CMP(N)302: Design and Analysis of Algorithms

Lecture 01: Sorting & Recurrences

Ahmed Hamdy

Computer Engineering Department

Cairo University

Fall 2019

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 $\langle a_1, a_2, \dots, a_n \rangle$
- **Output:** A permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$
- **Algorithms:**
  - Insertion sort: $2n^2$ instructions
  - Merge sort: $50n \log_2 n$ instructions
  - Which is better??

Insertion vs Merge sort

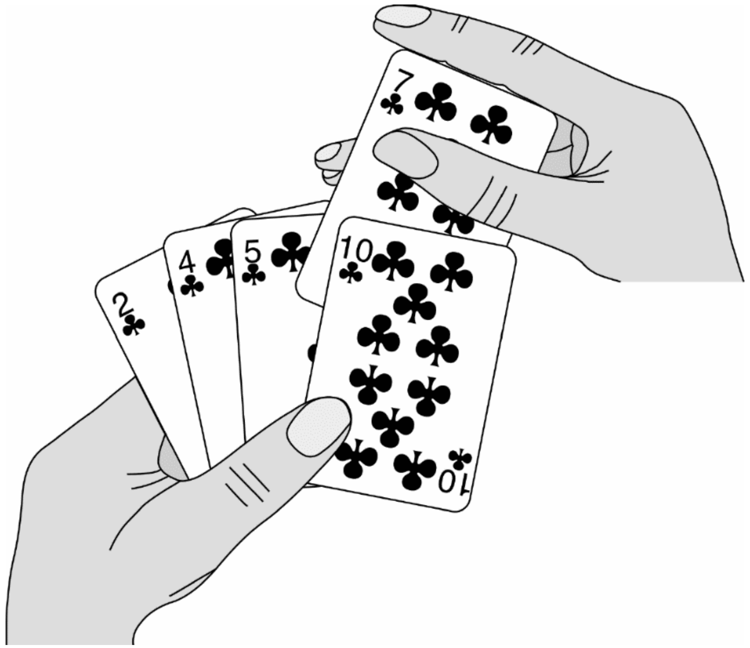
Input size	Insertion sort	Merge sort
n	$2n^2$	$50n \log_2 n$
2	8	100
10	200	1661
100	20,000	33,219
1K	2,000,000	498,289
10K	200,000,000	6,643,856

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

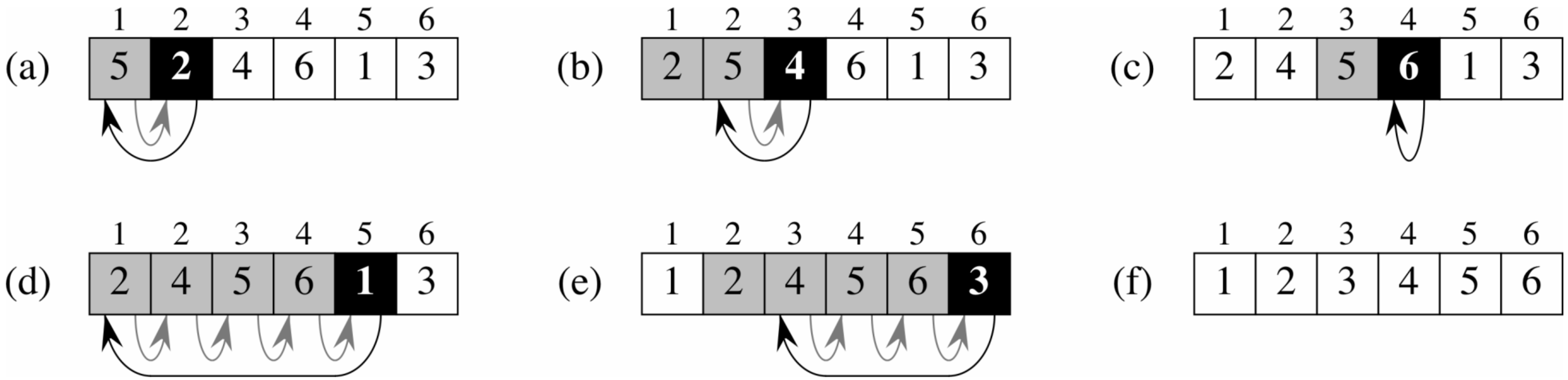
*Merge sort constants can be less than the above, they are just for illustration

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 do you typically sort them??



Insertion sort



https://visualgo.net/en/sorting
https://www.toptal.com/developers/sorting-algorithms

Insertion sort

- How to write it in pseudo-code??

```
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
```

- What if the while loop had “`A[i] >= key`” instead?

Insertion sort analysis

- We used what is called "***incremental approach***".

- Having a sorted subarray $A[1..j-1]$, we insert the new element into its proper place to yield the sorted subarray $A[1..j]$.

Algorithm analysis

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

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

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

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 $\mathrm{T(n)} \downarrow$ as $n \uparrow$: in quadratic running time, linear term is insignificant with large n
  - Care for the case $n \to \infty$, the highest-order term dominates
- Highest-order term represents ***order of growth***
- Neglect constants

Asymptotic analysis

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

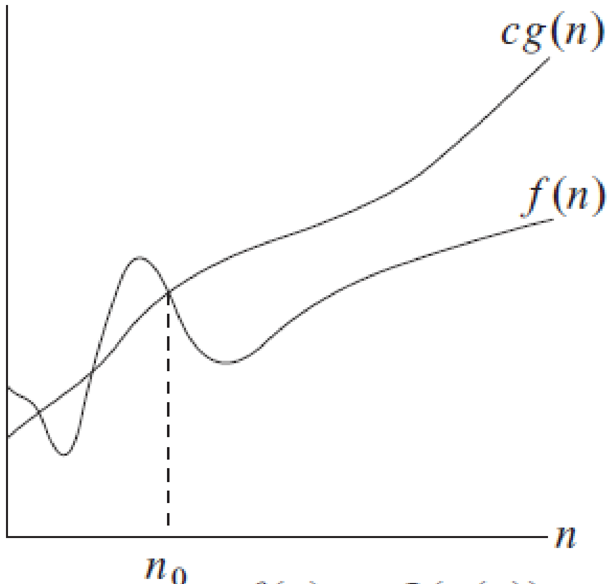
$$O(g(n)) = \{f(n)\!: \text{ there exist positive constants } c \text{ and } n_0 \text{ such that}$$

$$0 \le f(n) \le cg(n)\ for\ all\ n \ge n_0\}$$

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

- May be asymptotically tight; $2n^2 = O(n^2)$

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



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

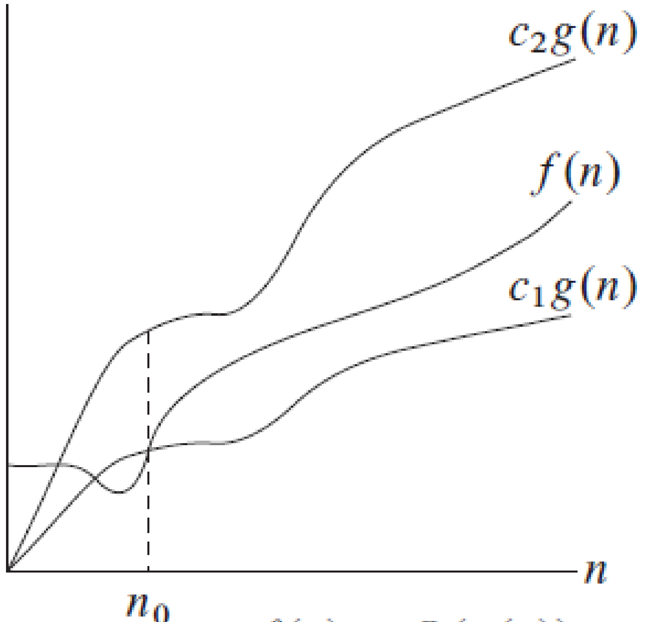
Asymptotic analysis

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

$$\Theta(g(n)) = \{f(n)\text{: there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that}$$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\ for\ all\ n \geq n_0\}$$

- We say that $g(n)$ is an ***asymptotically tight bound*** for $f(n)$
- $10n^2 = \Theta(n^2)$



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

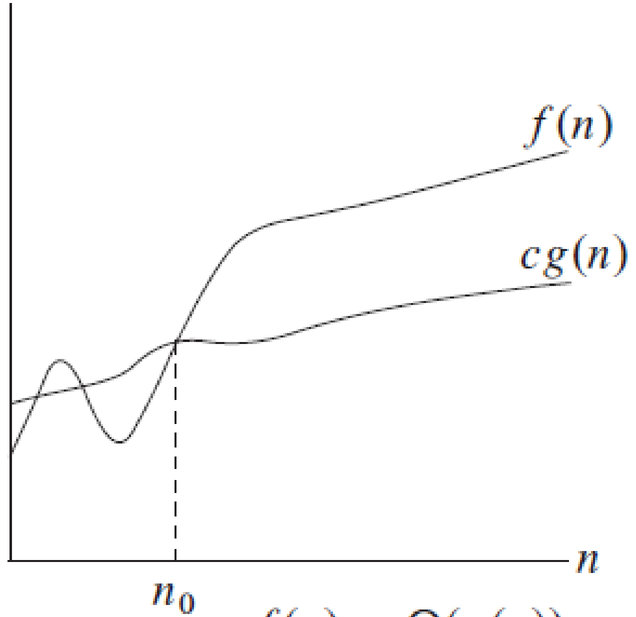
Asymptotic analysis

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

$$\Omega(g(n)) = \{f(n)\text{: there exist positive constants } c \text{ and } n_0 \text{ such that}$$

$$0 \leq cg(n) \leq f(n)\ for\ all\ n \geq n_0\}$$

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



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

- May be asymptotically tight; $2n^2 = \Omega(n^2)$
- May not be asymptotically tight; $2n^3 = \Omega(n^2)$

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

Insertion sort: 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 $t_j = j/2$ which is $\Theta(n^2)$.
- Overall: insertion sort is $O(n^2)$, why??

Recurrences

- What is a recurrence?
  - Simply formulate the time or space complexity of a program as a mathematical function $T(n)$
- Straightforward for easy programs:

```
foo(n) {
    for i = 1..n
        for j = 1..n
            Line1
}
```

bar(n)

- Clearly $T_{foo}(n) = n^2$??
- What if **Line1** is **memcpy(p1, p2 ,n,…)**??

Recurrences

- Let is start nesting functions

```
bar(n) {
    for j = 1..n
        Line1
}
foo(n) {
    for i = 1..n
        bar(n)
}
```

- $T_{bar}(n) = n$ (assuming **Line1** is $O(1)$)
- Then $T_{foo}(n) = nT_{bar}(n) = n^2 = O(n^2)$

Recurrences

- Now to recursion:

```
treeTraversal(root) {
        if  root not NULL {
                print root.data
                treeTraversal(root.left)
                treeTraversal(root.right)
        }
}
```

- $T(n) = 2T\left(\frac{n}{2}\right) + O(1) = O(n)$

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 do they typically do??

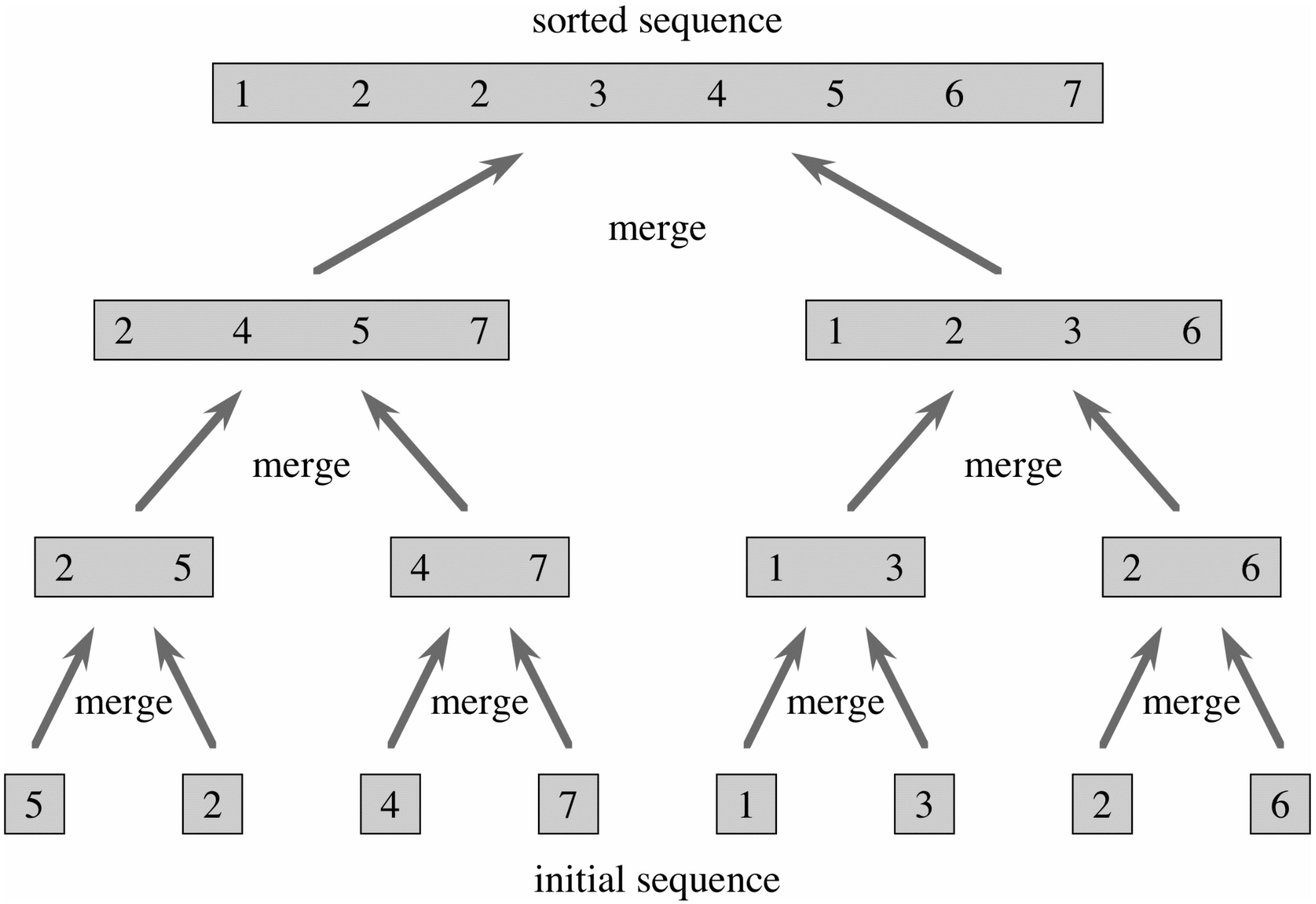
Design strategies

- 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

Merge sort

- ***Divide-and-conquer:***
  - **Divide:** Recursively divide the unsorted n-element sequence into two subsequences down to the lowest level of $n = 2$ elements each
  - **Conquer:** Sort the two subsequences using merge sort up recursively

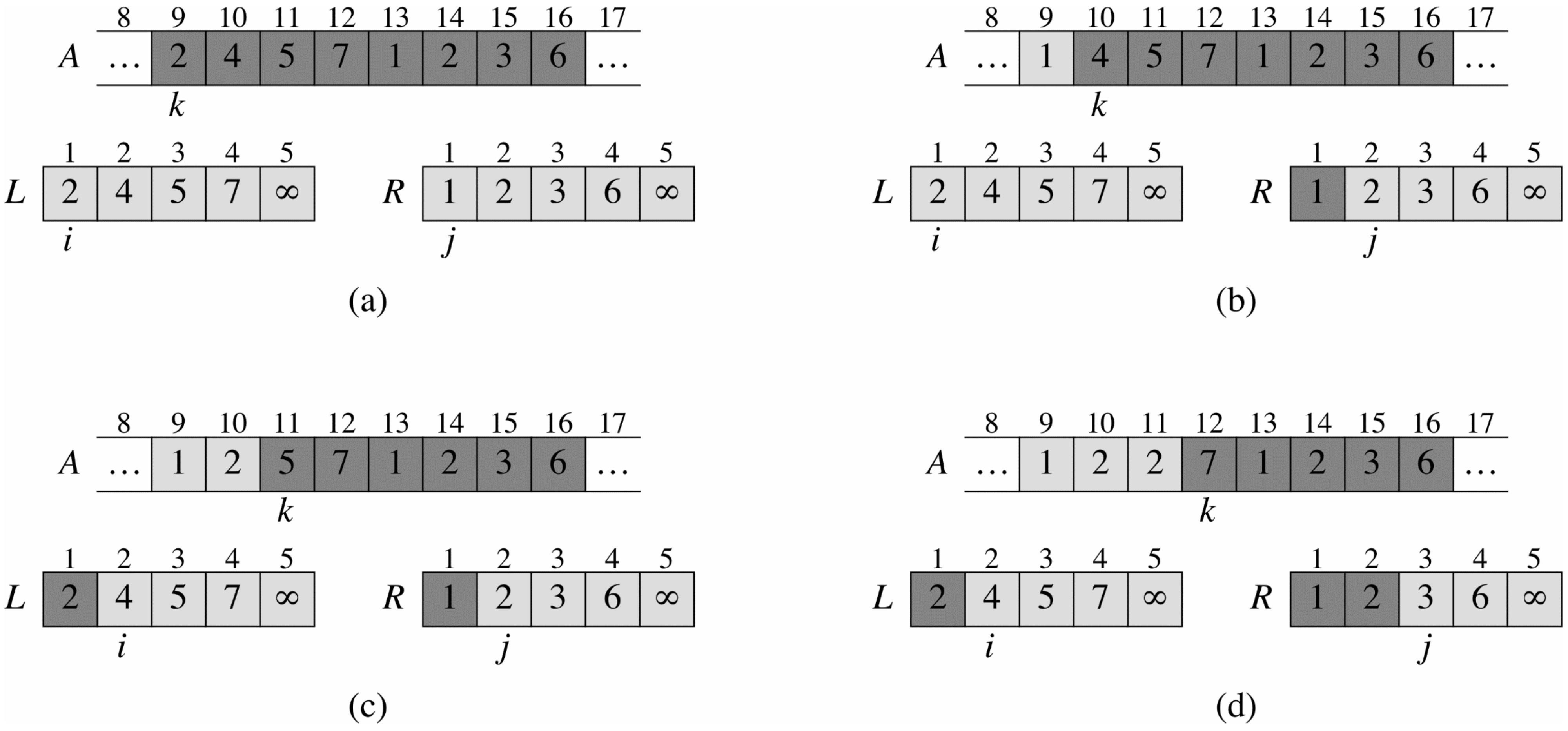
Merge sort



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

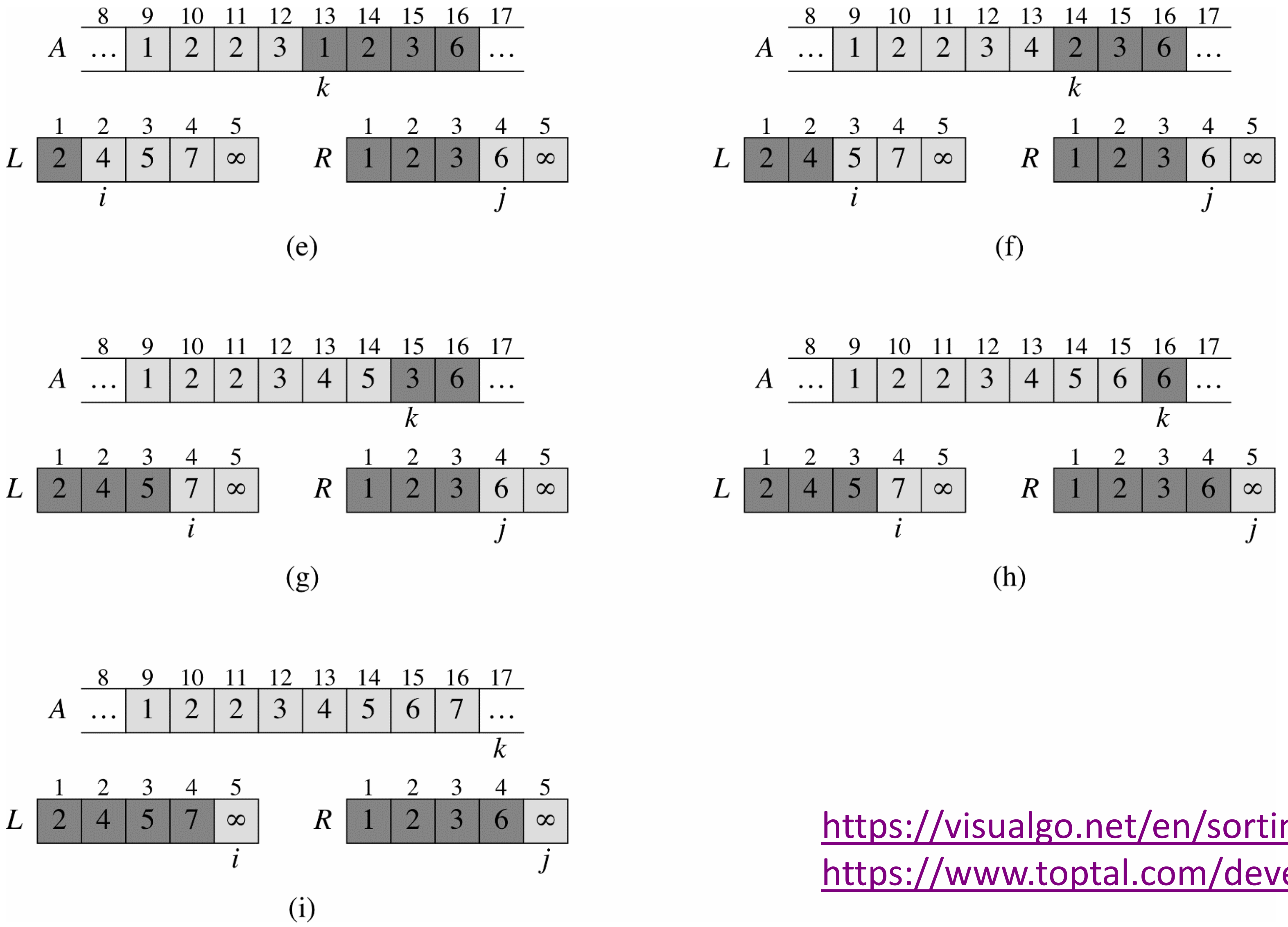
Merge sort

- Merge step



Merge sort

- Merge step (cont.)



https://visualgo.net/en/sorting
https://www.toptal.com/developers/sorting-algorithms

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 = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

- Initial call:

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

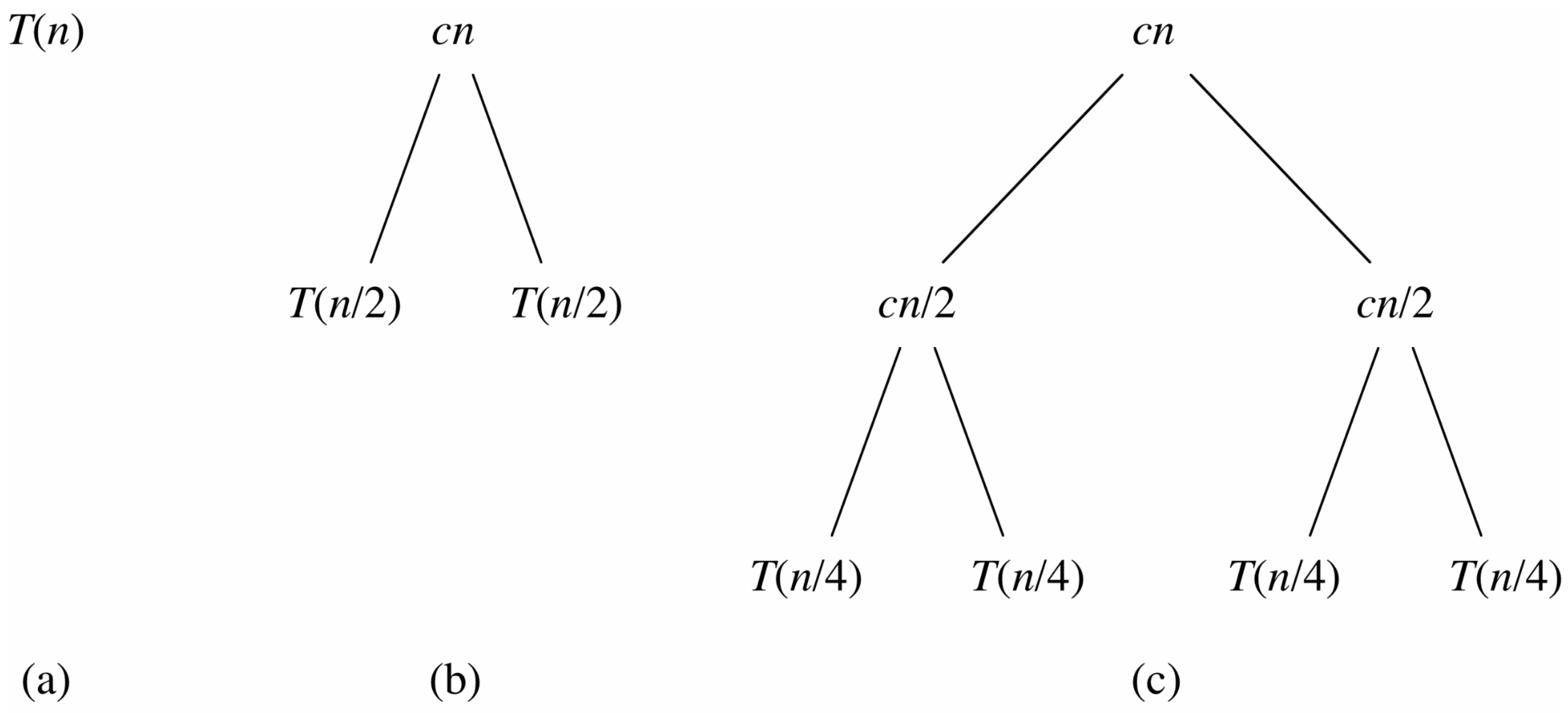
Merge sort analysis

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

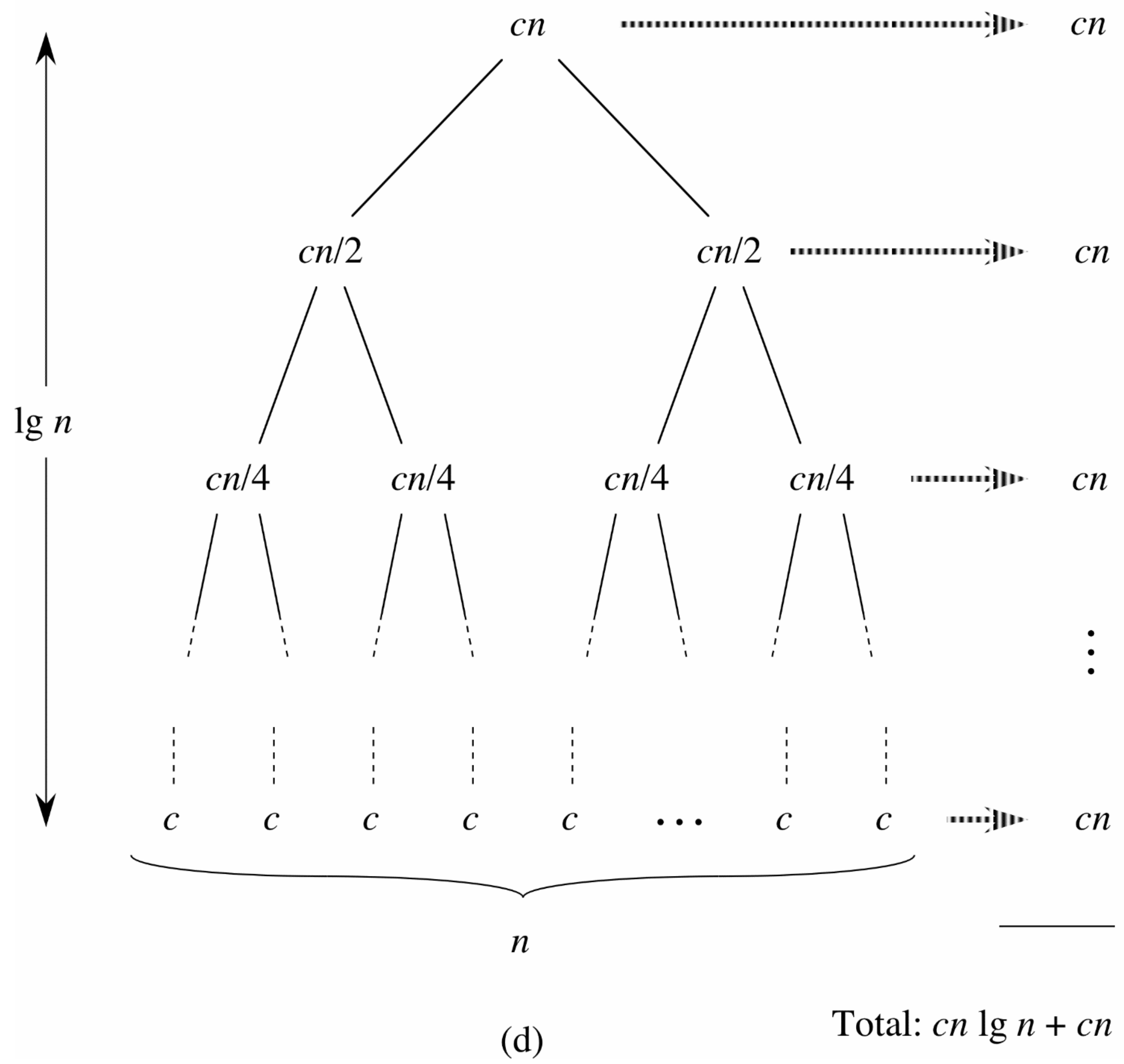
- **Divide:** just computes the middle of the subarray. Thus, $D(n) = \Theta(1)$.
- **Conquer:** recursively solve two subproblems, each of size $n/2$, which contributes $2T\ (n/2)$ to the running time.
- **Combine:** MERGE procedure on an n-element subarray takes time $\Theta(n)$, and so $C(n) = \Theta(n)$.

Merge sort analysis

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



Merge sort analysis



Space complexity

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

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

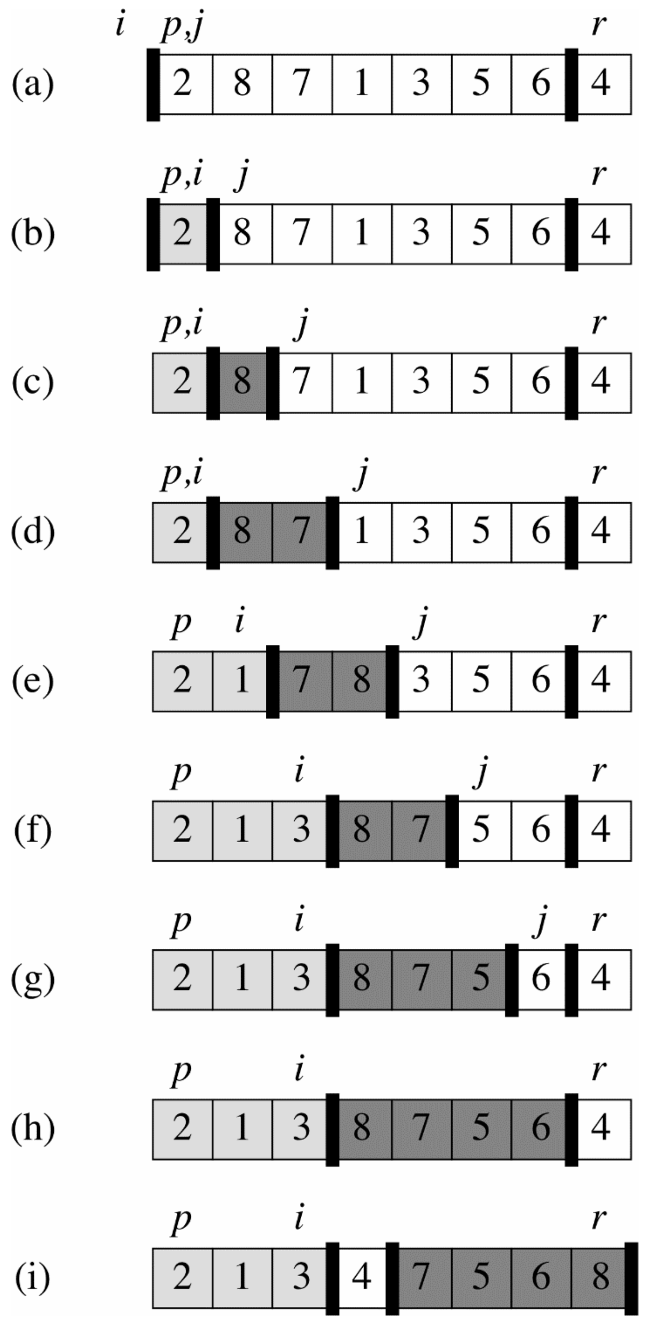
Insertion vs Merge sort

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

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

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

Quicksort



(a) i p,j r 2 8 7 1 3 5 6 4
(b) p,i j r 2 8 7 1 3 5 6 4
(c) p,i j r 2 8 7 1 3 5 6 4
(d) p,i j r 2 8 7 1 3 5 6 4
(e) p i j r 2 1 7 8 3 5 6 4
(f) p i j r 2 1 3 8 7 5 6 4
(g) p i j r 2 1 3 8 7 5 6 4
(h) p i r 2 1 3 8 7 5 6 4
(i) p i r 2 1 3 4 7 5 6 8

Quicksort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

- Stable?

Quicksort

Performance

- Worst case partitioning:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

$$T(n) = \Theta(n^2)$$

- Best case partitioning:

$$T(n) = 2T(n/2) + \Theta(n)$$

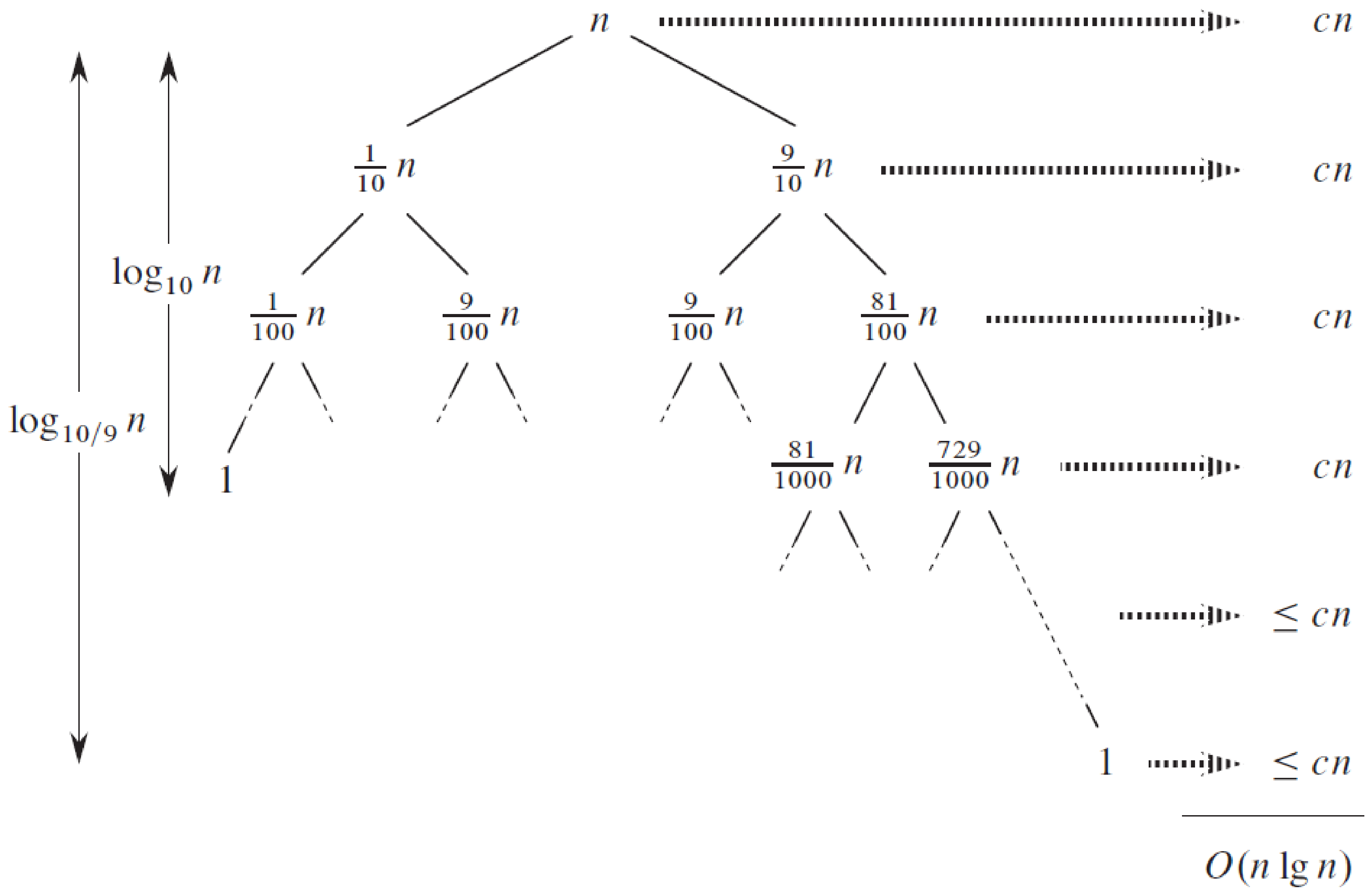
$$T(n) = \Theta(n \lg n)$$

Quicksort

Performance

- Balanced partitioning:

$$T(n) = T(9n/10) + T(n/10) + cn$$



Randomized quicksort

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q - 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

- Expected running time: $O(n \log n)$ when element values are distinct

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

    $$T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor).$$

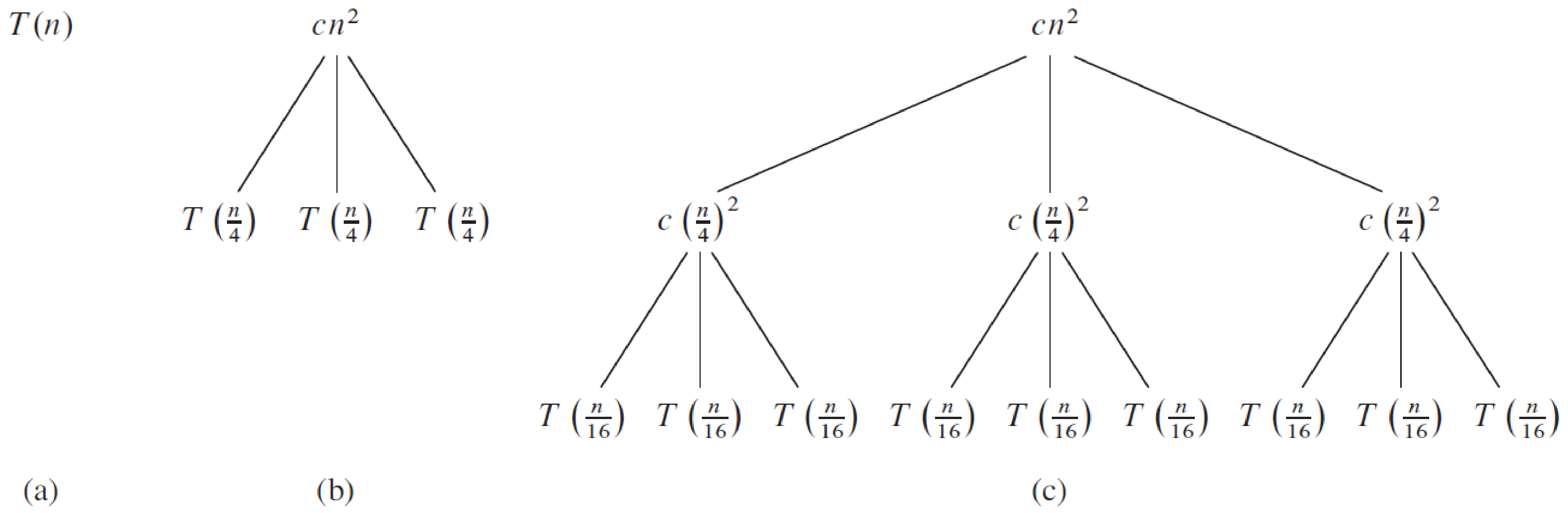
  - Induction:

    $$\begin{aligned} T(n) &\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n , \end{aligned}$$

    - Holds for $c \geq 1$

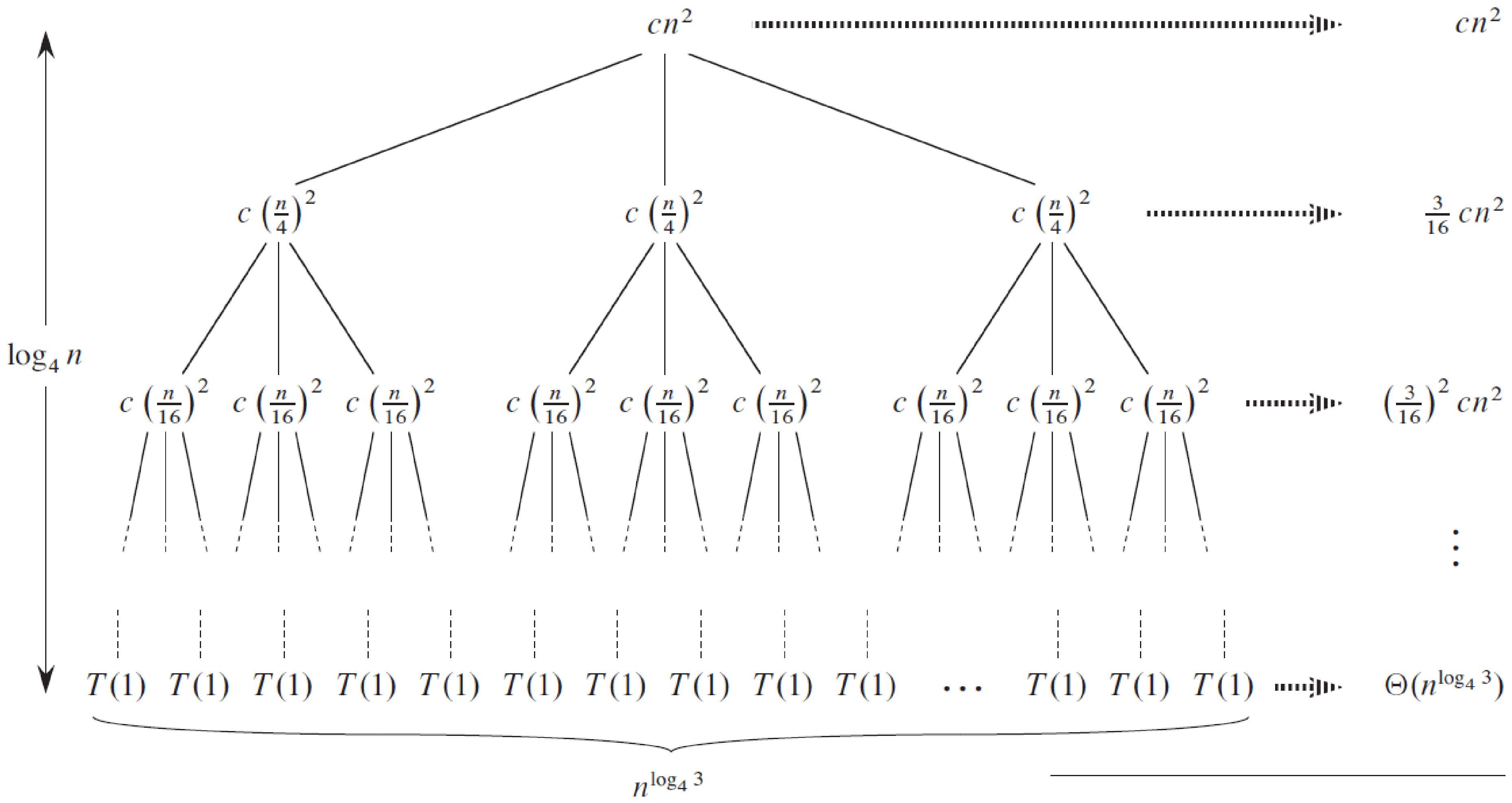
Recursion-tree method

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



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

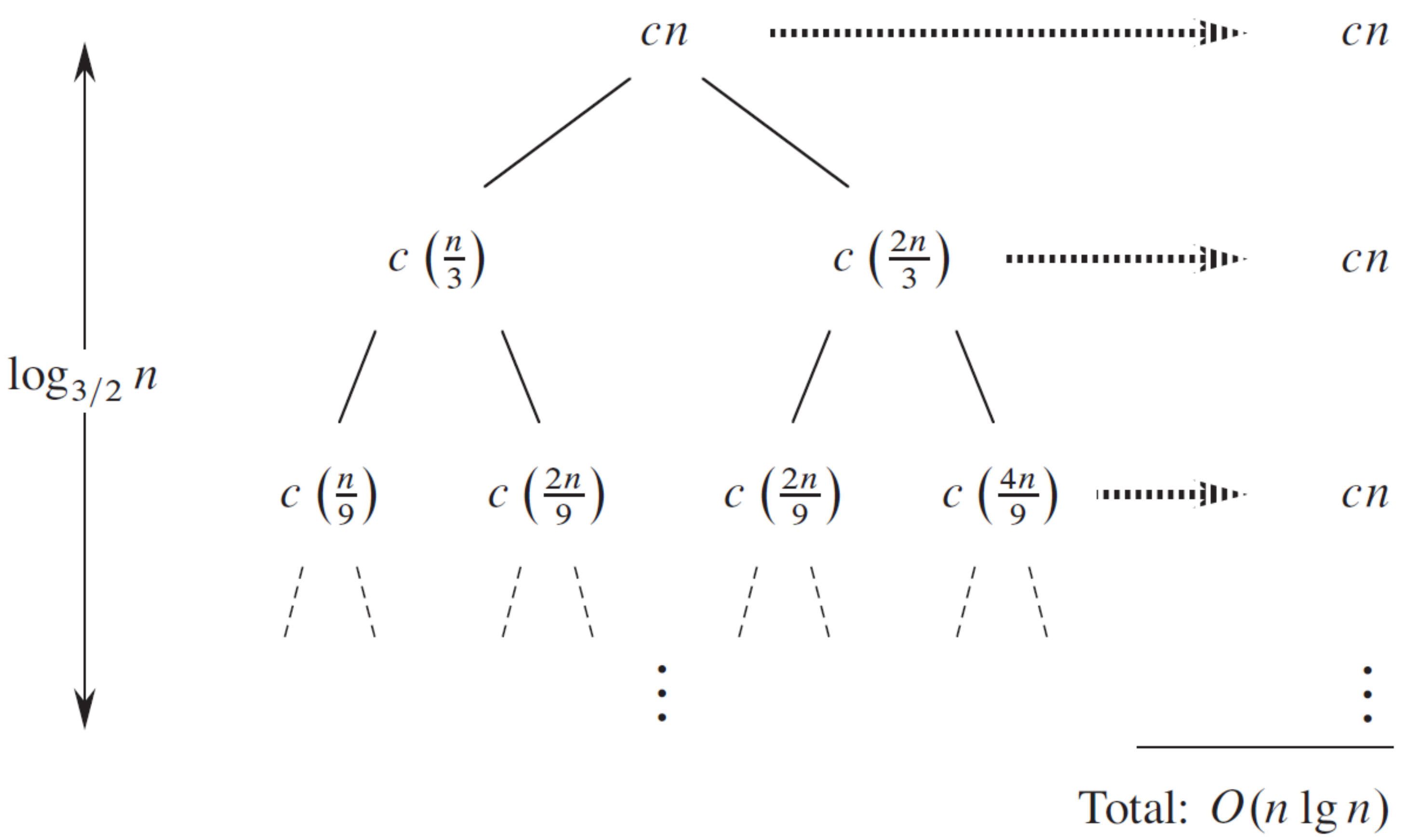
- Verify using substitution method
- Base case: $T(n) = O(n^2)$
- Induction:

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

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

Recursion-tree method

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



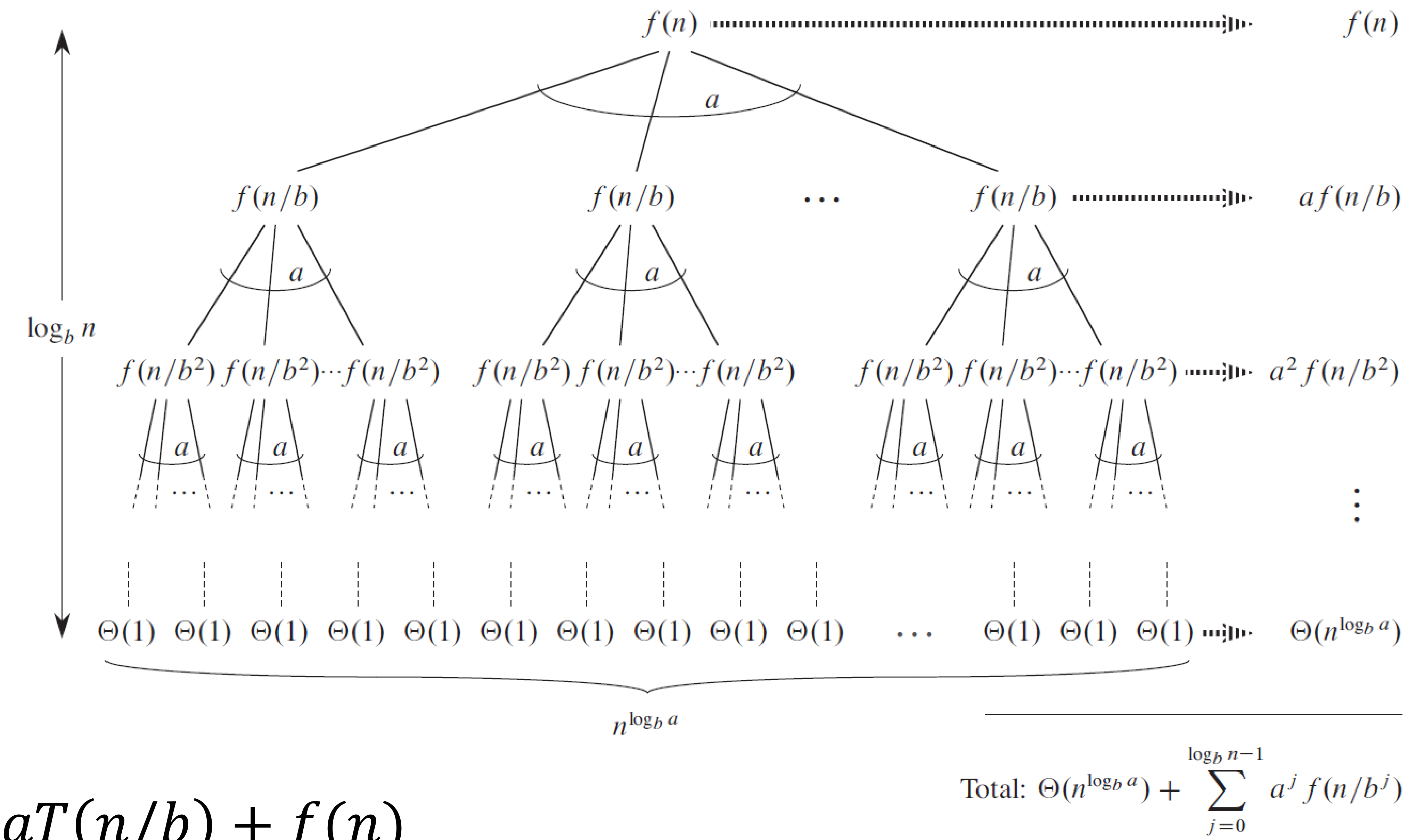
Recursion-tree method

- Verify using substitution 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

$$T(n) = 9T(n/3) + n\ .$$

For this recurrence, we have $a = 9$, $b = 3$, $f(n) = n$, and thus we have that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9-\epsilon})$, where $\epsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta(n^2)$.

Now consider

$$T(n) = T(2n/3) + 1,$$

in which $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and thus the solution to the recurrence is $T(n) = \Theta(\lg n)$.

For the recurrence

$$T(n) = 3T(n/4) + n \lg n\ ,$$

we have $a = 3$, $b = 4$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = \Omega(n^{\log_4 3+\epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n, we have that $af(n/b) = 3(n/4)\lg(n/4) \le (3/4)n \lg n = cf(n)$ for $c = 3/4$. Consequently, by case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

Master method

The master method does not apply to the recurrence

$$T(n) = 2T(n/2) + n \lg n ,$$

even though it appears to have the proper form: $a = 2$, $b = 2$, $f(n) = n \lg n$, and $n^{\log_b a} = n$. You might mistakenly think that case 3 should apply, since $f(n) = n \lg n$ is asymptotically larger than $n^{\log_b a} = n$. The problem is that it is not *polynomially* larger. The ratio $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ is asymptotically less than n^ϵ for any positive constant ϵ. Consequently, the recurrence falls into the gap between case 2 and case 3. (See Exercise 4.6-2 for a solution.)

Let's use the master method to solve the recurrences we saw in Sections 4.1 and 4.2. Recurrence (4.7),

$$T(n) = 2T(n/2) + \Theta(n) ,$$

characterizes the running times of the divide-and-conquer algorithm for both the maximum-subarray problem and merge sort. (As is our practice, we omit stating the base case in the recurrence.) Here, we have $a = 2$, $b = 2$, $f(n) = \Theta(n)$, and thus we have that $n^{\log_b a} = n^{\log_2 2} = n$. Case 2 applies, since $f(n) = \Theta(n)$, and so we have the solution $T(n) = \Theta(n \lg n)$.

Recurrence (4.17),

$$T(n) = 8T(n/2) + \Theta(n^2) ,$$

describes the running time of the first divide-and-conquer algorithm that we saw for matrix multiplication. Now we have $a = 8$, $b = 2$, and $f(n) = \Theta(n^2)$, and so $n^{\log_b a} = n^{\log_2 8} = n^3$. Since n^3 is polynomially larger than $f(n)$ (that is, $f(n) = O(n^{3-\epsilon})$ for $\epsilon = 1$), case 1 applies, and $T(n) = \Theta(n^3)$.

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

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

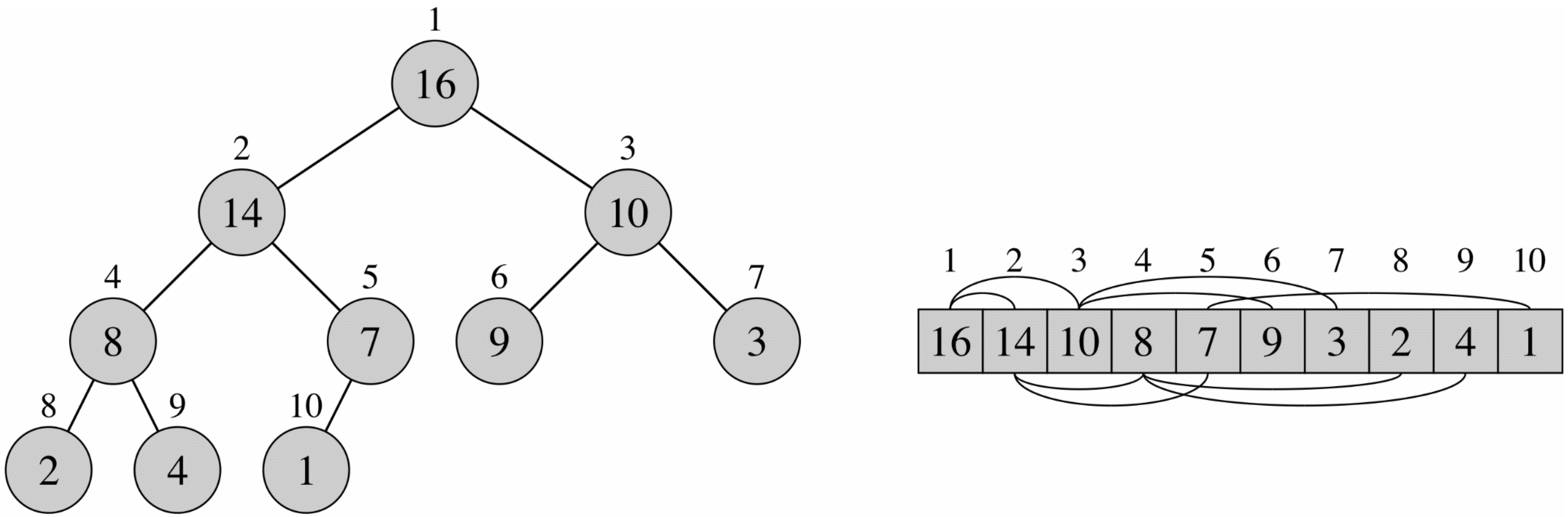
$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} &= n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1 \\
&= n^{\log_b a} \log_b n .
\end{aligned}$$

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

Selection sort

- Loop at all positions of the array (i):
  - Find the minimum element in subarray $A[i..n]$
  - $A[i] = \min(A[i..n])$
- What is the complexity??
- Stable?

Heaps



(a) (b)

PARENT(i)

return $\lfloor i/2 \rfloor$

LEFT(i)

return $2i$

RIGHT(i)

return $2i + 1$

Heaps

Optimizations of heap operations

- $2i$ computed as shift left
- $2i + 1$ computed as shift left and adding 1 / ORing 1
- $\lfloor i/2 \rfloor$ computed as shift right
- Implement heap operations (parent, left, right) as macros or inline functions

Heaps

- Max-heap:

$$A[\textsc{Parent}(i)] \geq A[i]$$

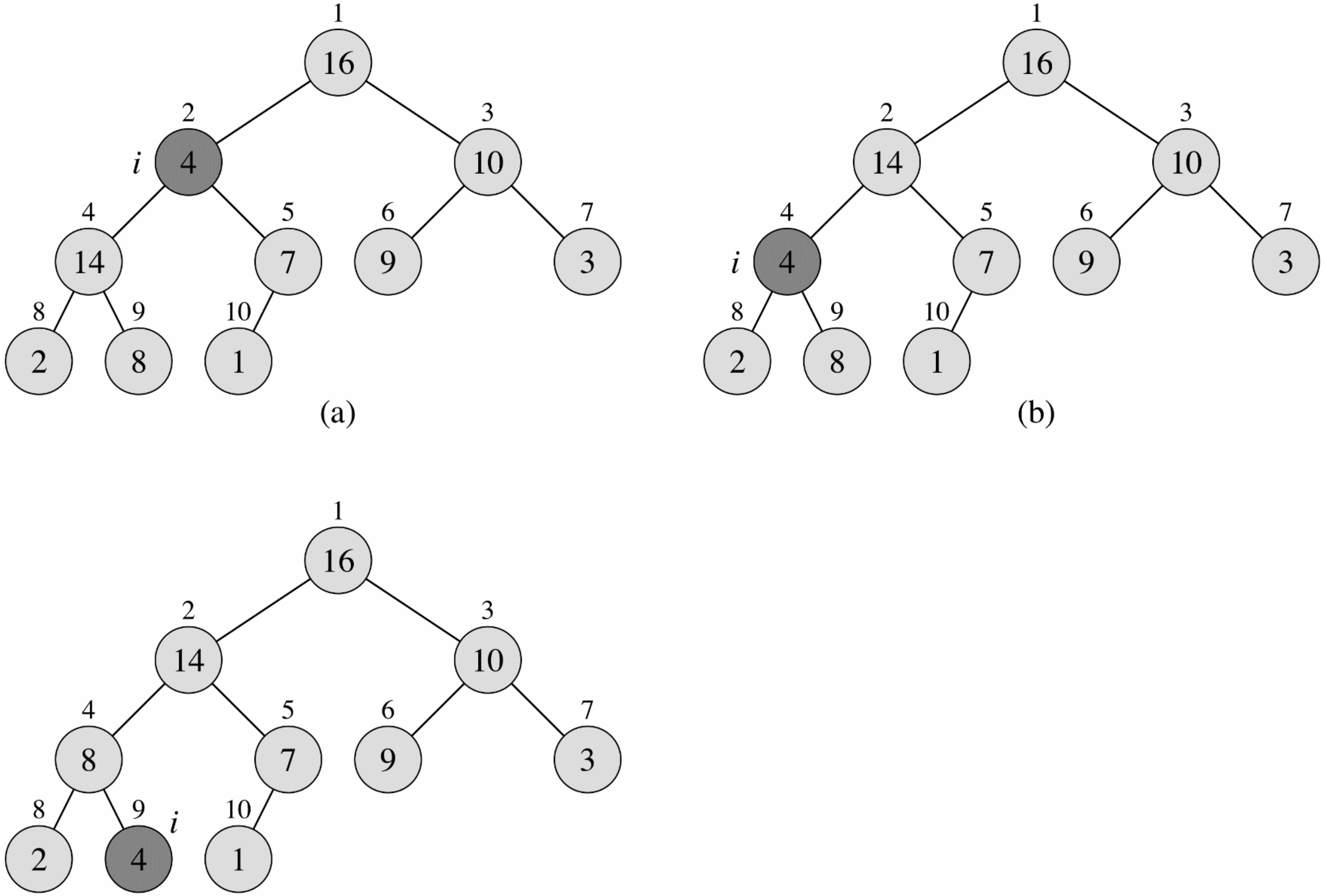
- Min-heap:

$$A[\textsc{Parent}(i)] \leq A[i]$$

- Height: $\Theta(\log n)$
- Operations: $O(\log n)$

Heaps

Max-Heapify



(a) (b) (c)

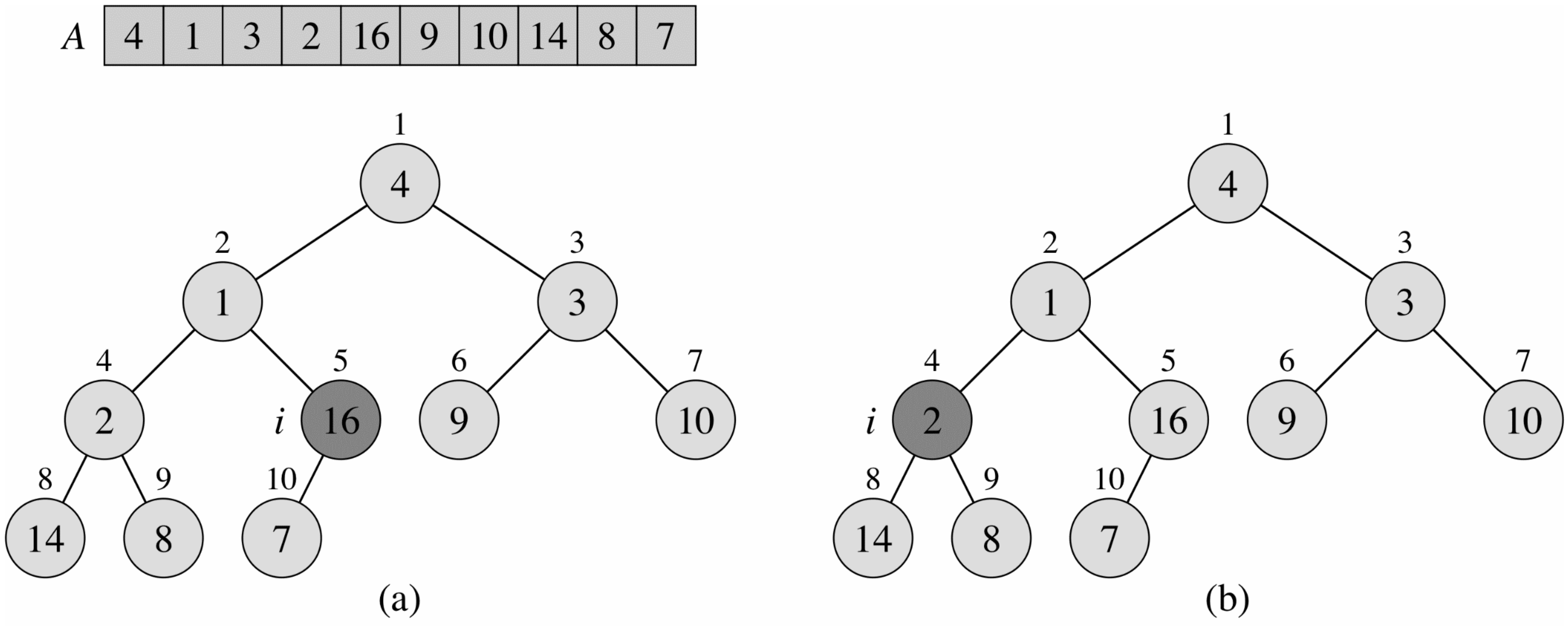
Heaps

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

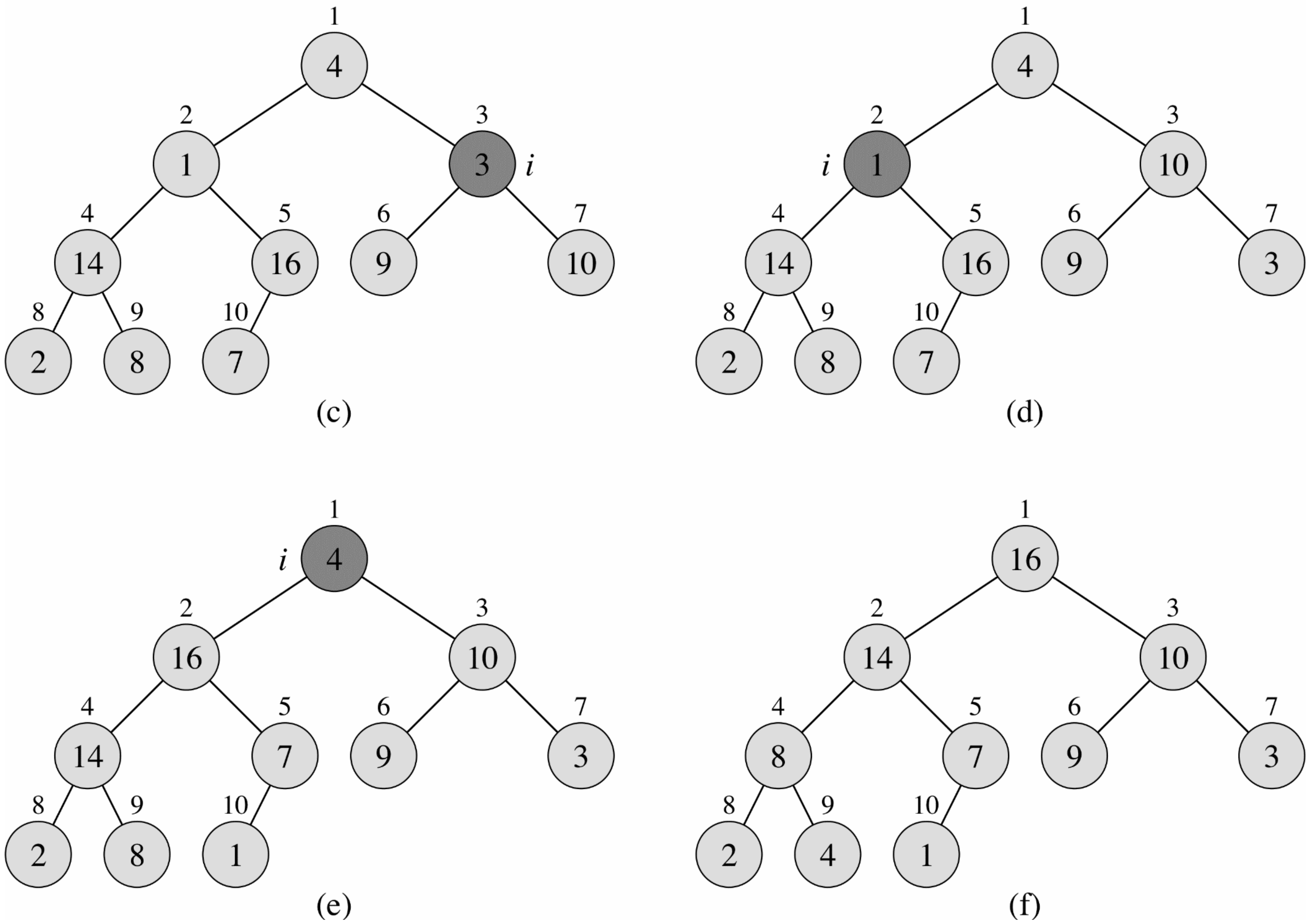
- Max-Heapify complexity: $O(\log n)$

Heaps

- How to organize an array to be a heap?



Heaps



1
4
2
1
3
3
i
4
14
5
16
6
9
7
10
8
2
9
8
10
7
(c)
1
4
2
i
1
3
10
4
14
5
16
6
9
7
3
8
2
9
8
10
7
(d)
1
i
4
2
16
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1
(e)
1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
4
10
1
(f)

Heaps

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

- Build-Max-Heap Complexity:
  - $O(n)$ calls to Max-Heapify
  - Thus overall complexity of Build-Max-Heap: $O(n \log n)$
  - Is this tight bound??

Heaps

- Height of n-element heap: $\lfloor \log n \rfloor$
- Number of nodes at height h: $\left\lceil \frac{n}{2^{h+1}} \right\rceil$
- Build-Max-Heap is bounded by

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

- Using

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \quad \Rightarrow \quad \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2\,.$$

for $|x| < 1$.

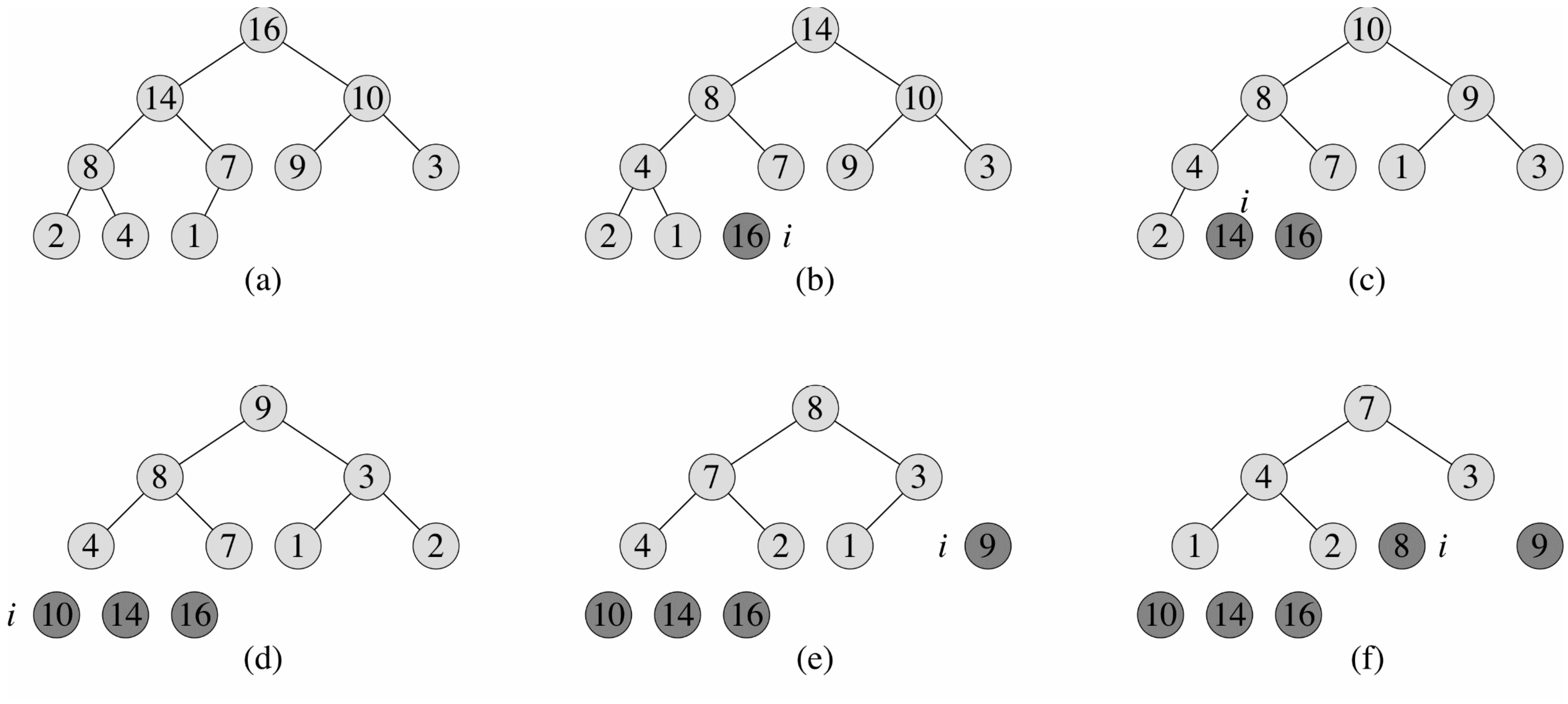
Heaps

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

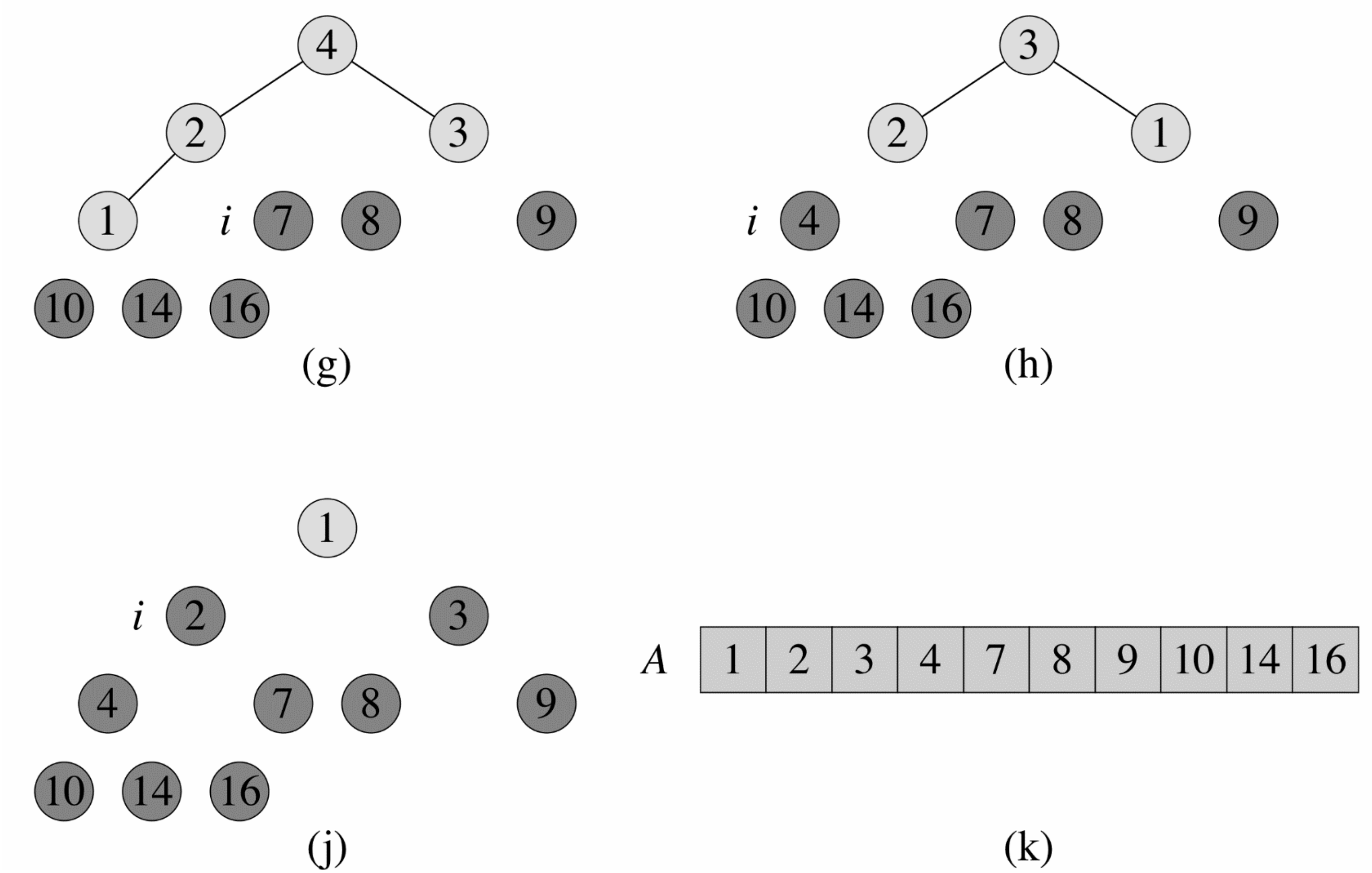
Becomes

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n) . \end{aligned}$$

Heapsort


16
14
10
8
7
9
3
2
4
1
(a)
14
8
10
4
7
9
3
2
1
16
i
(b)
10
8
9
4
7
1
3
i
2
14
16
(c)
9
8
3
4
7
1
2
i
10
14
16
(d)
8
7
3
4
2
1
i
9
10
14
16
(e)
7
4
3
1
2
8
i
9
10
14
16
(f)

Heapsort


4
2
3
1
i
7
8
9
10
14
16
(g)
3
2
1
i
4
7
8
9
10
14
16
(h)
2
1
3
i
4
7
8
9
10
14
16
(i)
1
i
2
3
4
7
8
9
10
14
16
(j)
A
1
2
3
4
7
8
9
10
14
16
(k)

Heapsort

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

Complexity

- $O(n)$ calls to Max-Heapify
- Thus overall complexity of Heapsort: $O(n \log n)$

Priority queues

- Useful in applications where priority is the criteria for selection
- Example, scheduling jobs
- Max-priority or min-priority queues

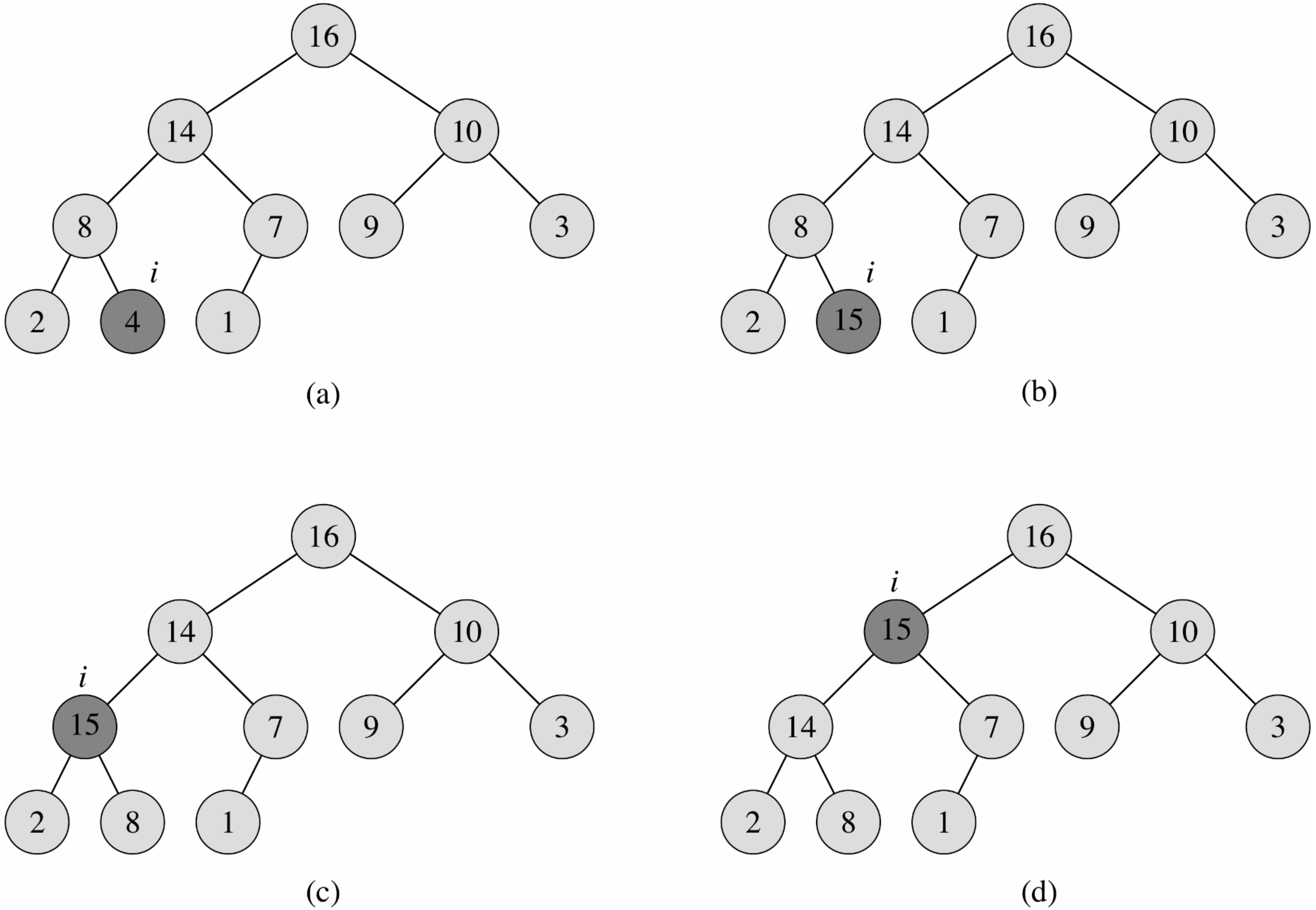
Priority queues

```
HEAP-MAXIMUM(A)
1  return A[1]
```

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2      error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size - 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

Priority queue

- Heap-Increase-Key



Priority queues

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

```
MAX-HEAP-INSERT(A, key)
A.heap-size = A.heap-size + 1
A[A.heap-size] = -∞
HEAP-INCREASE-KEY(A, A.heap-size, key)
```

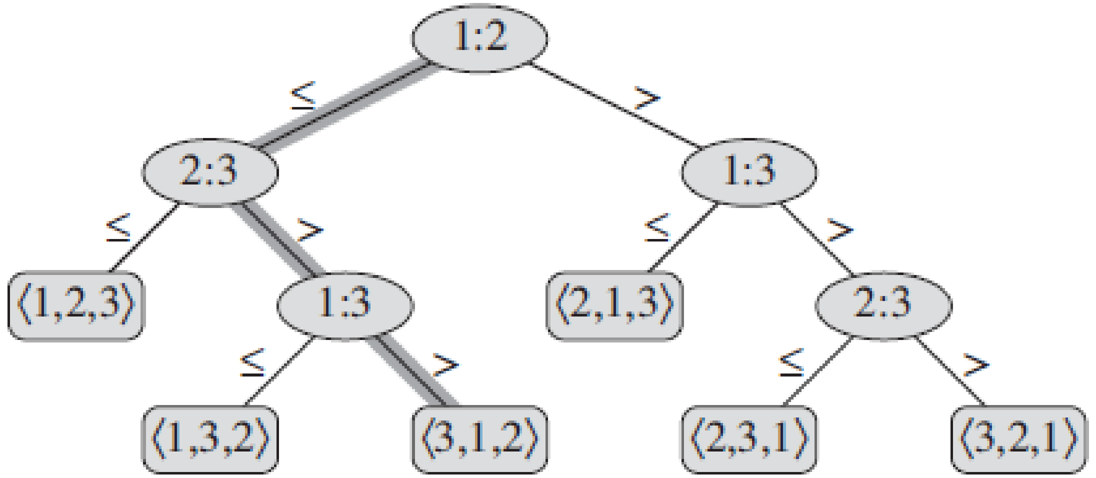
Comparison sort

- Insertion-sort, mergesort, heapsort and quicksort all use comparisons to gain order

- Can we obtain a lower bound for any comparison sort?

Comparison sort

- Decision tree



- Number of leaves in decision tree: $\geq n!$
- Number of leaves in binary tree: 2^h

$$n! \leq l \leq 2^h$$

$$\begin{aligned} h &\geq \lg(n!) \\ &= \Omega(n \lg n) \end{aligned}$$

Comparison sort

- Mergesort and heapsort are asymptotically optimal comparison sorts

- Both have upper bound of $O(n \log n)$

- Quicksort isn’t, why?

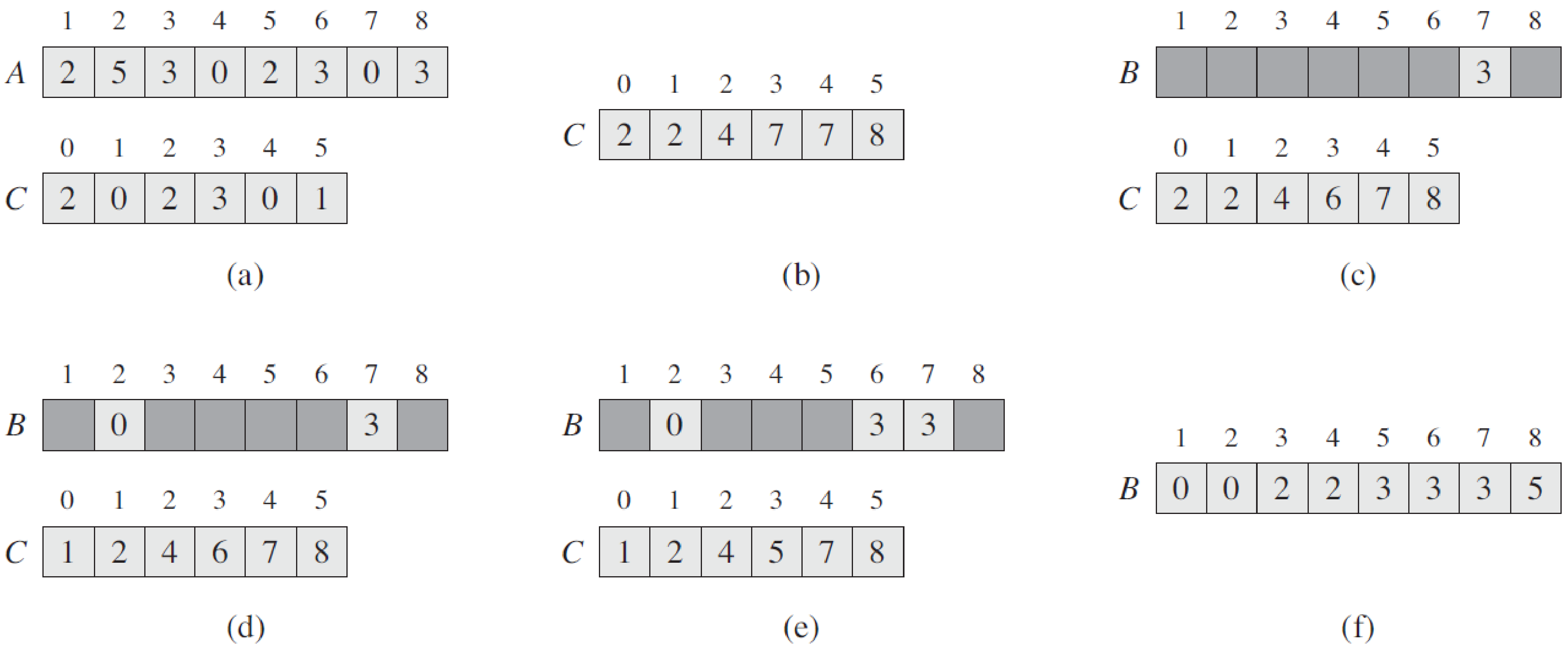
Linear-time sorting

- If there is an assumption about the data, we can use to optimize sorting.

General rule in Computer Engineering

- For integer data, counting sorting, radix sorting and bucket sorting are useful techniques.

Counting sort



Counting sort

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i − 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1
```

When $k = O(n)$, the sort runs in $\Theta(n)$ time.

Counting sort

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

- Complexity: $\Theta(n+k)$
- Typically used when $k = O(n)$, complexity $\Theta(n)$
- Stable

Sorting problem

- Sort a set of files according to their timestamps in ascending order??

- Assume just year, month, day.

Year	Month	Day
2010	2	5
2010	3	4
2005	3	4
2005	2	5
2000	2	5
2000	3	4

Radix sort

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

```
RADIX-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```

- Use counting sort to sort each digit
- Complexity: $\Theta(d(n+k))$

Radix sort

- What is the optimal d to use given n b-bit numbers?

- Ask it in a different way, what is the optimal r-bits out of the b-bits to use for every digit ($d = \lceil b/r \rceil$)?

$$T(n, b) = \Theta(d(n + k)) = \Theta\left(\frac{b}{r}(n + 2^r)\right)$$

Radix sort

$$T(n,b) = \Theta(d(n+k)) = \Theta\left(\frac{b}{r}(n+2^r)\right)$$

- If $b < \lfloor \log n \rfloor$, $(n+2^r) = \Theta(n)$.
- It is best also to choose $\frac{b}{r} = 1$, which translates to $\boldsymbol{r = b}$.

Radix sort

$$T(n,b) = \Theta(d(n+k)) = \Theta\left(\frac{b}{r}(n+2^r)\right)$$

If $b \geq \lfloor \log n \rfloor$:

- As r decreases ($2^r \ll n$), $\frac{b}{r}$ increases and $(n+2^r)$ stays the same as $\Theta(n)$
- As r increases ($2^r \gg n$), $\frac{b}{r}$ decreases but $(n+2^r)$ increases much more than the decrease of $\frac{b}{r}$.
- Logically, $(\mathrm{n}+2^r) = \Theta(n)$, then $\boldsymbol{r = \lfloor \log n \rfloor}$

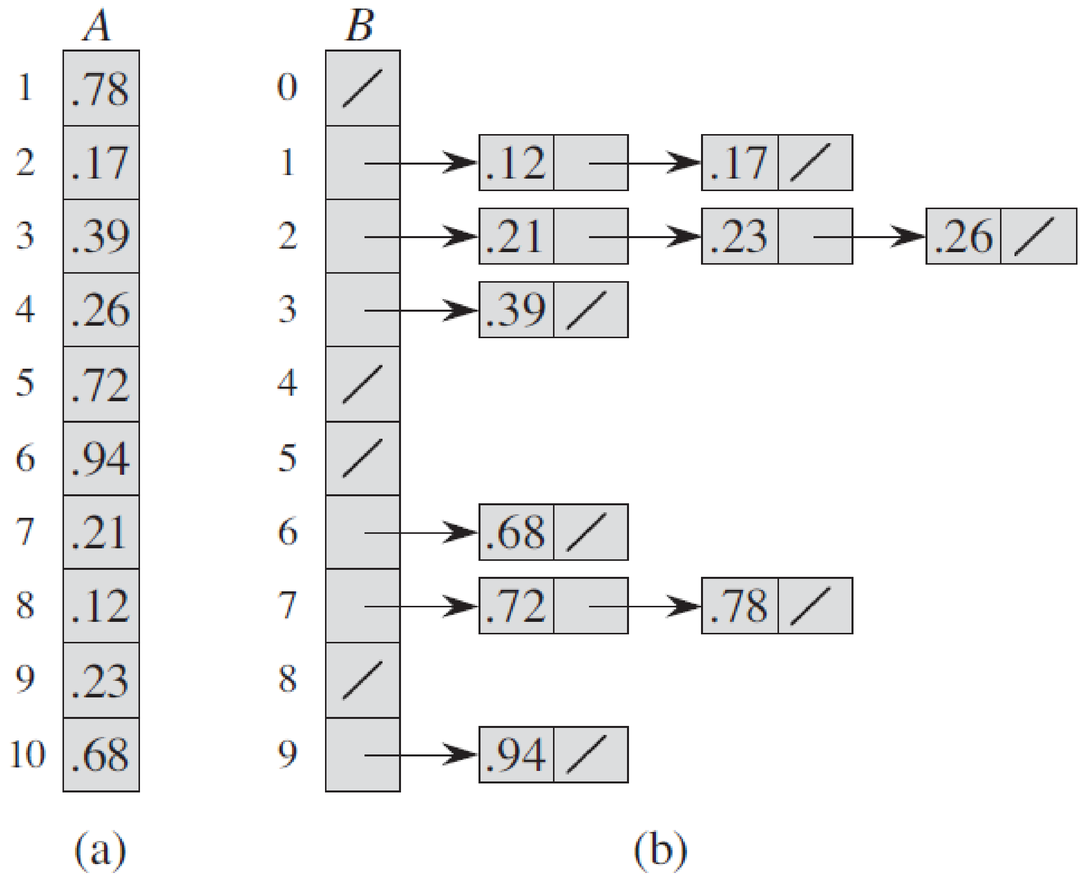
$$T(n,b) = \Theta\left(\frac{b}{r}(n+2^r)\right) = \Theta\left(\frac{bn}{\log n}\right)$$

Radix sort

- Counting sort is NOT in-place

- Quicksort is better than radix sort
  - Better cache utilization
  - In-place

Bucket sort



A
B
1
2
3
4
5
6
7
8
9
10
.78
.17
.39
.26
.72
.94
.21
.12
.23
.68
0
1
2
3
4
5
6
7
8
9
.12
.17
.21
.23
.26
.39
.68
.72
.78
.94
(a)
(b)

Bucket sort

```
BUCKET-SORT(A)
let B[0..n − 1] be a new array
n = A.length
for i = 0 to n − 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n − 1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

Bucket sort

- When input is uniformly distributed, the average case is $O(n)$
- It is fast, as counting-sort because it makes an assumption about the input
- RULE OF THUMB:
  - Extra assumption/extra information → room for optimization/customization

Order statistics

- The i^{th} *order statistic* of a set of n elements is the i^{th} *smallest element*

Order statistics

- Minimum:

```
MINIMUM(A)
min = A[1]
for i = 2 to A.length
    if min > A[i]
        min = A[i]
return min
```

- Requires $O(n)$, optimal as $n-1$ comparisons are needed

Selection algorithm

- Find i^{th} *smallest element*:

```
RANDOMIZED-SELECT(A, p, r, i)
if p == r
    return A[p]
q = RANDOMIZED-PARTITION(A, p, r)
k = q - p + 1
if i == k          // the pivot value is the answer
    return A[q]
elseif i < k
    return RANDOMIZED-SELECT(A, p, q - 1, i)
else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

- $O(n)$, prove it?

Selection algorithm

- On average, the left or right subarray that will be selected for next iteration will be of size $n/2$
- The recurrence becomes

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n)$$

- Using the master method with $a = 1$, $b = 2$, $f(n) = \Theta(n)$
- Thus, $O(n)$ on average