CMP(N)302: Algorithms Design and Analysis

Lecture 01: Introduction

Ahmed Hamdy

Computer Engineering Department

Cairo University

Fall 2017

What is an *Algorithm?*

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

An algorithm is thus a sequence of computational steps that transform the input into the output.

Problem example: Sorting

- **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 $a'_1 \leq a'_2 \leq \cdots \leq a'_n$
- **Algorithms:**
  - Insertion sort: $2n^2$ instructions
  - Merge sort: $50n \log_2 n$ instructions

Insertion vs Merge sort

Input size	Insertion sort	Merge sort
n	$2n^2$	$50n \log_2 n$
2	8	100
10	200	1661
100	20,000	33,219
1K	2,000,000	498,289
10K	200,000,000	6,643,856

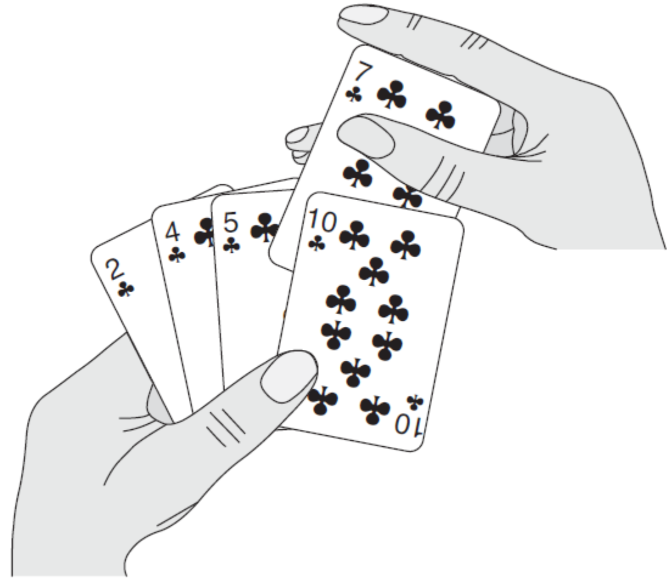
- For $n = 2, 10, 100$, Insertion sort is faster
- For $n = 1K, 10K$, …, Merge sort is faster
- Recommendation?

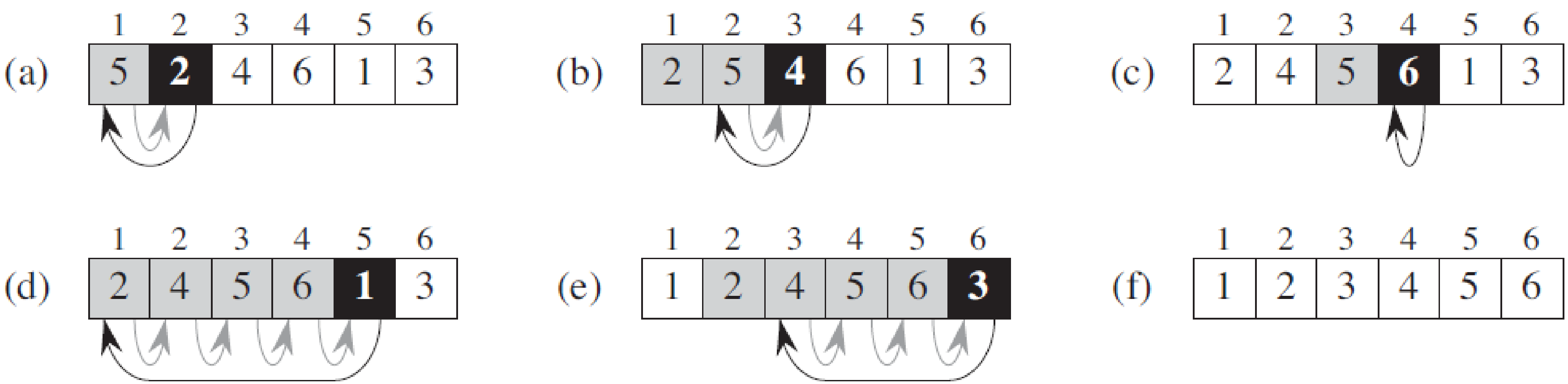
*Merge sort constants can be less than the above, they are just for illustration

Sorting algorithm

- How to come up with an algorithm based on our daily problems?
- Imagine you are playing cards and you have 13 cards of the same suit. How typically you sort them??

Insertion sort

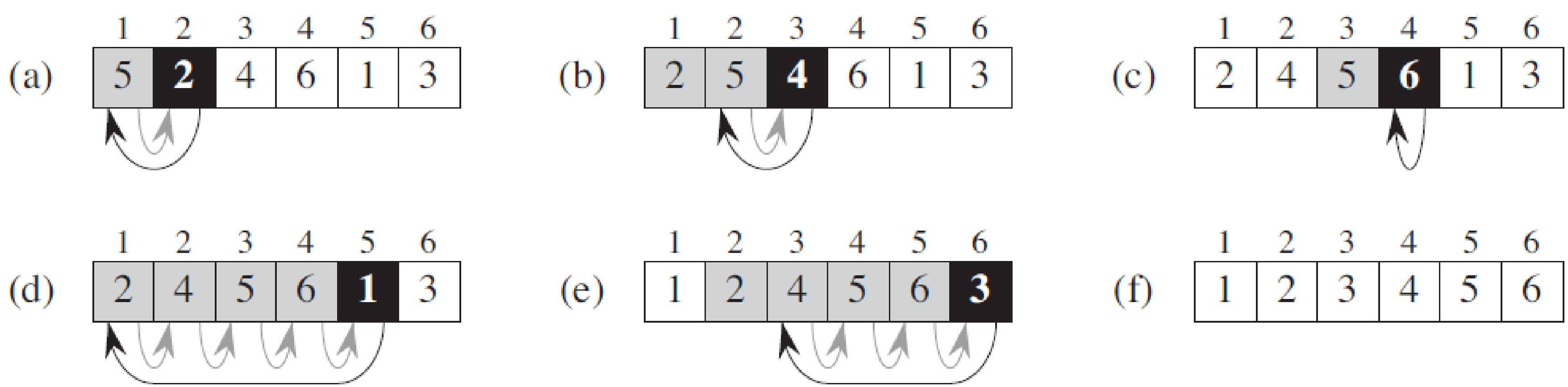




How to write it in pseudo-code??

Insertion sort

```
INSERTION-SORT(A)
for j = 2 to A.length
    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
```



Algorithm analysis

- Based on loops, what is the running time $T(n)$ in terms of the size of the input n?

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	$n-1$

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

Algorithm analysis

- **Worst-case:** Max($T(n)$)
  – applies to certain input cases
- **Best-case:** Min($T(n)$)
  – applies to certain input cases
- **Average-case:** $E[T(n)]$, requires knowledge of statistical distribution of inputs (can be biased)
  - Approx. to worst-case (when the best-case is the exception)
  - Approx. to best-case (when the worst-case is the exception)

Order of growth

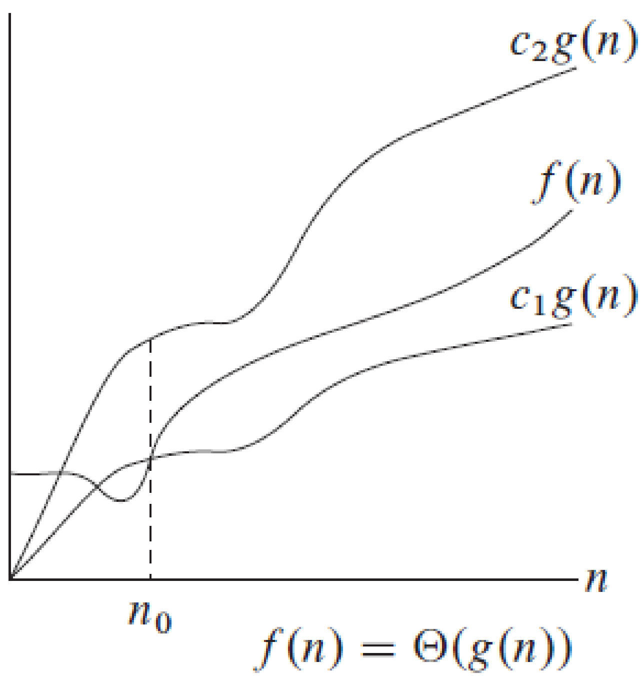
- Instruction delays are machine-dependent
  - IPC (Instructions per cycle) is machine dependent
  - CPU frequency varies even with same IPC
- Exact running time is overly complex
  - Significance of Lower-order terms in $\mathrm{T(n)}$ ↓ as n ↑: in quadratic running time, linear term is insignificant with large n
  - Care for the case $n \rightarrow \infty$, the highest-order term dominates
- Highest-order term represents ***order of growth***
- Neglect constants

Asymptotic analysis

- Define $\boldsymbol{\theta - notation}$ (Theta):

$$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \} .^1$$

- We say that $g(n)$ is an ***asymptotically tight bound*** for $f(n)$



Algorithm analysis

- What about Insertion sort?
- Best-case: when input is (nearly) sorted, $\Theta(n)$.
- Worst-case: when input is (nearly) sorted in reverse, $\Theta(n^2)$.
- Average-case:
  - On average, half of the checks in the inner loop condition are true, so $t_j = j/2$. So $\Theta(n^2)$.

Another sorting algorithm

- How to come up with an algorithm based on our daily problems?
- Imagine 10 persons want to sort 1000 student exam papers by sequential ID#. What typically they do??

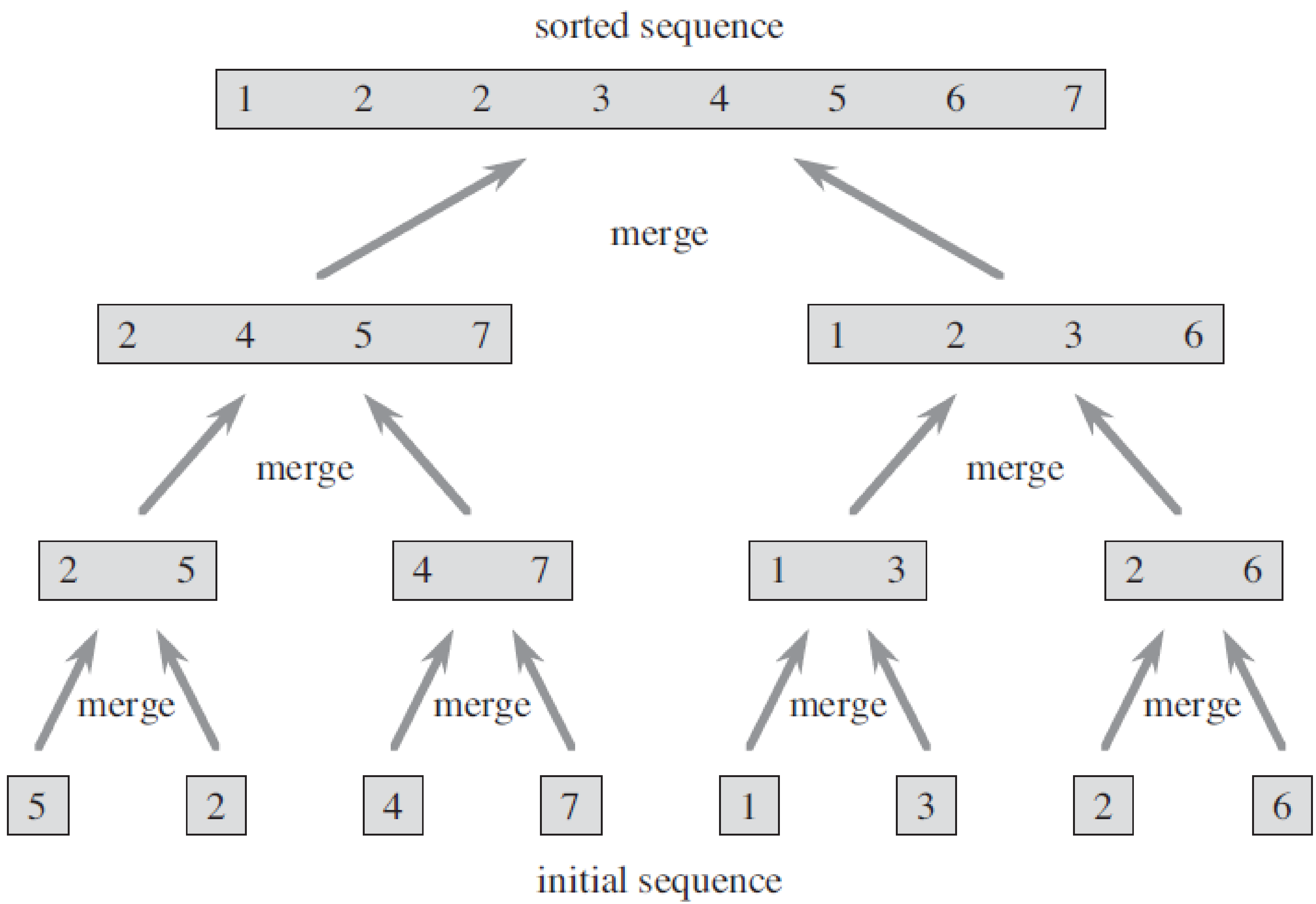
Design strategies

- In Insertion sort, one element at a time is inserted in the previously sorted subarray → *Incremental approach*
- ***Divide-and-conquer**:*
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
  - **Conquer**
    - Solve subproblems in a straightforward manner if simple
    - Otherwise solve subproblems recursively
  - **Combine** the solutions to the subproblems into the solution for the original problem.

Merge sort

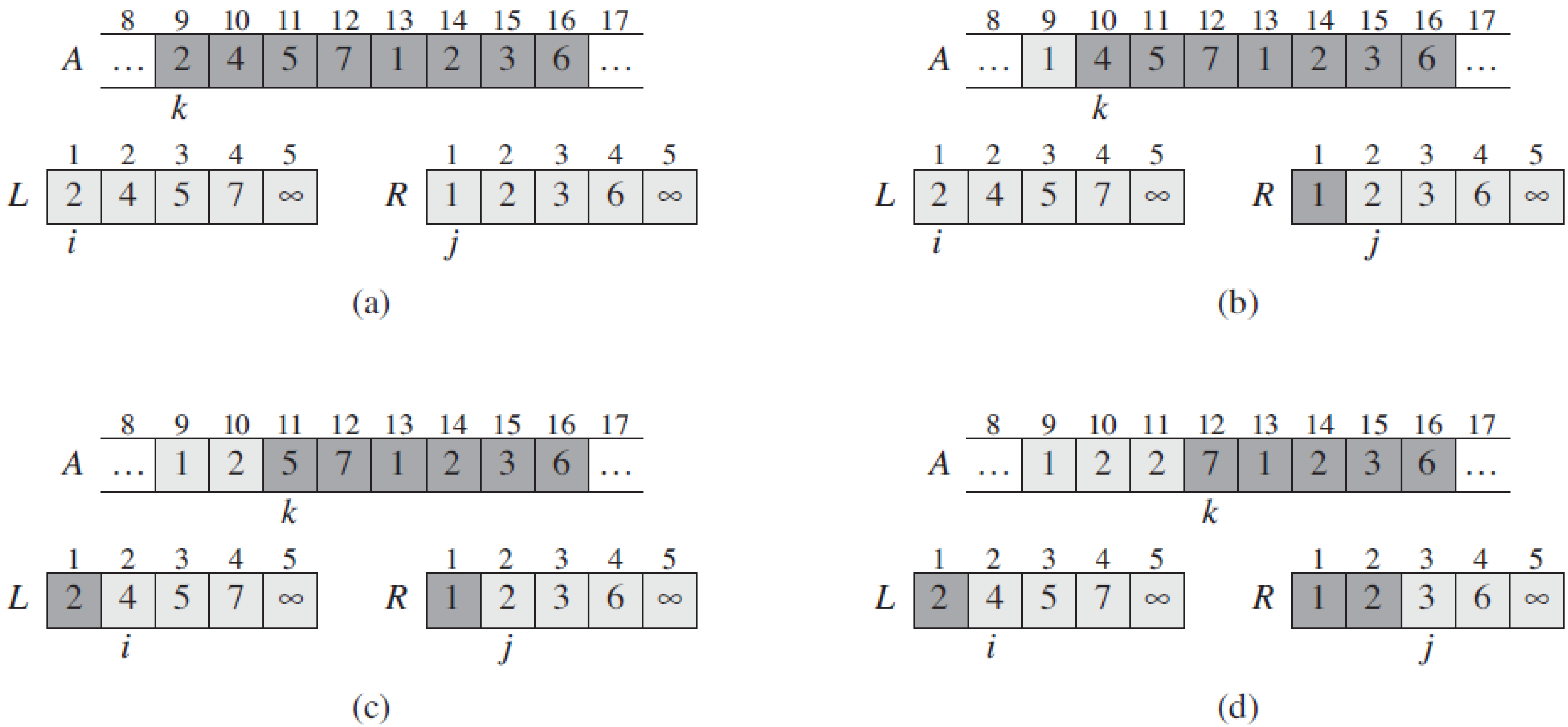
- ***Divide-and-conquer:***
  - **Divide:** Divide the n-element sequence to be sorted into two subsequences of $n = 2$ elements each
  - **Conquer:** Sort the two subsequences recursively using merge sort
  - **Combine** the solutions to the subproblems into the solution for the original problem.

Merge sort


sorted sequence
1 2 2 3 4 5 6 7
merge
2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6
initial sequence

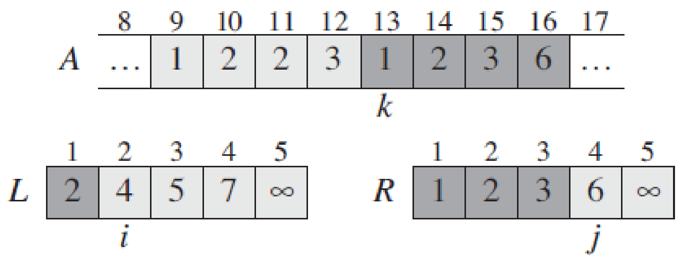
Merge sort

- Merge step

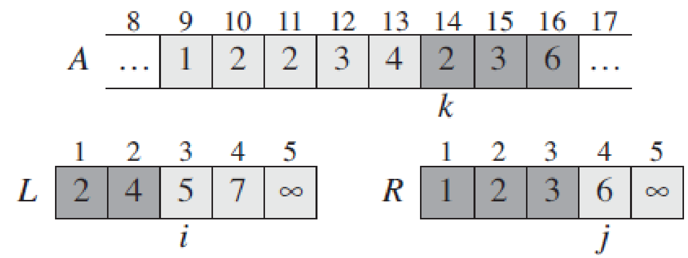


Merge sort

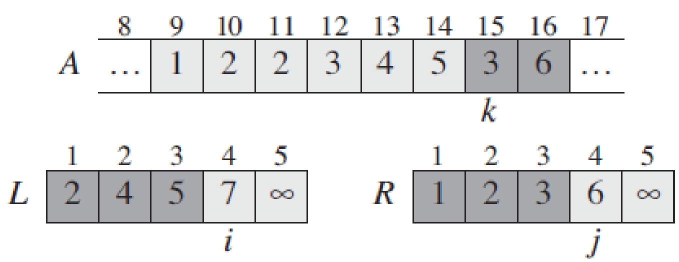
- Merge step (cont.)



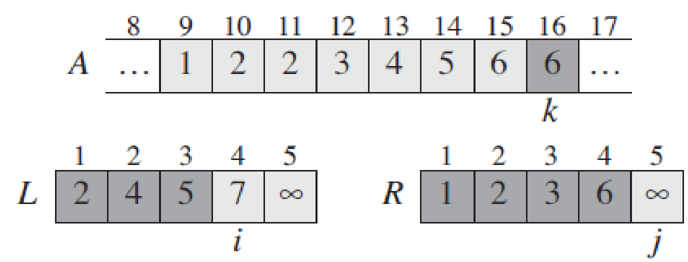
(e)



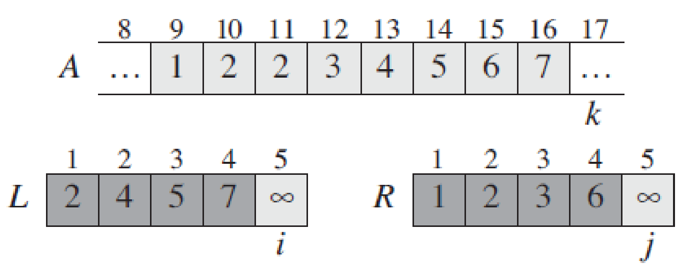
(f)



(g)



(h)



(i)

Merge sort

```
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
```

Merge sort

- Main subroutine

MERGE-SORT(A, p, r)

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

- Initial call:

MERGE-SORT$(A, 1, A.length)$

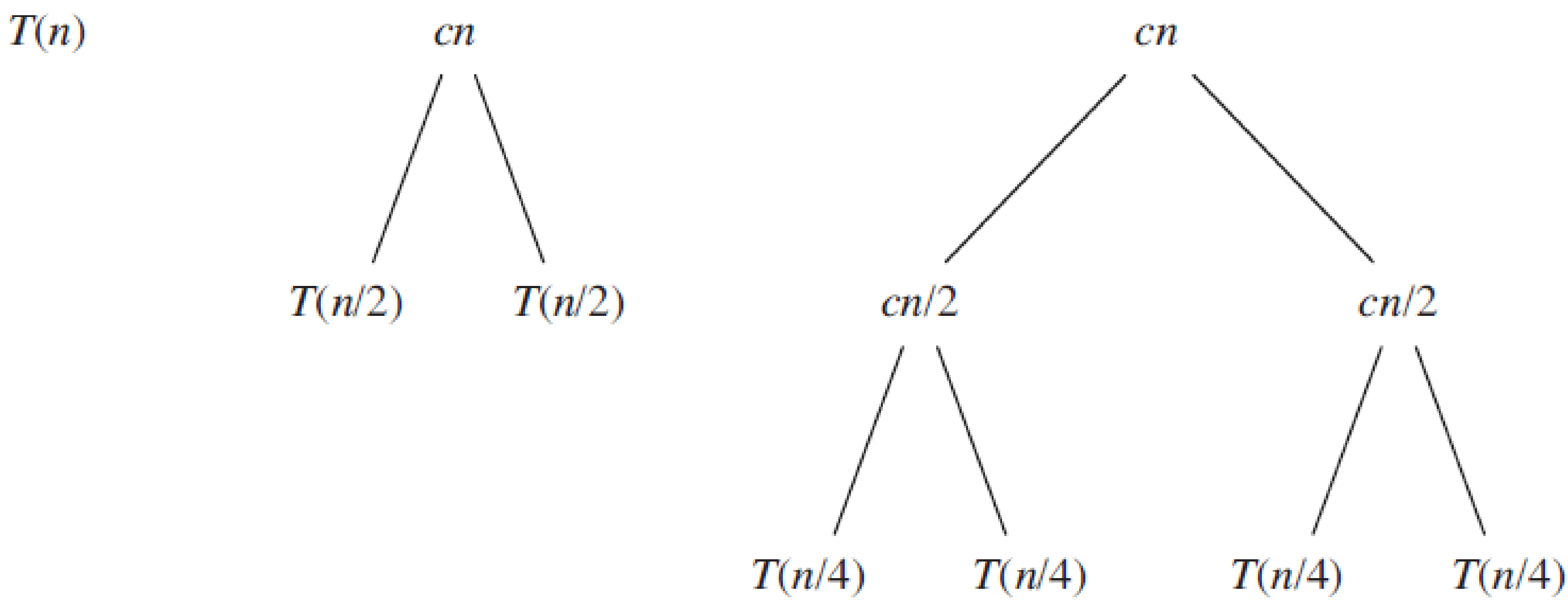
Merge sort analysis

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

- **Divide:** just computes the middle of the subarray. Thus, $D(n) = \Theta(1)$.
- **Conquer:** recursively solve two subproblems, each of size $n/2$, which contributes $2T\,(n/2)$ to the running time.
- **Combine:** MERGE procedure on an n-element subarray takes time $\Theta(n)$, and so $C(n) = \Theta(n)$.

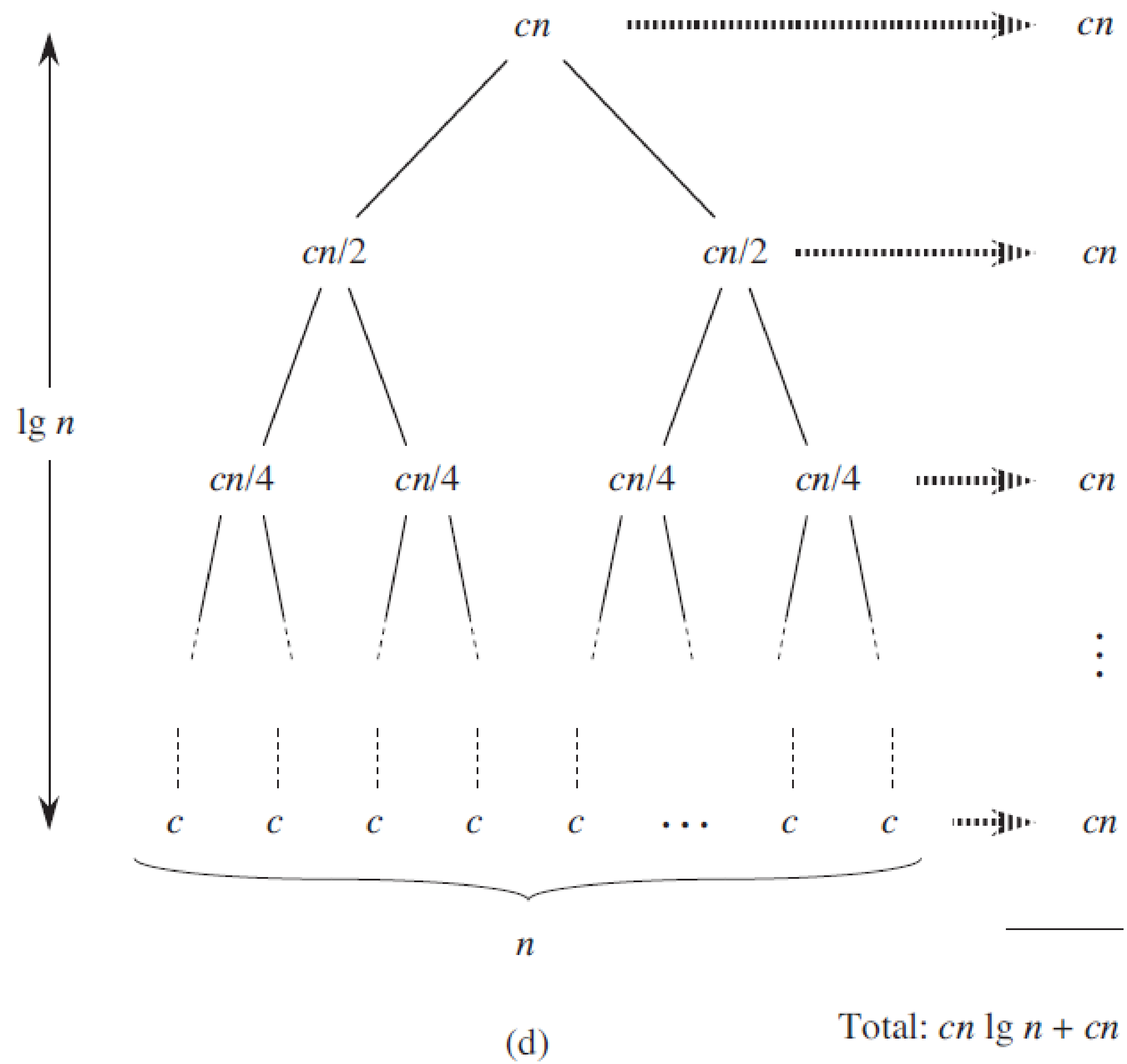
Merge sort analysis

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



(a) (b) (c)

Merge sort analysis



(d)

Space complexity

- Insertion sort:
  - Does sorting in-place, thus $\Theta(1)$.

- Merge sort:
  - Merge subroutine requires temporary space with complexity $\Theta(n)$.

Insertion vs Merge sort

- Insertion sort:
  - Time complexity $\mathrm{T(n)} = \Theta(n^2)$.
  - Space complexity (in-place) $S(\mathrm{n}) = \Theta(1)$.

- Merge sort:
  - Time complexity $\mathrm{T(n)} = \Theta(n \log n)$.
  - Space complexity $S(\mathrm{n}) = \Theta(n)$.

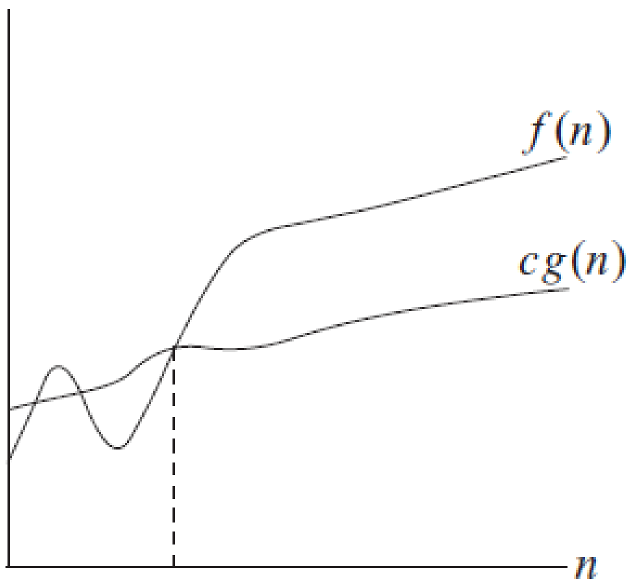
- For small n, Insertion sort is better while Merge sort is better for large n.

Asymptotic analysis

- Define $\Omega - \boldsymbol{notation}$ (Big-Omega):

$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \} .$$

- We say that $g(n)$ is an ***asymptotically lower bound*** for $f(n)$



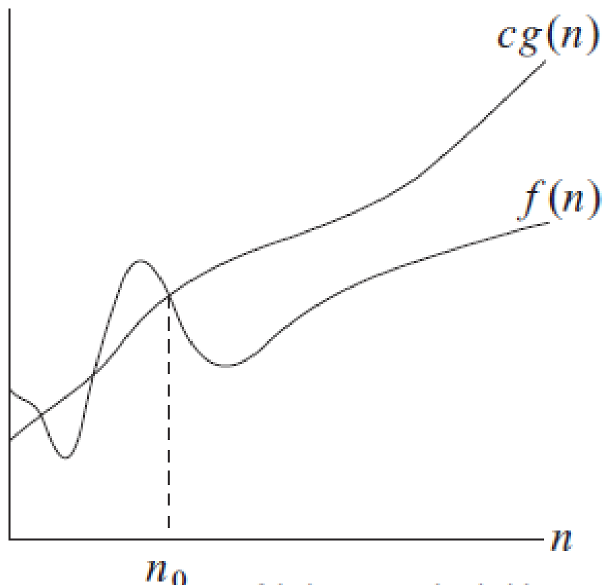
$f(n) = \Omega(g(n))$

Asymptotic analysis

- Define $\boldsymbol{O-notation}$ (Big-O):

$$O(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$

- We say that $g(n)$ is an ***asymptotically upper bound*** for $f(n)$



$f(n) = O(g(n))$

- May be asymptotically tight; $2n^2 = O(n^2)$
- May not be asymptotically tight; $2n = O(n^2)$

Asymptotic analysis

Non-asymptotically tight bounds:

- Define $\boldsymbol{o-notation}$ (little-o)
  - Similar to Big-O, but not tight

- Define $\boldsymbol{\omega-notation}$ (little-omega)
  - Similar to Big-Omega, but not tight

Recurrences

- Substitution method
  - Guess a bound
  - Use mathematical induction to prove
- Recursion-tree method
  - Convert recurrence into a tree
  - Use techniques for bounding summations
- Master method
  - Provides bounds for recurrences with the form

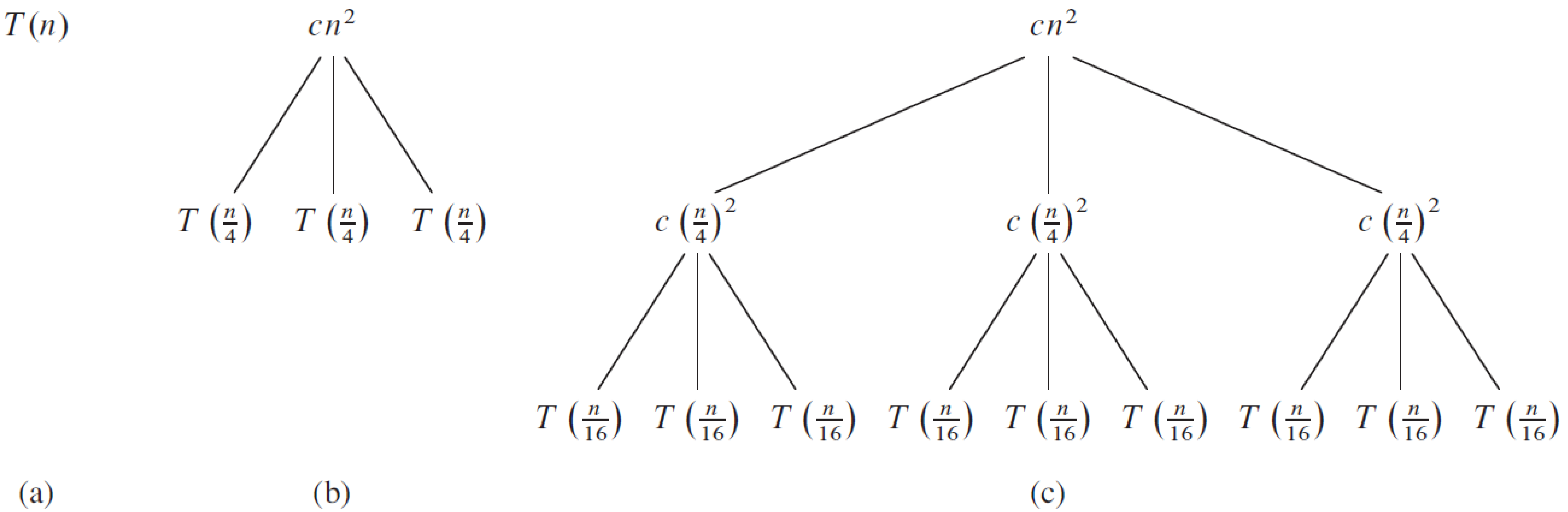
$$T(n) = aT(n/b) + f(n)$$

Substitution method

- Find upper bound for $T(n) = 2T(\lfloor n/2 \rfloor) + n$
- Guess $T(n) = O(n \lg n)$
- Use mathematical induction:
  - Base case: assume it holds for all $m < n$, say $m = \lfloor n/2 \rfloor$, thus $T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$.
  - Induction:
$$\begin{aligned} T(n) &\le 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\le cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\le cn \lg n , \end{aligned}$$
    - Holds for $c \ge 1$

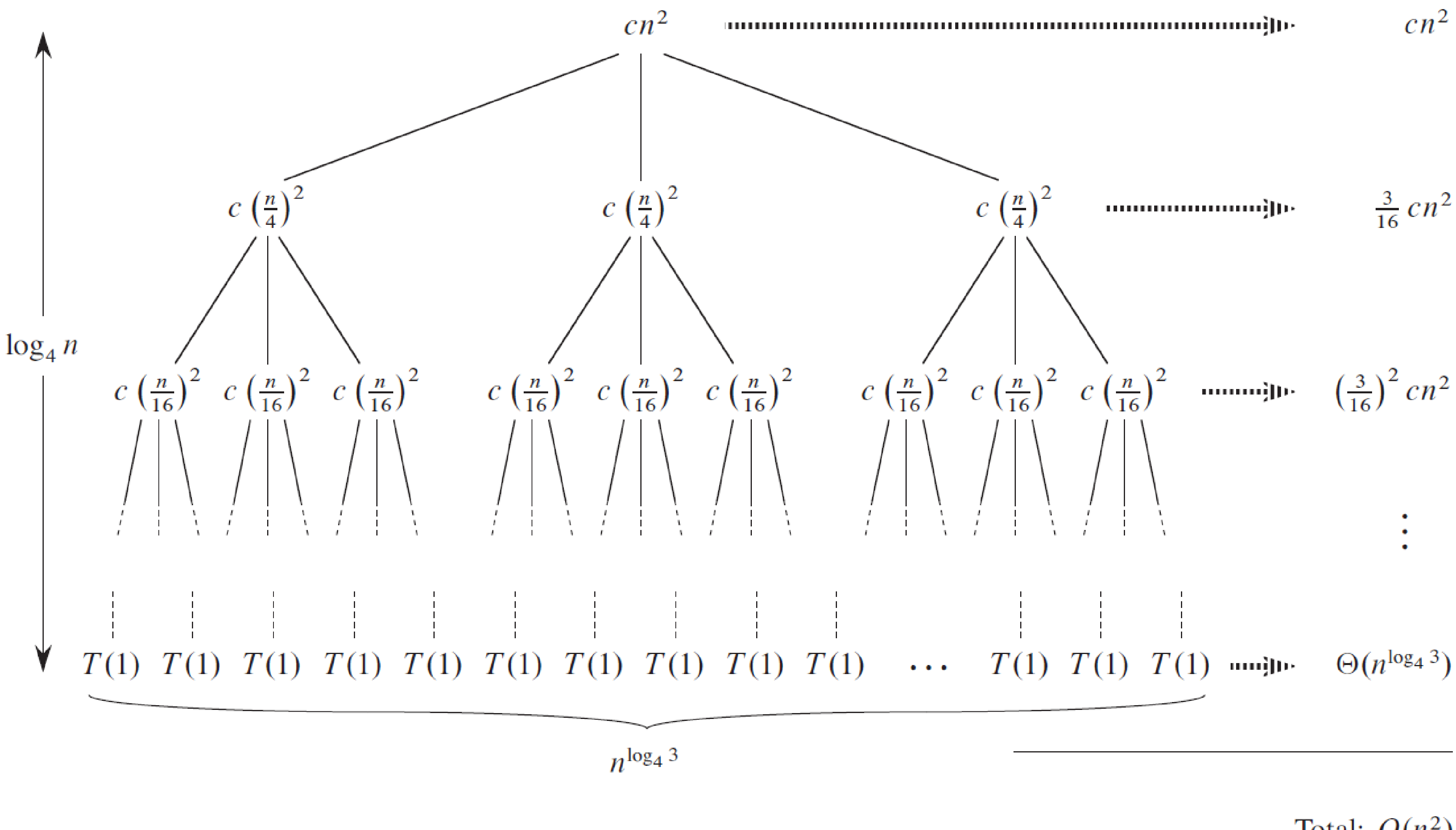
Recursion-tree method

$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$



Recursion-tree method

$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$



(d)

Recursion-tree method

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}) \qquad \text{(by equation (A.5))}\ .
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2)\ .
\end{aligned}
$$

Recursion-tree method

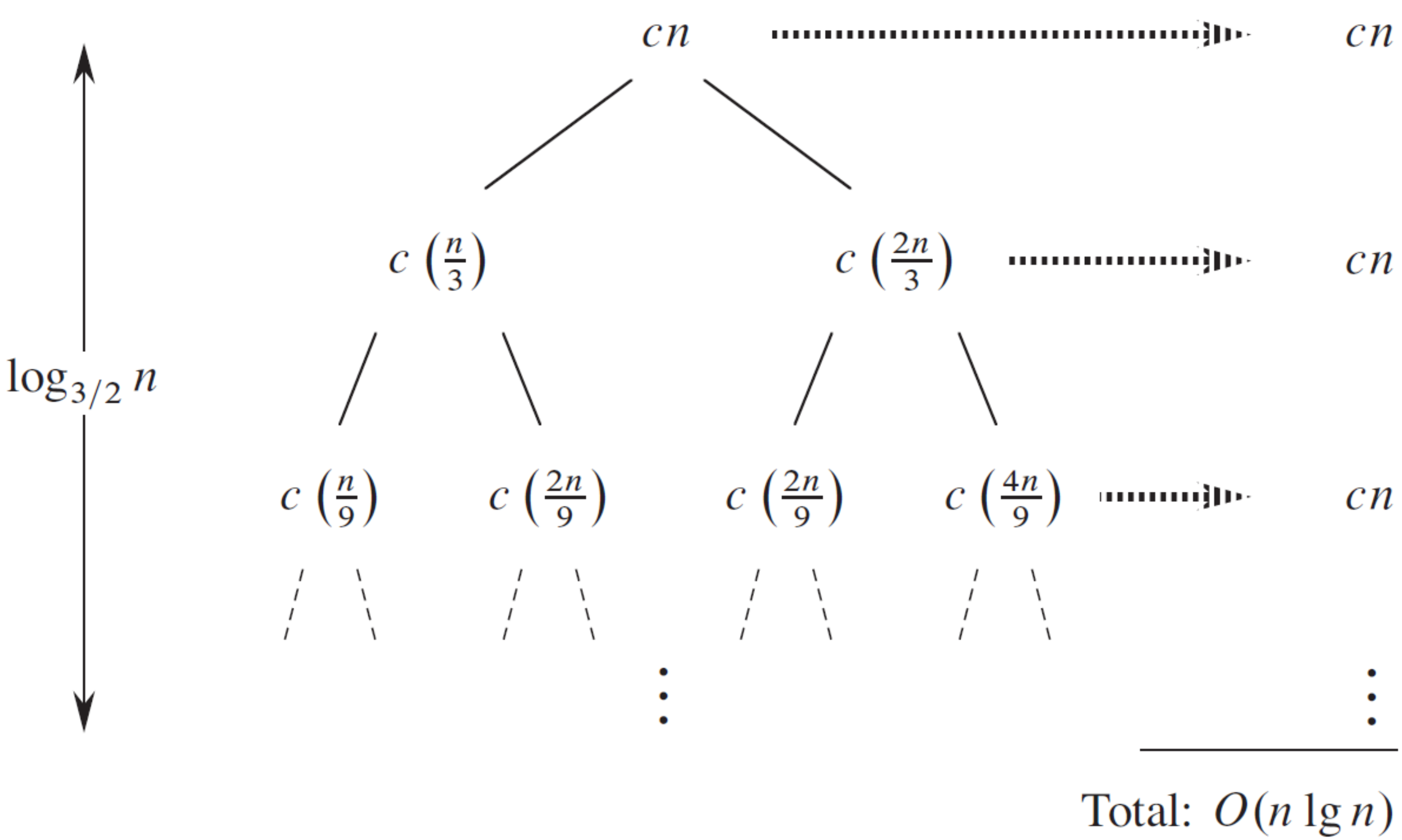
- Verify using substitution method
- Base case: $T(n) = O(n^2)$
- Induction:

$$
\begin{aligned}
T(n) &\le 3T(\lfloor n/4 \rfloor) + cn^2 \\
&\le 3d \lfloor n/4 \rfloor^2 + cn^2 \\
&\le 3d(n/4)^2 + cn^2 \\
&= \frac{3}{16} dn^2 + cn^2 \\
&\le dn^2 ,
\end{aligned}
$$

where the last step holds as long as $d \ge (16/13)c$.

Recursion-tree method

$$T(n) = T(n/3) + T(2n/3) + cn$$

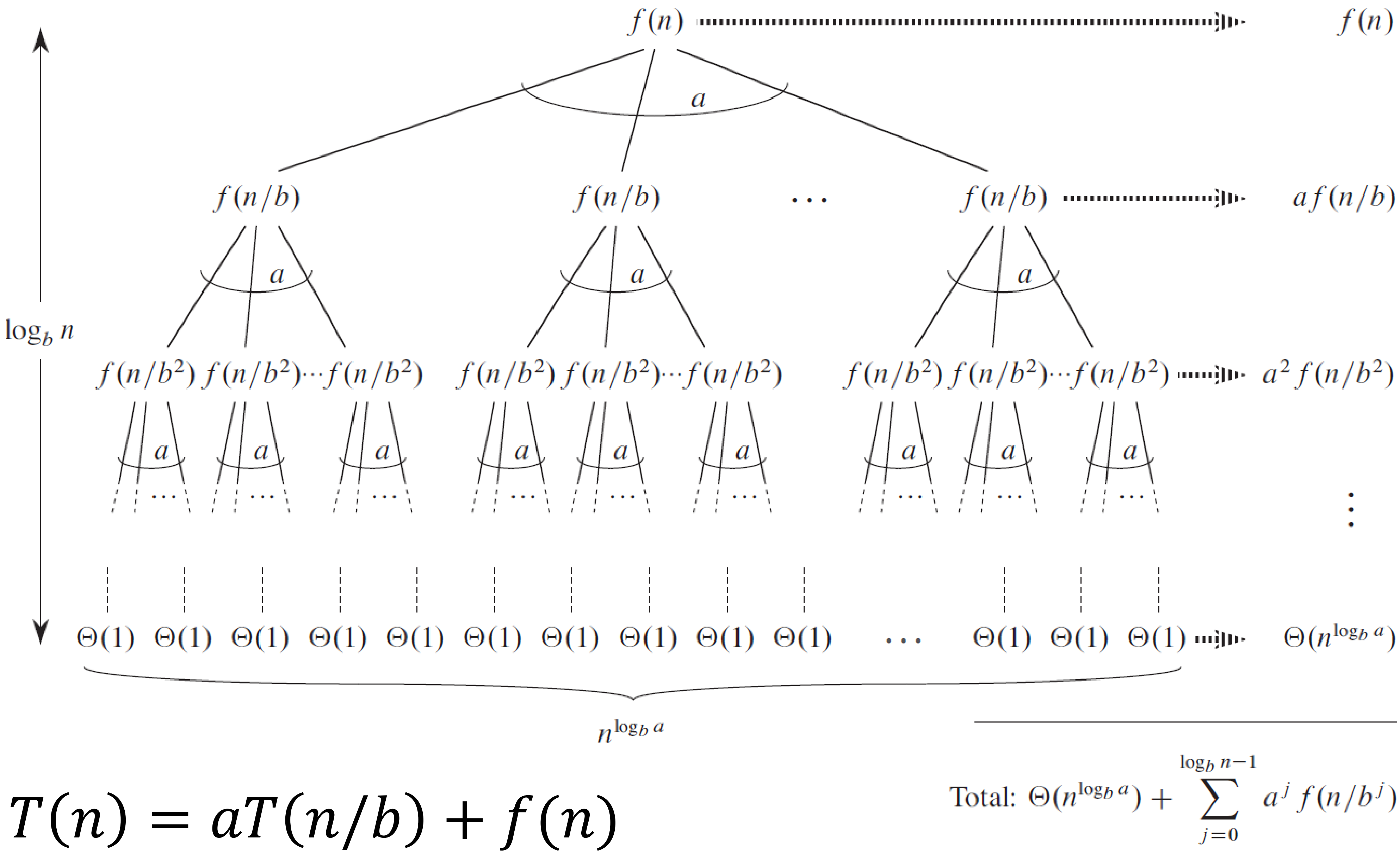


Recursion-tree method

$$
\begin{aligned}
T(n) &\le T(n/3) + T(2n/3) + cn \\
&\le d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn \\
&= (d(n/3)\lg n - d(n/3)\lg 3) \\
&\qquad + (d(2n/3)\lg n - d(2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn \\
&= dn\lg n - dn(\lg 3 - 2/3) + cn \\
&\le dn\lg n\,,
\end{aligned}
$$

as long as $d \ge c/(\lg 3 - (2/3))$

Master method



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

Master method

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ∎

Master method

Lemma 4.2

Let $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a nonnegative function defined on exact powers of b. Define $T(n)$ on exact powers of b by the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ aT(n/b) + f(n) & \text{if } n = b^i , \end{cases}$$

where i is a positive integer. Then

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) . \qquad (4.21)$$

Master method

Lemma 4.3

Let $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a nonnegative function defined on exact powers of b. A function $g(n)$ defined over exact powers of b by

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \tag{4.22}$$

has the following asymptotic bounds for exact powers of b:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $g(n) = O(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $g(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $af(n/b) \leq cf(n)$ for some constant $c < 1$ and for all sufficiently large n, then $g(n) = \Theta(f(n))$.

Proof For case 1, we have $f(n) = O(n^{\log_b a - \epsilon})$, which implies that $f(n/b^j) = O((n/b^j)^{\log_b a - \epsilon})$. Substituting into equation (4.22) yields

$$g(n) = O\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right) . \tag{4.23}$$

Master method

$$\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} = n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j$$

$$= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} (b^\epsilon)^j$$

$$= n^{\log_b a-\epsilon} \left(\frac{b^{\epsilon \log_b n} - 1}{b^\epsilon - 1}\right)$$

$$= n^{\log_b a-\epsilon} \left(\frac{n^\epsilon - 1}{b^\epsilon - 1}\right)$$

$$n^{\log_b a-\epsilon} O(n^\epsilon) = O(n^{\log_b a})$$

$$g(n) = O(n^{\log_b a})$$

Master method

Because case 2 assumes that $f(n) = \Theta(n^{\log_b a})$, we have that $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$. Substituting into equation (4.22) yields

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right) . \tag{4.24}$$

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} &= n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1 \\
&= n^{\log_b a} \log_b n .
\end{aligned}$$

$$\begin{aligned}
g(n) &= \Theta(n^{\log_b a} \log_b n) \\
&= \Theta(n^{\log_b a} \lg n) ,
\end{aligned}$$