n numbers. Also suppose that merge sort is implemented in high-level language with an inefficient compiler on computer B, with the resulting code taking 50n lg n instructions.
- To sort the array, computer A takes: $\frac{2.\left(10^7\right)^2 instructions}{10^{10}\, instructions\ per\ second}$=20,000 sec.
- The computer B takes: $\frac{50.10^7 \lg 10^7 instructions}{10^7\, instructions\ per\ second}$ $\approx$1163 sec.
- By using an algorithm whose running time grows more slowly, even with a poor compiler, computer B runs approx. 17 times faster than computer A.

Describing Algorithms

- A complete description of an algorithm consists of three parts:
  1. the algorithm e*xpressed in a way that is clear and concise (can be pseudocode*)
  2. a proof of the algorithm’s correctness
  3. a derivation of the algorithm’s running time

InsertionSort

- Like sorting a hand of playing cards:
  - start with empty left hand, cards on table
  - remove cards one by one, insert into correct position
  - to find position, compare to cards in hand from right to left
  - cards in hand are always sorted

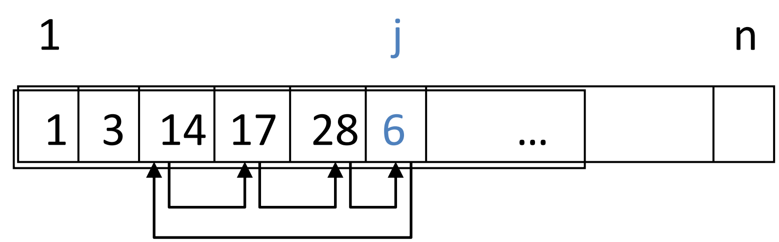
InsertionSort is

- a good algorithm to sort a small number of elements
- an **incremental algorithm**
  - having sorted the subarray A[1.. j-1], we inserted the single element A[j] into its proper place, yielding the sorted subarray A[1..j].

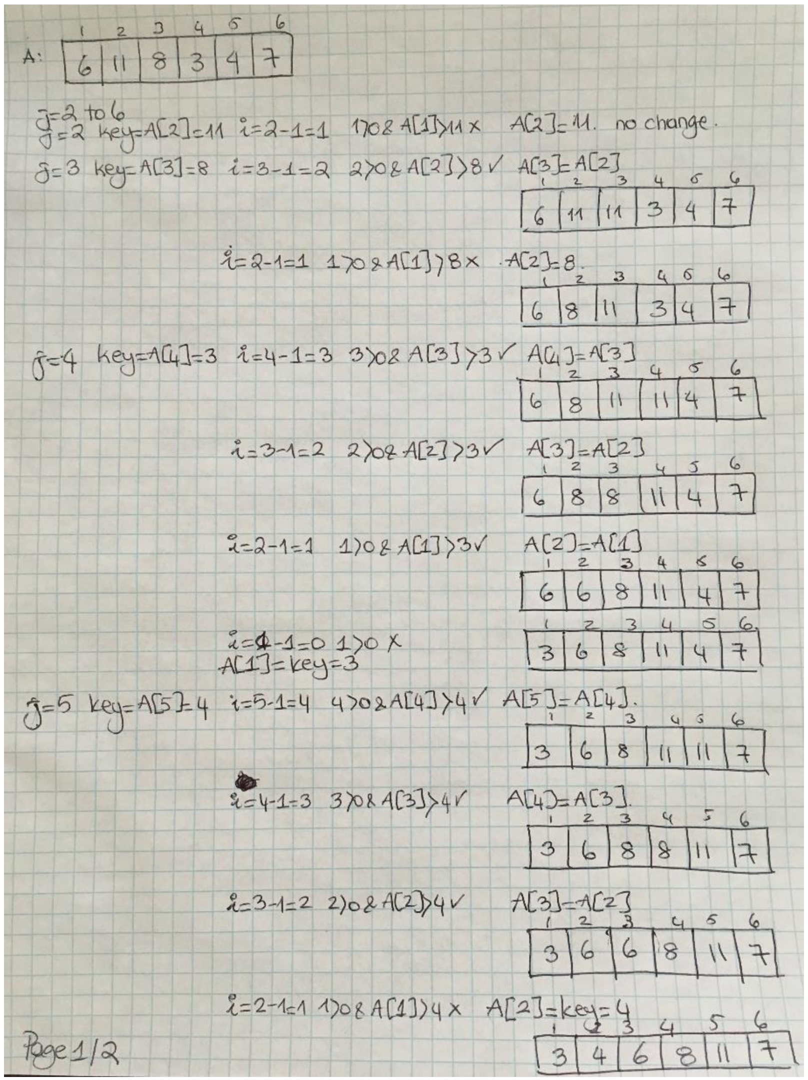
InsertionSort

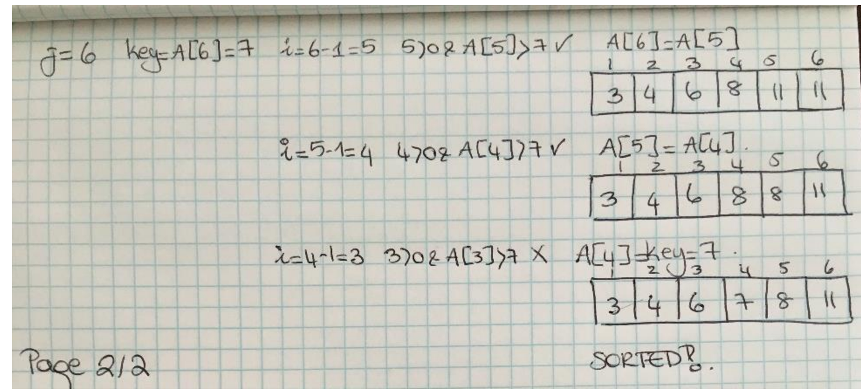
```
InsertionSort(A)
    for j ← 2 to length[A]
      key ← A[j]
      i ← j -1
      while i > 0  and A[i] > key
         A[i+1] ← A[i]
         i ← i -1
      A[i +1] ← key
```

InsertionSort is an **in place** algorithm: the numbers are rearranged within the array with only constant extra space.



Insertion Sort Example





InsertionSort(A)

1. **for** j ← 2 **to** length[A]
2. key ← A[j]
3. i ← j -1
4. **while** i > 0 and A[i] > key
5. A[i+1] ← A[i]
6. i ← i -1
7. A[i +1] ← key

Correctness proof

- Use a loop invariant to understand why an algorithm gives the correct answer.
- A **loop invariant** is a condition among program variables that is true before and after each iteration of a loop.
  - Loop invariant for InsertionSort: At the start of each iteration of the **for** loop (lines 1-7) the subarray A[1..j-1] consists of the elements originally in A[1..j-1] in sorted order.

```
InsertionSort(A)
1.    for j ← 2 to length[A]
2.       key ← A[j]
3.       i ← j -1
4.       while i > 0  and A[i] > key
5.          A[i+1] ← A[i]
6.          i ← i -1
7.       A[i +1] ← key
```

Correctness proof

- For proof correctness with a loop invariant we need to show three things:

1. **Initialization:** Invariant is true prior to the first iteration of the loop.
2. **Maintenance:** If the invariant is true before an iteration of the loop, it remains true before the next iteration.
3. **Termination:** When the loop terminates, the invariant (usually along with the reason that the loop terminated) gives us a useful property that helps show that the algorithm is correct.

Analyzing Algorithms: Insertion Sort

- ***Running time*** for a particular input is the number of *primitive operations* (steps) executed
- ***Assumption***: Constant time c_i for the execution of the i^{th} line (of pseudocode)

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $\quad key = A[j]$	c_2	$n-1$
3 $\quad$ // Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
4 $\quad i = j-1$	c_4	$n-1$
5 $\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $\qquad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $\qquad i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $\quad A[i+1] = key$	c_8	$n-1$

Note: t_j is the number of times the ***while*** loop test in line 5 is executed for that value of j.

Analyzing Algorithms: Insertion Sort

- 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.

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

Analyzing Algorithms: Insertion Sort

INSERTION-SORT(A)	cost	times
for $j = 2$ **to** $A.length$	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	$n-1$

- ***Best case***
  - Array is already sorted, so $t_j = 1$ for $j = 2, 3, \ldots, n$.
  - $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$
    $= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$
    $= an + b$ (linear function of n)

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.

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

Analyzing Algorithm

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	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	$n-1$

- ***Worst case***
  - Array is in reverse sorted order
  - Must compare each element A[j] w/ each element in the sorted subarray A[1..j-1]. So, t_j=j for j=2, 3, .., n. Note that:

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

  - Therefore running time is:

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

= $an^2 + bn + c$ (quadratic function of n)

Analyzing Algorithms

- **Average Case:**
- Concentrate on <u>worst-case</u> running time
  - Provides the upper bound
  - Occurs often
  - Average case is often as bad as the worst case

Order of Growth

- The order of a running-time function is the fastest growing term, discarding constant factors
- As an example for Insertion sort
  - Best case: $an + b \rightarrow \Theta(n)$
  - Worst case: $an^2 + bn + c \rightarrow \Theta(n^2)$

Analyzing Algorithms: An Example

Assume:

X Sort: $15 n^2 + 7n - 2$

Y Sort: $300 n \log n + 50 n$

	n=10	n=100	n=1000
X Sort	1568	150698	1.5×10^7
Y Sort	10466	204316	3.0×10^6

XSort
6 times faster

XSort
1.35 times faster

YSort
5 times faster

n = 1,000,000 XSort 1.5×10^{13}

YSort 6×10^9 YSort 2500 times faster !

Analyzing Algorithms

- It is extremely important to have efficient algorithms for large inputs
- The rate of growth (or order of growth) of the running time is far more important than constants

InsertionSort	: $\Theta(n^2)$
MergeSort	: $\Theta(n \log n)$

- We'll see the growth of functions in Ch 3

Designing Algorithms

- Several approaches are possible:
- ***Incremental*** design:
  - Iterative
  - Example: insertion sort. having sorted the subarray A[1..j-1], we inserted the single element A[j] into its proper place, yielding the sorted subarray A[1..j].

Designing Algorithms

- *Divide-and-conquer* approach
  - Recursive
  - Example: merge sort
- Three steps in the divide-and-conquer paradigm
  - *Divide* the problem into smaller subproblems that are smaller instances of the same problem
  - *Conquer* subproblems by solving them recursively
  - *Combine* solutions of subproblems

Designing Algorithms: Merge Sort

- Uses Divide-and-conquer approach:
  - *Divide* the *n*-element sequence into two subsequences of *n/2* elements each
  - *Conquer* sort the two subsequences recursively using merge sort
  - *Combine* merge the two sorted subsequences to produce the sorted answer

Merge Sort Algorithm

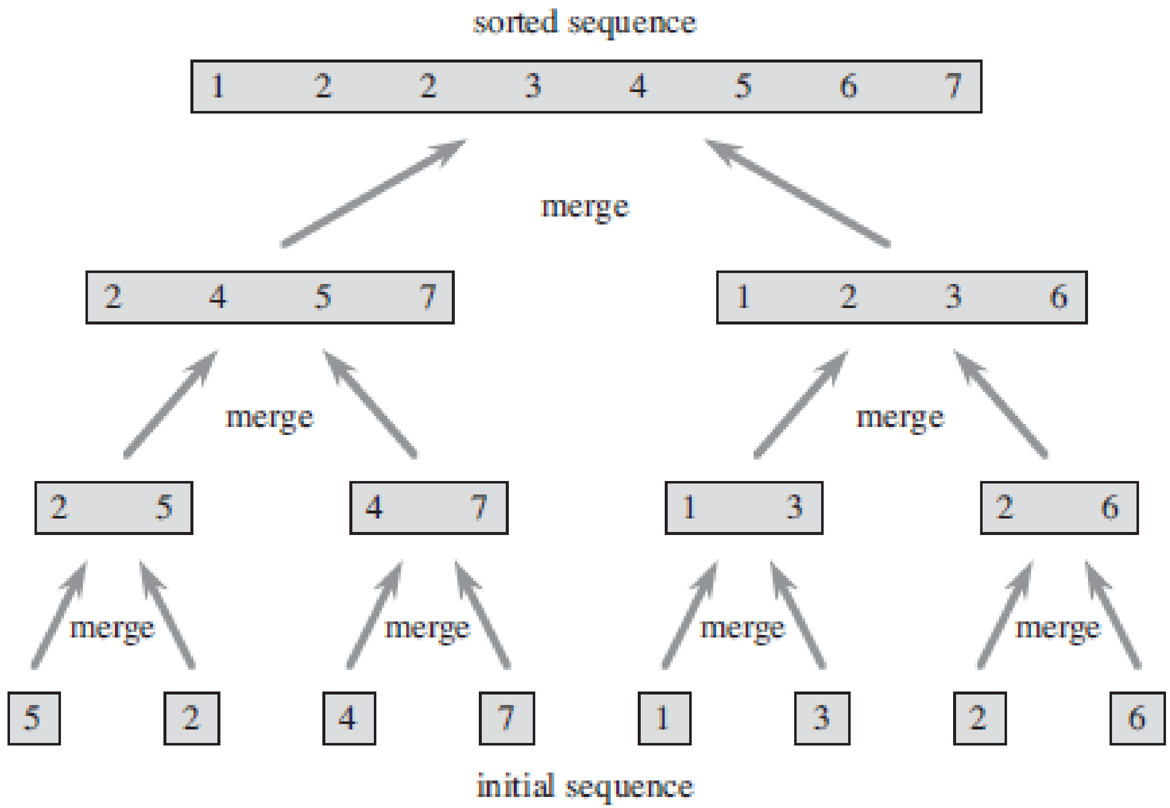
```
MERGE_SORT(A, p, r)
    if p < r
    then q ← ⌊(p + r) / 2⌋
        MERGE_SORT(A, p, q)
        MERGE_SORT(A, q+1, r)
        MERGE(A, p, q, r)
```

Merge Sort Algorithm

- Note:
  - To sort an array A[1 .. n], we call MERGE_SORT(A, 1, n)
  - The MERGE_SORT(A, p, r) sorts the elements in the subarray A[p .. r]
  - If $p >= r$, the subarray has at most one element and is therefore already sorted
  - The procedure MERGE(A, p, q, r), where $p <= q < r$, merges two already sorted subarrays A[p ..q] and A[$q+1$ .. r]. It takes $\Theta(n)$ time

Merge Sort Algorithm

- Operation of merge sort on the array A =<5, 2, 4, 7, 1, 3, 2, 6>
- The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.



MERGE(A, p, q, r)

- Performs merging of two sorted sequences in the "combine" step, where A is an array and p, q, and r are indices into the array s.t. $p \leq q < r$.
- The procedure MERGE assumes that the subarrays A[p..q] and A[q+1..r] are in sorted order, and it ***merges*** them to form a single sorted subarray that replaces the current subarray A[p..r].

MERGE(A, p, q, r)

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

The way we call MergeSort

	1	2	3	4	5	6	7	8
A:	5	2	4	7	1	3	2	6

p r
MERGE_SORT(A, 1, 8)
1<8✓ q=(1+8)/2=4

MERGE_SORT(A,1,4)
MERGE_SORT(A,5,8)
MERGE(A,1,4,8)

p r
MERGE_SORT(A,1,4)
1<4 q=(1+4)/2=2.

MERGE-SORT(A,1,2)
MERGE_SORT(A,3,4)
MERGE(A,1,2,4)

p r
MERGE_SORT(A,1,2)
1<2 q=(1+2)/2=1.

MERGE_SORT(A,1,1) Don't do
MERGE_SORT(A,2,2)
MERGE(A,1,1,2)

MERGE_SORT(A,5,8)
5<8✓ q=(5+8)/2=6.

MERGE_SORT(A,5,6)
MERGE_SORT(A,7,8)
MERGE(A,5,6,8).

MERGE(A, p, q, r)

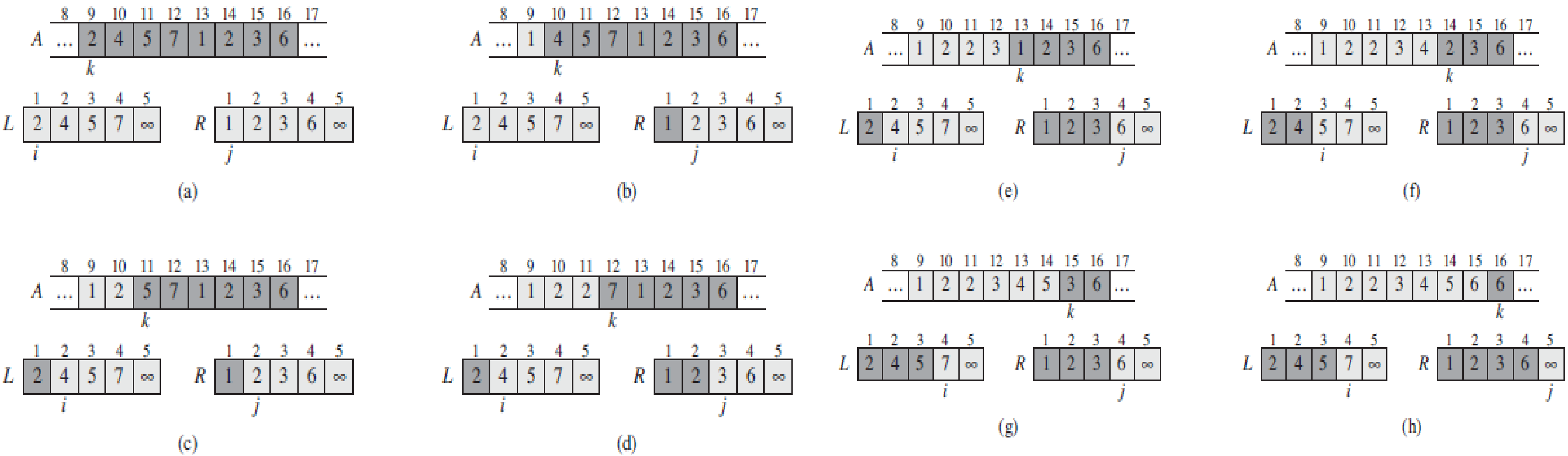
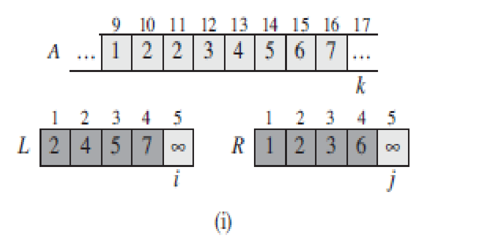


Figure 2.3 The operation of lines 10–17 in the call MERGE$(A, 9, 12, 16)$, when the subarray $A[9 \,..\, 16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$. After copying and inserting sentinels, the array L contains $\langle 2, 4, 5, 7, \infty \rangle$, and the array R contains $\langle 1, 2, 3, 6, \infty \rangle$. 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.

Figure 2.3, continued (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.



Analysis of Merge Sort

- A recurrence for the running time of a divide-and-conquer algorithm is obtained using its three steps.
- If the problem size is small enough, say n ≤ c for some constant c, the straightforward solution takes constant time, i.e. $\Theta(1)$.
- Suppose division of the problem yields *a* subproblems, each of which is 1/*b* the size of the original. (For merge sort, both *a* and *b* are 2, though there exists other divide-and-conquer algorithms in which *a* ≠ *b*.)
- It takes time *T(n/b)* to solve one subproblem of size *n/b*, and so it takes time *aT(n/b)* to solve *a* of them. If we take *D(n)* time to divide the problem into subproblems and *C(n)* time to combine the solutions, we get the recurrence:

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

Analysis of Merge Sort

- Merge sort on just one element takes constant time. When we have n > 1 elements, we break down the running time as follows.
  - **Divide:** The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.
  - **Conquer:** We recursively solve two subproblems, each with size $n/2$, which contributes $2T(n/2)$ to the running time.
  - **Combine:** We have already noted that the MERGE procedure on an n-element subarray takes time $\Theta(n)$, and so $C(n) = \Theta(n)$.
- Adding functions D(n) and C(n) requires adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of n, i.e. $\Theta(n)$. Adding it to the $2T(n/2)$ term from the "conquer" step yields the recurrence for the worst-case running time:

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

Analysis of Merge Sort

- Rewriting the recurrence as follows, where the constant c represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps:

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

- The recursion tree (next slide) has lg n + 1 levels, each costing cn, for a total cost of cn(lg n + 1) = cn lg n + cn.
- Ignoring the low-order term and the constant c yields Θ(n lg n) running time for Merge Sort.

Recursion Tree for Merge Sort

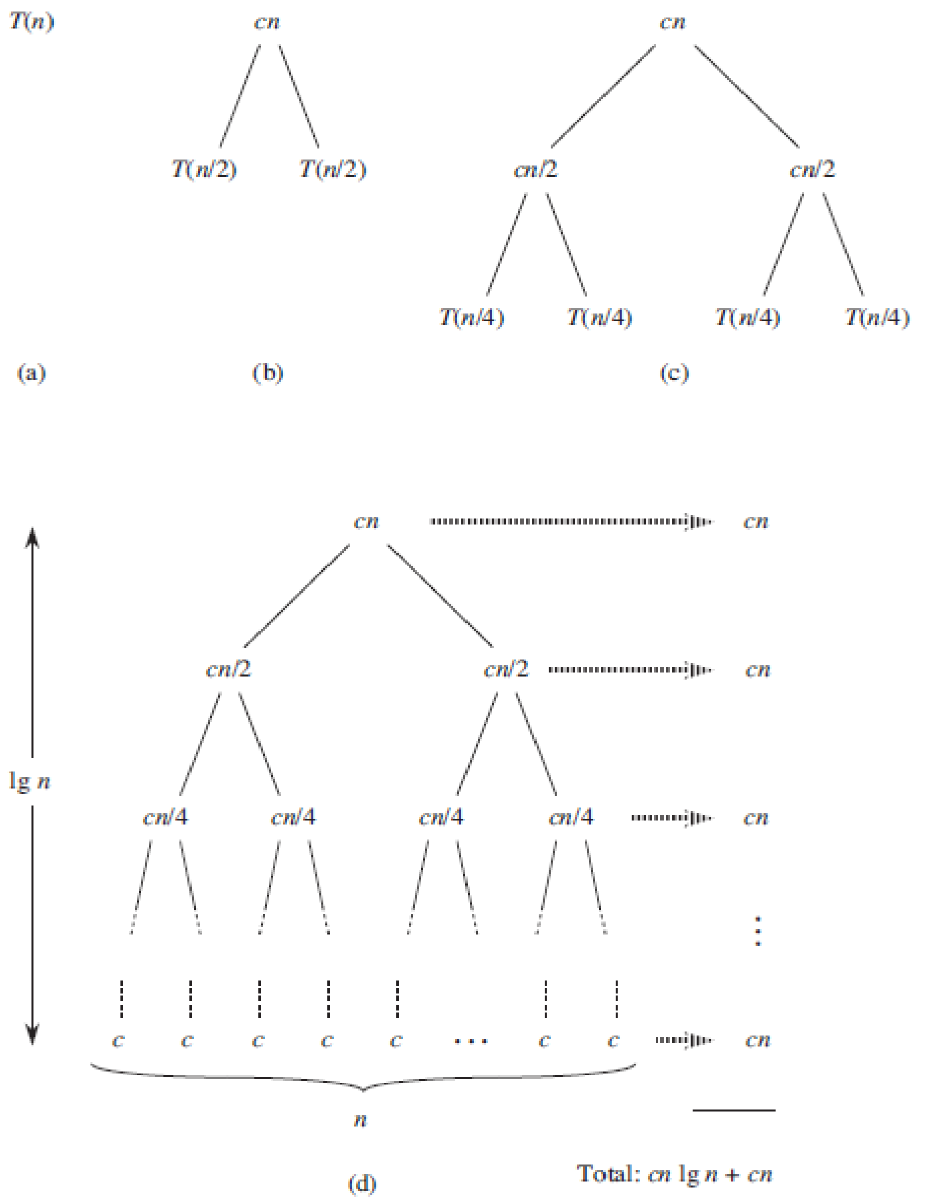


Figure 2.5 How to construct a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part (a) shows $T(n)$, which progressively expands in (b)–(d) to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes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

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

- The recursion tree has lg n + 1 levels, each costing cn, for a total cost of cn(lg n + 1) = cn lg n + cn.
- Ignoring the low-order term and the constant c yields Θ(n lg n) running time for Merge Sort.