Writing Pseudocode

- You must have finished reading Section 2.1 by now.
- Make sure that you understand the pseudocode conventions in this section.

- Read 2.2 now!

Computational Problem Solving

- Problems
- Designing solution strategies
- Developing algorithms (iterative and recursive)
  - Writing algorithms that implement the strategies
  - Understand existing algorithms and modify/reuse
- Understanding an algorithm by simulating its operation on an input
- Checking/proving correctness
- Analyzing and comparing performance: complexity or efficiency
  - Theoretically: Using a variety of mathematical tools
  - Empirically: Code, run and collect performance data
- Choosing the best

Analyzing algorithms

- **Assumption: the computer that runs the algorithms is a** one processor random access machine (RAM)

Analyzing Algorithms

- The time needed by an algorithm to solve a problem depends on the **steps of the algorithm that have to be executed** (the number of primitive "instructions" or "operations" executed before it halts on a given input and produces an output) and **how big the given input is**.
- We specify this using an expression T(n) where n is the size of the input.
- So if we know T(n), we can estimate how much time that computer will take to solve that problem (see example on page 11). Alternately, if we know how much time we have, we can decide the largest size of a problem that can be solved (see problem 1-1 in Chapter 1).

Detailed Analysis of Algorithm Complexity

- Complexity or efficiency of an algorithm is characterized by an equation T(n) that represents the time taken by the algorithm to solve a problem of size n.

Computing T(n)

Arithmetic operations, assignment, single or multi-dimensional array reference, execution of statements such as return, print, read-from-file etc. take CONSTANT time.

Consecutive Steps

temp = i+1	c_1
A[i] = A[i-1]-3	c_2
*A[i] = A[i-1]*3*	c_3

Total time = $c_1 + c_2 + c_3$
Total time = c = T(n)

Conditional Step

if k=2 then	c_1
temp = i+1	c_2
A[i] = A[i-1]-3	c_3
else if k=0 then	c_4
temp = 0	c_5
else	
A[i] = A[i-1]+3	c_6
end if	

Total time T(n) < $c_1 + c_2 + c_3 + c_4 + c_5 + c_6$

In fact, a **tighter upper bound** is
T(n) ≤ $max(c_1 + c_2 + c_3\,,\ c_1 + c_4 + c_5\,,\ c_1 + c_4 + c_6)$

But we will generally add up all the costs even though that is a looser upper bound.

Loop

	cost of a step	*# times executed*
for i=1 to n	c_1	$n+1$
if A[i]>m then	c_2	n
m = A[i]	c_3	n

Total cost = $(n+1)c_1 + nc_2 + nc_3 = (c_1+c_2+c_3)n + c_1$
$T(n) = nc + c_1$ *where* $c=c_1+c_2+c_3$

Independent Nested Loops

for i=1 to n	n+1 times	$c_1(n+1)$
for j=1 to n	*n(*n+1) times	$c_2n(n+1)$
temp=i+j+1	n^2 times	c_3n^2
A[i,j]=temp	n^2 times	c_4n^2

$T(n)=c_5n^2+c_6n+c_1$

Dependent Nested Loops

```
1  for i =1 to n
2      key=A[j]
3       for j =1 to i
4           A[j +1] =A[j]
```

$$T(n) = c_1(n+1) + c_2 n + c_3 \sum_{i=1}^{n} (i+1) + c_4 \sum_{i=1}^{n} i$$

learn to expand these summations to obtain a polynomial

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

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

the running time

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

$$c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- $t_j = 1$ for j = 2,3,…,*n*; best case – why?

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

the running time

$t_j = j$ for j = 2,3,…,n; quadratic function on n; worst case – why?

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

$$c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

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

$$-(c_2 + c_4 + c_5 + c_8)$$

Best-case, worst-case and average-case analysis

- Usually, we calculate only the ***worst-case running time***
- Reason:
  - It is an upper bound on the running time
  - The worst case occurs fairly often
- The average case is often as bad as the worst case. The best case is usually one particular input amongst the large number of possible inputs.

Example

```
Max (A: Array [1..n] of integer)
m: integer
begin
1   m = A[1]
2    for i = 2 to n
3          if A[i]>m then
4                    m = A[i]
5    return m
```

$$T(n) = c_1 + c_2n + c_3(n-1) + c_4(n-1) + c_5$$

$$T(n) = c_6n + c_7$$

Determining the constants

- Count a cost of 1 for
  - reading from a variable
  - writing a value to a variable (i.e., assigning a value to a variable)
  - using an array index to locate the corresponding memory location (or accessing a memory location such as a tree or linked list node using a pointer)
  - reading from or writing into that location (i.e. reading or writing an array cell or a tree or linked list node)
  - an arithmetic operation: + - / *
  - a comparison: =, <, >, !=
  - note that ≤ and ≥ involve two comparisons
  - boolean operations: and, or, not, xor

What about loops?

- a loop statement containing one or more of: loop variable initialization, loop variable update and exit condition:
  - for loop
  - while loop
  - repeat-until loop

Determining the constants

- Examples:

  m = 12: cost 1

  m = n: cost 2

  A[5] = A[1]: cost 4

  A[i] = A[j]: cost 6

  m = A[i+1]: cost 5

  A[i] = A[i+1]: cost 7

  for i = 2 to n : depends, so we will use a cost of 1

  while i < n : cost 3

  if position ≥ m: cost 4

T(n) calculation

How detailed should the T(n) calculation be?

As detailed as needed for your purpose!

Approximate vs detailed analyses

Approximate Big-Oh Analysis of Complexity Nonrecursive Algorithms

1. For each step that is not a loop, decide whether it has a constant cost or not.
   - If constant cost, use a cost of O(1) for the entire step.
   - If not constant cost, estimate maximum (worst case) cost as a function of n and state the appropriate Big-Oh complexity.
2. For loops, start from the innermost loop and work outwards, updating the complexity as each loop is considered. For each loop:
   - Estimate the maximum (worst case or upper bound) number of times the loop statement (for, while, etc.) will execute as a function of n and state the appropriate Big-Oh complexity.
   - Calculate the single execution cost of the loop statement and each step of the body of the loop as in (1) above.
   - Multiply the two Big-Ohs.
3. Complexity of the algorithm is the same as the largest Big-Oh complexity of any step after all estimations in steps (1) and (2) are done.

Approximate Big-Oh Analysis of Complexity Recursive Algorithms

1. For each step that is not a loop, decide whether it has a constant cost or not. Ignore any recursive calls.
   - If constant cost, use a cost of O(1) for the entire step.
   - If not constant cost, estimate maximum (worst case) cost as a function of n and state the appropriate Big-Oh complexity.
2. For loops, start from the innermost loop and work outwards, updating the complexity as each loop is considered. For each loop:
   - Estimate the maximum (worst case or upper bound) number of times the loop statement (for, while, etc.) will execute as a function of n and state the appropriate Big-Oh complexity.
   - Calculate the single execution cost of the loop statement and each step of the body of the loop as in (1) above.
   - Multiply the two Big-Ohs.
3. Complexity of one execution of the algorithm is the same as the largest Big-Oh complexity of any step after all estimations in steps (1) and (2) are done.
   - Estimate the maximum (worst case or upper bound) number of recursive executions that will take place as a function of n and state the appropriate Big-Oh complexity.
   - Multiply the two Big-Oh complexities from steps above. This is the recursive algorithm's overall complexity.

Example

algorithm A1

1 **for** *i* =1 **to** n

2 key=A[*j*]

3 **for** *j* =1 **to** i

4 *A*[*j* +1] =*A*[*j*]

Suppose you want to compare this “algorithm” with another A3 that you know has three nested loops each of which executes n times. This means that A3’s approx. complexity is $O(n^3)$.

The above algorithm’s approx. complexity, on the other hand, is $O(n^2)$; so A1 is a more efficient algorithm.

But suppose you want to compare the previous algorithm A1 with algorithm A2:

```
1  for i =1 to n
2      key=A[j]
3      for j =1 to n
4          A[j +1] =A[j]
```

By approximate analysis, this and the algorithm A1 on the previous slide are both $O(n^2)$ algorithms.

But you can tell A2 will be less efficient than A1. Why?

But suppose you also want to know how much more inefficient this algorithm is. Then you need to do a detailed calculation.

algorithm A1:

1 **for** *i* =1 **to** n

2 key=A[*j*]

3 **for** *j* =1 **to** i

4 *A*[*j* +1] =*A*[*j*]

- $T(n) = 4n^2+10n+1$

algorithm A2:

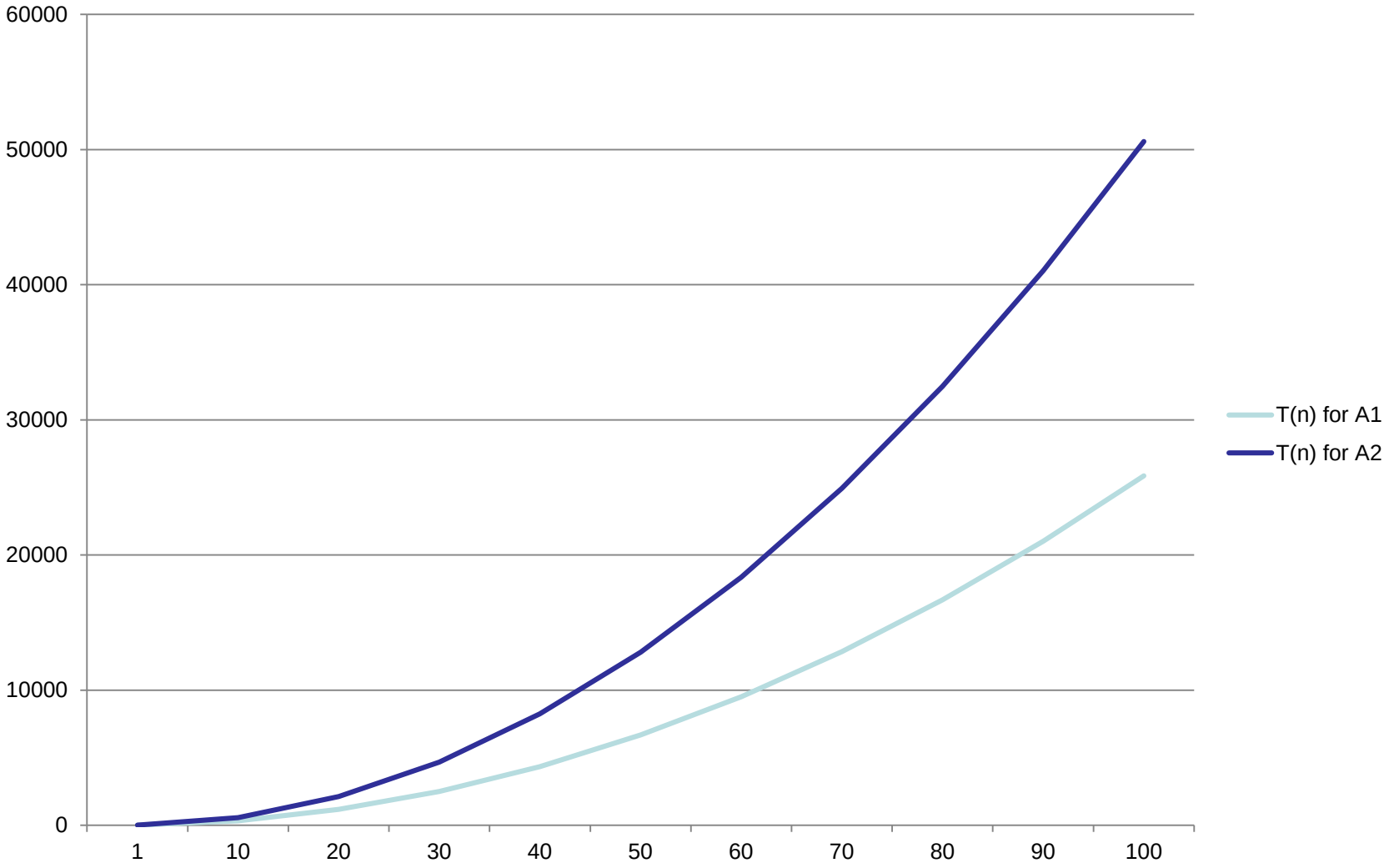
1 **for** *i* =1 **to** n

2 key=A[*j*]

3 **for** *j* =1 **to** n

4 *A*[*j* +1] =*A*[*j*]

- $T(n) = 8n^2+6n+1$


60000
50000
40000
30000
20000
10000
0
1
10
20
30
40
50
60
70
80
90
100
T(n) for A1
T(n) for A2

You can see that though the number of steps executed or the complexity of both algorithms are close to each other for very small values of n, as the input becomes larger the efficiency gap between them increases. Only a detailed complexity calculation would alert you that for large inputs A1 will be significantly faster.

Reading Assignment

- Read 2.3

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)

How many base cases – one, two, more?

Which?

Merge(A,p,q,r)

1 $n_1 = q - p + 1$

2 $n_2 = r - q$

3 create array L[1 .. $n_1 + 1$] and R[1 .. $n_2 + 1$]

4 **for** $i = 1$ ***to*** n_1

5 L[i] =A[$p + i - 1$]

6 **for** $j = 1$ ***to*** n_2

7 R[j] =A[$q + j$]

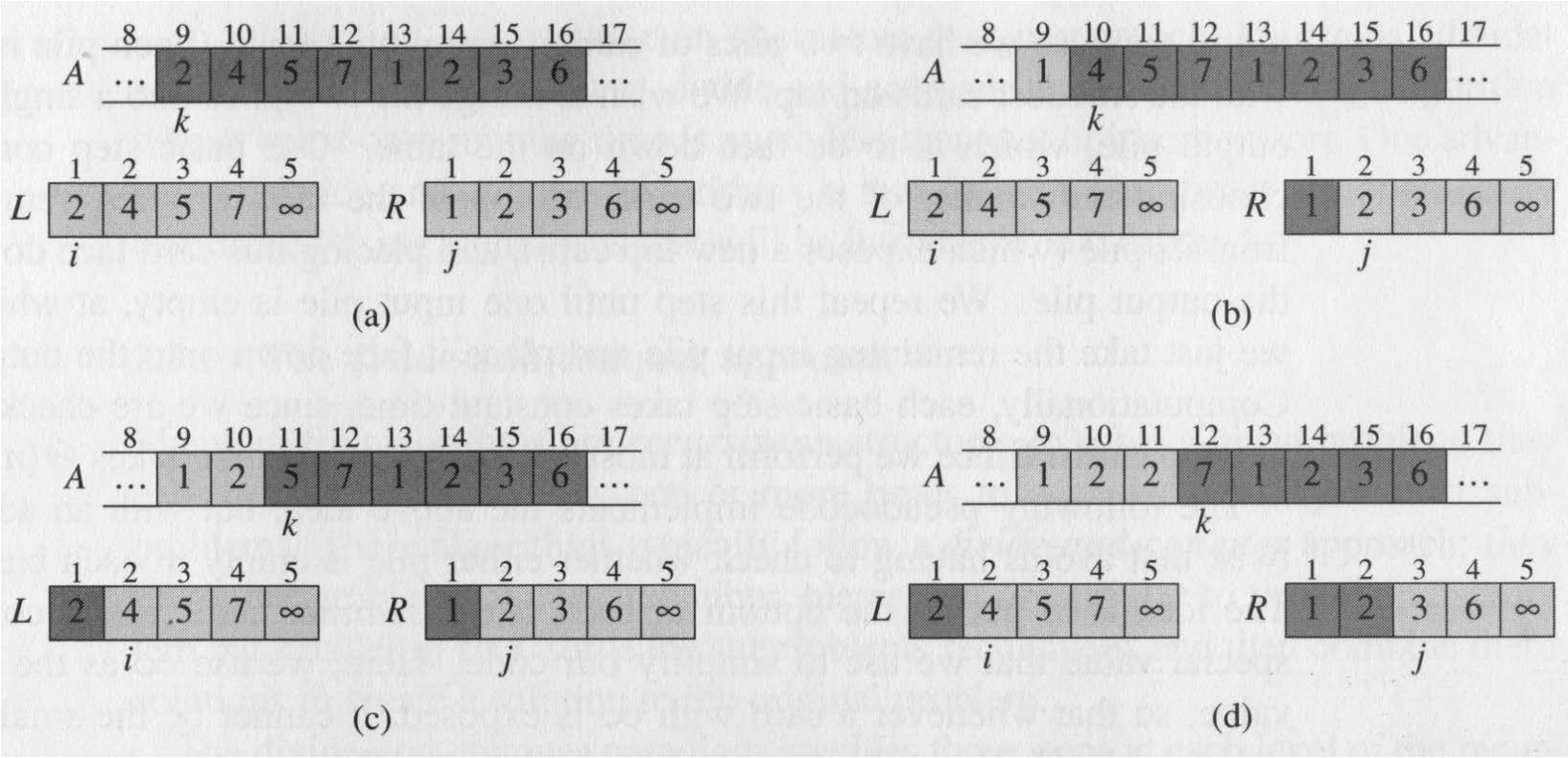
8 L[$n_1 + 1$] = ∞

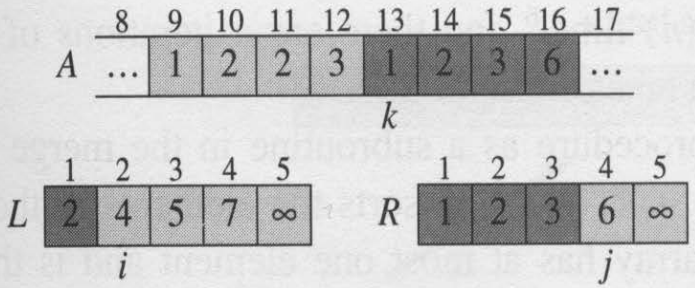
9 R[$n_2 + 1$] = ∞

Merge(A,p,q,r)

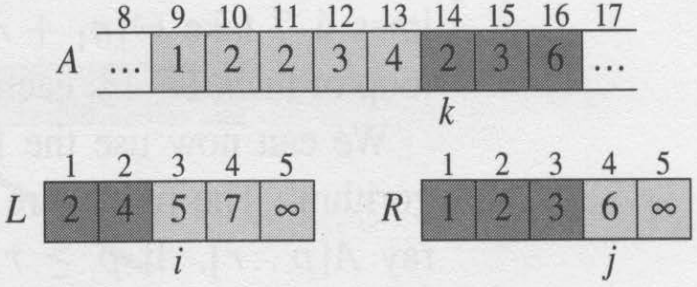
```
10    i =1
11    j =1
12  for k = p to r
13      if  L[ i ] ≤ R[ j ]
14              then A[ k ] = L[ i ]
15           i = i + 1
16      else A[ k ]  = R[ j ]
17           j  = j + 1
```

Operation of Merge

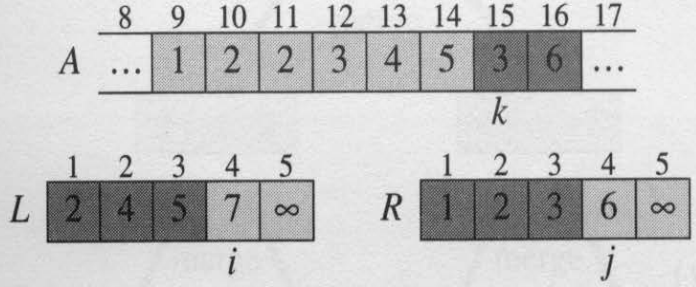



8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

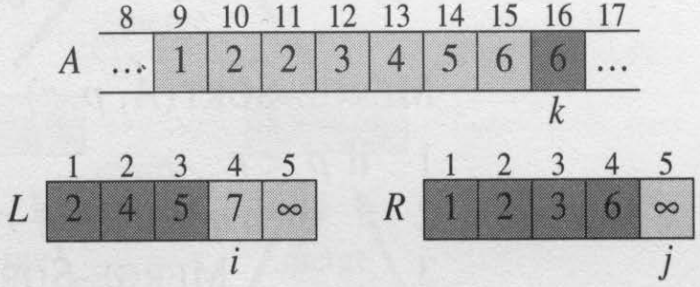
(e)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

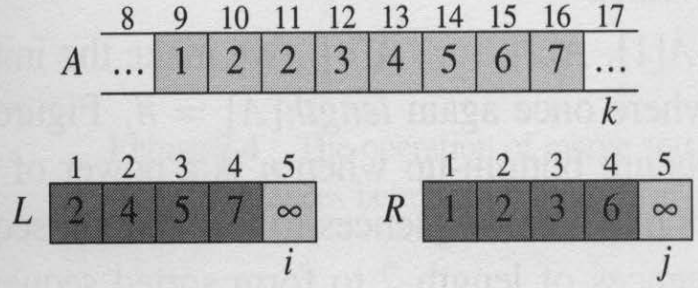
(f)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(g)

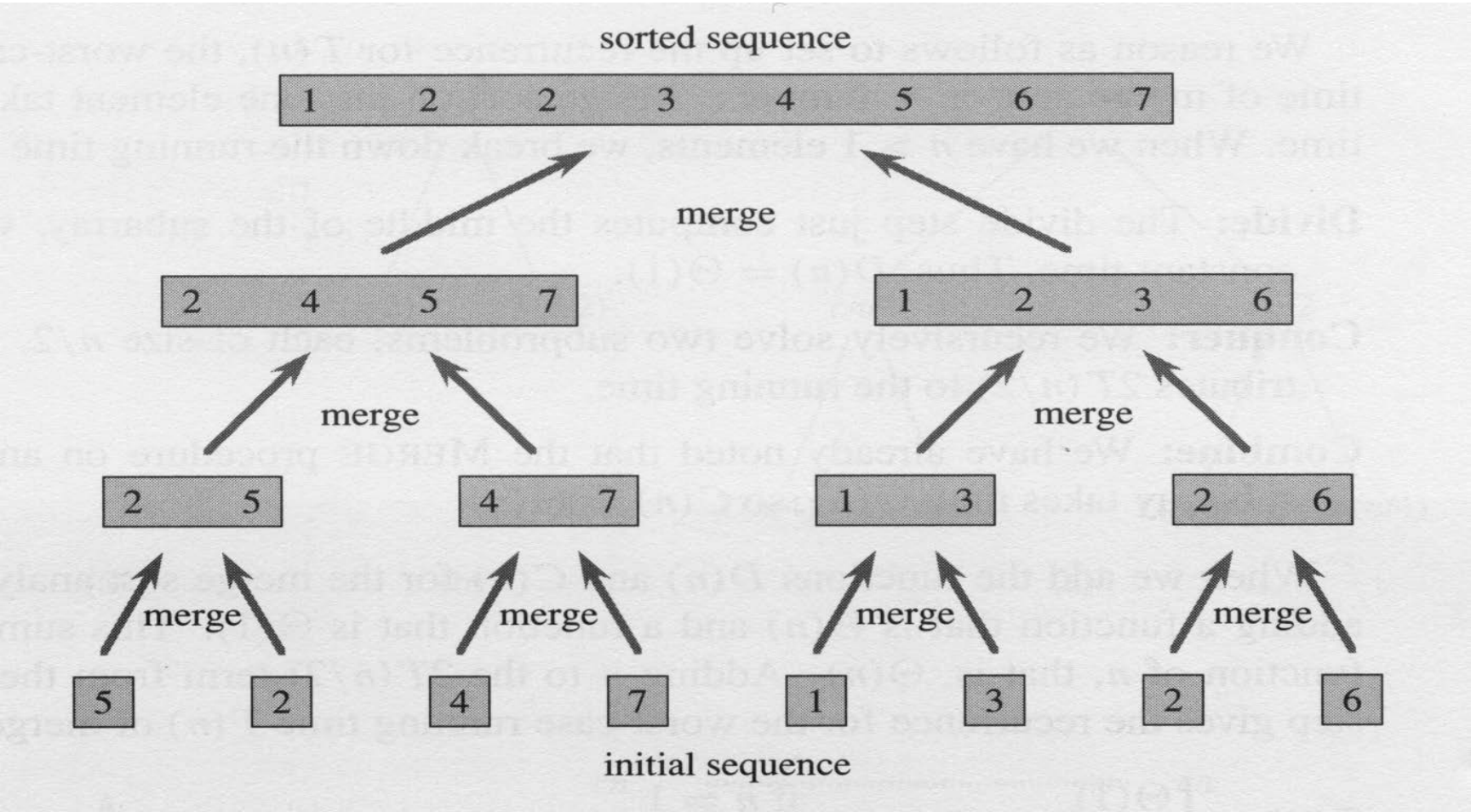

8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(h)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 7 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(i)

Operation of Merge Sort



How complex is Merge Sort?

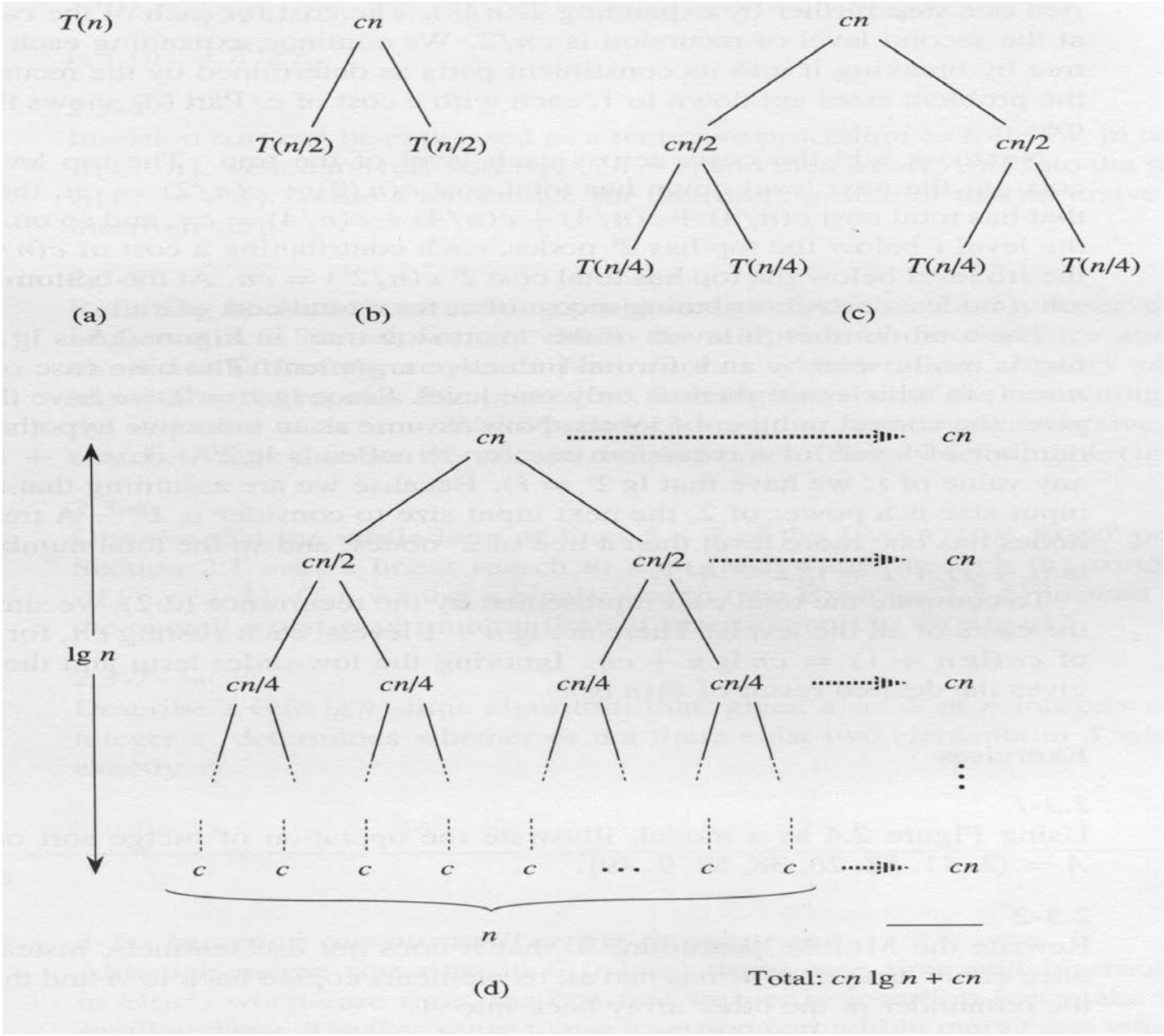
Let us try the step-by-step approach we've used for analyzing algorithms.

Step#	# of times	Cost	
1 **if** p < r	1	c_1	
2 **then** q=⌊(p+r)/2⌋	1	c_2	
3 MERGE-SORT(A,p,q)	1	???	problem!
4 MERGE-SORT(A,q+1,r)	1	???	problem!
5 MERGE(A,p,q,r)	1	cn	why?

Analysis of merge sort

$$T(n) = \begin{cases} c & if\ n = 0\ or\ 1 \\ 2T(n/2) + cn & if\ n > 1 \end{cases}$$

where the constant *c* represents both the time required to solve base cases and the time required for all steps other than the recursive executions for non-base-cases.


T(n)
cn
T(n/2)
T(n/2)
(a)
cn
cn/2
cn/2
T(n/4)
T(n/4)
T(n/4)
T(n/4)
(b)
(c)
cn
cn
cn/2
cn/2
cn
lg n
cn/4
cn/4
cn/4
cn/4
cn
c
c
c
c
c
c
c
cn
n
(d)
Total: cn lg n + cn

Which is faster? Insertion or Merge?

Designing algorithms

There are many ways to design algorithms. We will discuss these at various times throughout the semester. You have already seen two:

- **Incremental approach**: divide the problem into smaller problems (that may overlap) and use iteration (looping) to solve the smaller problems incrementally until the whole problem is solved.

  E.g., boggle problem, insertion sort

- **Divide-and-conquer**: divide the problem into non-overlapping subproblems, solve the subproblems recursively and then finish up by combining the subproblem solutions into a solution of the overall problem. E.g., merge sort.

At this point you must have completed reading Chapters 1 & 2. I also suggest that you try out some of the end-of-chapter problems.