

Design and Analysis of Algorithms

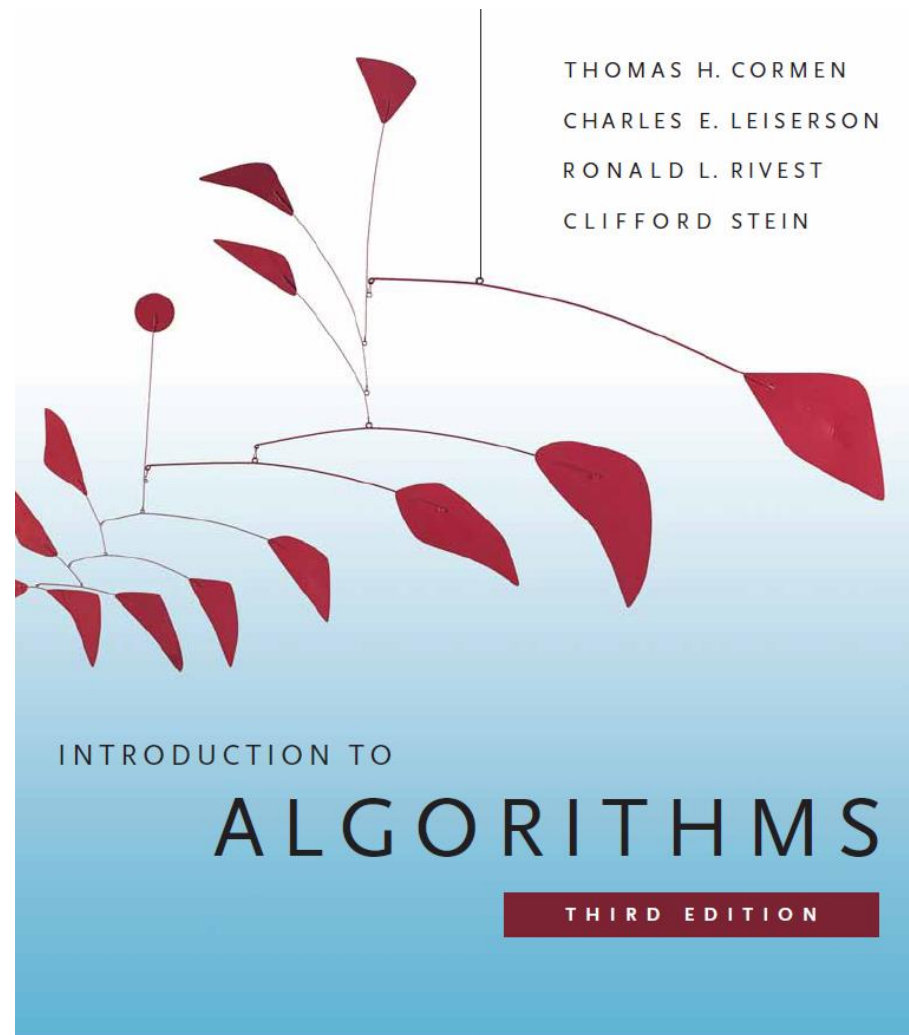


Lecture 01: Sorting & Recurrences

Ahmed Hamdy

Textbook

- Introduction to Algorithms, Thomas H. Cormen Charles E. Leiserson Ronald L. Rivest Clifford Stein, Third Edition.



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 $a'_1 \leq a'_2 \leq \dots \leq a'_n$
- **Algorithms:**
 - Insertion sort: $2n^2$ instructions
 - Merge sort: $50n \log_2 n$ instructions
 - Which is better??

Insertion vs Merge sort

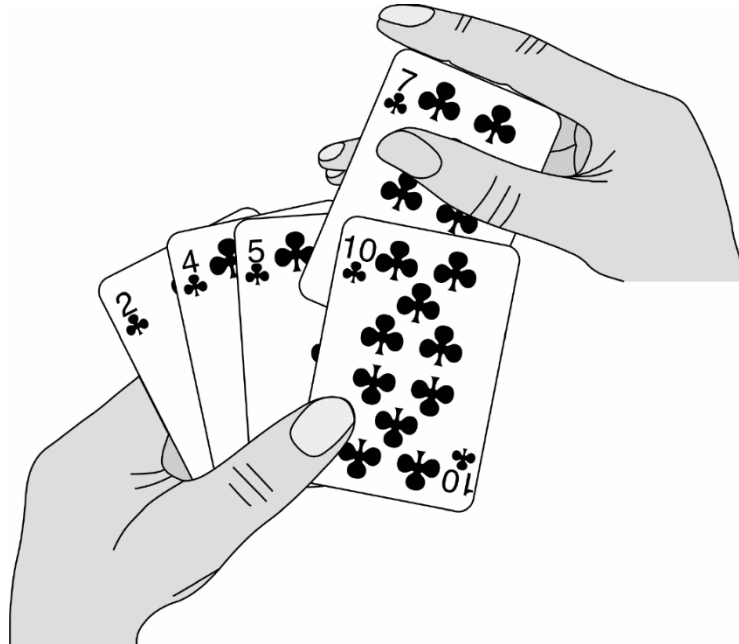
Input size	Insertion sort	Merge sort
n	$2n^2$	$50n \log_2 n$
2	8	100
10	200	1661
100	20,000	33,219
1K	2,000,000	498,289
10K	200,000,000	6,643,856

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

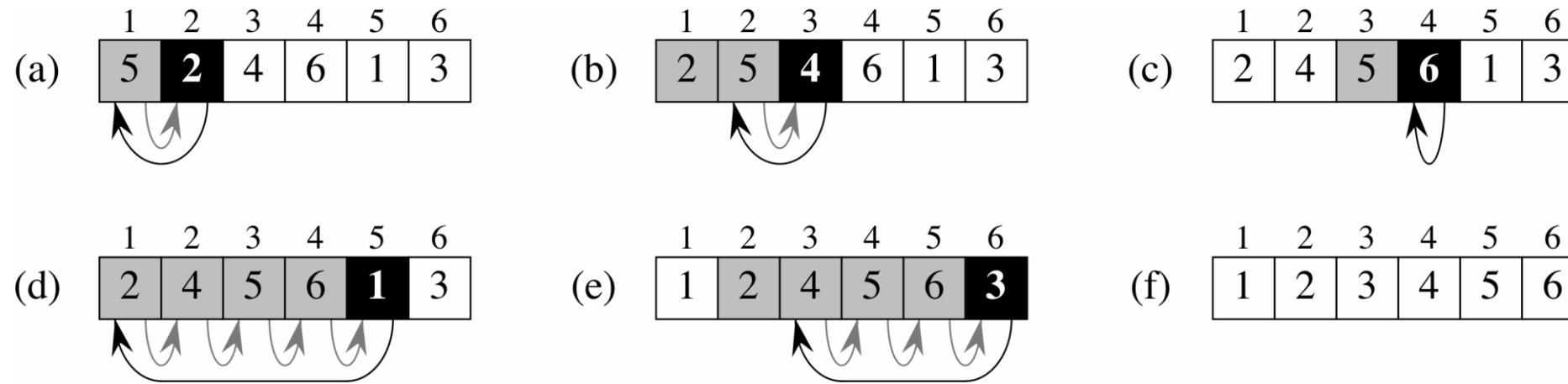
*Merge sort constants can be less than the above, they are just for illustration

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)	<i>cost</i>	<i>times</i>
for $j = 2$ to n	c_1	n
$key = A[j]$	c_2	$n - 1$
// Insert $A[j]$ into the sorted sequence $A[1 \dots j - 1]$.	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:** $\text{Max}(T(n))$
 - applies to certain input cases
- **Best-case:** $\text{Min}(T(n))$
 - 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 $T(n)$ ↓ 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** in high-level comparisons, fall back to them when comparing two algorithms with equal order of growth.

Asymptotic analysis

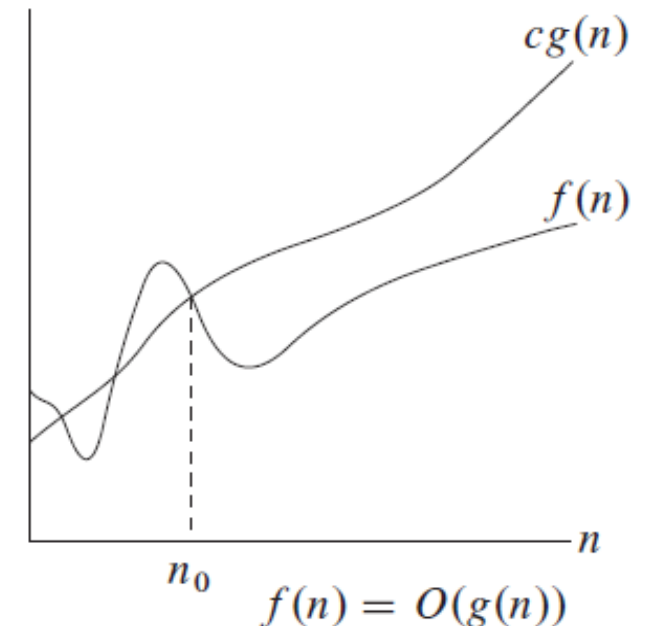
- Define ***O – notation*** (Big-O):

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

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

- 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; $2n = O(n^2)$



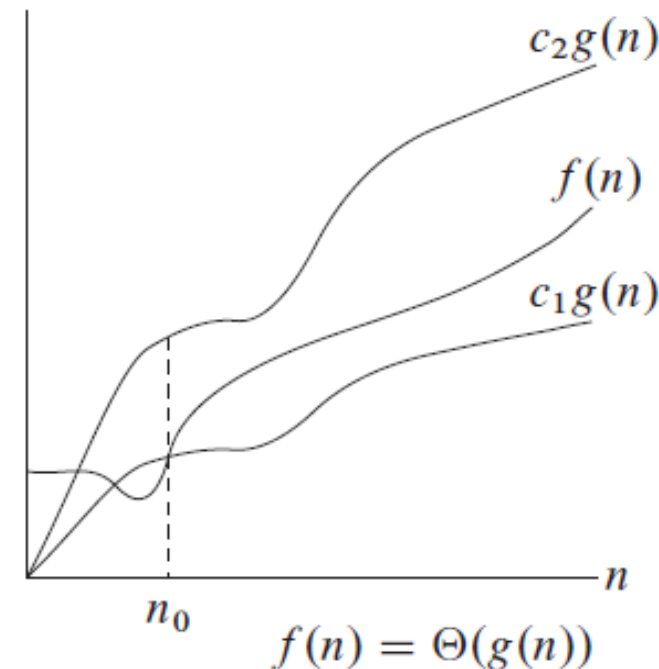
Asymptotic analysis

- Define Θ – *notation* (Theta):

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

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

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



Asymptotic analysis

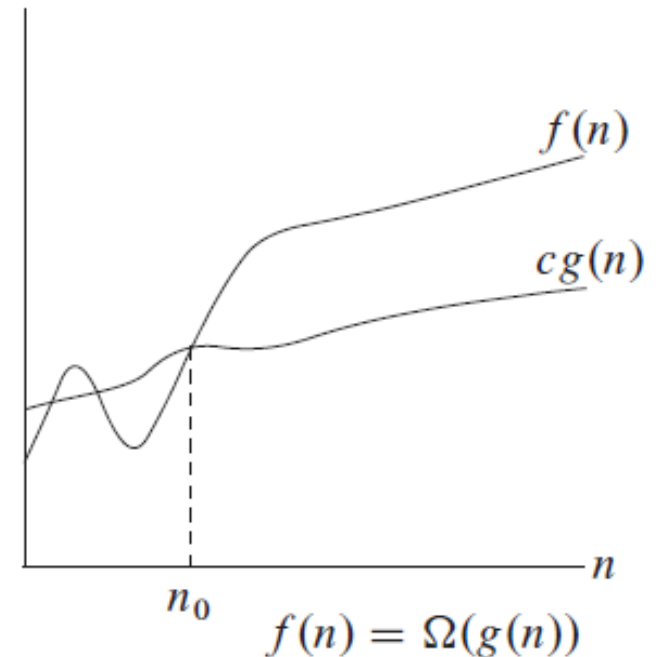
- Define Ω – *notation* (Big-Omega):

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

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

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

- 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 ***o – notation*** (little-o)
 - Similar to Big-O, but not tight
- Define ***ω – 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 us 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

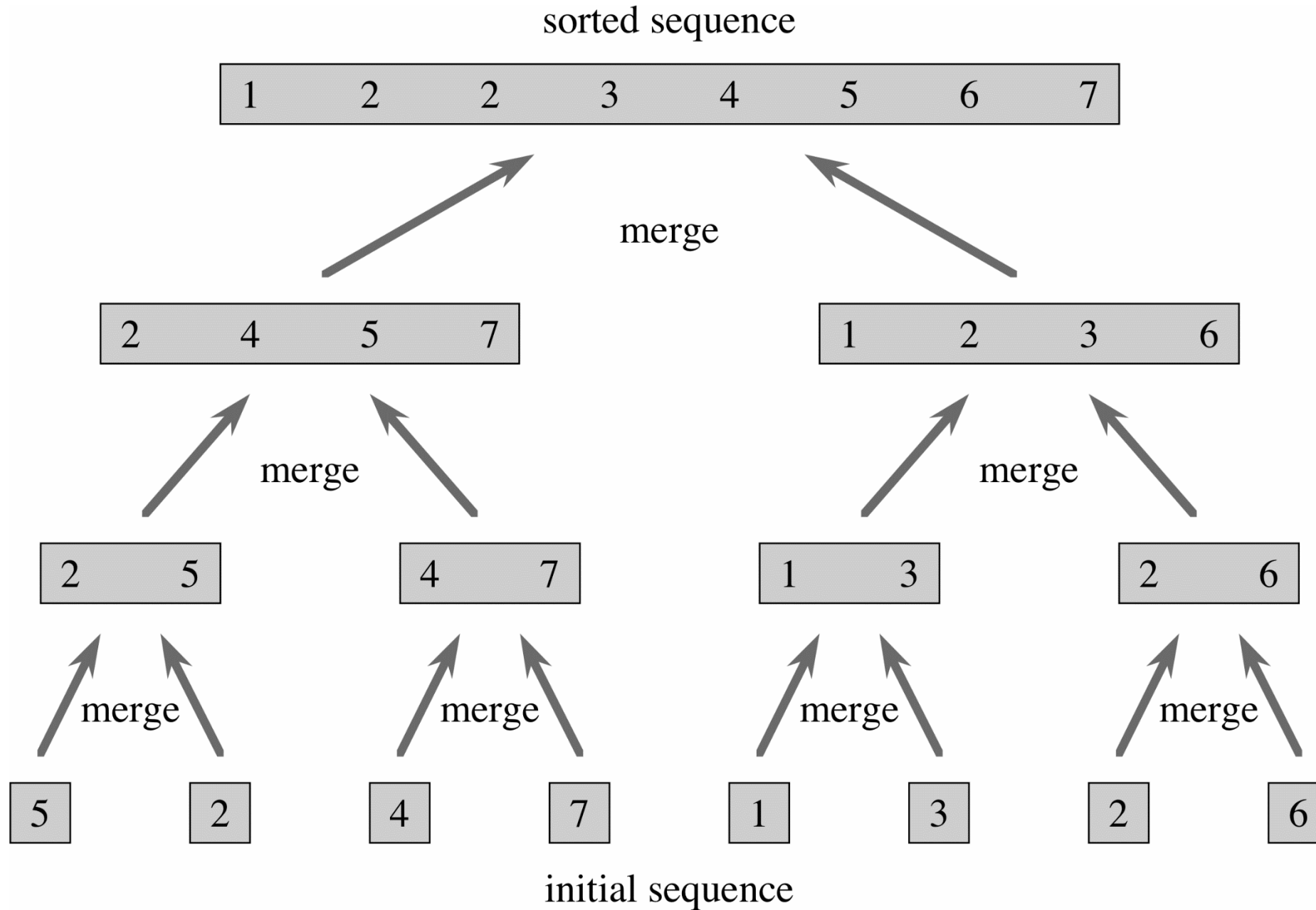
- In Insertion sort, one element at a time is inserted in the previously sorted subarray → *Incremental approach*
- ***Divide-and-conquer:***
 - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
 - **Conquer**
 - Solve subproblems in a straightforward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

Merge sort

- *Divide-and-conquer:*

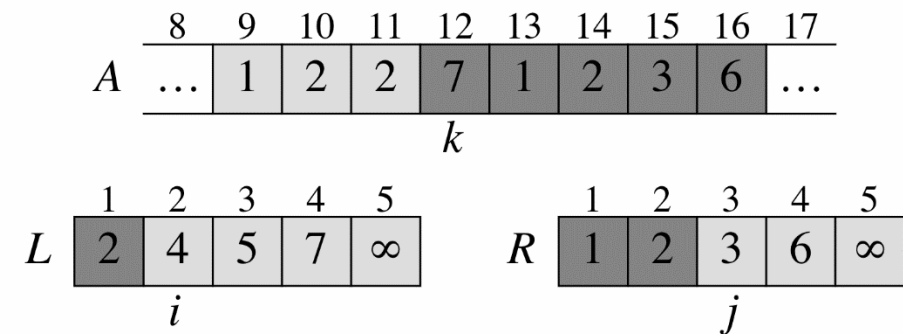
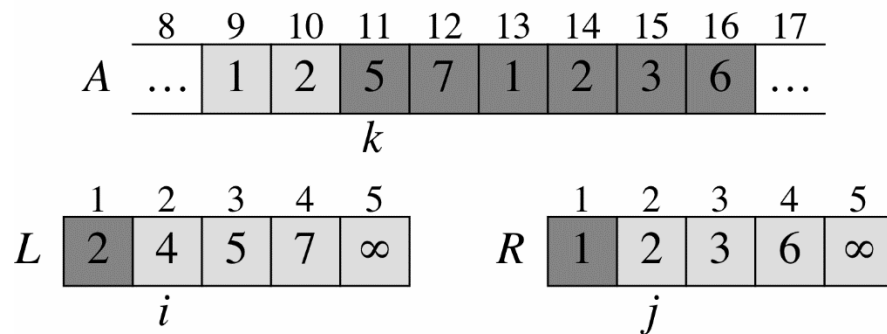
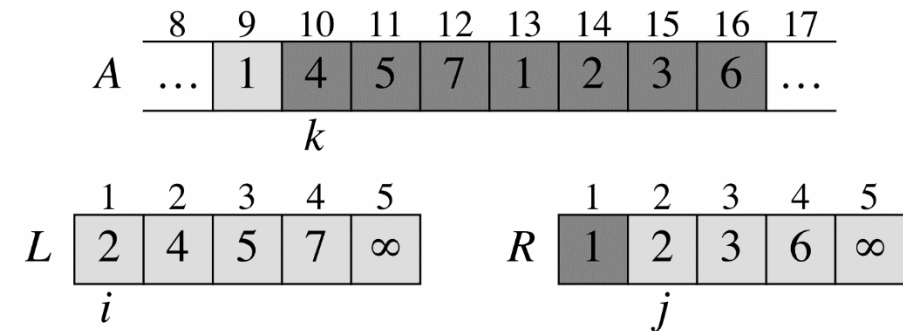
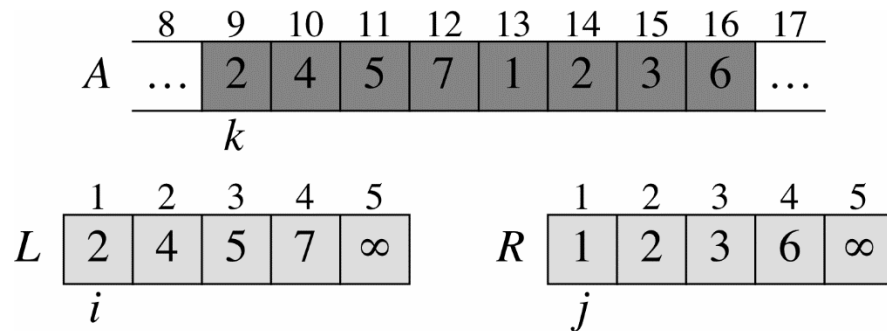
- **Divide:** Recursively divide the unsorted n -element sequence into two subsequences down to the lowest level of $n = 2$ elements each
- **Conquer:** Sort the two subsequences using merge sort up recursively

Merge sort



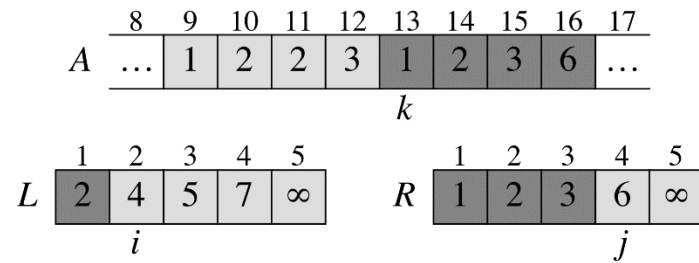
Merge sort

- Merge step

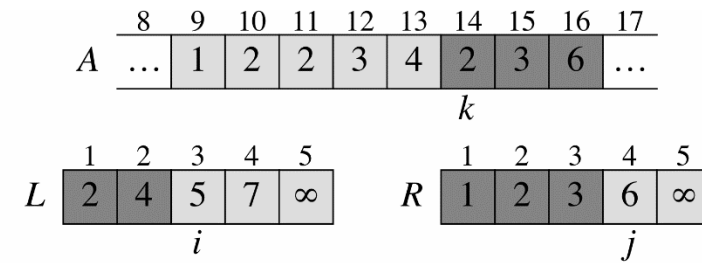


Merge sort

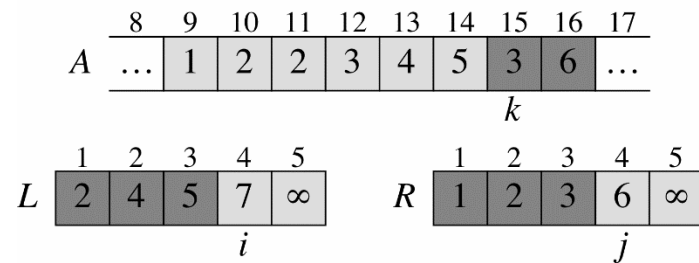
- Merge step (cont.)



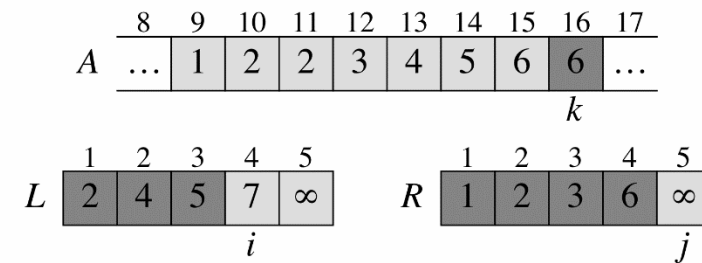
(e)



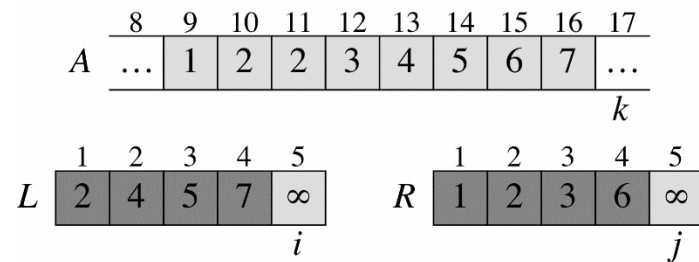
(f)



(g)



(h)



(i)

<https://visualgo.net/en/sorting>

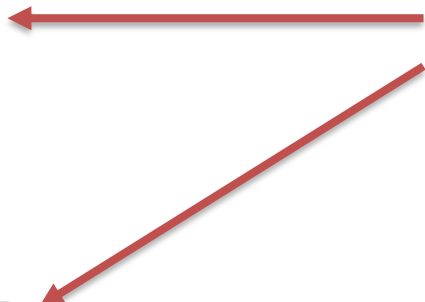
<https://www.toptal.com/developers/sorting-algorithms>

Merge sort

MERGE(A, p, q, r)

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
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   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 
```

Without the last entry with ∞ in each array, we would have to add extra checks for i and j in the if condition which would be checked in all loop iterations which is a waste of time.



Merge sort

- Main subroutine

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)

- Initial call:

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

Merge sort analysis

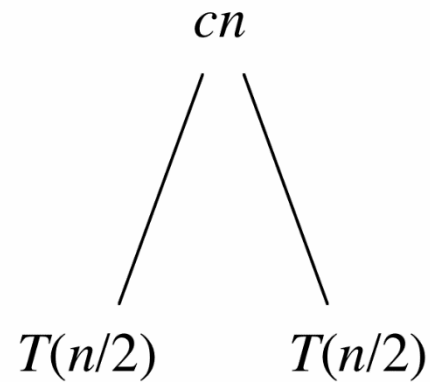
$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c , \\ aT(n/b) + D(n) + C(n) & \text{otherwise .} \end{cases}$$

- **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 $C(n) = \Theta(n)$.

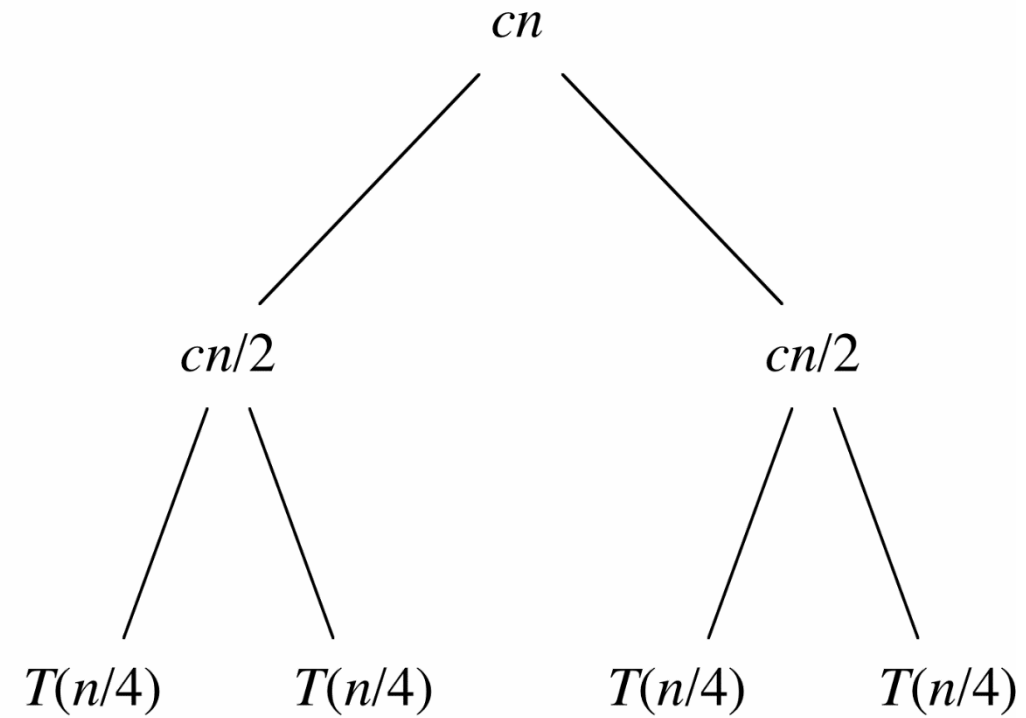
Merge sort analysis

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

$T(n)$



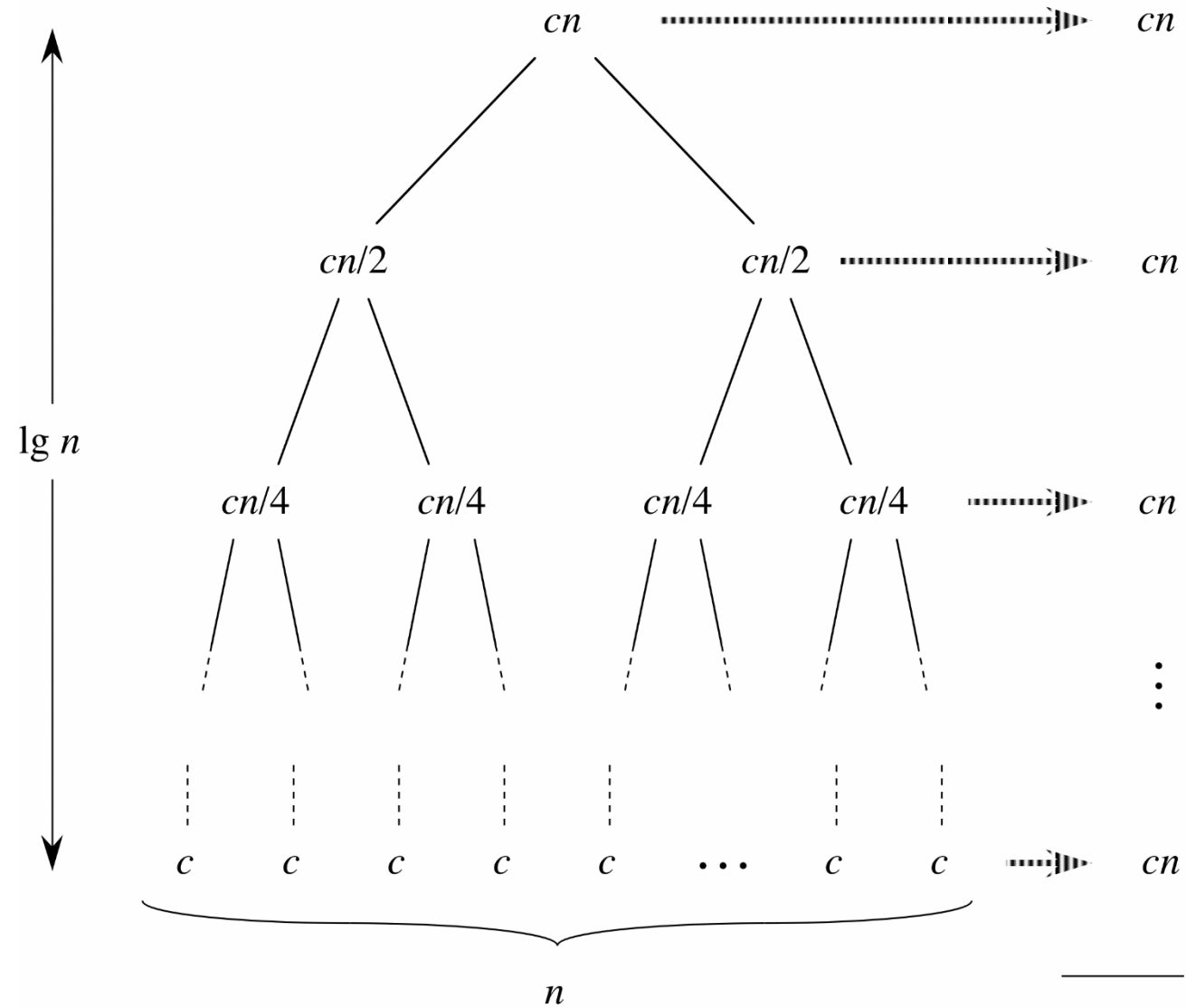
(a)



(b)

(c)

Merge sort analysis



(d)

Total: $cn \lg n + cn$

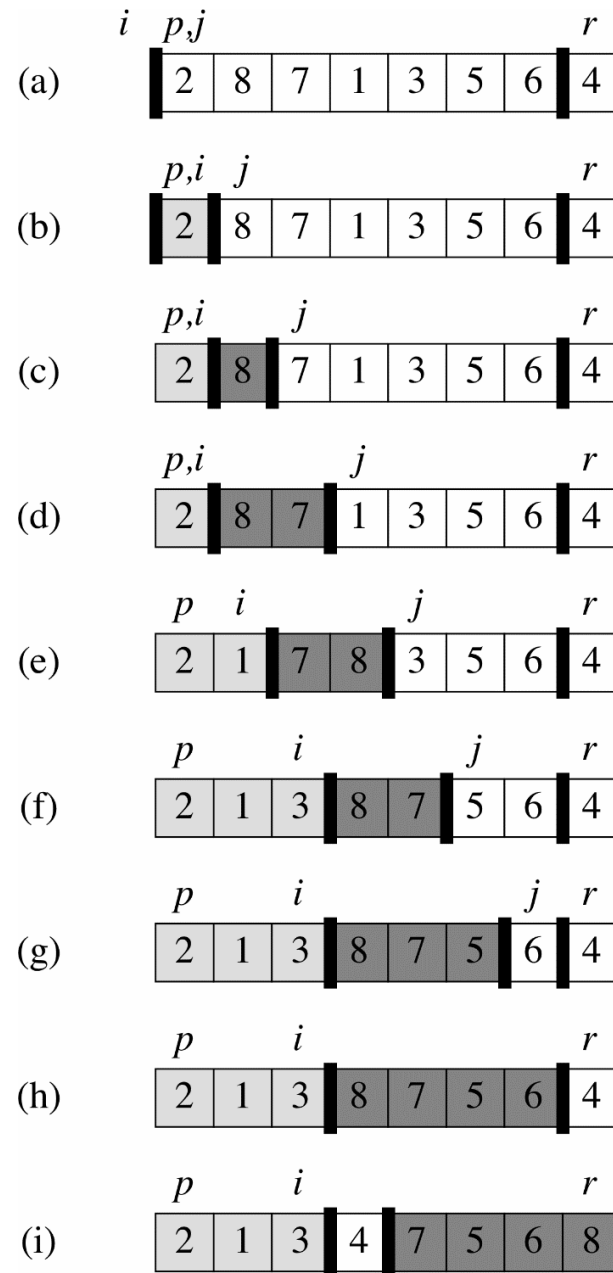
Space complexity

- 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 $T(n) = O(n^2)$.
 - Space complexity (in-place) $S(n) = \Theta(1)$.
- Merge sort:
 - Time complexity $T(n) = \Theta(n \log n)$.
 - Space complexity $S(n) = \Theta(n)$.
- For small n , Insertion sort is better while Merge sort is better for large n .

Quicksort



Quicksort

QUICKSORT(A, p, r)

```
1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3      QUICKSORT( $A, p, q - 1$ )
4      QUICKSORT( $A, q + 1, r$ )
```

PARTITION(A, p, r)

```
1   $x = A[r]$ 
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5           $i = i + 1$ 
6          exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  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:

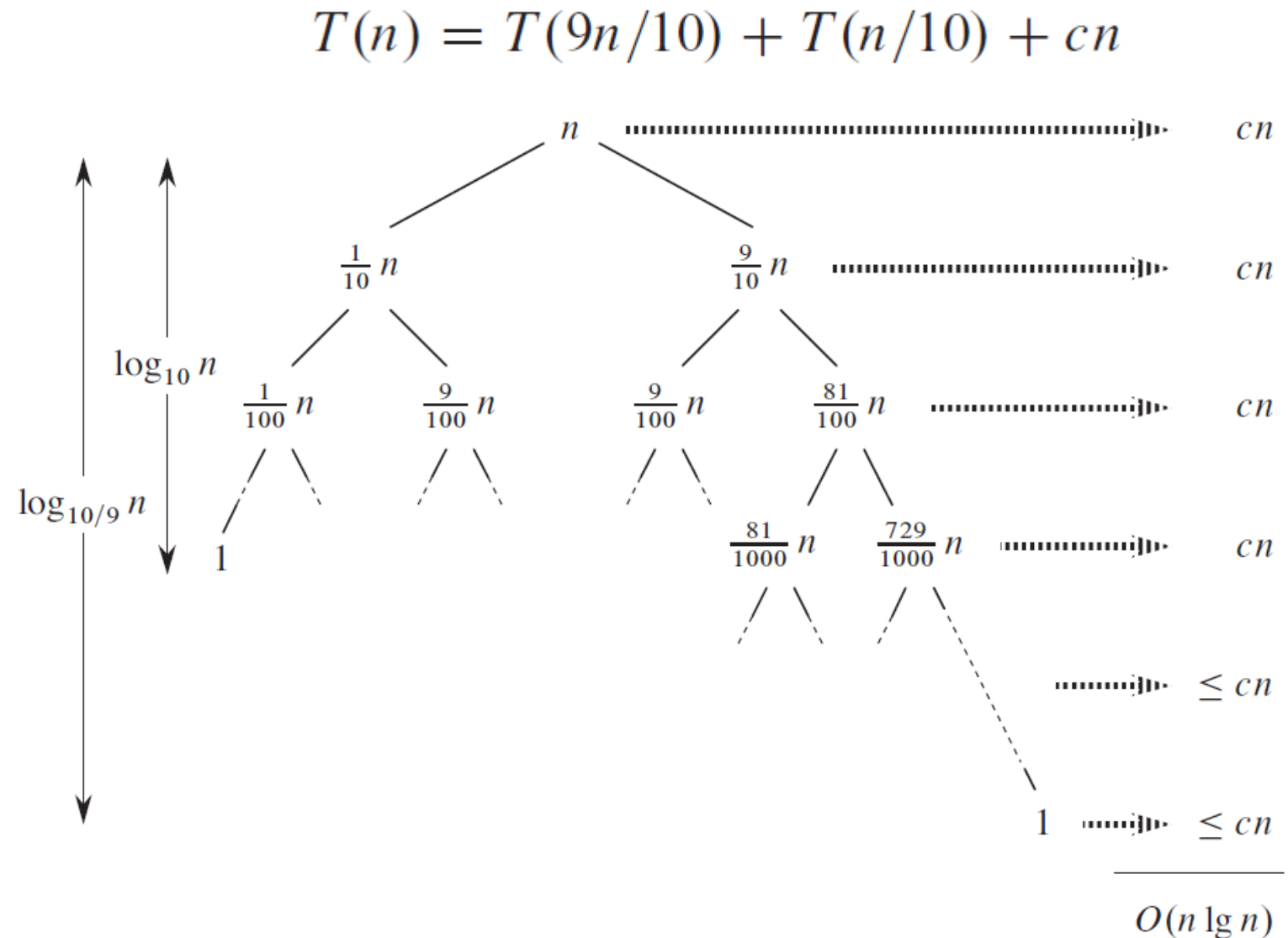
$$T(n) = 2T(n/2) + \Theta(n)$$

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

Quicksort

Performance

- Balanced partitioning:



Randomized quicksort

RANDOMIZED-PARTITION(A, p, r)

```
1   $i = \text{RANDOM}(p, r)$   
2  exchange  $A[r]$  with  $A[i]$   
3  return PARTITION( $A, p, r$ )
```

RANDOMIZED-QUICKSORT(A, p, r)

```
1  if  $p < r$   
2       $q = \text{RANDOMIZED-PARTITION}(A, p, r)$   
3      RANDOMIZED-QUICKSORT( $A, p, q - 1$ )  
4      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:

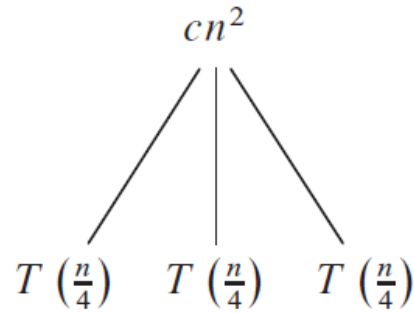
$$\begin{aligned} T(n) &\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n, \end{aligned}$$

- Holds for $c \geq 1$

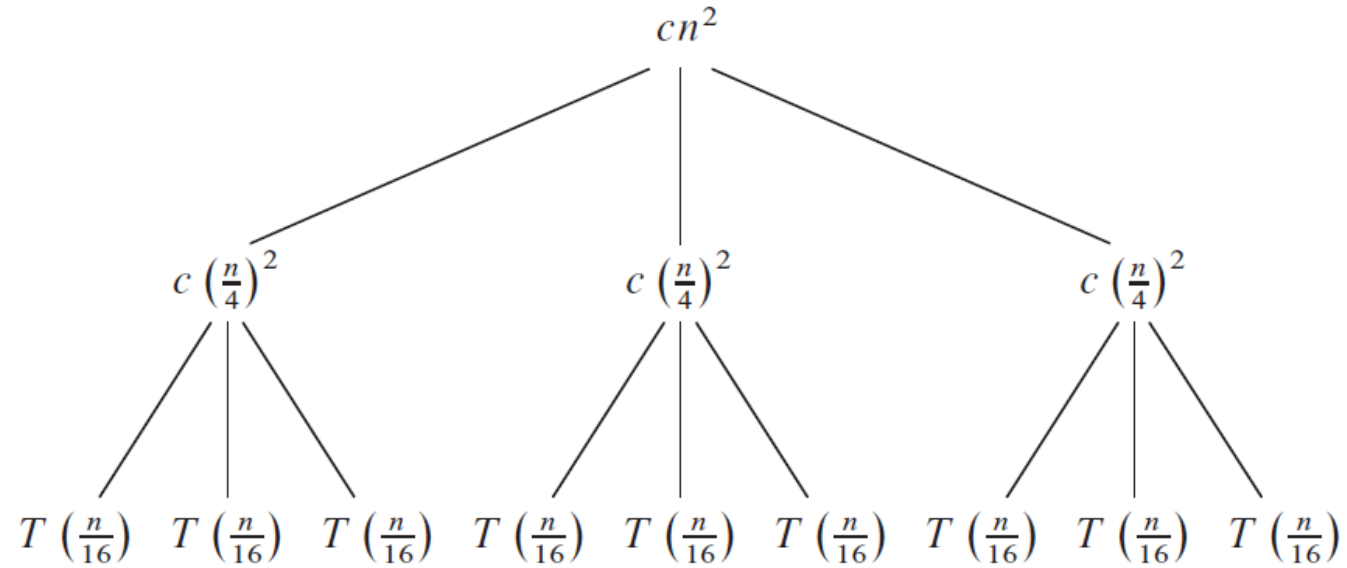
Recursion-tree method

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

$T(n)$



(a)

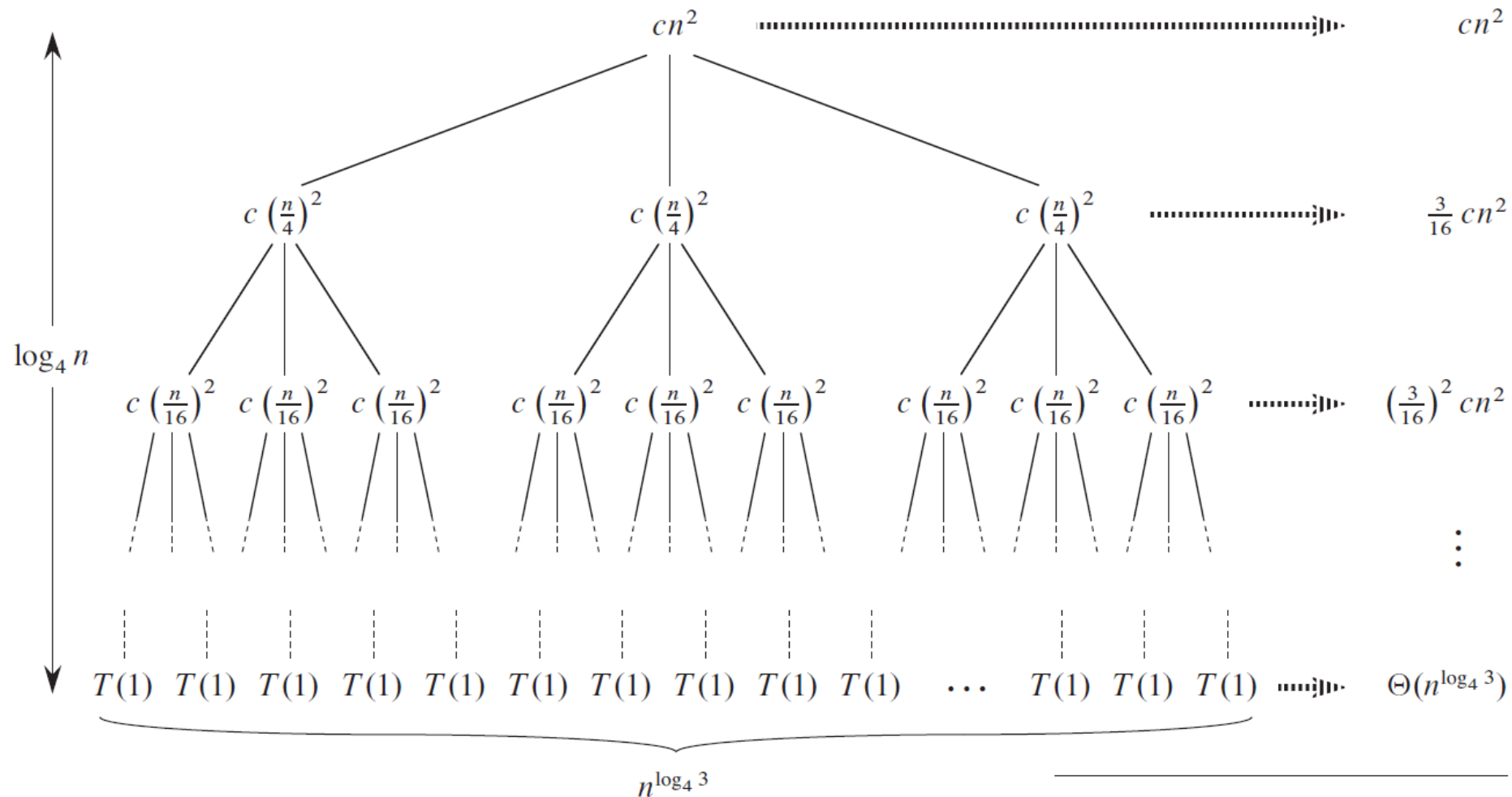


(b)

(c)

Recursion-tree method

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



(d)

Recursion-tree method

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

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

$$\begin{aligned}\sum_{i=0}^n x^i &= 1 + x + x^2 + \cdots x^n \\&= \frac{1 - x}{1 - x} (1 + x + x^2 + \cdots x^n) \\&= \frac{1 - x^{n+1}}{1 - x}\end{aligned}$$

When $x < 1$ and $n \rightarrow \infty$:

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1 - x}$$

Recursion-tree method

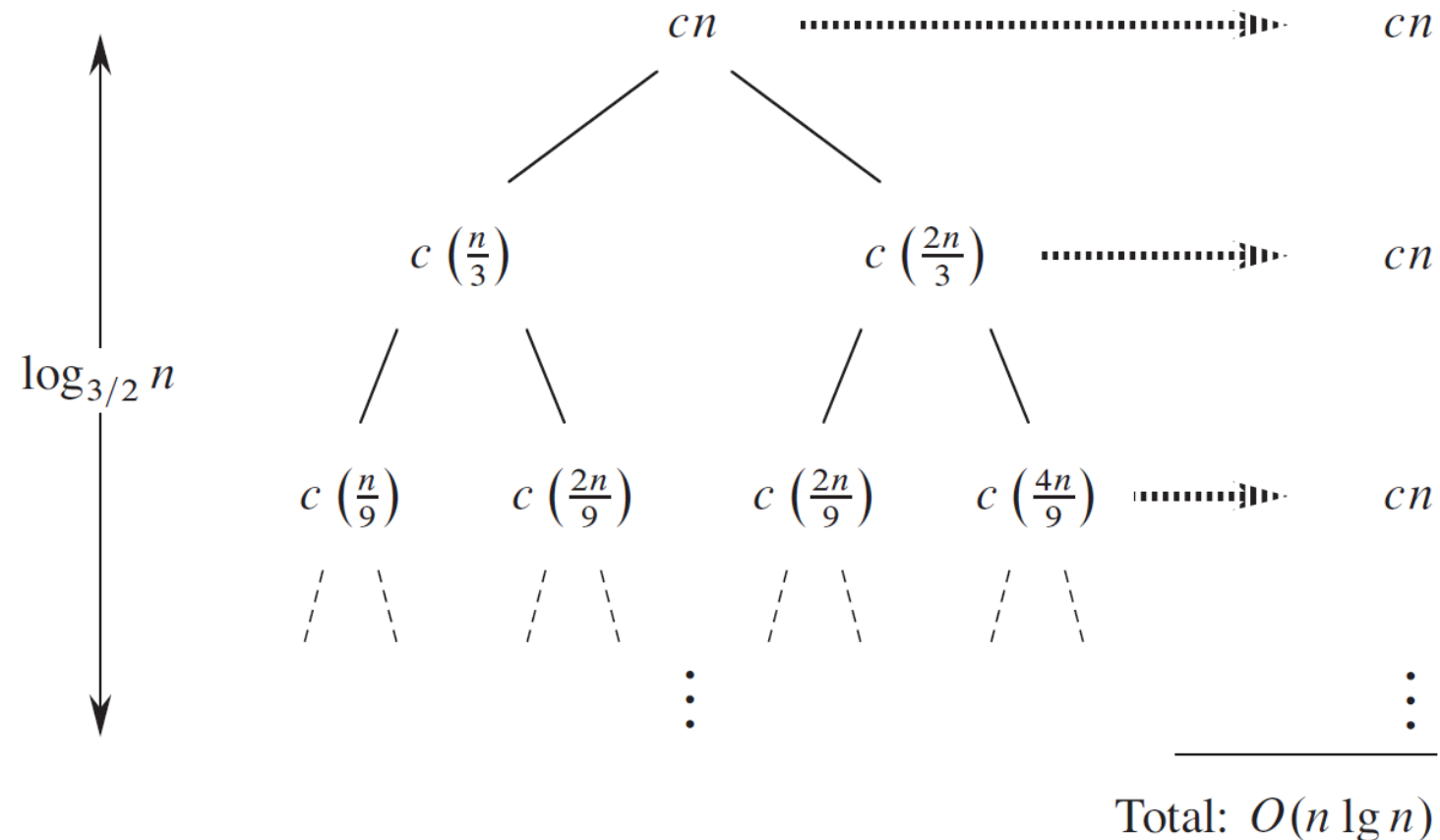
- Verify using substitution method $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$
- Base case: $T(\lfloor n/4 \rfloor) \leq d\lfloor n/4 \rfloor^2$
- Induction:

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

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

Recursion-tree method

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



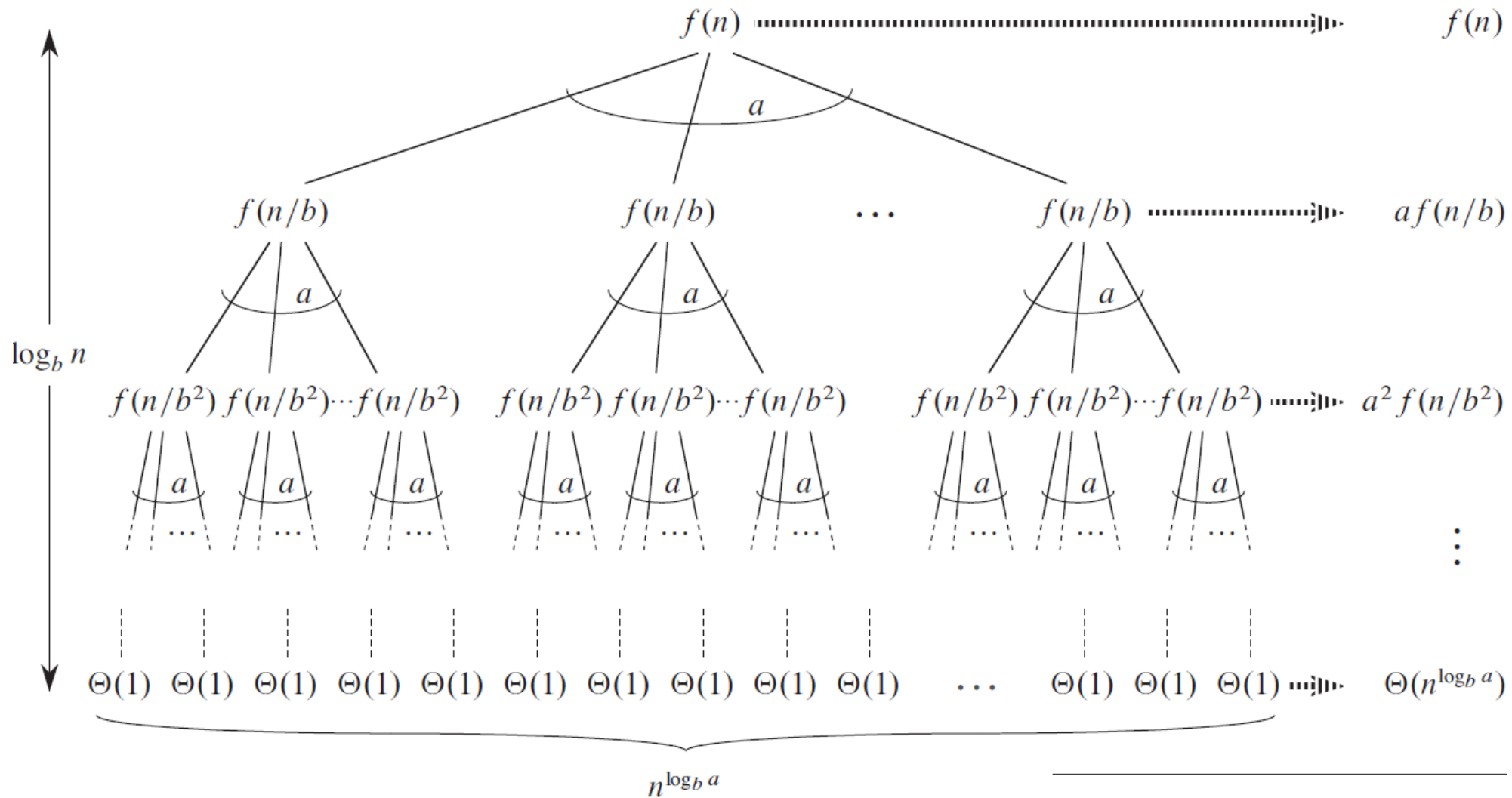
Recursion-tree method

- Verify using substitution method

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

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

Master method



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

$$\text{Total: } \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

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 $af(n/b) = 3(n/4) \lg(n/4) \leq (3/4)n \lg n = cf(n)$ 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

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

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

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

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

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

Master method

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

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

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

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

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

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

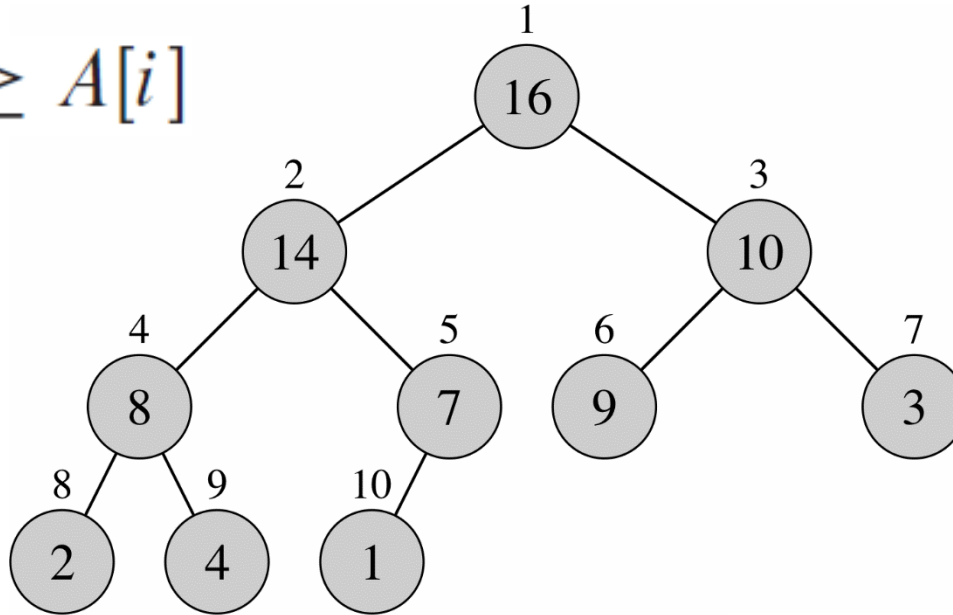
$$\begin{aligned} g(n) &= \Theta(n^{\log_b a} \log_b n) \\ &= \Theta(n^{\log_b a} \lg n), \end{aligned}$$

Selection sort

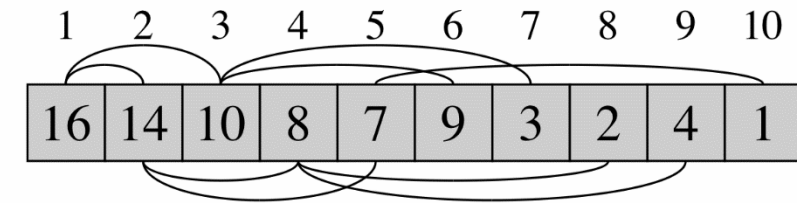
- Loop at all positions of the array (i):
 - Find the minimum element in subarray $A[i..n]$
 - $A[i] = \min(A[i..n])$
- What is the complexity??
- Stable?

Heaps

$$A[\text{PARENT}(i)] \geq A[i]$$



(a)



(b)

PARENT(i)

1 **return** $\lfloor i/2 \rfloor$

LEFT(i)

1 **return** $2i$

RIGHT(i)

1 **return** $2i + 1$

Heaps

Optimizations of heap operations

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

Heaps

- Max-heap:

$$A[\text{PARENT}(i)] \geq A[i]$$

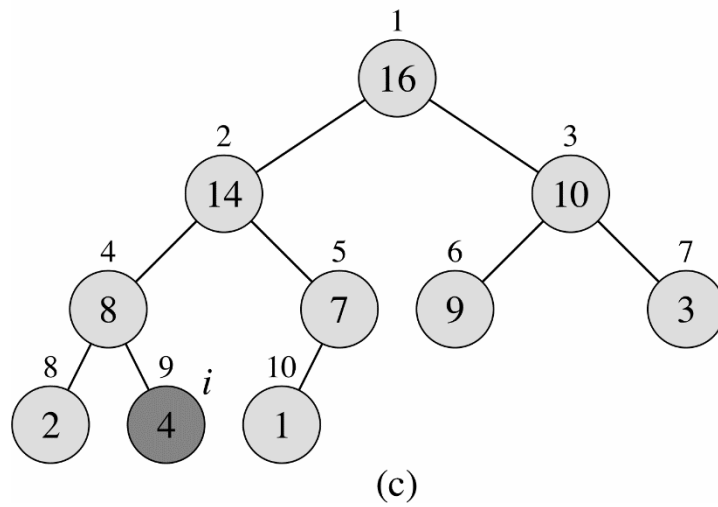
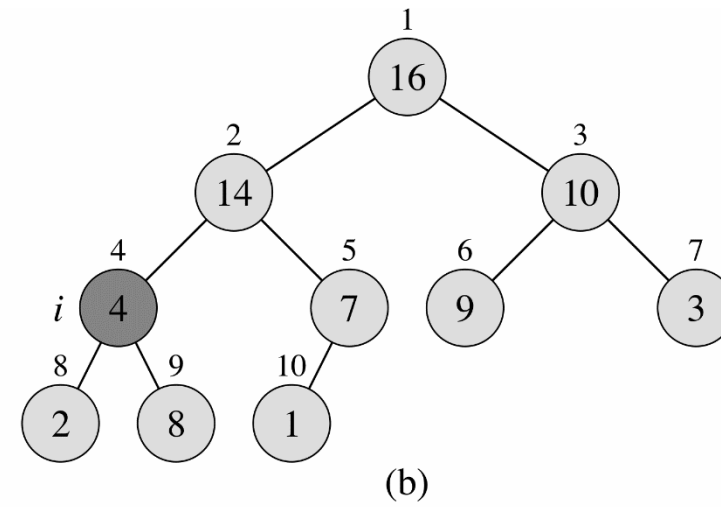
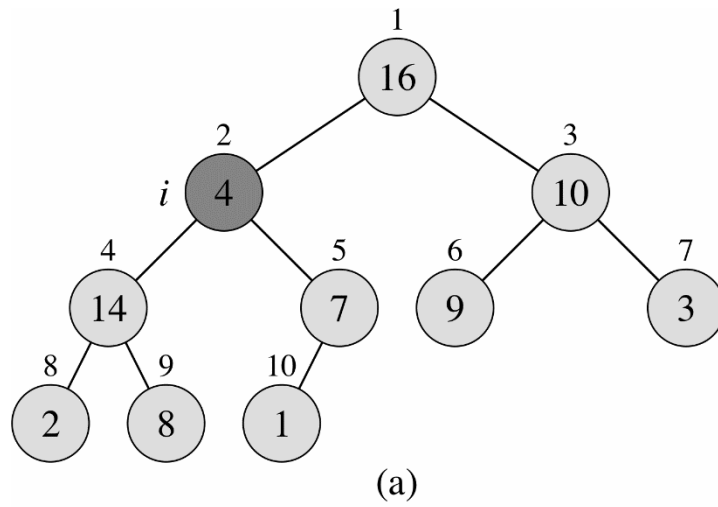
- Min-heap:

$$A[\text{PARENT}(i)] \leq A[i]$$

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

Heaps

Max-Heapify



Heaps

MAX-HEAPIFY(A, i)

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4       $\text{largest} = l$ 
5  else  $\text{largest} = i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7       $\text{largest} = r$ 
8  if  $\text{largest} \neq i$ 
9      exchange  $A[i]$  with  $A[\text{largest}]$ 
10     MAX-HEAPIFY( $A, \text{largest}$ )
```

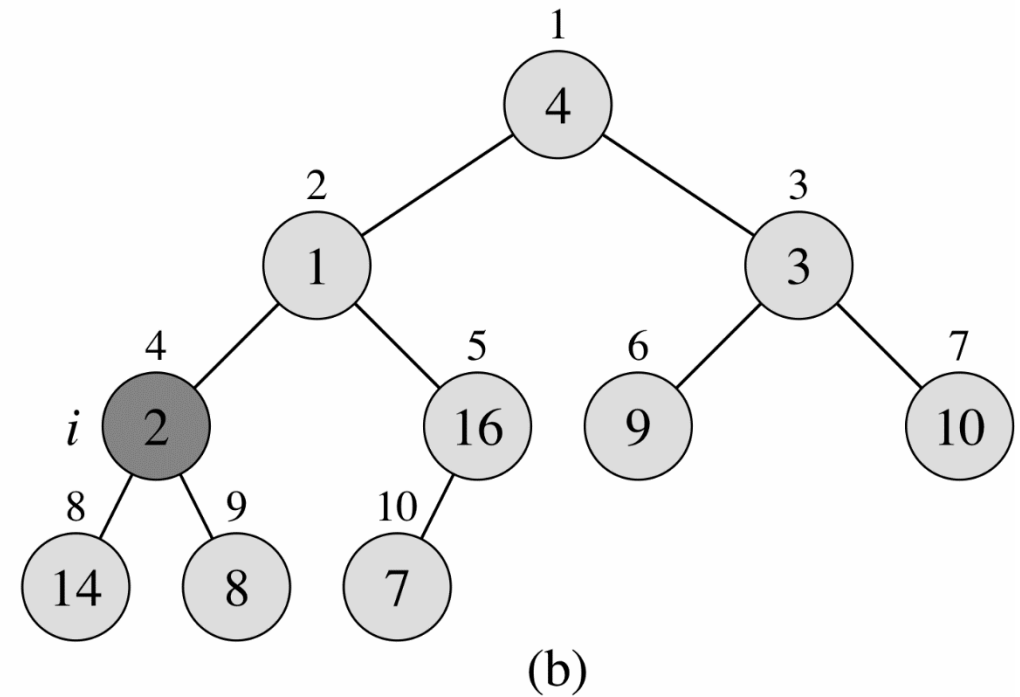
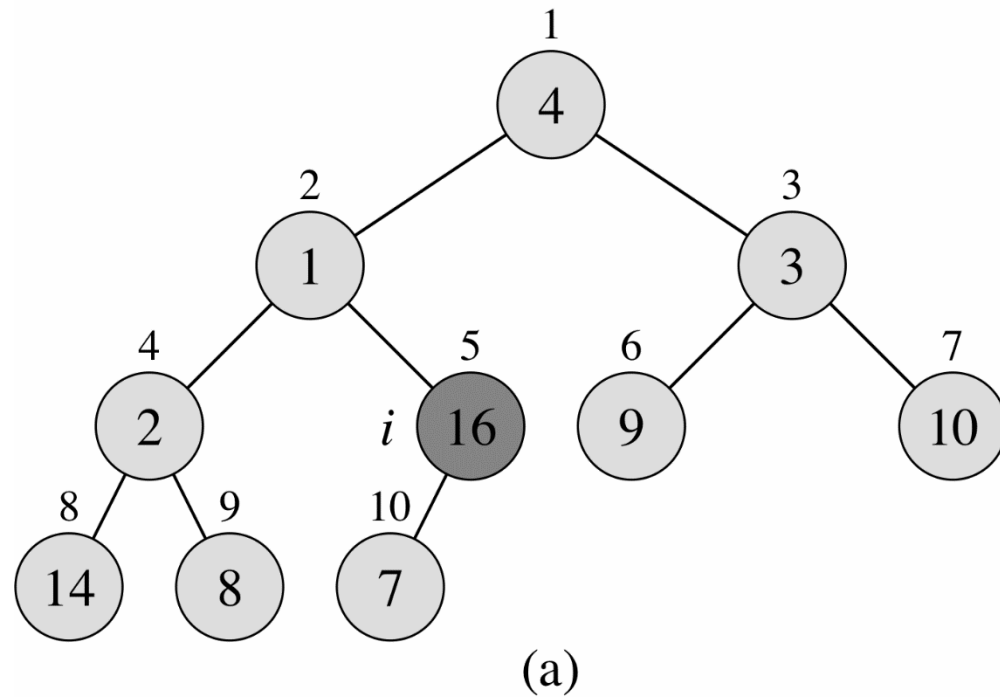
- Max-Heapify complexity: $O(\log n)$

Heaps

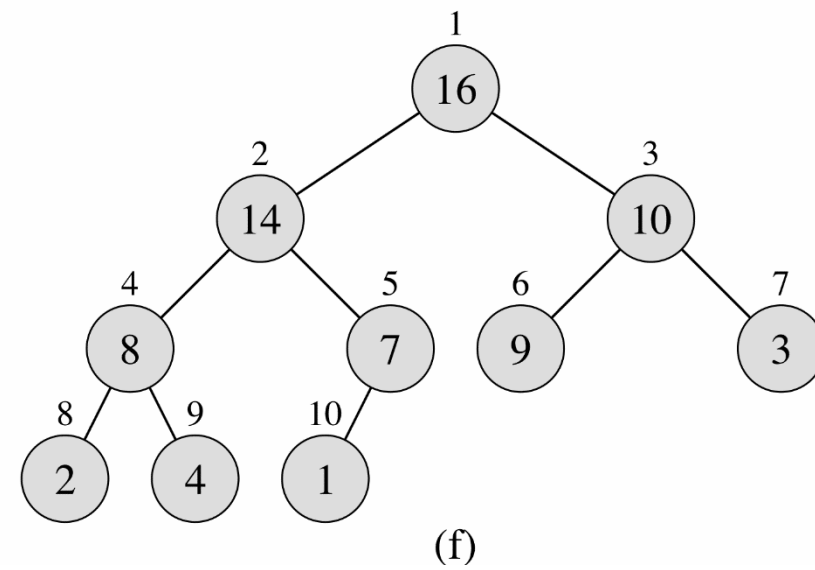
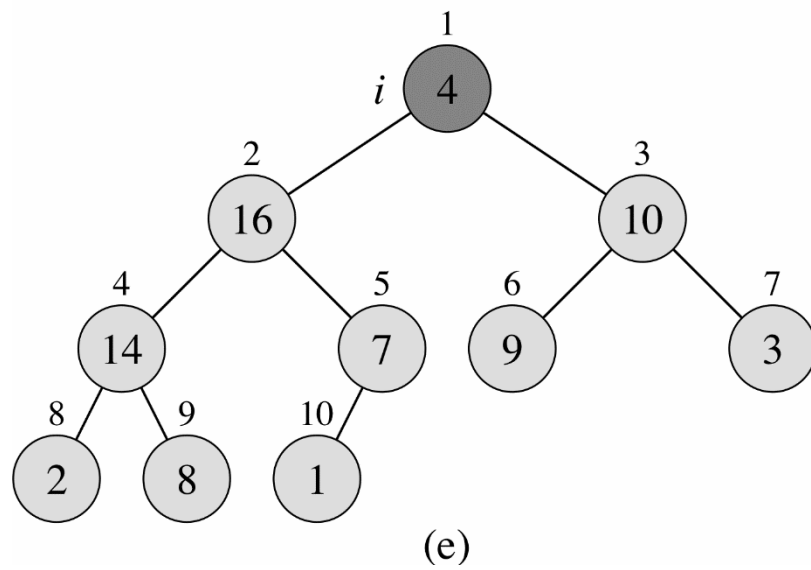
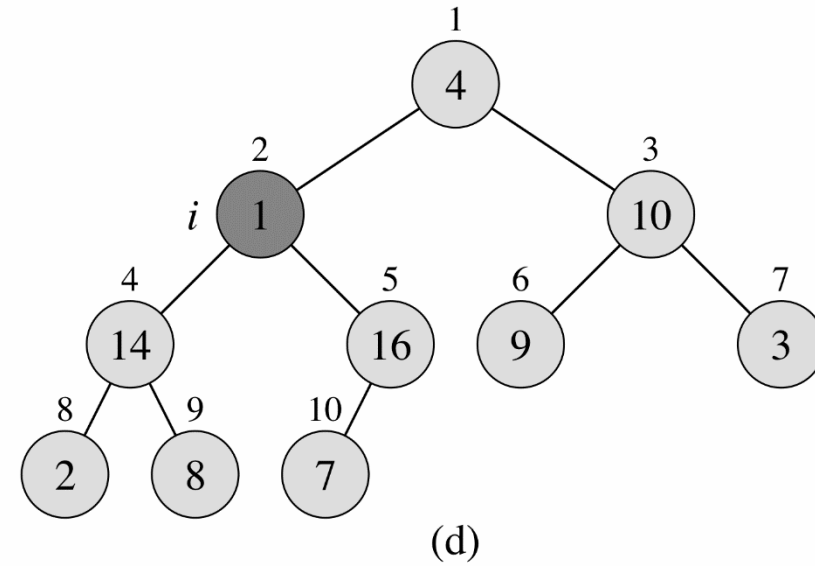
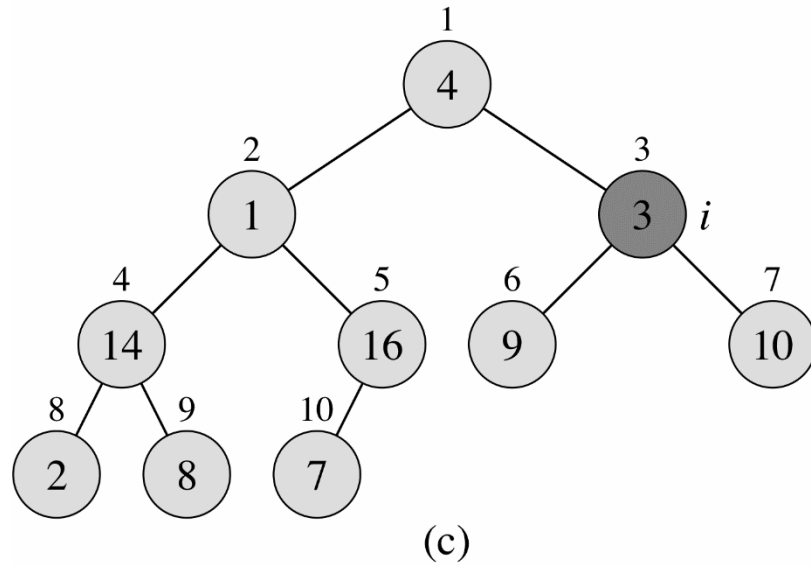
- How to organize an array to be a heap?

A

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---



Heaps



Heaps

BUILD-MAX-HEAP(A)

```
1   $A.heap-size = A.length$   
2  for  $i = \lfloor A.length/2 \rfloor$  downto 1  
3      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 $|x| < 1$.

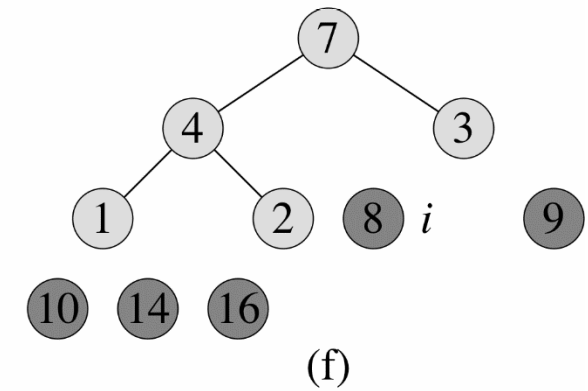
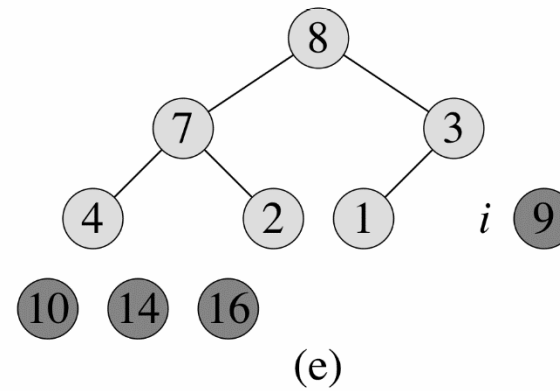
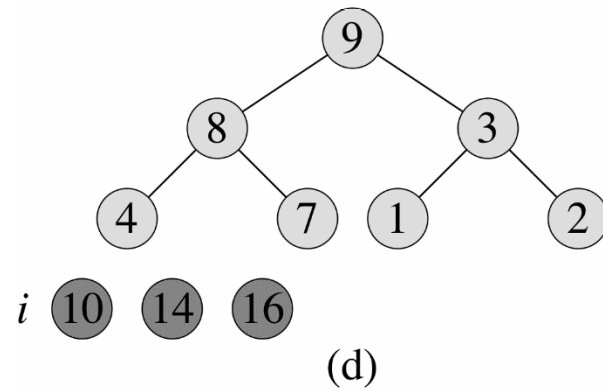
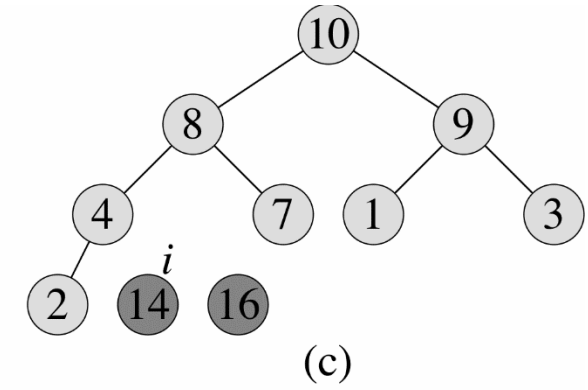
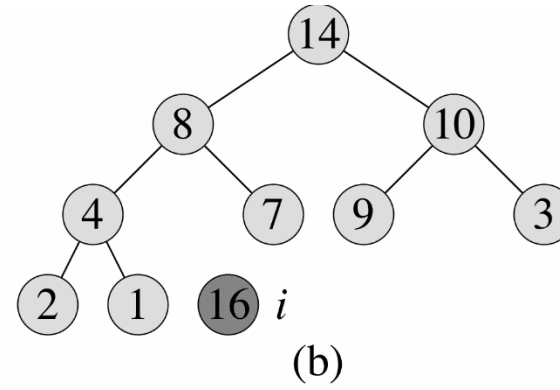
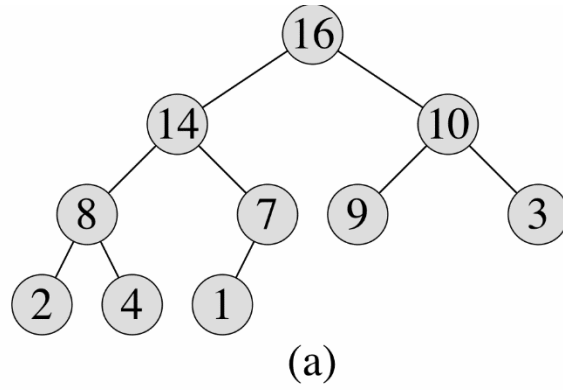
Heaps

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

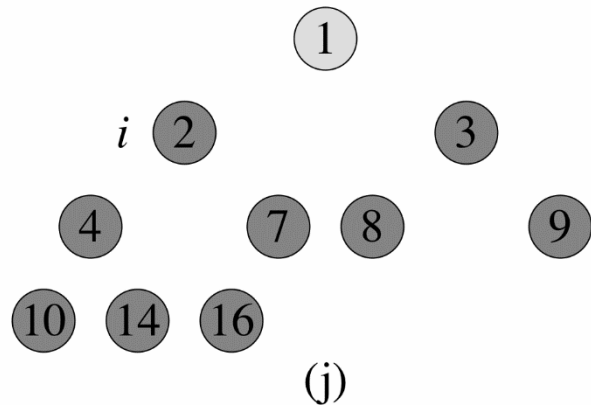
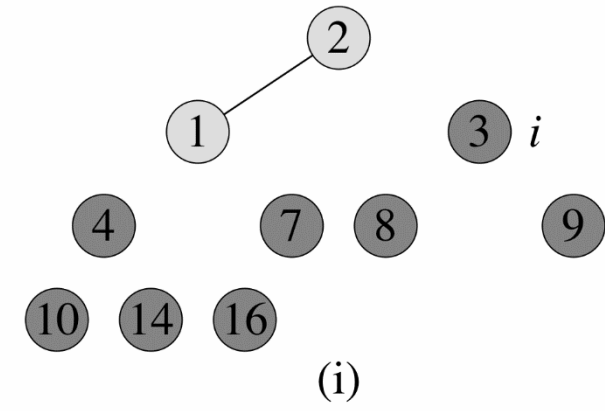
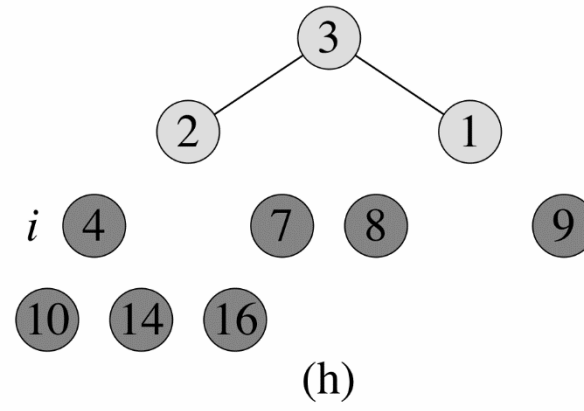
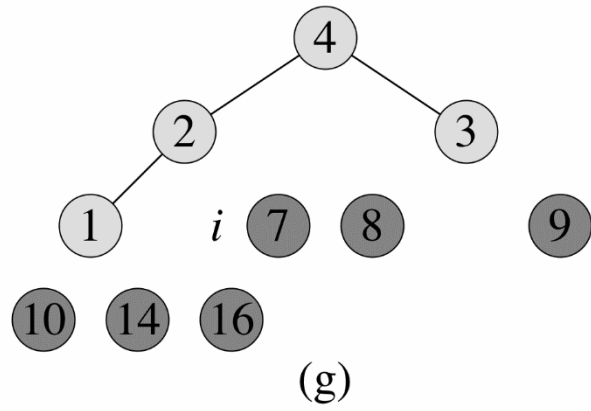
Becomes

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

Heapsort



Heapsort



A

1	2	3	4	7	8	9	10	14	16
---	---	---	---	---	---	---	----	----	----

(k)

Heapsort

HEAPSORT(A)

```
1  BUILD-MAX-HEAP( $A$ )
2  for  $i = A.length$  downto 2
3      exchange  $A[1]$  with  $A[i]$ 
4       $A.heap-size = A.heap-size - 1$ 
5      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** $A.heap\text{-}size < 1$

2 **error** “heap underflow”

3 $max = A[1]$

4 $A[1] = A[A.heap\text{-}size]$

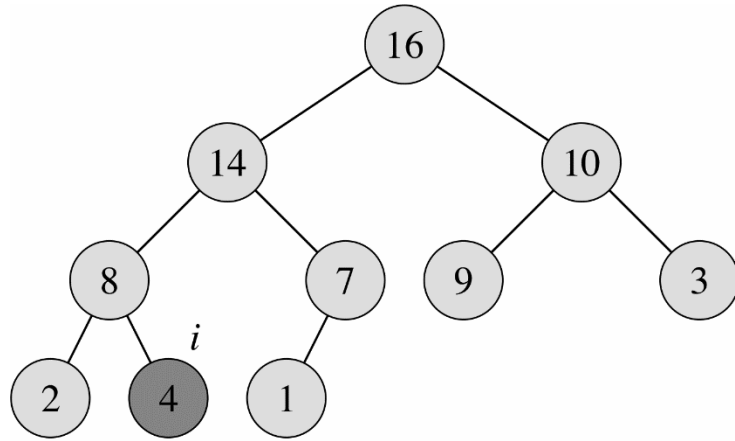
5 $A.heap\text{-}size = A.heap\text{-}size - 1$

6 MAX-HEAPIFY($A, 1$)

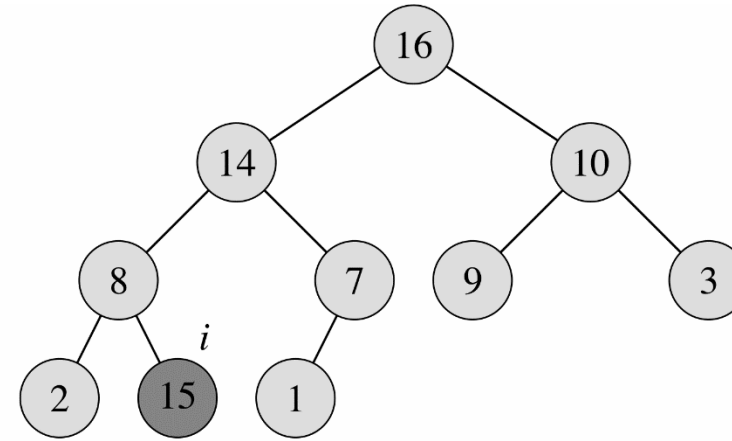
7 **return** max

Priority queue

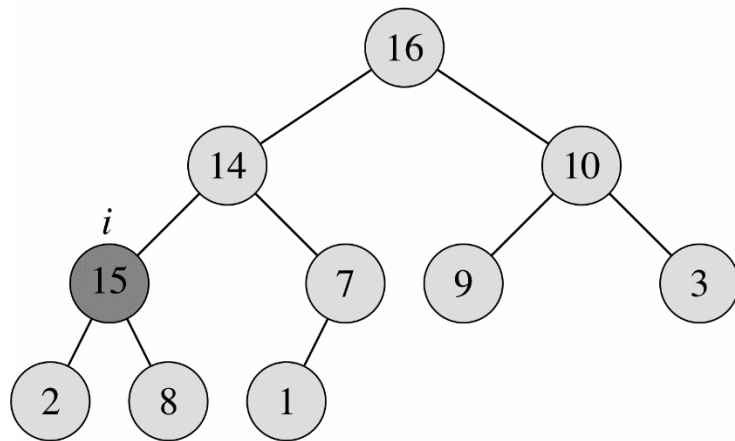
- Heap-Increase-Key



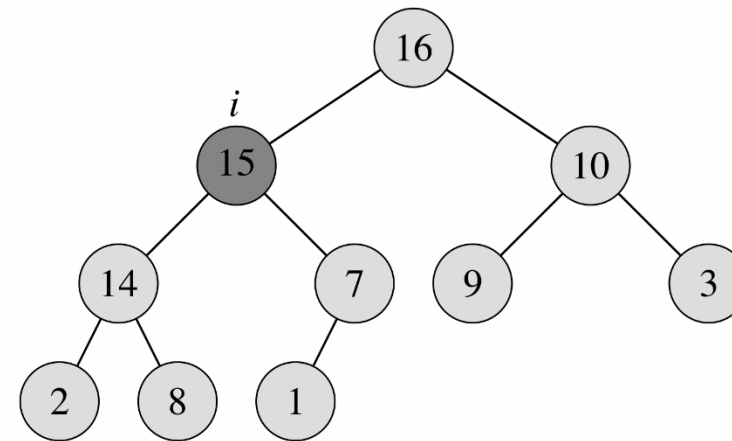
(a)



(b)



(c)



(d)

Priority queues

HEAP-INCREASE-KEY(A, i, key)

```
1  if  $key < A[i]$ 
2      error “new key is smaller than current key”
3   $A[i] = key$ 
4  while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$ 
5      exchange  $A[i]$  with  $A[\text{PARENT}(i)]$ 
6       $i = \text{PARENT}(i)$ 
```

MAX-HEAP-INSERT(A, key)

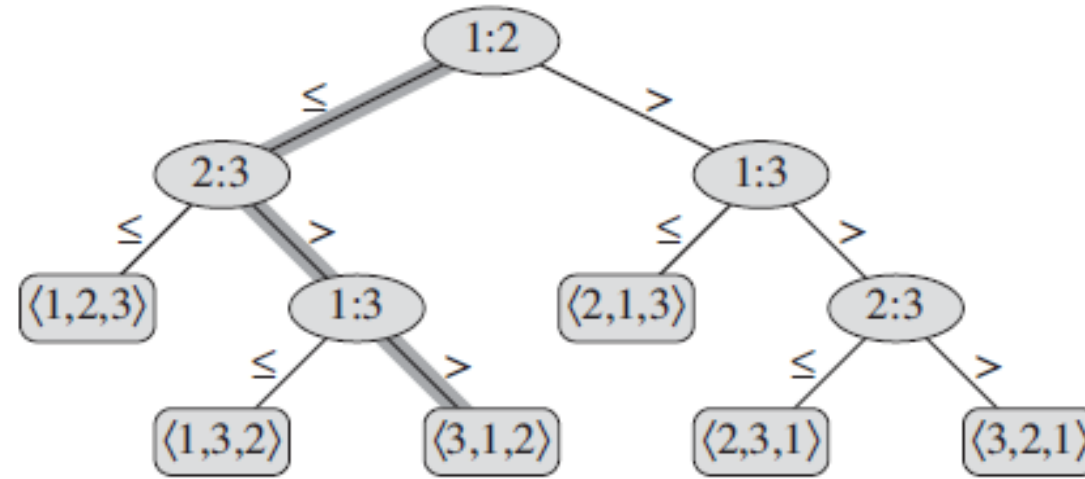
```
1   $A.\text{heap-size} = A.\text{heap-size} + 1$ 
2   $A[A.\text{heap-size}] = -\infty$ 
3  HEAP-INCREASE-KEY( $A, A.\text{heap-size}, key$ )
```


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 sort, radix sort and bucket sort are useful techniques.

Counting sort

	1	2	3	4	5	6	7	8
<i>A</i>	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
<i>C</i>	2	0	2	3	0	1

(a)

	0	1	2	3	4	5
<i>C</i>	2	2	4	7	7	8

(b)

	1	2	3	4	5	6	7	8
<i>B</i>							3	

	0	1	2	3	4	5
<i>C</i>	2	2	4	6	7	8

(c)

	1	2	3	4	5	6	7	8
<i>B</i>		0					3	

	0	1	2	3	4	5
<i>C</i>	1	2	4	6	7	8

(d)

	1	2	3	4	5	6	7	8
<i>B</i>		0				3	3	

	0	1	2	3	4	5
<i>C</i>	1	2	4	5	7	8

(e)

	1	2	3	4	5	6	7	8
<i>B</i>	0	0	2	2	3	3	3	5

(f)

Counting sort

COUNTING-SORT(A, B, k)

```
1  let  $C[0..k]$  be a new array
2  for  $i = 0$  to  $k$ 
3       $C[i] = 0$ 
4  for  $j = 1$  to  $A.length$ 
5       $C[A[j]] = C[A[j]] + 1$ 
6  //  $C[i]$  now contains the number of elements equal to  $i$ .
7  for  $i = 1$  to  $k$ 
8       $C[i] = C[i] + C[i - 1]$ 
9  //  $C[i]$  now contains the number of elements less than or equal to  $i$ .
10 for  $j = A.length$  downto 1
11      $B[C[A[j]]] = A[j]$ 
12      $C[A[j]] = C[A[j]] - 1$ 
```

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

Counting sort

COUNTING-SORT(A, B, k)

```
1  let  $C[0..k]$  be a new array
2  for  $i = 0$  to  $k$ 
3       $C[i] = 0$ 
4  for  $j = 1$  to  $A.length$ 
5       $C[A[j]] = C[A[j]] + 1$ 
6  //  $C[i]$  now contains the number of elements equal to  $i$ .
7  for  $i = 1$  to  $k$ 
8       $C[i] = C[i] + C[i - 1]$ 
9  //  $C[i]$  now contains the number of elements less than or equal to  $i$ .
10 for  $j = A.length$  downto 1
11      $B[C[A[j]]] = A[j]$ 
12      $C[A[j]] = C[A[j]] - 1$ 
```

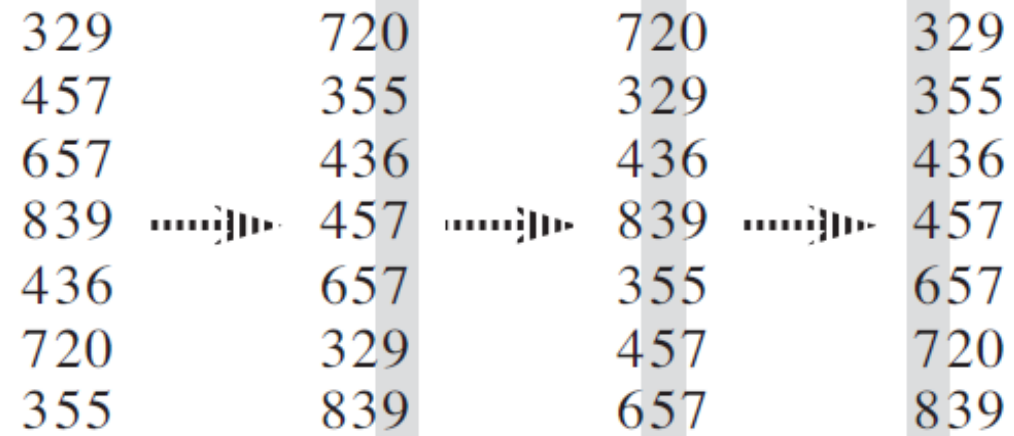
- Complexity: $\Theta(n + k)$
- Typically used when $k = O(n)$, complexity $\Theta(n)$
- Stable

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



RADIX-SORT(A, d)

- 1 **for** $i = 1$ **to** d
- 2 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 **$r = b$** .

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, $(n + 2^r) = \Theta(n)$, then **$r = \lfloor \log n \rfloor$**

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

Radix sort

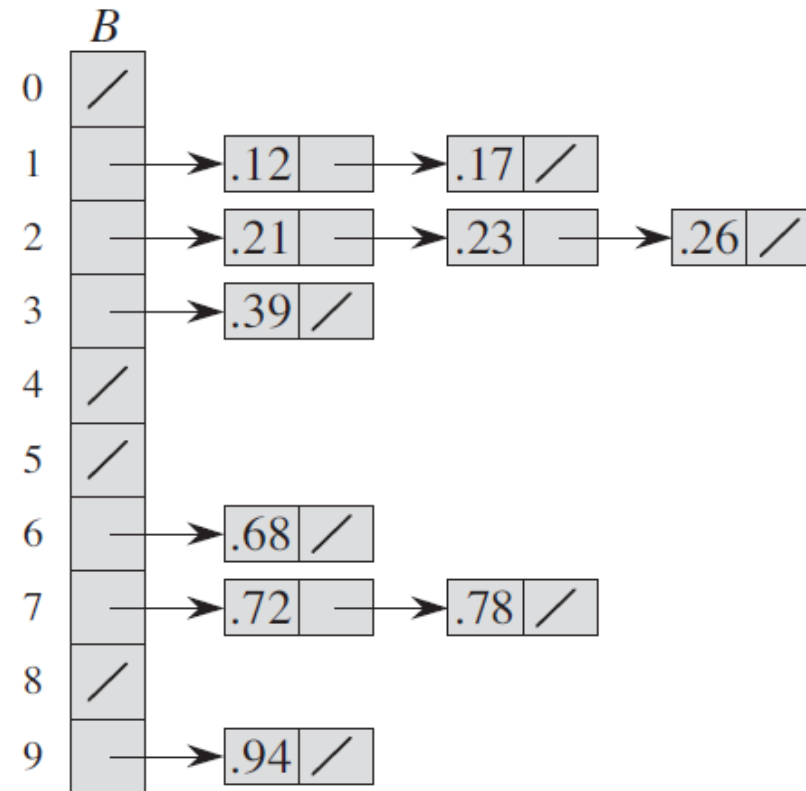
- Counting sort is NOT in-place
- Quicksort is better than radix sort
 - Better cache utilization
 - In-place

Bucket sort

A

1	.78
2	.17
3	.39
4	.26
5	.72
6	.94
7	.21
8	.12
9	.23
10	.68

(a)



(b)

Bucket sort

BUCKET-SORT(A)

```
1  let  $B[0..n-1]$  be a new array
2   $n = A.length$ 
3  for  $i = 0$  to  $n - 1$ 
4      make  $B[i]$  an empty list
5  for  $i = 1$  to  $n$ 
6      insert  $A[i]$  into list  $B[\lfloor nA[i] \rfloor]$ 
7  for  $i = 0$  to  $n - 1$ 
8      sort list  $B[i]$  with insertion sort
9  concatenate the lists  $B[0], B[1], \dots, B[n-1]$  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*)

```
1  min = A[1]
2  for i = 2 to A.length
3      if min > A[i]
4          min = A[i]
5  return min
```

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

Selection algorithm

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

```
RANDOMIZED-SELECT( $A, p, r, i$ )
1  if  $p == r$ 
2      return  $A[p]$ 
3   $q = \text{RANDOMIZED-PARTITION}(A, p, r)$ 
4   $k = q - p + 1$ 
5  if  $i == k$            // the pivot value is the answer
6      return  $A[q]$ 
7  elseif  $i < k$ 
8      return RANDOMIZED-SELECT( $A, p, q - 1, i$ )
9  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