

1. Pseudo-code – describe algorithms
2. Asymptotic notation – discuss efficiency
3. Design techniques – design algorithms

The sorting problem

Input: A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Insertion Sort

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3      // Insert  $A[j]$  into the sorted sequence  $A[1 .. j - 1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 
```

- True at **Initialization**
- **Maintained** during each iteration of the loop
- It and **termination condition** implies correctness

Algorithm design technique:

Incremental Design

You should know about **loop invariants** to show correctness (of loops)

(Read pages 18–20 of the text)

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.

A model for analyzing running times

The **Random Access Machine (RAM)** model [p 23-24]

Assume instructions commonly found on most real computers take **constant time per instruction**

- Arithmetic: add, subtract, **multiply**, divide, remainder, floor, ceiling). Also shift left/shift right (good for multiplying/dividing by 2^k).
- Data movement: load, store, copy.
- Control: conditional/unconditional branch, subroutine call and return.

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 a very frequently used shorthand for $\log_2 n$.)

- $c \geq 1 \Rightarrow$ we can hold the value of $n \Rightarrow$ we can index the individual elements.
- c is a constant \Rightarrow the word size cannot grow arbitrarily.

A model for analyzing running times

An algorithm's running time depends upon **input size** and **input value**

- Takes more time to sort more elements
 - Usually, **size** is the **number of elements** in the input
 - Sometimes, (e.g., number problems) the **number of bits** needed
 - Sometimes, **multiple** parameters (e.g., graphs)
- Primality testing – trivial for even numbers
- [We'll see that] Insertion sort takes least/ most time on sorted/ reverse-sorted input

A model for analyzing running times

For each j , let t_j denote the number of times the **while** test is evaluated

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3      // Insert  $A[j]$  into the sorted sequence  $A[1 .. j - 1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 
```

A model for analyzing running times

For each j , let t_j denote the number of times the **while** test is evaluated

INSERTION-SORT(A)

1 **for** $i = 2$ **to** $A.length$

2 Best case: each t_j is 1 $T(n)$ is a linear function of n

3 Worst case: each t_j is $j-1$ $T(n)$ is a quadratic function of n

4 We focus mainly on **worst-case** running times

5 **while** $i > 0$ and $A[i] > key$

cost times

$c_1 \quad n$

$n - 1$

$n - 1$

$c_4 \quad n - 1$

$c_5 \quad \sum_{j=2}^n t_j$

Order of growth

Theta notation – $\Theta(n^2)$

Another abstraction to ease analysis and focus on the important features.


Look only at the leading term of the formula for running time.

- Drop lower-order terms.
- Ignore the constant coefficient in the leading term.

Take-away message

1. Concentrate on the **worst case**
2. Ignore **constant factors/ lower-order terms**
3. **Asymptotic** analysis – for large values of n

A FAST algorithm is one for which the **worst-case running time grows slowly with input size**

- 
1. Pseudo-code – describe algorithms
 2. Asymptotic notation – discuss efficiency
 3. Design techniques – design algorithms

Insertion sort – incremental design

Divide and conquer

Generally **recursive** in structure – make sure you understand recursion!

The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

Divide and conquer – merge sort

The *merge sort* algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

Divide: Divide the n -element sequence to be sorted into two subsequences of $n/2$ elements each.

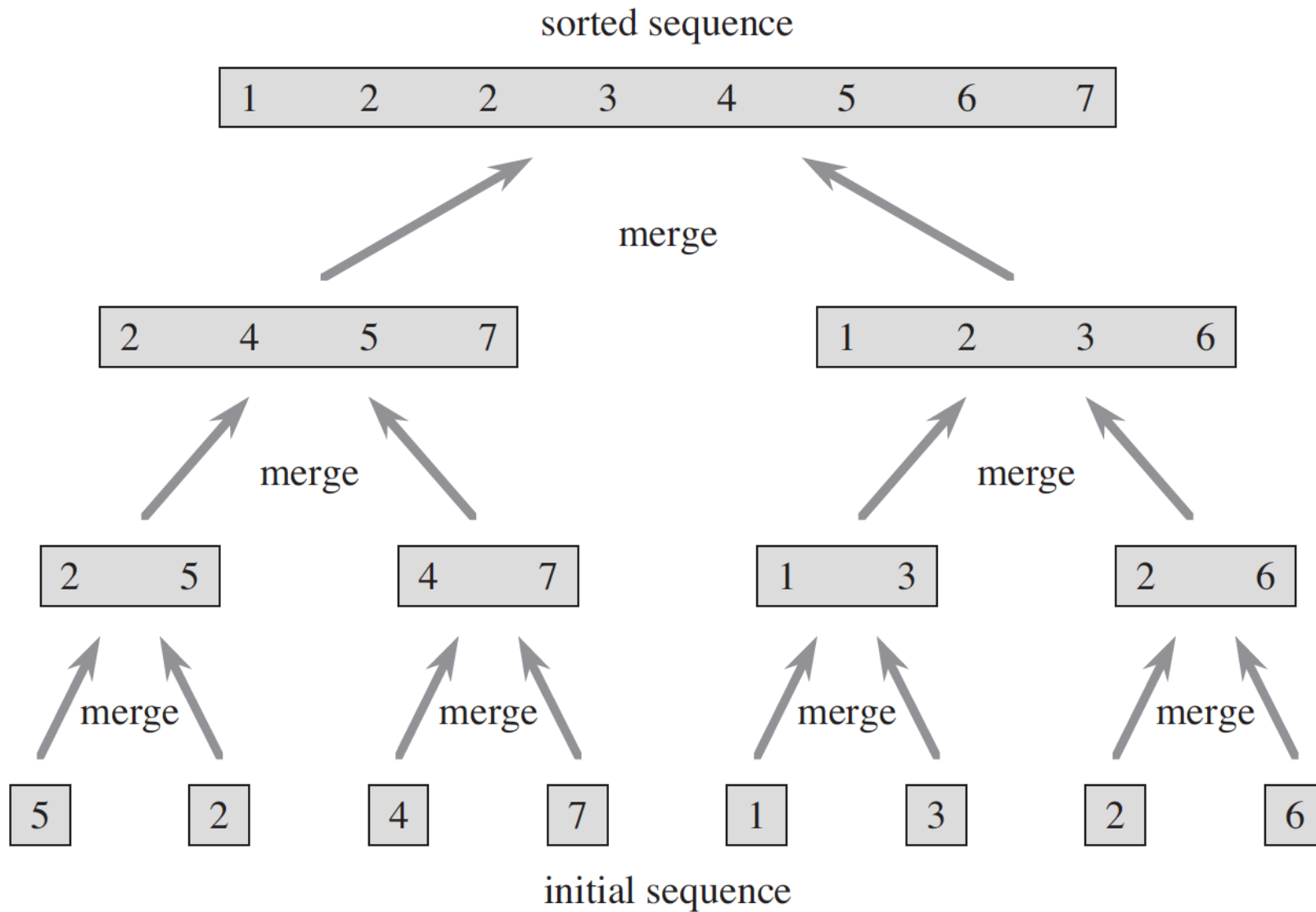
Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

The recursion “bottoms out” when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

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

Divide and conquer – merge sort



Divide and conquer analysis - recurrences

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. (In merge sort, $a = b = 2$.)
- Let the time to divide a size- n problem be $D(n)$.
- Have a subproblems to solve, each of size $n/b \Rightarrow$ each subproblem takes $T(n/b)$ time to solve \Rightarrow we spend $aT(n/b)$ time solving subproblems.
- Let the time to combine solutions be $C(n)$.
- We get the recurrence

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

For merge sort

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

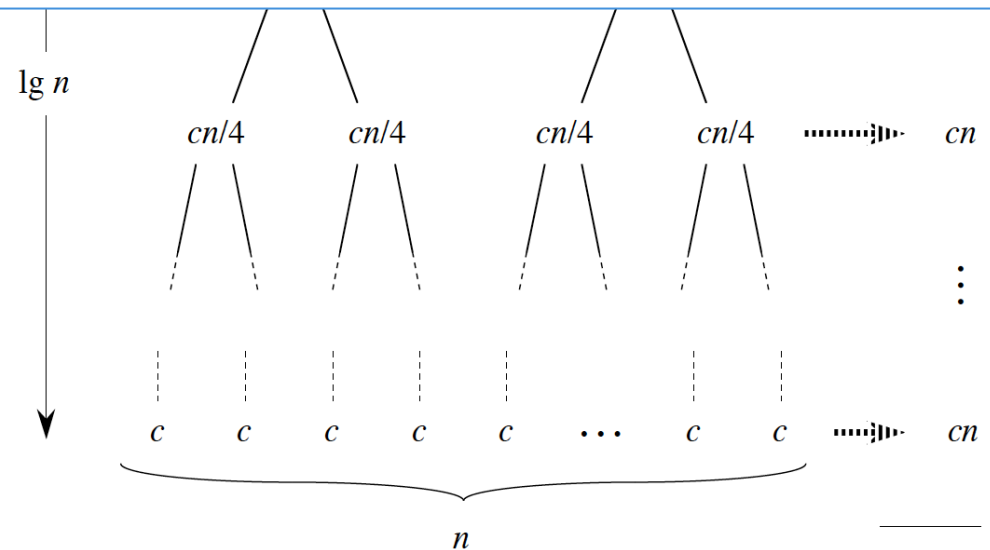
Divide and conquer analysis - recurrences

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

The Master Method yields $T(n) = \Theta(n \lg n)$

The *master method* provides bounds for recurrences of the form

$$T(n) = aT(n/b) + f(n),$$



Total: $cn \lg n + cn$