Design and Analysis of Algorithms

Unit-1

Definition of Algorithm

- An **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.

- An **algorithm** is a set of steps to accomplish or complete a task that is described precisely enough that a computer can run it.

Characteristics of an algorithm

- ❖ **Input:** Algorithm must contain 0 or more input values.
- ❖ **Output:** Algorithm must contain 1 or more input values.
- ❖**Finiteness:** Algorithm must complete after finite number of steps.
- ❖**Definiteness:** Each instruction of the algorithm must be defined precisely or clearly.
- ❖**Effectiveness:** Every instruction must be basic i.e. simple instruction.

Design Strategy of Algorithm

- Divide and Conquer
- Dynamic Programming
- Branch and Bound
- Greedy Approach
- Backtracking
- Randomized Algorithm

Analysis of algorithms

Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Time complexity: The amount of time required by an algorithm is called time complexity.

Space complexity: The amount of space/memory required by an algorithm is called space complexity.

Analysis of algorithms

The main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis –

Worst case – The maximum number of steps taken on any instance of size **n**.
Best case – The minimum number of steps taken on any instance of size **n**.
Average case – An average number of steps taken on any instance of size **n**.

Running time of an algorithm

- **Running time of an algorithm is** the time taken by the algorithm to execute it successfully.
- The ***running time*** of an algorithm on a particular input is the number of primitive operations or "steps" executed.
- It is convenient to define the notion of step so that it is as machine independent as possible.
- It is denoted by **T(n)**, where n is the input size.

Pseudo code conventions

We use the following conventions in our pseudo code.

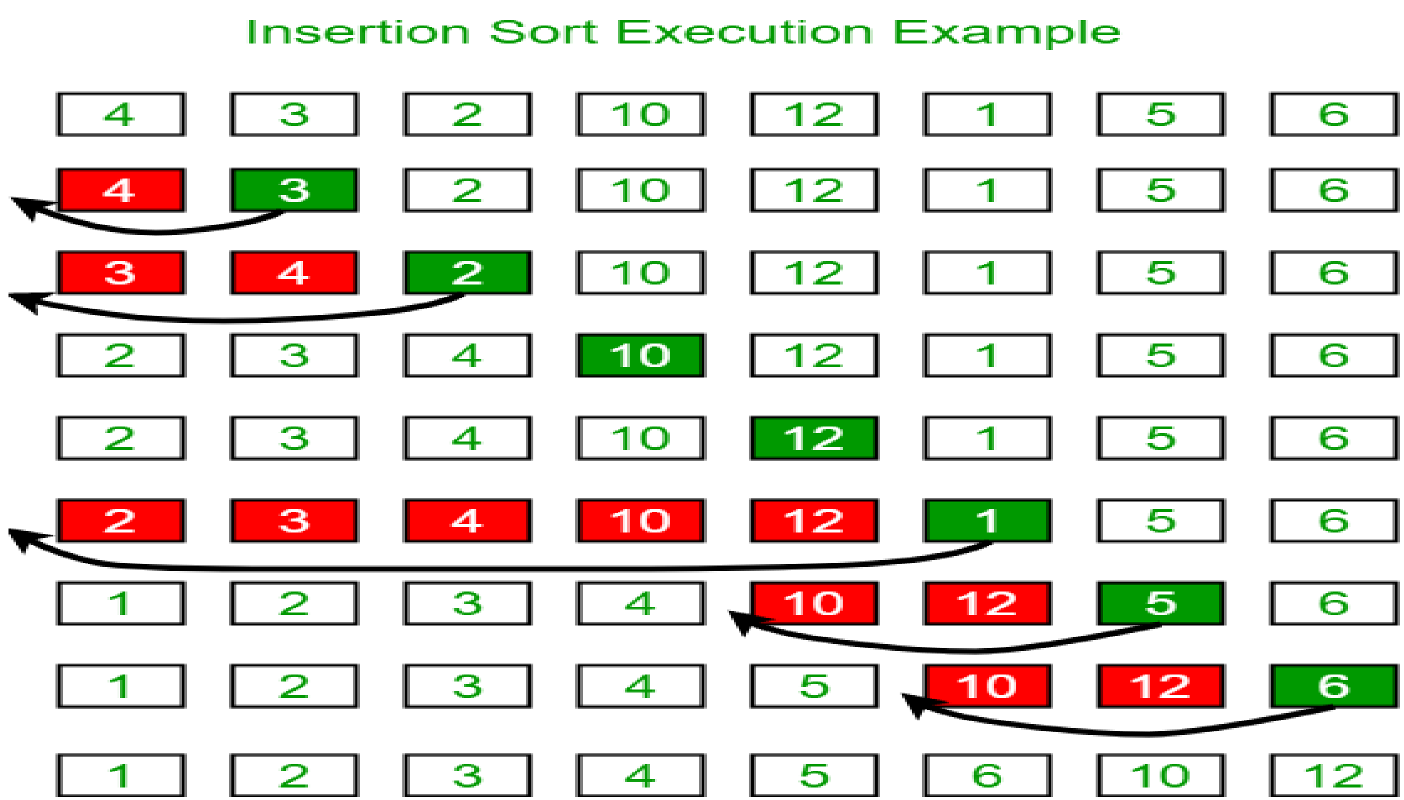
- Indentation is used to indicate a block.
- The looping constructs **while**, **for**, and **repeat-until** and the **if-else** conditional construct have interpretations similar to those in C.
- The symbol "//" indicates that the remainder of the line is a comment.
- A multiple assignment of the form i←j ← e assigns to both variables i and j the value of expression e; it should be treated as equivalent to the assignment j ← e followed by the assignment i ← j .

Pseudo code conventions

- We access array elements by specifying the array name followed by the index in square brackets. For example, A[i] indicates the i^{th} element of the array A.
- The notation ".." is used to indicate a range of values within an array. Thus, A[1..j] indicates the sub-array of A consisting of the j elements A[1], A[2],…………,A[j].
- The Boolean operators "and" and "or" are used.
- A **return** statement immediately transfers control back to the point of call in the calling procedure.

Insertion Sort

Ex. Sort the following elements by insertion sort:- 4, 3, 2, 10, 12, 1, 5, 6.



Insertion Sort Algorithm

Insertion_Sort(A)

1. n← length[A]
2. for j←2 to n
    1. a←A[j]
    2. //Insert A[j] into the sorted sequence A[1 .. j-1].
    3. i← j-1
    4. While i > 0 and a < A[i]
        1. A[i+1]←A[i]
        2. i← i-1
    5. A[i+1] ← a

Insertion Sort Algorithm Analysis

Insertion_Sort(A)

Instruction	cost	times
n← length[A]	c_1	1
for j←2 to n	c_2	n
a←A[j]	c_3	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 < A[i]	c_5	$\sum_{j=2}^{n} t_j$
A[i+1]←A[i]	c_6	$\sum_{j=2}^{n}(tj\text{-}1)$
i← i-1	c_7	$\sum_{j=2}^{n}(tj\text{-}1)$
A[i+1] ← a	c_8	n-1

Time complexity of Insertion sort

$$T(n) = c_1.1 + c_2.n + c_3.(n\text{-}1) + c_4.(n\text{-}1) + c_5.\sum_{j=2}^{n} t_j + c_6.\sum_{j=2}^{n}(tj\text{-}1) + c_7.\sum_{j=2}^{n}(tj\text{-}1) + c_8.(n\text{-}1) \;\ldots\ldots\ldots\ldots(1)$$

Here, there will be three case.

(1) Best case

(2) Worst case

(3) Average case

Time complexity of Insertion sort

Best case: This case will be occurred when data is already sorted.

In this case, value of t_j will be 1. Put the value of t_j in equation (1) and find T(n). Therefore,

$$\begin{aligned} T(n) &= c_1.1 + c_2.n + c_3.(n-1) + c_4.(n-1) + c_5.\textstyle\sum_{j=2}^{n} 1 \\ &\quad + c_6.\textstyle\sum_{j=2}^{n}(1-1) + c_7.\textstyle\sum_{j=2}^{n}(1-1) + c_8.(n-1) \\ &= (c_2+c_3+c_4+c_5+c_8).n + (c_1 - c_3 - c_4 - c_5 - c_8) \\ &= an + b \end{aligned}$$

Clearly T(n) is in linear form, therefore,

$$\mathbf{T(n) = \theta(n)}$$

Time complexity of Insertion sort

Worst case: This case will be occurred when data is in reverse sorted order.

In this case, value of t_j will be j. Put the value of t_j in equation (1) and find T(n). Therefore,

$$
\begin{aligned}
T(n) = {} & c_1.1 + c_2.n + c_3.(n-1) + c_4.(n-1) + c_5.\textstyle\sum_{j=2}^{n} j \\
& + c_6.\textstyle\sum_{j=2}^{n}(j-1) + c_7.\textstyle\sum_{j=2}^{n}(j-1) + c_8.(n-1) \\
& = c_1.1 + c_2.n + c_3.(n-1) + c_4.(n-1) + c_5.\,(n+2).(n-1)/2 \\
& + c_6.n(n-1)/2 + c_7.\,n(n-1)/2 + c_8.(n-1) \\
& = (c_5+c_6+c_7).n^2/2 + (c_2+c_3+c_4+(c_5 - c_6 - c_7)/2+c_8).n + \\
& \quad (c_1 - c_3 - c_4 - c_5 - c_8) \\
& = an^2 + bn + c
\end{aligned}
$$

Clearly T(n) is in quadratic form, therefore, $\mathbf{T(n) = \theta(n^2)}$

Time complexity of Insertion sort

Average case: This case will be occurred when data is in any order except best and worst case.

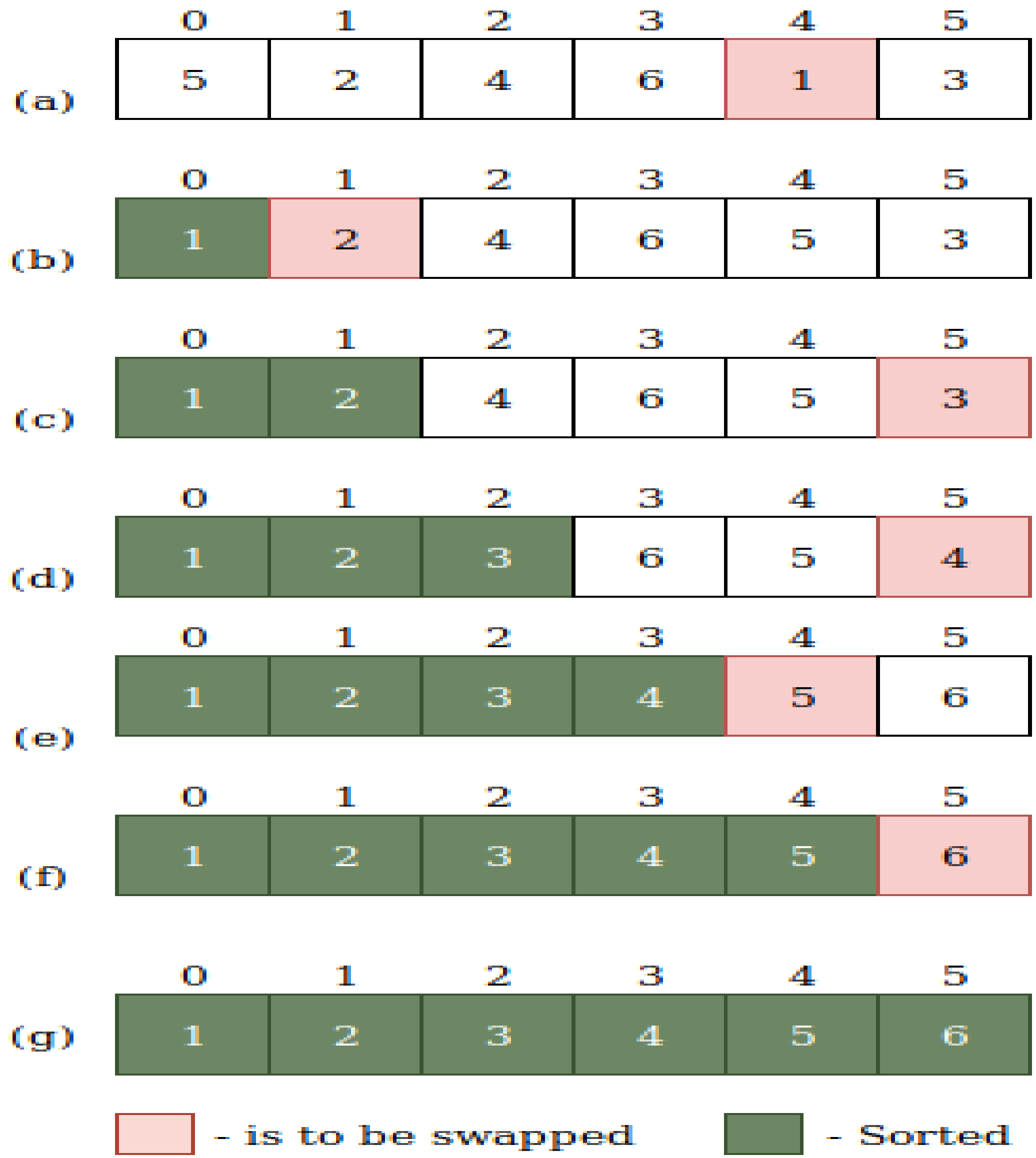
In this case, value of t_j will be j/2. Put the value of t_j in equation (1) and find T(n). Therefore,

$$
\begin{aligned}
T(n) = {} & c_1.1 + c_2.n + c_3.(n\text{-}1) + c_4.(n\text{-}1) + c_5.\textstyle\sum_{j=2}^{n} (j/2) \\
& + c_6.\textstyle\sum_{j=2}^{n}(j/2\text{-}1) + c_7.\textstyle\sum_{j=2}^{n}(j/2\text{-}1) + c_8.(n\text{-}1) \\
& = c_1.1 + c_2.n + c_3.(n\text{-}1) + c_4.(n\text{-}1) + c_5.\ (n+2).(n\text{-}1)/4 \\
& + c_6.(n\text{-}2)(n\text{-}1)/4 + c_7.\ (n\text{-}2)(n\text{-}1)/4 + c_8.(n\text{-}1) \\
& = (c_5+c_6+c_7).n^2/4 + (c_2+c_3+c_4+(c_5 - 3c_6 - 3c_7)/4+c_8).n + \\
& \quad (c_1 - c_3 - c_4 - c_5/2 + c_6/2 + c_7/2 - c_8) \\
& = an^2 + bn + c
\end{aligned}
$$

Clearly T(n) is in quadratic form, therefore, $\mathbf{T(n) = \theta(n^2)}$

Some other sorting algorithms

- **Selection Sort**



Selection Sort

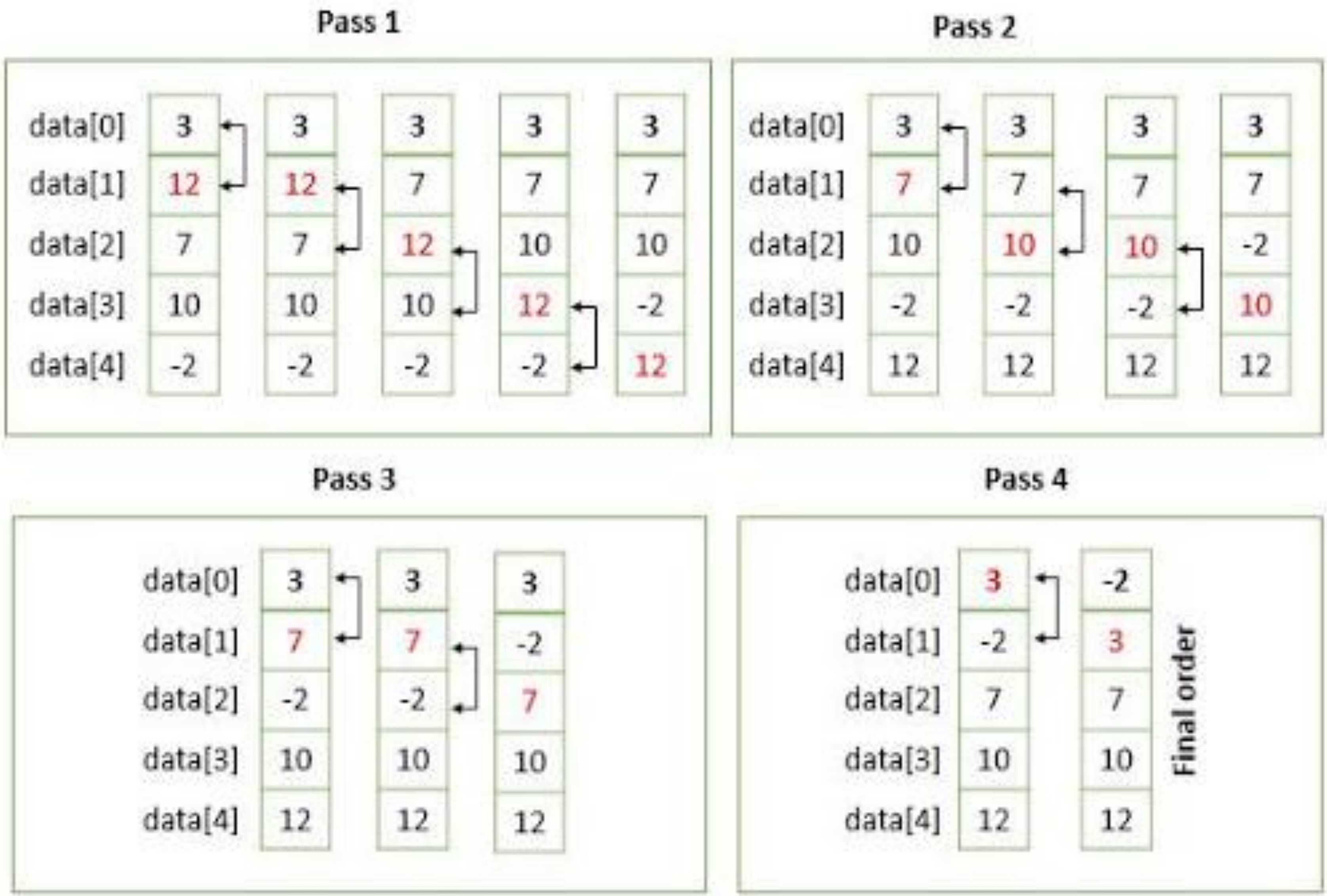
Some other sorting algorithms

```
        Selection_sort(A)
n ←length[A]
for i←1 to n-1
        min ← i
        for j←i+1 to n
                if(A[j] < A[min])
                        min ←j
        Interchange  A[i]  ↔ A[min]
```

Time complexity, $T(n) = \theta\ (n^2)$

Some other sorting algorithms

Bubble Sort



Some other sorting algorithms

Bubble _Sort(A)

```
Procedure bubblesort (List array, number length_of_array)
    for i=1 to length_of_array - 1;
        for j=1 to length_of_array – I;
            if array [j] > array [j+1] then
                temporary = array [j+1]
                array[j+1] = array [j]
                array[j] = temporary
            end if
        end of j loop
    end of i loop
return array
End of procedure
```

Divide and Conquer approach

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

Divide the problem into a number of sub-problems that are smaller instances of the same problem.

Conquer the sub-problems by solving them recursively. If the sub-problem sizes are small enough, however, just solve the sub problems in a straightforward manner.

Combine the solutions to the sub-problems into the solution for the original problem.

Analysis of Divide and Conquer based algorithm

When an algorithm contains a recursive call to itself, we can often describe its running time by a ***recurrence equation*** or ***recurrence***, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs. We can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

Analysis of Divide and Conquer based algorithm

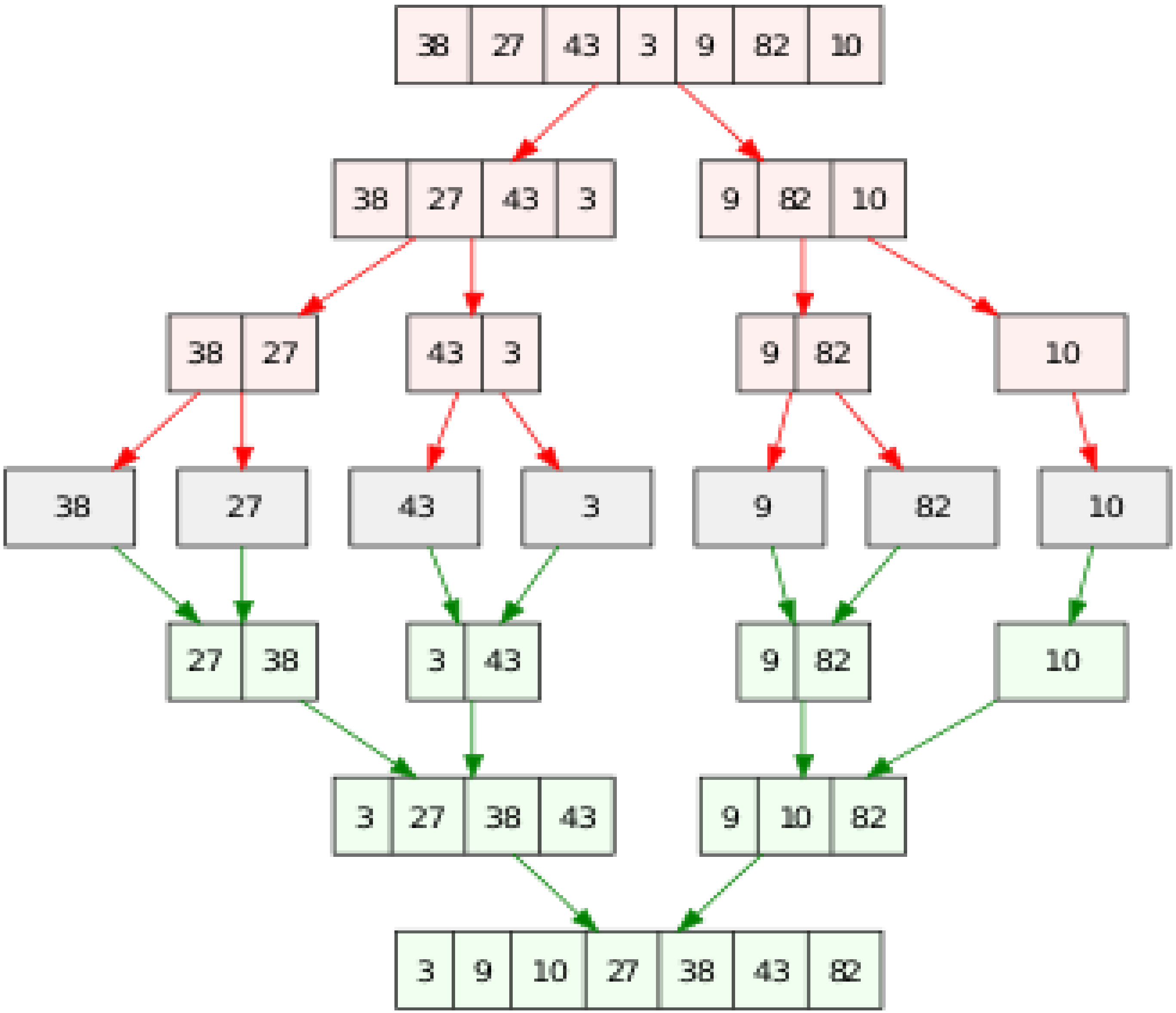
A recurrence equation for the running time of a divide-and-conquer algorithm uses the three steps of the basic paradigm.

The recurrence equation for the running time of a divide-and-conquer algorithm is the following:-

$$
\begin{aligned}
T(n) &= aT(n/b) + D(n) + C(n), \quad && \text{if } \; n > c \\
&= \theta(1) \quad , && \text{otherwise}
\end{aligned}
$$

Where, n is the size of original problem. a is the number of sub-problems in which the original problem divided at an instant. Each sub-problems has size n/b. c is a small integer.

Merge Sort



38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Merge Sort Algorithm

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)

Merge Sort Algorithm

```
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 Algorithm Analysis

The recurrence equation for the running time of merge sort algorithm will be

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

It can be modified as:-

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

Merge Sort Algorithm Analysis

When we solve recurrence equation, modify it as:-

$$T(n) = 2T(n/2) + \mathbf{cn}, \qquad \text{if } n > 1$$
$$= \mathbf{d}, \qquad \text{otherwise}$$

Here, c and d are some constants.

Merge Sort Algorithm Analysis

Iterative method:

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&= 2(2T(n/4) + cn/2) + cn \\
&= 2^2T(n/4) + 2cn \\
&= 2^2(2T(n/8)+cn/4) + 2cn \\
&= 2^3T(n/8)+ 3cn \\
&= \ldots\ldots\ldots\ldots\ldots\ldots \\
&= 2^kT(n/2^k) + kcn \\
&= nT(1) + c\, n\log(n) \quad \textbf{(Let } \mathbf{n = 2^k} \textbf{)} \\
&= dn + cn\log(n) \quad \textbf{(since } \mathbf{T(1) = d}\textbf{)} \\
&= \theta(n\log(n))
\end{aligned}
$$

Therefore, **$T(n) = \theta(n\log(n))$**

Asymptotic Notations

- The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers N={ 0, 1, 2, ………….}.
- We will use asymptotic notations to describe the running times of algorithms.
- Following notations are used to define the running time of algorithms.
  1. Θ–notation
  2. O-notation
  3. Ω-notation
  4. o-notation
  5. ω-notation

Asymptotic Notations

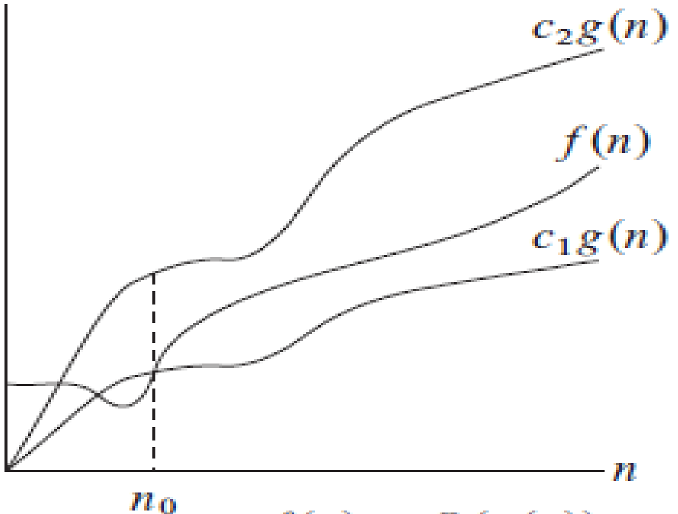
Θ–notation (Theta notation)

- For a given function g(n), it is denoted by Θ(g(n)).
- It is defined as following:-

$\Theta(g(n)) = \{ f(n) \,!\, \exists$ positive constants c_1, c_2 and n_0 such that

$0 \le c_1 g(n) \le f(n) \le c_2 g(n), \forall\ n \ge n_0 \}$

- This notation is said to be tight bound.
- If $f(n) \in \Theta(g(n))$ then $f(n) = \Theta(g(n))$



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

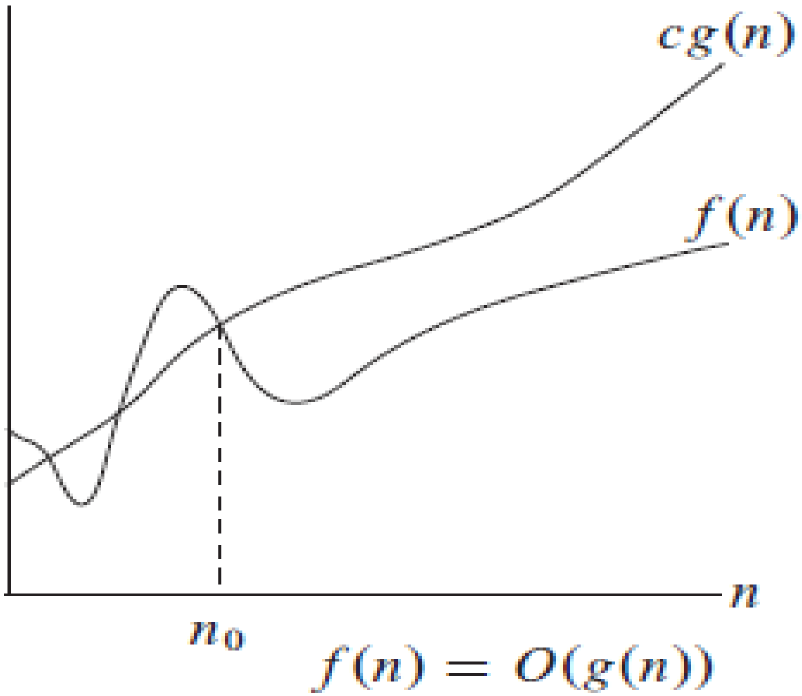
Asymptotic Notations

O–notation (Big-oh notation)

- For a given function g(n), it is denoted by O(g(n)).
- It is defined as following:-

$O(g(n)) = \{ f(n) \mid \exists$ positive constants c and n_0 such that

$0 \le f(n) \le cg(n), \forall\, n \ge n_0 \}$



- This notation is said to be upper bound.
- If $f(n) \in O(g(n))$ then $f(n) = O(g(n))$
- If $f(n) = \Theta(g(n))$ then $f(n) = O(g(n))$

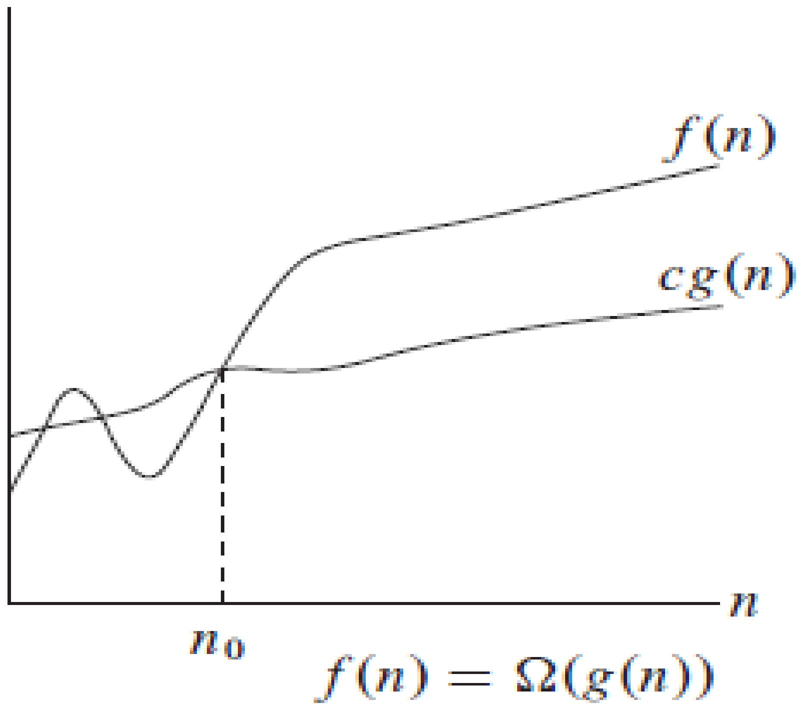
Asymptotic Notations

Ω–notation (Big-omega notation)

- For a given function g(n), it is denoted by $\Omega(g(n))$.
- It is defined as following:-

$\Omega(g(n)) = \{ f(n) ! \exists$ positive constants c and n_0 such that

$0 \le cg(n) \le f(n), \forall n \ge n_0 \}$



- This notation is said to be lower bound.
- If $f(n) \in \Omega(g(n))$ then $f(n) = \Omega(g(n))$
- If $f(n) = \Theta(g(n))$ then $f(n) = \Omega(g(n))$
- $f(n) = \Theta(g(n))$ iff $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$

Asymptotic Notations

o–notation (little-oh notation)

- The asymptotic upper bound provided by O-notation may or may not be asymptotically tight.
- o-notation denotes an upper bound that is not asymptotically tight.
- For a given function g(n), it is denoted by o(g(n)).
- It is defined as following:-

o(g(n)) = { f(n) ! for any positive constants c, there exists a constant n_o such that

$0 \le f(n) < cg(n), \forall\ n \ge n_o$ }

Asymptotic Notations

ω–notation (little-omega notation)

- The asymptotic lower bound provided by **Ω**-notation may or may not be asymptotically tight.
- **ω**-notation denotes an upper bound that is not asymptotically tight.
- For a given function g(n), it is denoted by **ω**(g(n)).
- It is defined as following:-

ω(g(n)) = { f(n) ! for any positive constants c, there exists a constant n_o such that $0 \leq cg(n) < f(n), \forall\ n \geq n_o$ }

Asymptotic Notations

Example: Show that $(1/2)n^2 - 3n = \theta(n^2)$.

Solution: Using definition of θ-notation,

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \quad \forall\, n \geq n_0$$

In this question, $f(n) = (1/2)n^2 - 3n$ and $g(n) = n^2$, therefore

$$c_1\, n^2 \leq (1/2)n^2 - 3n \leq c_2\, n^2, \quad \forall\, n \geq n_0$$

We divide above by n^2, we get

$$c_1 \leq (1/2)\text{-}\,(3/n) \leq c_2\,, \quad \forall\, n \geq n_0 \quad \text{……………….}(1)$$

Now, we have to find c_1, c_2 and n_0, such that equation (1) is satisfied.

Consider, left part of (1), $c_1 \leq (1/2)\text{-}\,(3/n)$ …………. (2)

The value of c_1 will be positive value less than or equal to the minimum value of $(1/2)\text{-}\,(3/n)$. Minimum value of $(1/2)\text{-}\,(3/n) = 1/14$. Therefore, $c_1 = 1/14$. This value of c_1 will satisfy equation (2) for $n \geq 7$.

Here, $c_1 = 1/14$ and $n \geq 7$ which satisfy (2).

Asymptotic Notations

Consider, right part of (1), $(1/2) - (3/n) \le c_2$, ... (3)

The value of c_2 will be positive value greater than or equal to the maximum value of $(1/2) - (3/n)$. Maximum value of $(1/2) - (3/n) = 1/2$. Therefore, $c_2 = 1/2$. This value of c_2 will satisfy equation (3) for $n \ge 1$.

Here, $c_2 = 1/2$ and $n \ge 1$ which satisfy (3).

Therefore, for $c_1 = 1/14$, $c_2 = 1/2$ and $n_0 = 7$, equation (1) is satisfied.

Hence by using definition of θ-notation ,

$$(1/2)n^2 - 3n = \theta(n^2).$$

It is proved.

Asymptotic Notations

Example: Show that $2n+5 = O(n^2)$.

Solution: Using definition of O-notation,

$$f(n) \leq cg(n) \;,\; \forall\, n \geq n_0$$

In this question, $f(n) = 2n+5$ and $g(n) = n^2$, therefore

$$2n+5 \leq c\, n^2 \quad \forall\, n \geq n_0$$

We divide above by n^2, we get

$$(2/n)+(5/n^2) \leq c \;,\; \forall\, n \geq n_0 \quad \ldots\ldots\ldots(1)$$

Now, we have to find c and n_0, such that equation (1) is satisfied.

Asymptotic Notations

The value of c will be positive value greater than or equal to the maximum value of $(2/n)+(5/n^2)$.

Maximum value of $(2/n)+(5/n^2) = 7$.

Therefore, $c = 7$.

Clearly equation (1) is satisfied for $c = 7$ and $n \geq 1$.

Hence by using definition of O-notation ,

$$2n+5 = O(n^2).$$

It is proved.

Asymptotic Notations

Example: Show that $2n^2+5n+6 = \boldsymbol{\Omega}(n)$.

Solution: Using definition of $\boldsymbol{\Omega}$ -notation,

$$cg(n) \le f(n)\ ,\ \forall\ n \ge n_0$$

In this question, $f(n) = 2n^2+5n+6$ and $g(n) = n$, therefore

$$cn \le 2n^2+5n+6\ ,\quad \forall\ n \ge n_0$$

We divide above by n, we get

$$c \le 2n + 5 + (6/n)\ ,\ \forall\ n \ge n_0 \quad \ldots\ldots\ldots(1)$$

Now, we have to find c and n_0, such that equation (1) is always satisfied.

Asymptotic Notations

The value of c will be positive value less than or equal to the minimum value of $2n + 5 + (6/n)$.

Minimum value of $2n + 5 + (6/n) = 12$.

Therefore, $c = 12$.

Clearly equation (1) is satisfied for $c = 12$ and $n \geq 2$.

Hence by using definition of **Ω** -notation ,

$$2n^2+5n+6 = \boldsymbol{\Omega}(n).$$

It is proved.

Asymptotic Notations

Example: Show that $2n^2 = o(n^3)$.

Solution: Using definition of o-notation,

$$f(n) < cg(n) \quad , \forall\, n \geq n_0$$

Here, $f(n) = 2n^2$, and $g(n) = n^3$. Therefore,

$$2n^2 < cn^3 , \qquad \forall\, n \geq n_0$$

We divide above by n^3, we get

$$(2/n) < c , \qquad \forall\, n \geq n_0 \ \ldots\ldots\ldots\ldots(1)$$

for $c = 1$, there will be $n_0 = 3$, which satisfy (1).

for $c = 0.5$, there will be $n_0 = 7$, which satisfy (1).

Therefore, for every c, there exists n_0 which satisfy (1).

Hence $2n^2 = o(n^3)$.

Asymptotic Notations

Example: Show that $2n^2 \neq o(n^2)$.

Solution: Using definition of o-notation,

$$f(n) < cg(n) \;, \forall\, n \geq n_0$$

Here, $f(n) = 2n^2$, and $g(n) = n^2$. Therefore,

$$2n^2 < cn^2 , \qquad \forall\, n \geq n_0$$

We divide above by n^2, we get

$$2 < c , \qquad \forall\, n \geq n_0 \;\ldots\ldots\ldots\ldots(1)$$

Clearly for $c = 1$, inequality (1) does not satisfy.

Therefore, for every c, there does not exist n_0 which satisfy (1). Hence $2n^2 \neq o(n^2)$.

Asymptotic Notations

Example: Show that $2n^2 = \boldsymbol{\omega}(n)$.

Solution: Using definition of $\boldsymbol{\omega}$ -notation,

$$cg(n) < f(n)\ ,\ \forall\ n \geq n_0$$

Here, $f(n) = 2n^2$, and $g(n) = n$. Therefore,

$$cn < 2n^2, \qquad \forall\ n \geq n_0$$

We divide above by n, we get

$$c < 2n, \qquad \forall\ n \geq n_0 \ \ldots\ldots\ldots\ldots(1)$$

for $c = 1$, there will be $n_0 = 1$, which satisfy (1).

for $c = 10$, there will be $n_0 = 6$, which satisfy (1).

Therefore, for every c, there exists n_0 which satisfy (1).

Hence $2n^2 = \boldsymbol{\omega}(n)$.

Asymptotic Notations

Example: Show that $2n^2 \neq \boldsymbol{\omega}(n^2)$.

Solution: Using definition of $\boldsymbol{\omega}$ -notation,

$$cg(n) < f(n)\ ,\ \forall\ n \geq n_0$$

Here, $f(n) = 2n^2$, and $g(n) = n^2$. Therefore,

$$cn^2 < 2n^2\ , \qquad \forall\ n \geq n_0$$

We divide above by n^2, we get

$$c < 2\ , \qquad \forall\ n \geq n_0 \ \ldots\ldots\ldots\ldots(1)$$

Clearly for $c = 3$, there does not exists n_0, which satisfy (1).

Therefore, for every c, there does not exist n_0 which satisfy (1). Hence $2n^2 \neq \boldsymbol{\omega}(n^2)$.

Asymptotic Notations

Example: Show that using definition of notations

(a) $3n^3 - 10n + 50 = \theta(n^3)$

(b) $5n^2 - 100n \neq \theta(n^3)$

(c) $3n^3 - 10n + 50 = O(n^3)$

(d) $5n^2 - 100n \neq O(n)$

(e) $3n^3 - 10n + 50 = \mathbf{\Omega}(n^3)$

(f) $5n^2 - 100n \neq \mathbf{\Omega}(n^3)$

Asymptotic Notations

Limit based method to compute notations for a function

First compute $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$.

(1) If c is a constant such that $0 < c < \infty$, then $f(n) = \theta(g(n))$.

(2) If c is a constant such that $0 \leq c < \infty$, then $f(n) = O(g(n))$.

(3) If c is a constant such that $0 < c \leq \infty$, then $f(n) = \boldsymbol{\Omega}(g(n))$.

(4) If c is a constant such that $c = 0$, then $f(n) = o(g(n))$.

(5) If c is a constant such that $c = \infty$, then $f(n) = \boldsymbol{\omega}(g(n))$.

Exercises

1. Let f(n) and g(n) be asymptotically non-negative functions. Using the basic definition of ,θ-notation, prove that **$\max(f(n), g(n)) = \theta(f(n)+g(n))$.**

2. Show that for any real constants a and b, where $b > 0$,

$$(n+a)^b = \theta(n^b)$$

3. Solve the followings:-

(a) Is $2^{n+1} = O(2^n)$?

(b) Is $2^{2n} = O(2^n)$?

4. Prove that $o(g(n)) \cap \boldsymbol{\omega}(g(n))$ is the empty set.

5. Prove that $n! = \boldsymbol{\omega}(2^n)$ and $n! = o(n^n)$.

6. Which is asymptotically larger: $\lg(\lg^* n)$ or $\lg^*(\lg n)$?

Exercise(cont.)

7. Arrange the following in ascending order of growth or rank the following functions by order of growth.

n^3, $(3/2)^n$, 2^n, n^2, $\log(n)$, 2^{2n}, $\log\log(n)$, $n!$, e^n.

8. Let f(n) and g(n) be two asymptotically positive functions. Prove or disprove the following:-

(a) $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.

(b) $f(n) + g(n) = \theta(\min(f(n),g(n)))$.

(c) $f(n) = O(g(n))$ implies $\lg(f(n)) = O(\lg(g(n)))$, where $\lg(g(n)) \geq 1$ and $f(n) \geq 1$ for all sufficiently large n.

(d) $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$.

(e) $f(n) = O(((f(n))^2)$

Exercise(cont.)

Solution-(4):

Assume $f(n) \in o(g(n)) \cap \boldsymbol{\omega}(g(n))$.

$\Rightarrow f(n) \in o(g(n))$ and $f(n) \in \boldsymbol{\omega}(g(n))$

$\Rightarrow f(n) < cg(n)$ and $cg(n) < f(n)$, for any c

$\Rightarrow$ Both inequality can not be true for any c.

Therefore, our assumption is incorrect.

Hence, $f(n) \notin o(g(n)) \cap \boldsymbol{\omega}(g(n))$. Therefore,

$o(g(n)) \cap \boldsymbol{\omega}(g(n))$ is the empty set.

Exercise(cont.)

Solution-(5): $n! = \boldsymbol{\omega}(2^n)$ and $n! = o(n^n)$

(i) $c2^n < n!$

for $c = 1$, $n_0 = 4$

for $c = 10$, $n_0 = 6$

(ii) $n! < cn^n$

for $c = 1$, $n_0 = 2$

for $c = 0.1$, $n_0 = 3$

Asymptotic Notations

Exercise(cont.)

Solution-(6): lg(lg*n) or lg*(lg n)

lg(lg*n) = lg(lg n)*

lg*(lg n) = (lg lg n)*

Solution-(7):

n^3, $(3/2)^n$, 2^n, n^2, log(n), 2^{2n}, loglog(n), n!, e^n

Ascending order is

loglog(n), log(n), n^2, n^3, $(3/2)^n$, 2^n , e^n, n!, 2^{2n}

Asymptotic Notations

AKTU questions

1. Take the following list of functions and arrange them in ascending order of growth rate. That is, if function g(n) immediately follows function f(n) in your list, then it should be the case that f(n) is O(g(n)). $f_1(n) = n^{2.5}$, $f_2(n) = \sqrt{2^n}$, $f_3(n) = n + 10$, $f_4(n) = 10^n$, $f_5(n) = 100^n$, and $f_6(n) = n^2 \log n$

1. Rank the following by growth rate: n, $2\lg \sqrt{n}$, $\log n$, $\log (\log n)$, $\log^2 n$, $(\lg n)^{\lg n}$, 4, $(3/2)^n$, $n!$

Recurrence relation

Recurrence equations will be of the following form:-

(1) $T(n) = aT(n/b) + f(n)$
(2) $T(n) = T(n-1) + n$
(3) $T(n) = T(n/3) + T(2n/3) + n$
(4) $T(n) = T(n-1) + T(n-2)$

Some approaches to sole recurrence relations

(1) Iterative method
(2) Substitution method
(3) Recurrence Tree
(4) Master theorem method

Substitution method

The ***substitution method*** for solving recurrences comprises two steps:

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

Example: Find the upper bound of following recurrence relation $T(n) = 2T(\lfloor n/2 \rfloor) + n$. ....(1)

Solution: We will solve this using substitution method.

Guess the upper bound of this equation is $T(n) = O(n\log n)$.

Now, we have to prove that this guessing solution is correct.

By definition of upper bound,

$$T(n) \le c\, n\log n, \quad \forall\, n \ge n_0. \quad ....(2)$$

Substitution method

We will prove inequality (2) using induction method.

Assume $T(1) = 1$.

Using equation (1),

$$T(2) = 2T(\lfloor 2/2 \rfloor) + 2 = 2T(1) + 2 = 4$$

$$T(3) = 2T(\lfloor 3/2 \rfloor) + 3 = 2T(1) + 3 = 5$$

For n=1.

$$T(1) \leq c\ 1\log 1 \Rightarrow 1 \leq c\ .\ 0 \Rightarrow 1 \leq 0 \text{ (False)}$$

Therefore, equation (2) is false for $n = 1$.

For n=2.

$$T(2) \leq c\ 2\log 2 \Rightarrow 4 \leq c\ .\ 2 \Rightarrow 4 \leq 2c \text{ (True for } c \geq 2)$$

Therefore, equation (2) is true for $n = 2$.

For n=3.

$$T(3) \leq c\ 3\log 3 \Rightarrow 5 \leq 3c.\log 3 \text{ (True for } c \geq 2)$$

Therefore, equation (2) is true for $n = 3$.

Substitution method

Assume equation (2) is true for n= n/2. We will prove for n.

Since equation (2) is true for n = n/2, therefore

$T(n/2) \leq c\,(n/2)\log(n/2), \quad \forall\, n \geq n_0$ ....(3)

Now for n,

$$
\begin{aligned}
T(n) &= 2T(\lfloor n/2 \rfloor) + n \\
&\leq 2c\,(\lfloor n/2 \rfloor)\log(\lfloor n/2 \rfloor) + n \\
&\leq 2c\,(n/2)\log(n/2) + n \\
&= cn\,(\log n - \log 2) + n \\
&= cn\,(\log n - 1) + n \\
&= cn\log n - cn + n \\
&\leq cn\log n \quad \text{if } c \geq 1
\end{aligned}
$$

Therefore, $T(n) \leq cn\log n$ if $c \geq 1$.

Hence, equation (2) is also proved for n. Therefore, guessing solution is correct.

Therefore the upper bound of given recurrence relation is

$T(n) = O(n\log n)$.

Substitution method

Example: Find the upper bound of following recurrence relation $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$

Solution: When n is large, the difference between $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 17$ is not that large. Therefore, given equation is equivalent to the following equation

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

Similarly, since $\lfloor n/2 \rfloor$ and $n/2$ are approximately same, therefore above equation is equivalent to the following equation

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

Since this equation is equivalent to the previous question therefore upper bound will be $T(n) = O(n\log n)$.

Substitution method

Example: Solve the following recurrence relation

$$T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \lg n$$

Solution: Here we change the variable n to m.

$$\text{Let } n = 2^m$$

Put $n = 2^m$ in given equation, we get

$$T(2^m) = 2T(\lfloor\sqrt{2^m}\rfloor) + m$$

$$= 2T(\lfloor 2^{m/2}\rfloor) + m$$

Let $T(2m) = S(m)$, therefore

$$S(m) = 2\, S(\lfloor m/2\rfloor) + m$$

Since this equation is equialent to the first question, therefore its solution will be $S(m) = m\log(m)$.

Hence $T(n) = T(2m) = S(m) = m\log(m) = \log(n)\,\log\log(n)$

i.e. $T(n) = \log(n)\,\log\log(n)$

Substitution method

Example: Solve the following recurrence relation

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

Solution: Since $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ are approximately equal to $n/2$, therefore given equation is equivalent to following equation

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

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

Now, we guess the solution is $T(n) = O(n)$. Therefore, we have to prove $T(n) \leq cn\text{-}d$, $\forall\ n \geq n_0$. ....(2)

Assume this is true for $n = n/2$, therefore

$$T(n/2) \leq cn/2 - d \text{ ....(3)}$$

Now for n,

$$\begin{aligned} T(n) &= 2T(n/2) + 1 \\ &\leq 2\,(cn/2 - d) + 1 \text{ (using equation (3))} \\ &= cn - 2d + 1 \\ &\leq cn\text{-}d \text{ if } d \geq 1 \end{aligned}$$

Hence $T(n) \leq cn\text{-}d$ for $d \geq 1$. Therefore **$T(n) = O(n)$.**

Recurrence tree method

- In a ***recursion tree***, each node represents the cost of a single sub-problem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion.

- A recursion tree is best used to generate a good guess, which you can then verify by the substitution method.

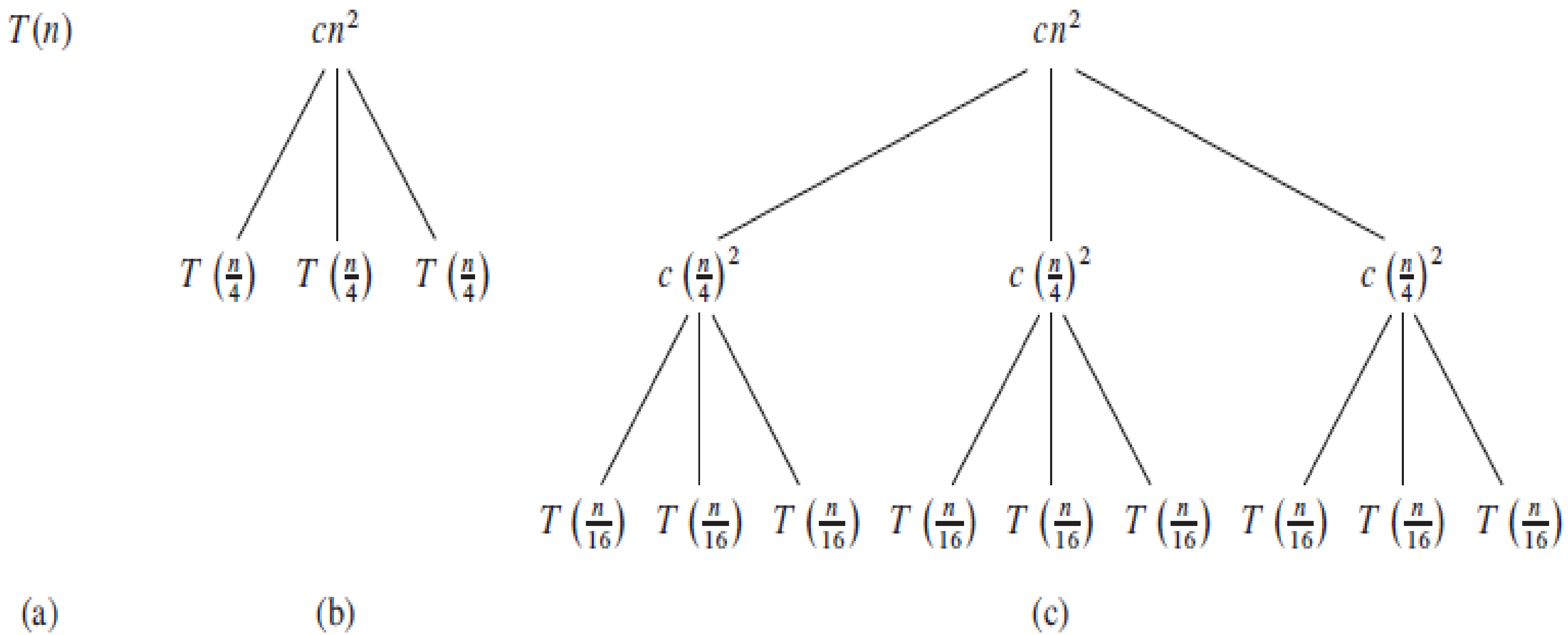
Recurrence tree method

Example: Solve the following recurrence equation

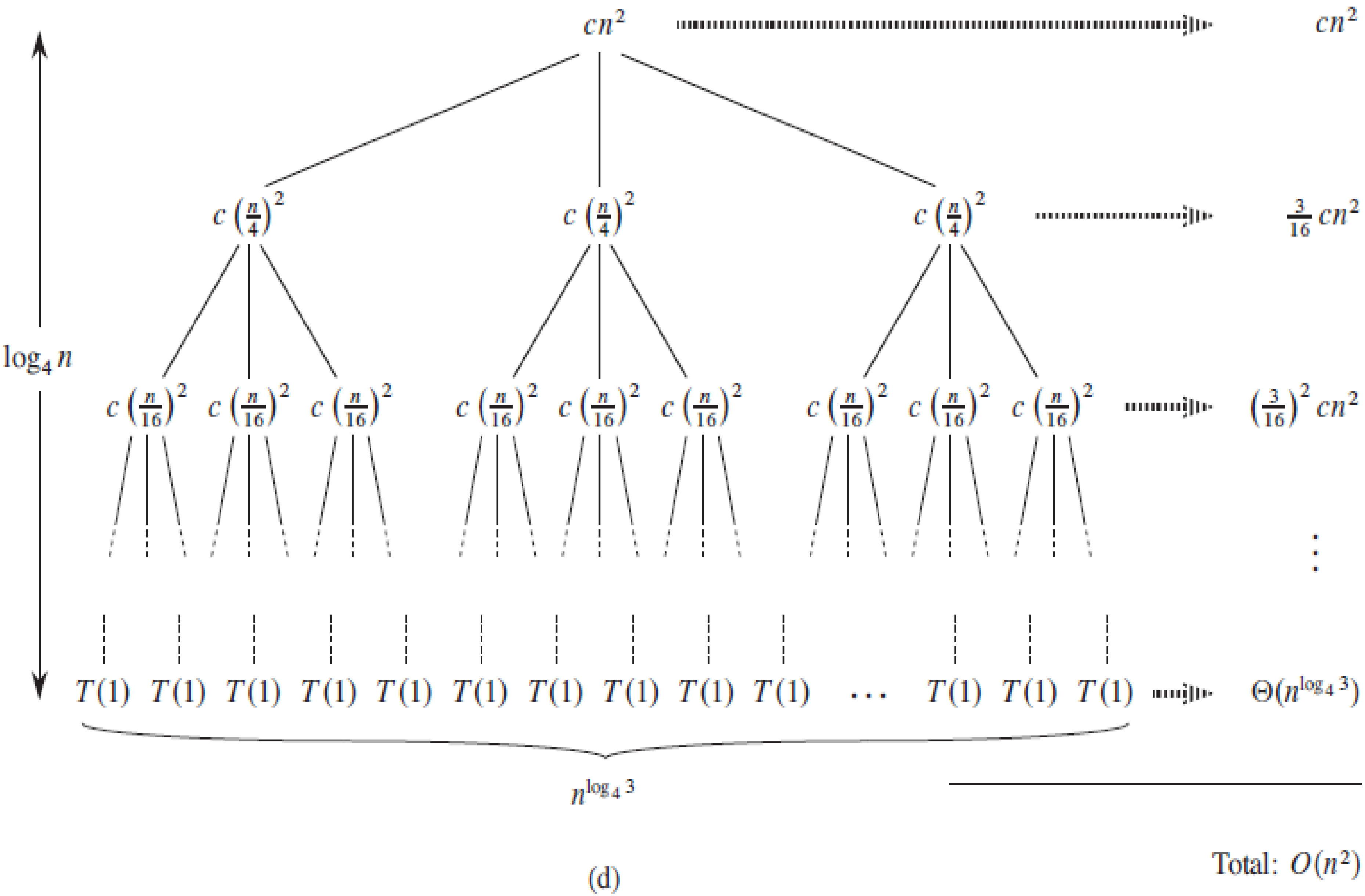
$$T(n) = 3T(\lfloor n/4 \rfloor) + \theta(n^2)$$

Solution: This equation is equivalent to the following equation $T(n) = 3T(n/4) + cn^2$

Here, we create recurrence tree for above equation.



Recurrence tree method



Recurrence tree method

Let h is the height of the tree. Then

$n/4^h = 1 \Rightarrow h = \log_4 n$

Now we add up the costs over all levels to determine the cost for the entire tree:

$T(n) = cn^2 + (3/16)cn^2 + (3/16)^2\, cn^2 + \ldots\ldots\ldots\ldots\ldots\ldots +$

$(3/16)^{\log_4 n - 1}\, cn^2 + \theta(n^{\log_4 3})$

$= \; cn^2 \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i + \theta(n^{\log_4 3})$

$< \; cn^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i + \theta(n^{\log_4 3})$

$= cn^2\, (1/(1-(3/16))) + \theta(n^{\log_4 3})$

$= (16/13)\, cn^2 + \theta(n^{\log_4 3})$

$= O(n^2)$

Therefore, solution will be **$T(n) = O(n^2)$.**

Recurrence tree method

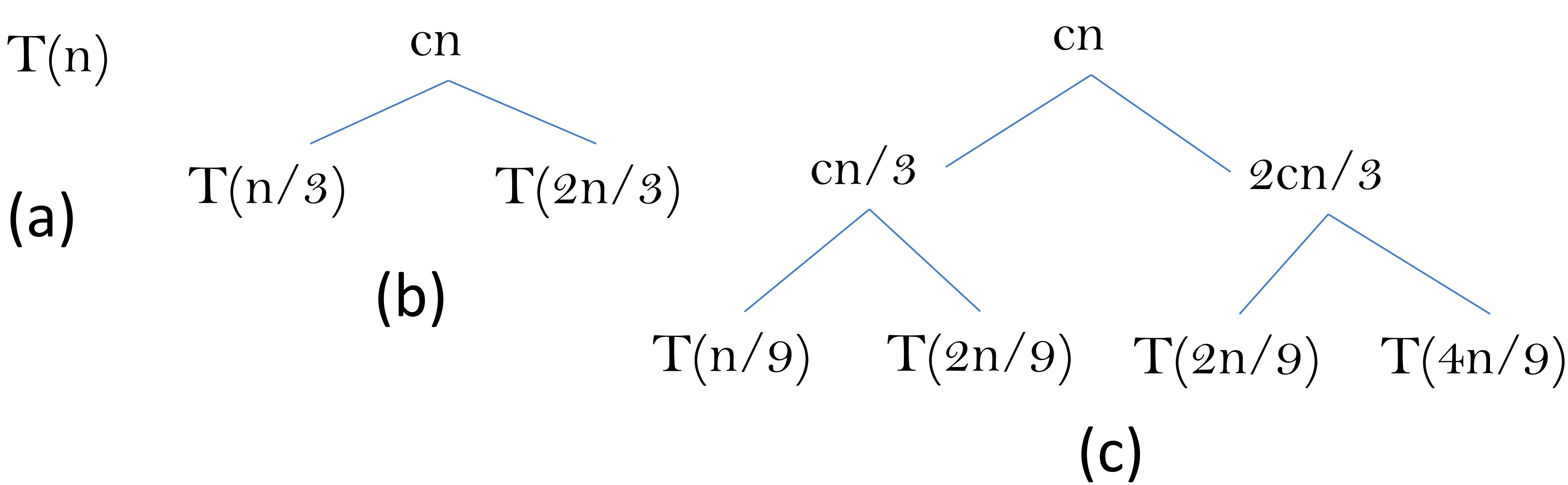
Example: Solve the following recurrence relation using recurrence tree method

$$T(n) = T(n/3) + T(2n/3) + \theta(n)$$

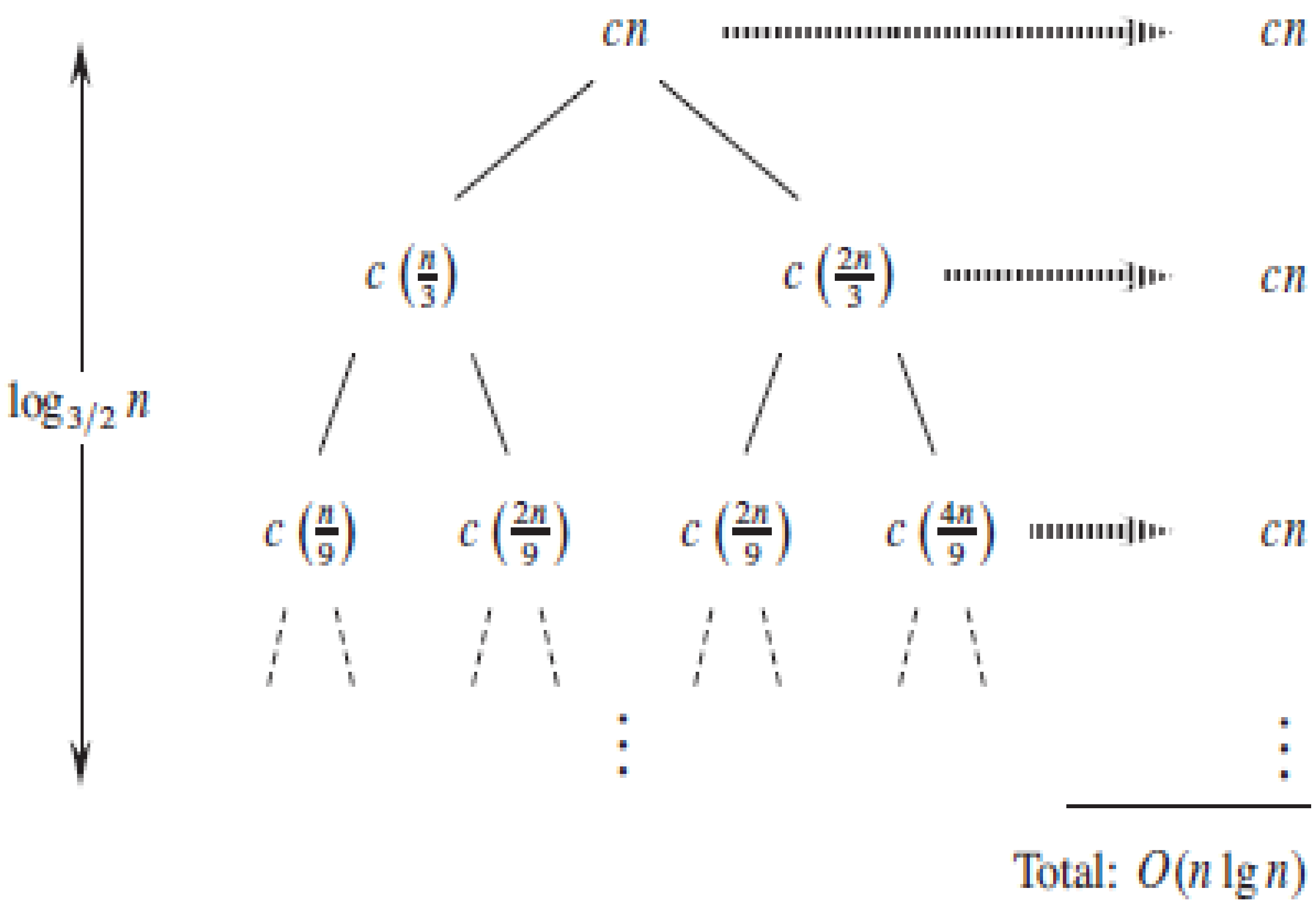
Solution: This recurrence relation is equivalent to the following relation

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

Recurrence tree for this equation will be the following:-



Recurrence tree method



(d)

Recurrence tree method

Let the height of the tree is h. Therefore

$$\frac{n}{\left(\frac{3}{2}\right)^{h}} = 1 \Rightarrow \text{ n} = \lg_{3/2}\text{n}$$

Therefore, the total cost of the tree is

$$T(n) < cn + cn + \ldots\ldots\ldots\ldots\ldots\ldots\ldots + cn$$

$$= (h+1)\ cn$$

$$= (\lg_{3/2}n + 1)cn$$

$$= cn\lg_{3/2}n + cn$$

$$= O(n\lg_{3/2}n)$$

$$= O(n\lg n)$$

Therefore **$T(n) = O(n\lg n)$**

Recurrence tree method

Exercise

(1) Draw the recursion tree for $T(n) = 4\,T(\lfloor n/2 \rfloor) + cn$, where c is a constant and provide a tight asymptotic bound on its solution. Verify your bound by the substitution method.

(2) Use a recursion tree to give an asymptotically tight solution to the recurrence $T(n) = T(n-a) + T(a) + cn$, where $a \geq 1$ and $c > 0$ are constants.

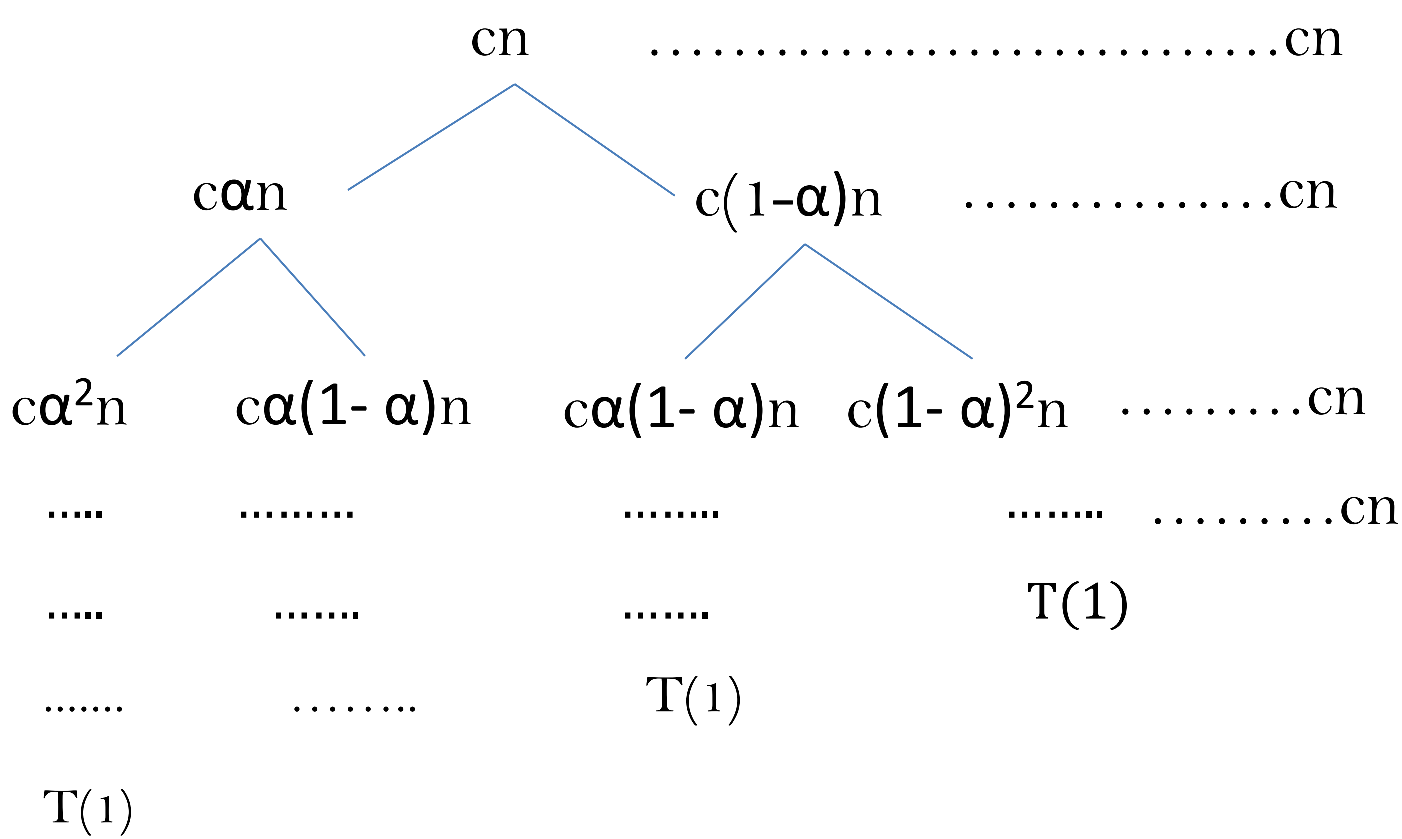
(3) Use a recursion tree to give an asymptotically tight solution to the recurrence $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$, where α is a constant in the range $0 < \alpha < 1$ and $c > 0$ is also a constant.

Recurrence tree method

Exercise(Solution)

(3.) Assume $\frac{1}{2} \le \alpha < 1$. Then $0 < (1-\alpha) \le \frac{1}{2}$.

Recurrence tree for this recurrence relation will be the following:



Let h is the height of the tree. Therefore,

$$\alpha^h n = 1 \Rightarrow h = \log_{1/\alpha}(n)$$

Therefore, total cost of the tree

$$T(n) = cn + cn + cn + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots + cn$$

$$= (h+1)\, cn$$

$$= (\log_{1/\alpha}(n) + 1)\, cn$$

$$= cn \log_{1/\alpha}(n) + cn$$

$$= O(n \log_{1/\alpha}(n))$$

$$= O(n\log n)$$

Therefore, $T(n) = O(n\log n)$.

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 Theorem Method

Example: Solve the following recurrence relations using master theorem method

(a) $T(n) = 9T(n/3) + n$

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

(c) $T(n) = 3T(n/4) + n \log n$

Solution:

(a) Consider $T(n) = 9T(n/3) + n$

In this recurrence relation, $a = 9$, $b = 3$ and $f(n) = n$.

Therefore, $n^{\log_b a} = n^{\log_3 9} = n^2$

Clearly, $n^{\log_b a} > f(n)$, therefore case1 can be applied.

Now determine ϵ such that $f(n) = O(n^{2-\epsilon})$. Here $\epsilon = 1$.

Therefore case 1 will be applied.

Hence solution will be $T(n) = \theta(n^2)$.

Master Theorem Method

Solution:

(b) Consider $T(n) = T(2n/3) + 1$

In this recurrence relation, $a = 1$, $b = 3/2$ and $f(n) = 1$.

Therefore, $n^{log_b a} = n^{\log_{3/2} 1} = 0$

Clearly, $f(n) = \theta(n^{\log_b a})$, therefore case 2 will be applied.

Hence solution will be $T(n) = \theta(\log n)$.

Master Theorem Method

Solution:

(c) Consider $T(n) = 3T(n/4) + n \log n$

In this recurrence relation, $a = 3$, $b = 4$ and $f(n) = n \log n$.

Therefore, $n^{\log_b a} = n^{\log_4 3} = n^{0.793}$

Clearly, $n^{\log_b a} < f(n)$, therefore case 3 can be applied.

Now determine ϵ such that $f(n) = \Omega(n^{0.793+\epsilon})$. Here $\epsilon = 0.207$.

Now, $af(n/b) \leq cf(n)$ imply that $3f(n/4) \leq cf(n)$

$$\Rightarrow 3(n/4)\log(n/4) \leq c\, n \log n$$

$$\Rightarrow (3/4)\log(n/4) \leq c \log n$$

Clearly above inequality is satisfied for $c = 3/4$. Therefore case 3 will be applied.

Hence solution will be $T(n) = \theta(n \log n)$.

Master theorem method

Example: Solve the following recurrence relation

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

Solution: Here, $a = 2$, $b=2$ and $f(n) = n \log n$.

$$n^{\log_b a} = n^{\log_2 2} = n$$

If we compare $n^{\log_b a}$ and $f(n)$, we get $f(n)$ is greater than $n^{\log_b a}$. Therefore, case 3 may be applied.

Now we have to determine $\epsilon > 0$ which satisfy $f(n) = \Omega(n^{\log_b a+\epsilon})$, i.e. $n \log n = \Omega(n^{1+\epsilon})$. Clearly there does not exist any ϵ which satisfy this condition. Therefore case 3 can not be applied. Other two cases are also not satisfied. Therefore Master theorem can not be applied in this recurrence relation.

Generalized Master theorem

Theorem: If $f(n) = \theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$, then the solution of recurrence will be $T(n) = \theta(n^{\log_b a} \lg^{k+1} n)$.

Now, consider the previous example:-

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

Solve it using aboe theorem,

Here a =2, b= 2, and k = 1. Therefore, the solution of this recurrence will be

$$T(n) = \theta(n^{\log_2 2} \lg^{1+1} n)$$

$$= \theta(n \lg^2 n)$$

Hence, $T(n) = \theta(n \lg^2 n)$

Recurrence relation

Exercise

1. Use the master method to give tight asymptotic bounds for the following recurrences:-

(a) $T(n) = 8T(n/2) + \theta(n^2)$

(b) $T(n) = 7T(n/2) + \theta(n^2)$

(c) $T(n) = 2T(n/4) + 1$

(d) $T(n) = 2T(n/4) + \sqrt{n}$

2. Can the master method be applied to the recurrence $4T(n/2) + n^2 \log n$? Why or why not? Give an asymptotic upper bound for this recurrence.

Recurrence relation

3. Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$. Make your bounds as tight as possible, and justify your answers.

(a) $T(n) = 2T(n/2) + n^4$

(b) $T(n) = T(7n/10) + n$

(c) $T(n) = 16T(n/4) + n^2$

(d) $T(n) = 2T(n/4) + \sqrt{n}$

(e) $T(n) = T(n-2) + n^2$

(f) $T(n) = 7T(n/3) + n^2$

(g) $T(n) = 3T(n/3 -2) + n/2$

Recurrence relation

4. Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for sufficiently small n. Make your bounds as tight as possible, and justify your answers.

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

(b) $T(n) = 3T(n/3) + n/\lg n$

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

(d) $T(n) = T(n/2) + T(n/4) + T(n/8) + n$

(e) $T(n) = T(n-1) + 1/n$

(f) $T(n) = T(n-1) + \lg n$

(g) $T(n) = T(n-2) + 1/\lg n$

(h) $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$

Recurrence relation

Some exercise solution

(3-e) Recurrence relation is $T(n) = T(n-2) + n^2$.

We will solve it using iteration method.

$T(n) = T(n-2) + n^2$

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

$= T(n-6) + (n-4)^2 + (n-2)^2 + n^2$

....

....

$= T(0) + 2^2 + 4^2 + 6^2 + \ldots\ldots\ldots\ldots\ldots\ldots + (n-2)^2 + n^2$

$= d + 2^2 + 4^2 + 6^2 + \ldots\ldots\ldots + (n-2)^2 + n^2$ (Let $T(0) = d$)

$= d + \frac{n(n+1)(n+2)}{6} = \theta(n^3)$

Recurrence relation

Some exercise solution

(3-g) Recurrence relation is $T(n) = 3T(n/3 - 2) + n/2$.

This recurrence relation is equivalent to the following equation $T(n) = 3T(n/3) + n/2$

Apply master theorem, here $a = 3$, $b = 3$ and $f(n) = n/2$.

$$n^{\log_b a} = n^{\log_3 3} = n$$

Clearly, $f(n) = n/2 = \theta(n) = \theta(n^{\log_b a})$

Therefore case 2 will be applied. Hence the solution is

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

Recurrence relation

Some exercise solution

(4-a) Recurrence relation is $T(n) = 4T(n/3) + n \lg n$.

Apply master theorem, here $a = 4$, $b = 3$ and $f(n) = n \lg n$.

$$n^{\log_b a} = n^{\log_3 4} = n^{1.26}$$

Clearly, $f(n) = n \lg n = O(n^{1.26-\epsilon}) = O(n^{\log_b a - \epsilon})$

Therefore case 1 will be applied. Hence the solution is

$$T(n) = \theta(n^{1.26})$$

Recurrence relation

Some exercise solution

(4-b) Recurrence relation is $T(n) = 3T(n/3) + n/\lg n$.

We will use recurrence tree method to solve it.

Recurrence tree will be

$$n/\lg n \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad n/\lg n$$

$$\left(\frac{n}{3}\right)/\lg\left(\frac{n}{3}\right) \quad \left(\frac{n}{3}\right)/\lg\left(\frac{n}{3}\right) \quad \left(\frac{n}{3}\right)/\lg\left(\frac{n}{3}\right) \quad \ldots\ldots\ldots\ldots \quad n/\lg\left(\frac{n}{3}\right)$$

$$\left(\frac{n}{9}\right)/\lg\left(\frac{n}{9}\right) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad \left(\frac{n}{9}\right)/\lg\left(\frac{n}{9}\right) \quad \ldots\ldots \quad n/\lg\left(\frac{n}{3^2}\right)$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$T(1)=d \quad T(1) = d \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots T(1) = d \ldots\ldots\; 3^h d$

Here d -> constant , h-> height of tree

Recurrence relation

Now, we calculate height of tree.

Clearly, $\frac{n}{3^h} = 1$, $\Rightarrow$ h = $log_3 n$.

Now, total cost of this tree is

$$T(n) = \frac{n}{lgn} + \frac{n}{\lg(\frac{n}{3})} + \frac{n}{\lg(\frac{n}{3^2})} + \ldots\ldots\ldots\ldots + \frac{n}{\lg(\frac{n}{3^{h-1}})} + 3^h d$$

$$= n\left(\frac{1}{lgn} + \frac{1}{\lg(\frac{n}{3})} + \frac{1}{\lg(\frac{n}{3^2})} + \ldots\ldots\ldots\ldots + \frac{1}{\lg(\frac{n}{3^{h-1}})}\right) + 3^h d$$

Substituting n = 3^h, we get

$$= n\left(\frac{1}{lg3^h} + \frac{1}{\lg(3^{h-1})} + \frac{1}{\lg(3^{h-2})} + \ldots\ldots\ldots\ldots + \frac{1}{\lg(3)}\right) + dn$$

$$= n\left(\frac{1}{h\ lg3} + \frac{1}{(h-1)\lg 3} + \frac{1}{(h-2)\lg 3} + \ldots\ldots + \frac{1}{\lg 3}\right) + dn$$

$$= (n/\lg 3)\left(\frac{1}{h} + \frac{1}{(h-1)} + \frac{1}{(h-2)} + \ldots\ldots + \frac{1}{2} + \frac{1}{1}\right) + dn$$

Recurrence relation

$$T(n) = \frac{n}{lg3}\left(1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\cdots\ldots\ldots\ldots+\frac{1}{h}\right)+dn$$

Suppose $h = 2^m$

$$=\frac{n}{lg3}\left(1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\cdots\ldots\ldots\ldots+\frac{1}{2^m}\right)+dn$$

$$< \frac{n}{lg3}\left(1+\left(\frac{1}{2}+\frac{1}{2}\right)+\left(\frac{1}{4}+\frac{1}{4}+\frac{1}{4}+\frac{1}{4}\right)+\cdots\ldots\ldots\ldots+\left(\frac{1}{2^m}+\frac{1}{2^m}+\ldots\ldots..+\frac{1}{2^m}\right)\right)+dn$$

$$= \frac{n}{lg3}(1+1+1+\ldots\ldots\ldots\ldots+1) + dn$$

$$= \frac{n}{lg3}(m+1) + dn \quad = \frac{n}{lg3}(\lg h + 1) + dn$$

$$= \frac{n}{lg3}(\lg \log_3 n + 1) + dn$$

$$= O(n \lg \lg n)$$

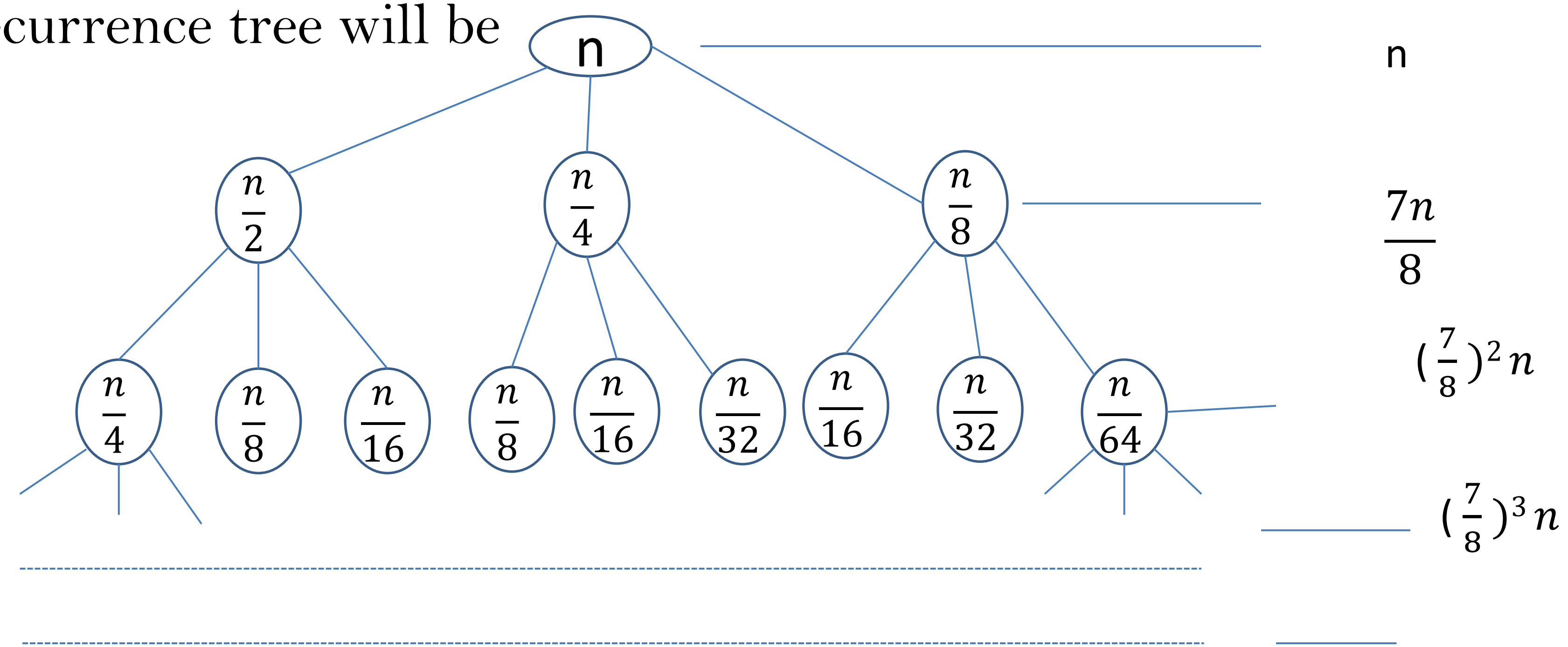
Some exercise solution

(4-d) Recurrence relation is

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n$$

We will use recurrence tree method to solve it.

Recurrence tree will be



Let h is the height of the tree. Therefore, $\frac{n}{2^h} = 1 \Rightarrow h = \lg n$.

Some exercise solution

Therefore total cost

$$T(n) = n + \frac{7n}{8} + \left(\frac{7}{8}\right)^2 n + \left(\frac{7}{8}\right)^3 n + \;\ldots\ldots\ldots\ldots. + \left(\frac{7}{8}\right)^h n$$

$$= n\left(1 + \frac{7}{8} + \left(\frac{7}{8}\right)^2 + \left(\frac{7}{8}\right)^3 + \;\ldots\ldots\ldots\ldots. + \left(\frac{7}{8}\right)^h \right)$$

$$< n\left(1 + \frac{7}{8} + \left(\frac{7}{8}\right)^2 + \left(\frac{7}{8}\right)^3 + \;\ldots\ldots\ldots\ldots\ldots\ldots.\right)$$

Here, we consider up to infinite term because common ratio of this series is $\frac{7}{8}$ that is less than 1.

$$= n\left(\frac{1}{(1-\frac{7}{8})} \right)$$

$$= 8n = O(n)$$

Therefore, $T(n) = O(n)$.

AKTU Examination Questions

1. Solve the recurrence $T(n) = 2T(n/2) + n^2 + 2n + 1$
2. Solve the recurrence using recursion tree method:

$T(n) = T(n/2) + T(n/4) + T(n/8) + n$

3. Use a recursion tree to give an asymptotically tight solution to the recurrence $T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$, where α is a constant in the range $0 < \alpha < 1$ and $c > 0$ is also a constant.
4. The recurrence $T(n) = 7T(n/3) + n^2$ describes the running time of an algorithm A. Another competing algorithm B has a running time of $S(n) = a\,S(n/9) + n^2$. What is the smallest value of 'a' such that A is asymptotically faster than B?
5. Solve the recurrence relation by substitution method

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

AKTU Examination Questions

6. Show that the solution to $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ is $O(n \lg n)$.

7. Solve the recurrence: $T(n) = 50\,T(n/49) + \log n!$

8. Solve the following recurrence using Master method:

$T(n) = 4T(n/3) + n^2$

9. Find the time complexity of the recurrence relation

$T(n) = n + T(n/10) + T(7n/5)$

10. Solve the following By Recursion Tree Method

$T(n) = n + T(n/5) + T(4n/5)$

11. The recurrence $T(n) = 7T(n/2) + n^2$ describe the running time of an algorithm A. A competing algorithm A has a running time of $T'(n) = aT'(n/4) + n^2$. What is the largest integer value for a A' is asymptotically faster than A?

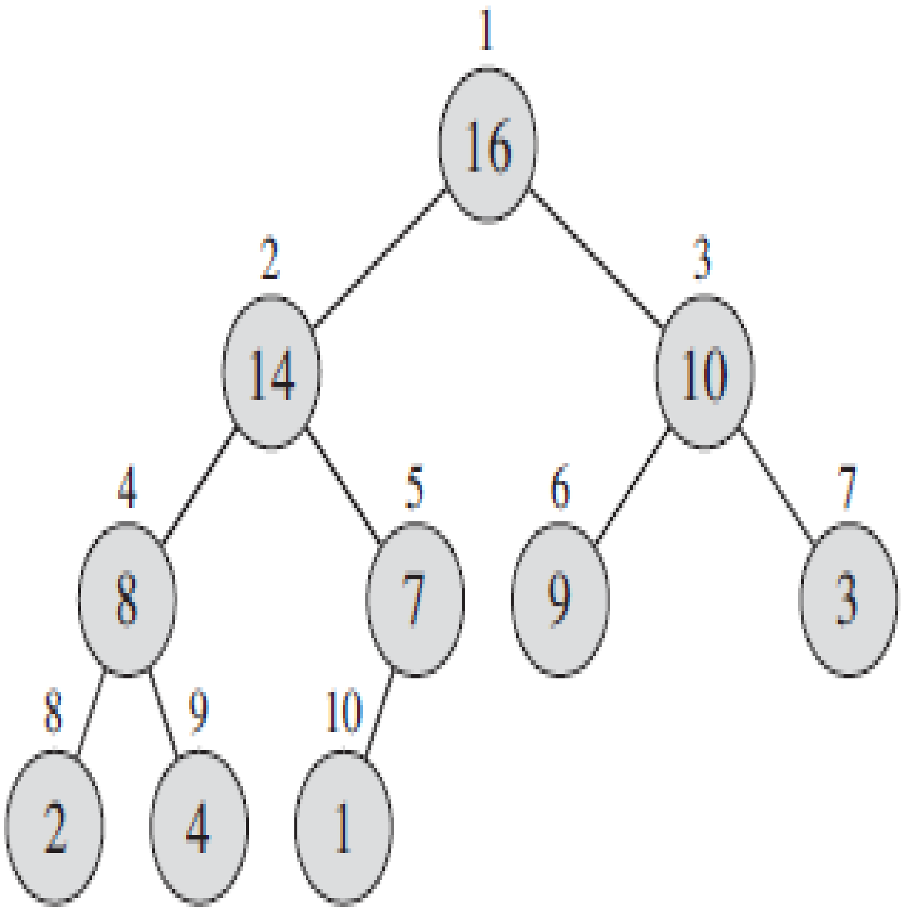
Heapsort

<u>Heap</u>

- The ***(binary) heap*** data structure is an array object that we can view as a nearly complete binary tree.

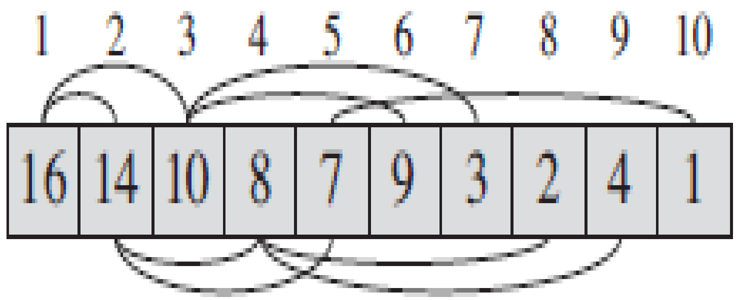
- Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

Heapsort



(a)

Max-heap



(b)

Array

Heapsort

Index: If i the index of a node, then the index of parent and its child are the following:-

$$\text{Parent}(i) = \lfloor i/2 \rfloor$$

$$\text{Left}(i) = 2i$$

$$\text{Right}(i) = 2i+1$$

Note: Root node has always index 1 i.e. A[1] is root element.

Heap-size: Heap-size is equal to the number of elements in the heap.

Height of a node: The height of a node in a heap is the number of edges on the longest simple downward path from the node to a leaf.

Height of heap: The height of the heap is equal to the height of its root.

Heapsort

Types of heap

There are two kinds of binary heaps:

(1) max-heaps (2) min-heaps

Max-heap: The heap is said to be max-heap if it satisfy the **max-heap property**.

The **max-heap property** is that the value at the parent node is always greater than or equal to value at its children.

Min-heap: The heap is said to be min-heap if it satisfy the **min-heap property**.

The **min-heap property** is that the value at the parent node is always less than or equal to value at its children.

Heapsort

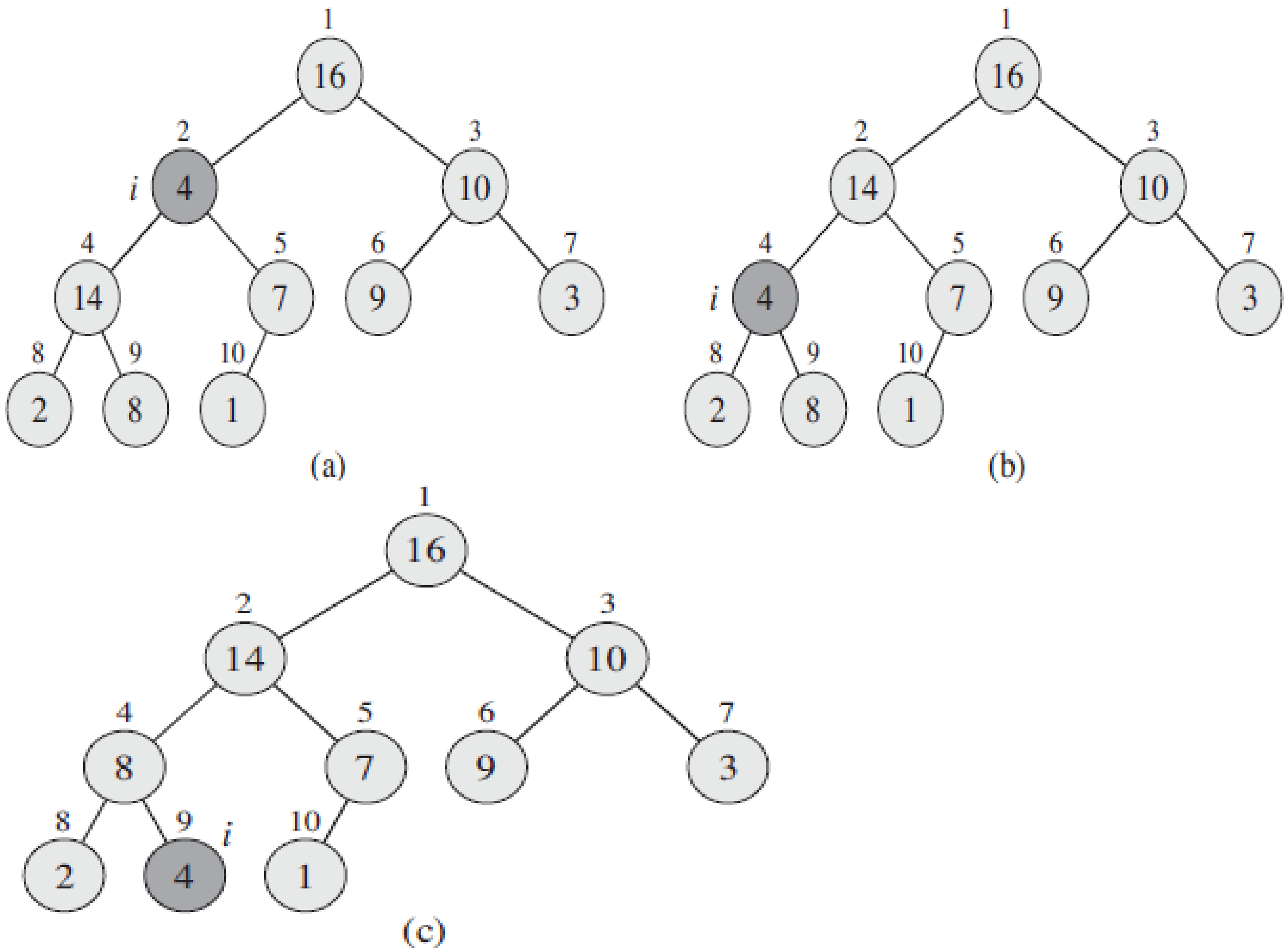
Heap sort algorithm consists of the following two sub-algorithms.

(1) Max-Heapify: It is used to maintain the max-heap property.

(2) Build-Max-Heap: It is used to construct a max-heap for the given set of elements.

Max-Heapify Algorithm

Action done by max-heapify algorithm is shown in the following figures:-



(a) (b) (c)

Heapsort

Max-Heapify Algorithm

```
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)
```

Heapsort

Time complexity of Max-Heapify Algorithm

The running time of max-heapify is determined by the following recurrence relation:-

$$T(n) \leq T(2n/3) + \theta(1)$$

Here n is the size of the sub-tree rooted at node i.

Using master theorem, the solution of this recurrence relation is

$$\mathbf{T(n) = \theta(lg\ n)}$$

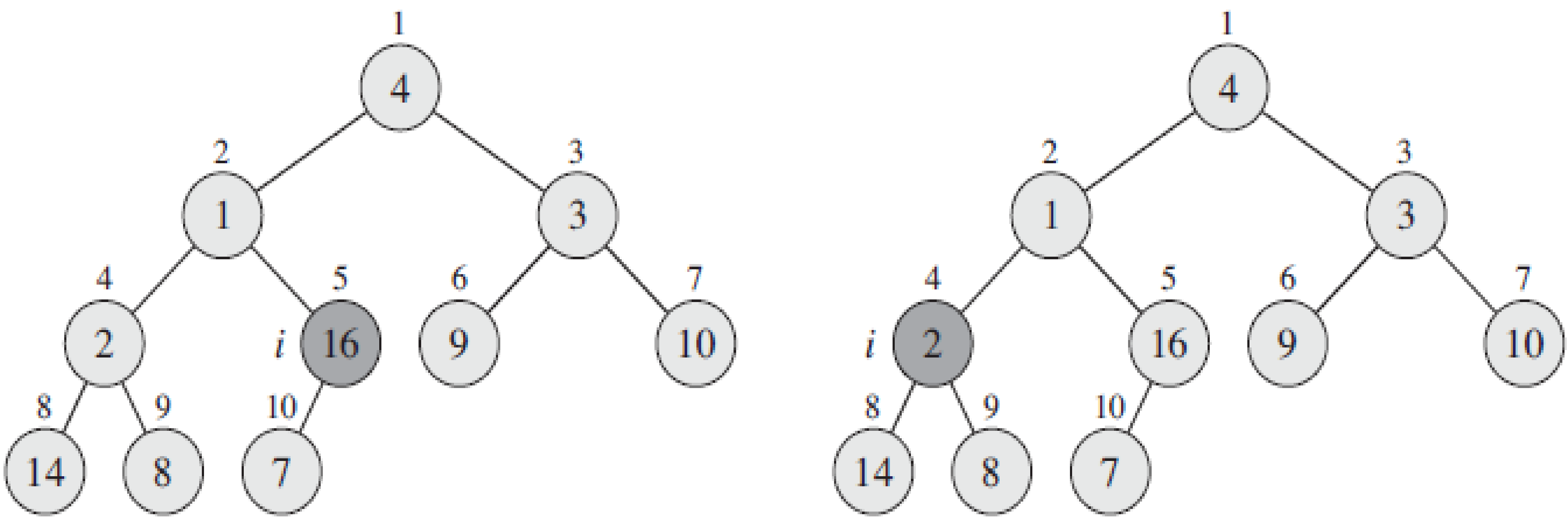
Heapsort

Build-Max-Heap Algorithm

Example: Construct max-heap corresponding to the following elements

4, 1, 3, 2, 16, 9, 10, 14, 8, 7.

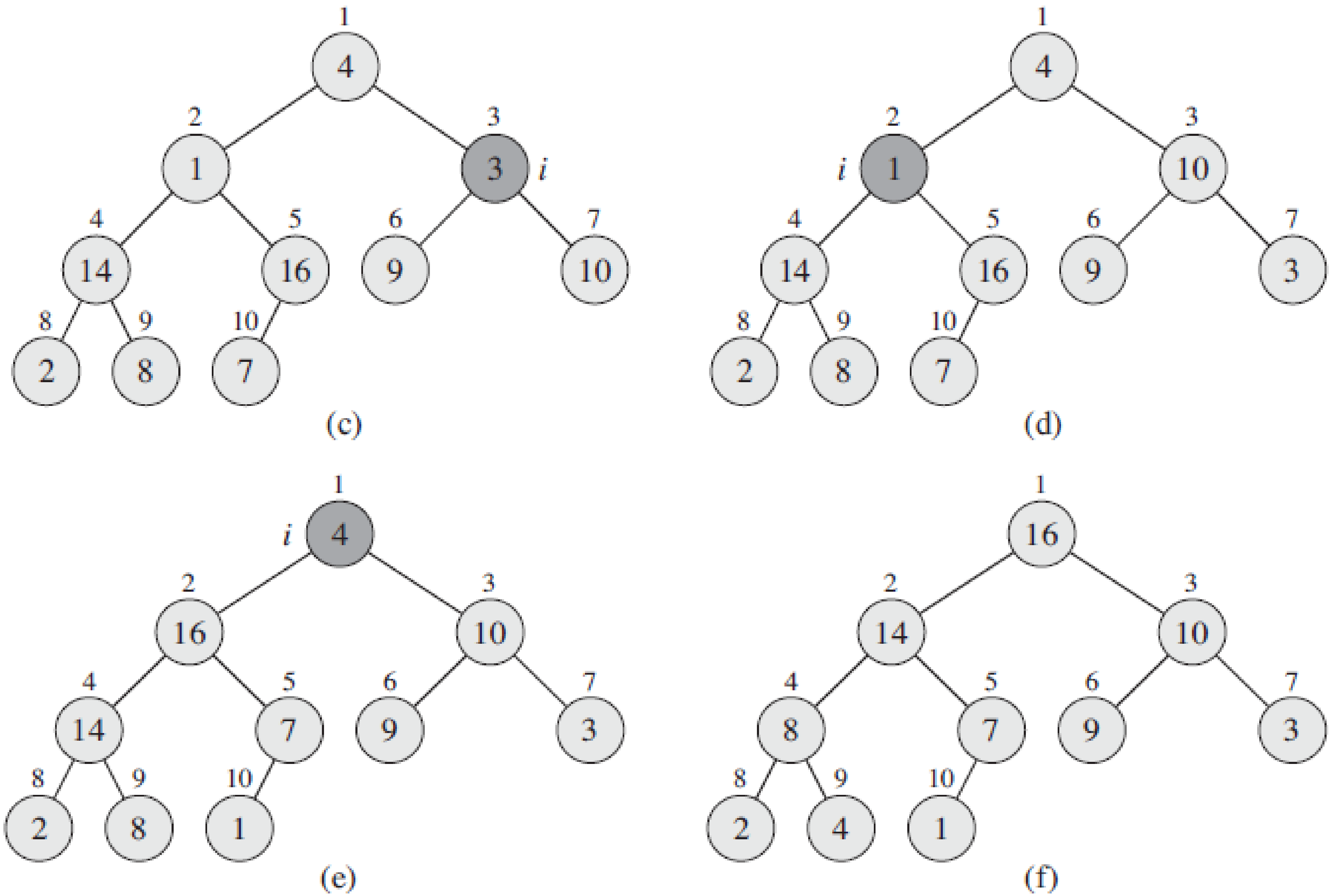
Solution:



(a) (b)

Heapsort

Build-Max-Heap Algorithm (cont.)



(c) (d) (e) (f)

Heapsort

Build-Max-Heap Algorithm (cont.)

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

The running time of this algorithm is

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

But this upper bound is not asymptotically tight.

Now, we shall calculate tight upper bound.

Heapsort

Time complexity of Build-Max-Heap Algorithm

We can derive a tighter bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small.

Our tighter analysis relies on the properties that an n-element heap has height $\lfloor \lg n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h.

If h is the height of the sub-tree then running time of max-heapify is O(h).

Therefore, total cost of build-max-heap is

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

Heapsort

Time complexity of Build-Max-Heap Algorithm

Now, $T(n) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$

$$= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right)$$

Since $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2}$

$$= 2 .$$

Therefore,

$$T(n) = O(2n)$$

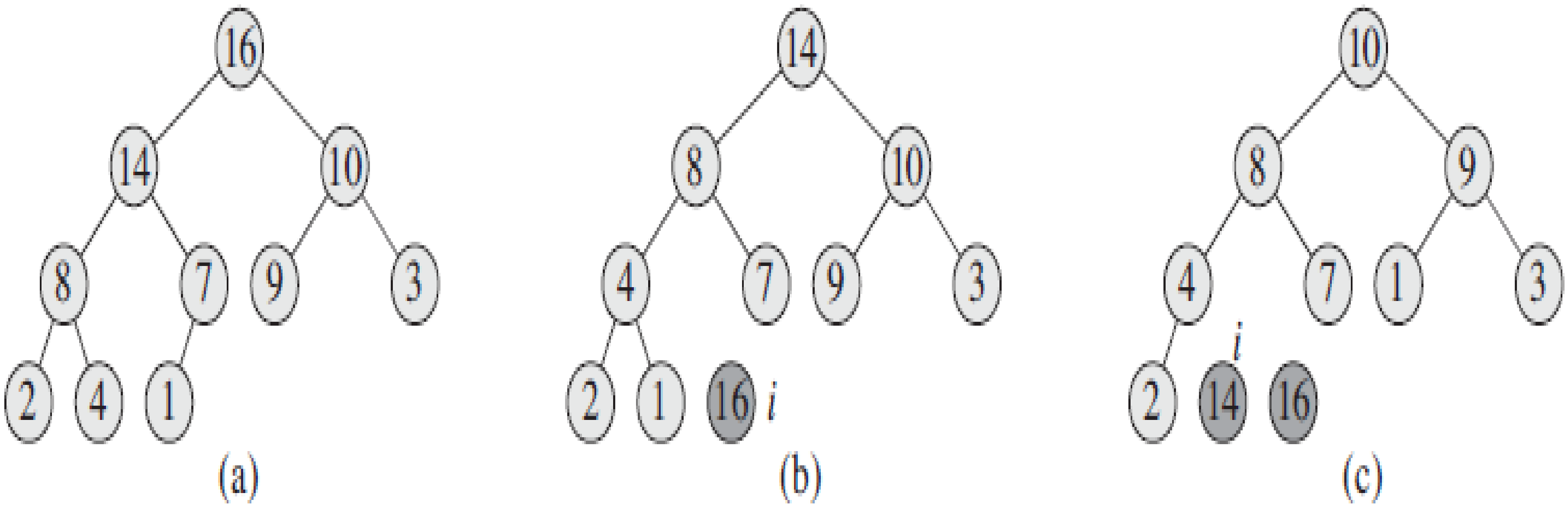
$$= \mathbf{O(n)}$$

Heapsort Algorithm

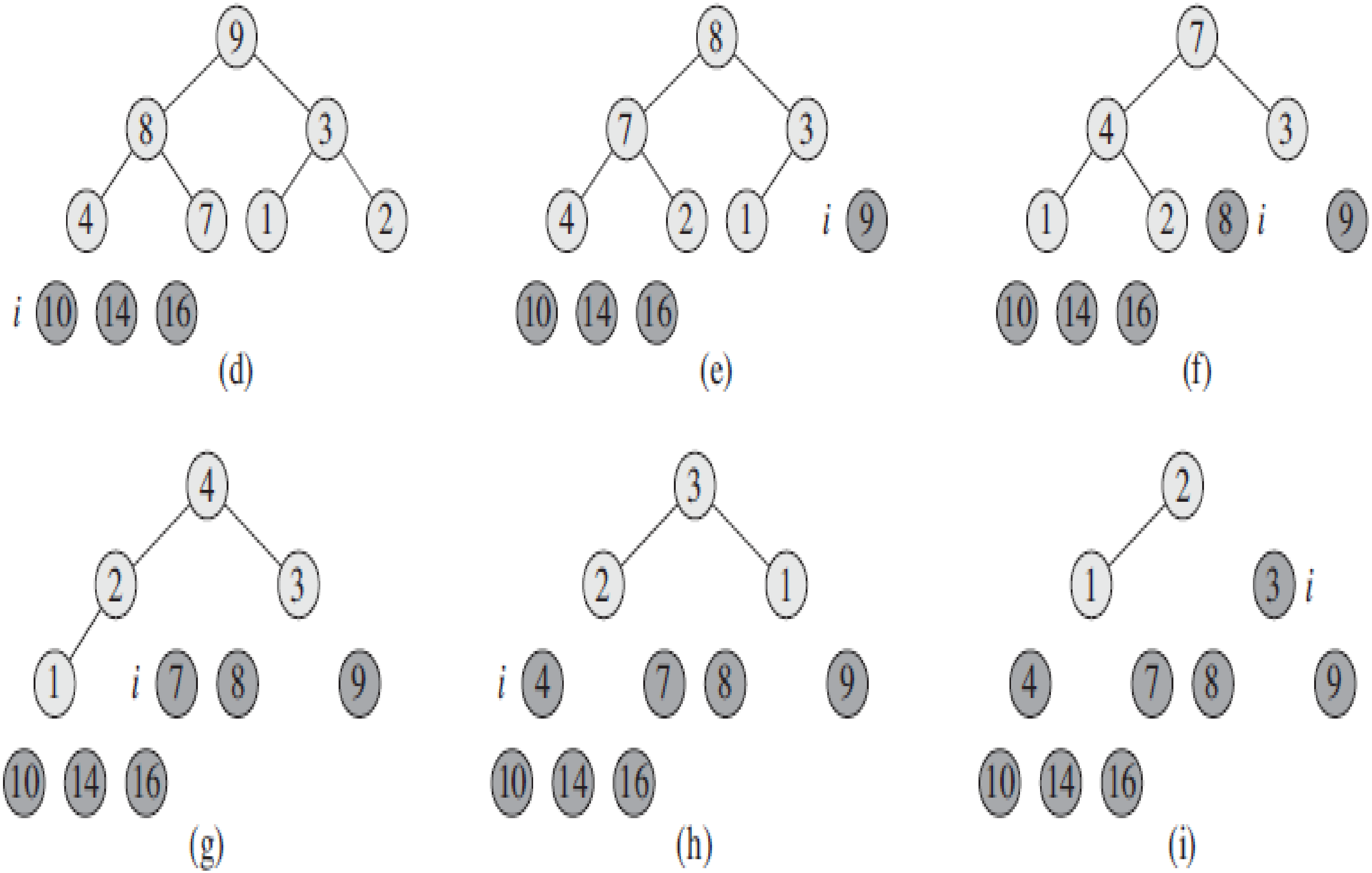
Example: Sort the following elements using heapsort

5, 13, 2, 25, 7, 17, 20, 8, 4.

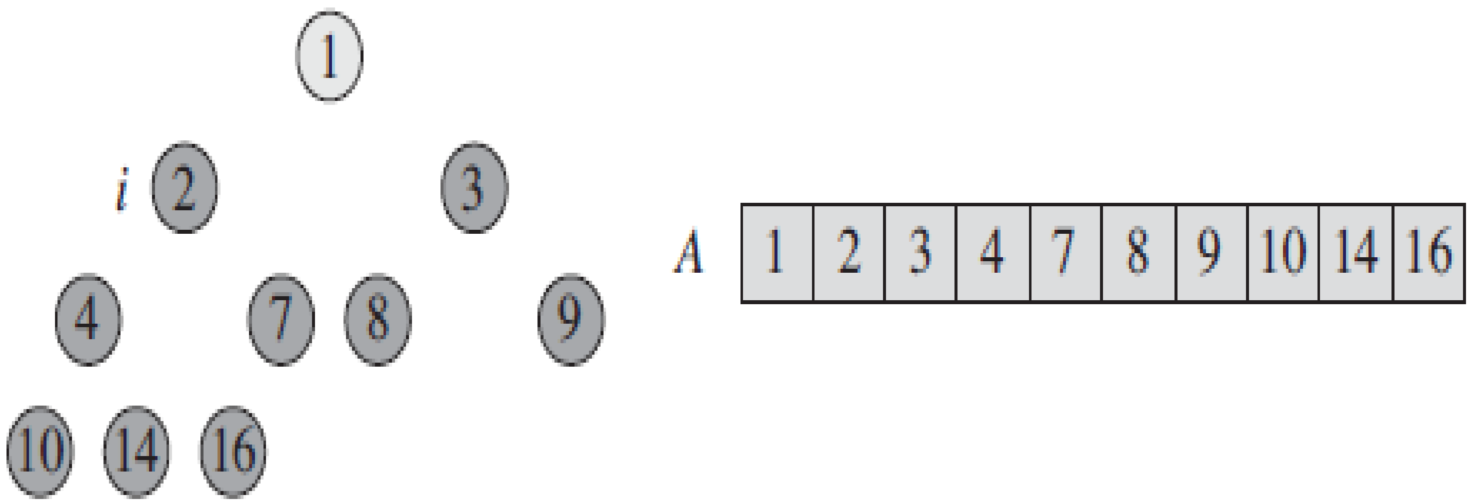
Solution: The operation of HEAPSORT is shown as following:-



Heapsort Algorithm



Heapsort Algorithm



(j) (k)

Heapsort Algorithm

```
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)
```

Time complexity: The HEAPSORT procedure takes time O(nlgn), since the call to BUILD-MAX-HEAP takes time O(n) and each of the n-1 calls to MAX-HEAPIFY takes time O(lgn).

Heapsort Algorithm

Exercise

1. What are the minimum and maximum numbers of elements in a heap of height h?
2. Show that an n-element heap has height $\lfloor \lg n \rfloor$.
3. Is the array with values 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 a max-heap?
4. Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.
5. What is the running time of HEAPSORT on an array A of length n that is already sorted in increasing order? What about decreasing order?

Heapsort Algorithm

AKTU Examination Questions

1. Sort the following array using Heap-Sort techniques—
   5,8,3,9,2,10,1,35,22
2. How will you sort following array A of elements using heap sort: A = (23, 9, 18, 45, 5, 9, 1, 17, 6).

Quicksort

Quicksort, like merge sort, applies the divide-and-conquer paradigm. The three-step divide-and-conquer process for sorting a typical subarray A[p .. r] is the following:-

Divide: Partition (rearrange) the array A[p .. r] into two (possibly empty) subarrays A[p .. q-1] and A[q+1 .. R] such that each element of A[p .. q-1] is less than or equal to A[q], which is, in turn, less than or equal to each element of A[q+1 .. r]. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays A[p .. q-1] and A[q+1 .. r] by recursive calls to quicksort.

Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array A[p .. r] is now sorted.

Quicksort

Algorithm

It divides the large array into smaller sub-arrays. And then quicksort recursively sort the sub-arrays.

Pivot

1. Picks an element called the "pivot".

Partition

2. Rearrange the array elements in such a way that the all values lesser than the pivot should come before the pivot and all the values greater than the pivot should come after it.

This method is called partitioning the array. At the end of the partition function, the pivot element will be placed at its sorted position.

Recursive

3. Do the above process recursively to all the sub-arrays and sort the elements.

Quicksort

Algorithm

Base Case

If the array has zero or one element, there is no need to call the partition method.

So we need to stop the recursive call when the array size is less than or equal to 1.

Pivot

There are many ways we can choose the pivot element.

i) The first element in the array

ii) The last element in the array

iii) The middle element in the array

iv) We can also pick the element randomly.

Note: In our algorithm, we are going to pick the last element as the pivot element.

Quicksort

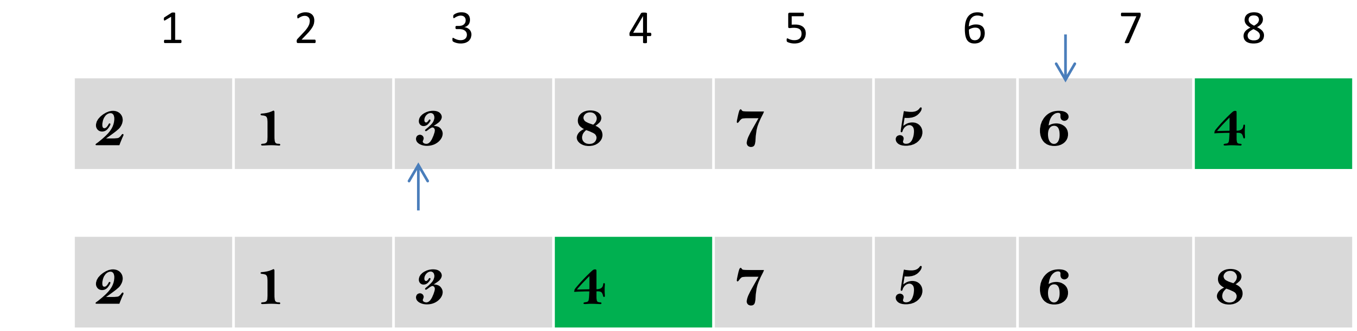
Example: Sort the following elements using quicksort

$2, 8, 7, 1, 3, 5, 6, 4$

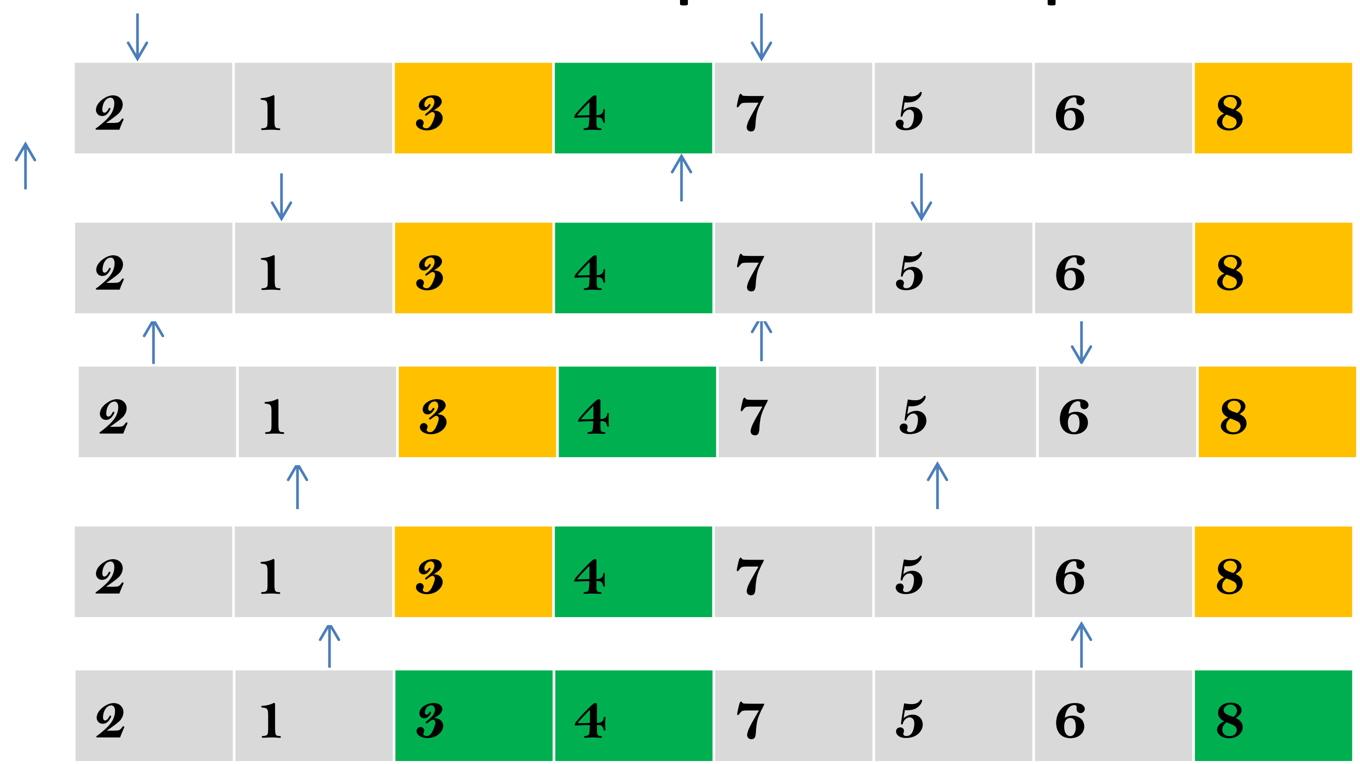
Solution: Here, we pick the last element in the list as a pivot

1	2	3	4	5	6	7	8
2	8	7	1	3	5	6	4
2	8	7	1	3	5	6	4
2	8	7	1	3	5	6	4
2	8	7	1	3	5	6	4
2	1	7	8	3	5	6	4
2	1	3	8	4	5	6	4

Quicksort



Partition completed in first pass



Partition completed in second pass

Quicksort

1	2	3	4	5	6	7	8
2	1	3	4	7	5	6	8

2	1	3	4	7	5	6	8

1	2	3	4	5	7	6	8

1	2	3	4	5	6	7	8

Partition completed in third pass

1	2	3	4	5	6	7	8

1	2	3	4	5	6	7	8

1	2	3	4	5	6	7	8

Process completed

Quicksort Algorithm

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

To sort an entire array A, the initial call is QUICKSORT(A, 1, *length[A])*.

Quicksort Algorithm

```
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
```

Quicksort Algorithm Analysis

The running time of quicksort algorithm is

$$T(n) = T(n_1) + T(n_2) + \theta(n) \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots(1)$$

Here n_1 and n_2 are the size of sub-problems.

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort.

Quicksort Algorithm Analysis

Worst-case partitioning

The worst-case behavior for quicksort occurs when the partitioning routine produces one sub-problem with n-1 elements and one with 0 elements. Let us assume that this unbalanced partitioning arises in each recursive call. Therefore,

$$T(n) = T(0) + T(n-1) + \theta(n)$$

$$T(n) = T(n-1) + \theta(n) \quad (\text{since } T(0) = \theta(1))$$

After solving this recurrence relation, we get

$$\mathbf{T(n) = \theta(n^2)}$$

Therefore the worst-case running time of quicksort is no better than that of insertion sort. Moreover, the $\theta(n^2)$ running time occurs when the input array is already completely sorted—a common situation in which insertion sort runs in $O(n)$ time.

Quicksort Algorithm Analysis

Best-case partitioning

If PARTITION produces two sub-problems, each of size no more than $n/2$, since one is of size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil - 1$. In this
case, quicksort runs much faster. The recurrence for the running time is then

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

After solving this recurrence relation, we get

$$\mathbf{T(n) = \theta(n\ lgn)}$$

Note: By equally balancing the two sides of the partition at every level of the recursion, we get an asymptotically faster algorithm.

Quicksort Algorithm Analysis

Balanced partitioning

The average-case running time of quicksort is much closer to the best case than to the worst case.

Suppose that the partitioning algorithm always produces a 9-to-1 proportional split. In this case, running time will be

$$T(n) = T(9n/10) + T(n/10) + \theta(n)$$

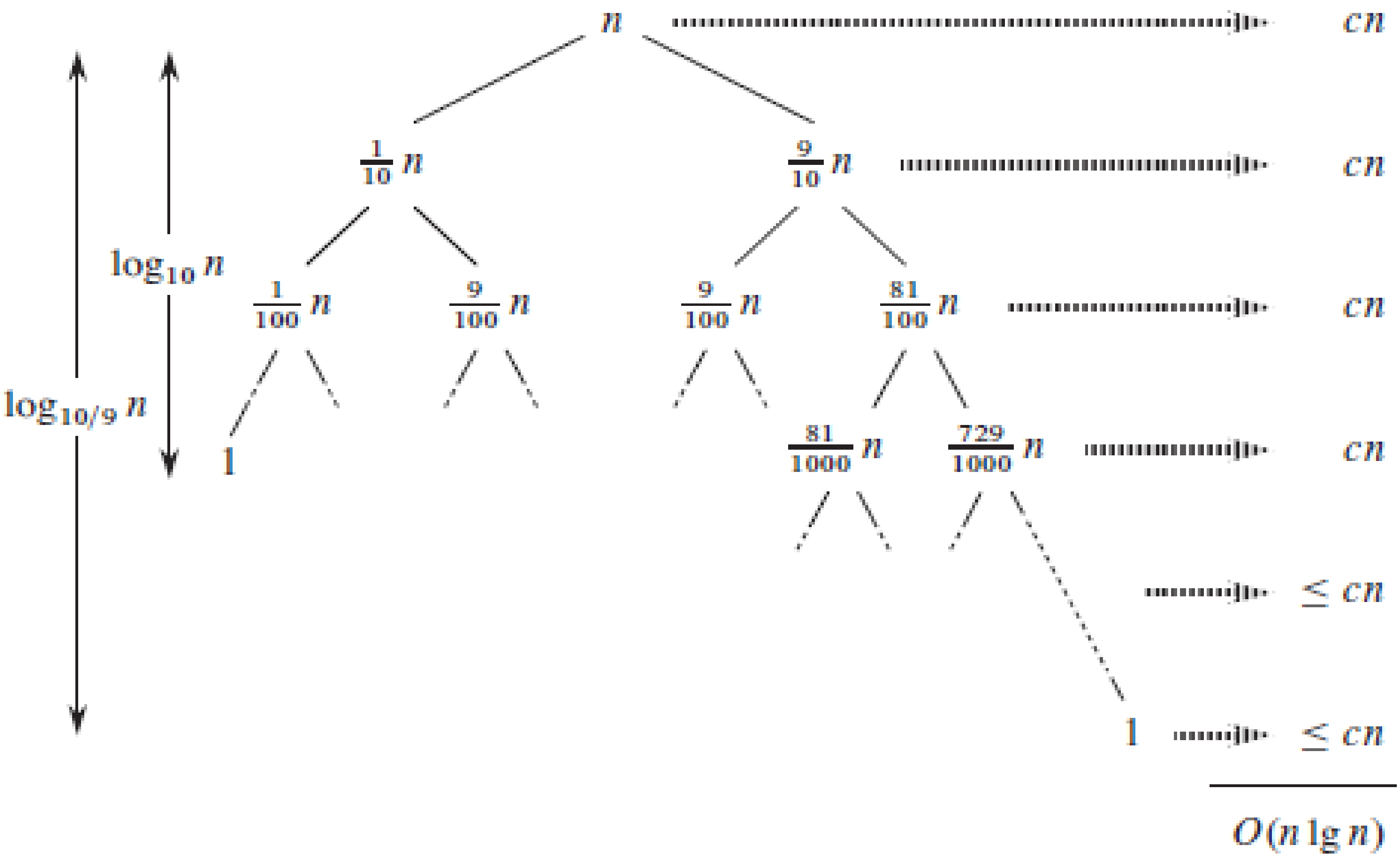
This relation is equivalent to the following

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

We can solve this recurrence using recurrence tree method.

Quicksort Algorithm Analysis

Recurrence tree will be



Quicksort Algorithm Analysis

Therefore, the solution of recurrence relation will be

$$T(n) \le cn + cn + cn + \ldots\ldots\ldots\ldots\ldots\ldots + cn$$

$$= cn(1+lg_{10/9}n)$$

$$= cn + cnlg_{10/9}n$$

Therefore, **$T(n) = O(nlgn)$**

Quicksort Algorithm Analysis

Exercise

1. Sort the following elements using quicksort

   13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11.

2. What is the running time of QUICKSORT when all elements of array A have the same value?

3. Show that the running time of QUICKSORT is, $\theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.

4. Suppose that the splits at every level of quicksort are in the proportion $1-\alpha$ to α, where $0 < \alpha \leq 1/2$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approximately $\lg n / \lg \alpha$ and the maximum depth is approximately $\lg n / \lg(1-\alpha)$. (Don't worry about integer round-off.)

A randomized version of quicksort

```
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)
```

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

A randomized version of quicksort

```
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
```

The running time of randomized quick sort will be

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

Sorting in Linear Time

Comparison Sort

In a comparison based sort, we use only comparisons between elements to gain order information about an input sequence $<a_1, a_2, \ldots\ldots\ldots\ldots\ldots, a_n>$. That is, given two elements a_i and a_j, we perform one of the tests $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$ to determine their relative order.

The decision-tree model

We can view comparison sorts abstractly in terms of decision trees.

A ***decision tree*** is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

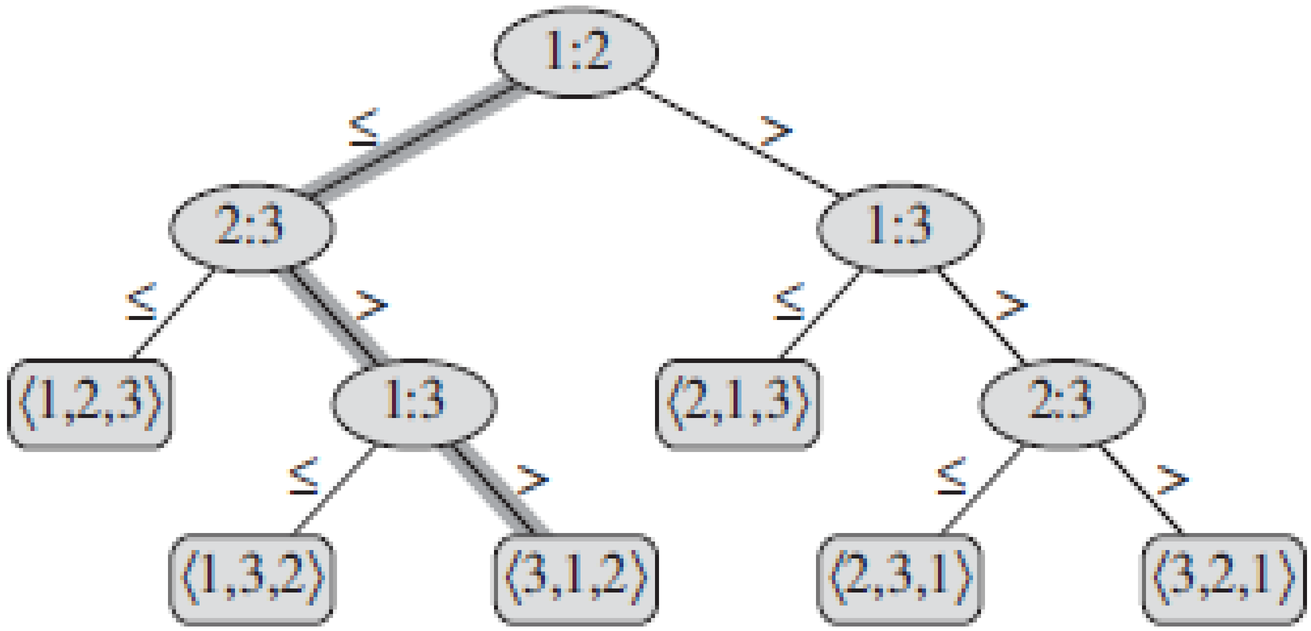
Sorting in Linear Time

In a decision tree, we annotate each internal node by **i:j** for some i and j in the range $1 \leq i, j \leq n$, where n is the number of elements in the input sequence. We also annotate each leaf by a permutation $< \boldsymbol{\pi}(1), \boldsymbol{\pi}(2), \boldsymbol{\pi}(3), \ldots\ldots\ldots\ldots, \boldsymbol{\pi}(n) >$.

- The execution of the sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf.
- Each internal node indicates a comparison $a_i \leq a_j$. The left sub-tree then dictates subsequent comparisons once we know that $a_i \leq a_j$, and the right sub-tree dictates subsequent comparisons knowing that $a_i > a_j$.
- When we come to a leaf, the sorting algorithm has established the ordering $<a_{\boldsymbol{\pi}(1)}, a_{\boldsymbol{\pi}(2)}, \ldots\ldots\ldots\ldots, a_{\boldsymbol{\pi}(n)}>$.

Sorting in Linear Time

The decision tree operating on three elements is the following:-



Sorting in Linear Time

Theorem:

Any comparison sort algorithm requires $\Omega(n\lg n)$ comparisons in the worst case.

Proof: The worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree.

Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements. Because each of the n! permutations of the input appears as some leaf, we have n! ≤ l . Since a binary tree of height h has no more than 2^h leaves, therefore

$$n! \le l \le 2h \;\Rightarrow\; h \ge \lg(n!)$$

Therefore $\quad h = \Omega(n\lg n)$

Counting sort

- ***Counting sort*** assumes that each of the n input elements is an integer in the range 0 to k, for some integer k.

- When k = O(n), the sorting algo. runs in θ(n) time.

- Counting sort determines, for each input element x, the number of elements less than x. It uses this information to place element x directly into its position in the output array.

Counting sort

Example: Sort the following elements using counting sort

2, 5, 3, 0, 2, 3, 0, 3

Solution:

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

(a)

	0	1	2	3	4	5
C	2	2	4	7	7	8

(b)

	1	2	3	4	5	6	7	8
B							3	

	0	1	2	3	4	5
C	2	2	4	6	7	8

(c)

	1	2	3	4	5	6	7	8
B		0					3	

	0	1	2	3	4	5
C	1	2	4	6	7	8

(d)

Counting sort

	1	2	3	4	5	6	7	8
B		0				3	3	

	0	1	2	3	4	5
C	1	2	4	5	7	8

(e)

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

(f)

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
```

- Time complexity of counting sort is

  $T(n) = \theta(n+k)$
- If $k \le n$, then $\mathbf{T(n) = \theta(n)}$

Stable sorting

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

Some stable sorting algorithms

1. Counting sort
2. Insertion sort
3. Merge Sort
4. Bubble Sort

Some unstable sorting algorithms

1. Selection sort
2. Quicksort
3. Heap sort

Radix sorting

- This sorting algorithm assumes all the numbers must be of equal number of digits.

- This algorithm sort all the elements on the basis of digits used in the number.

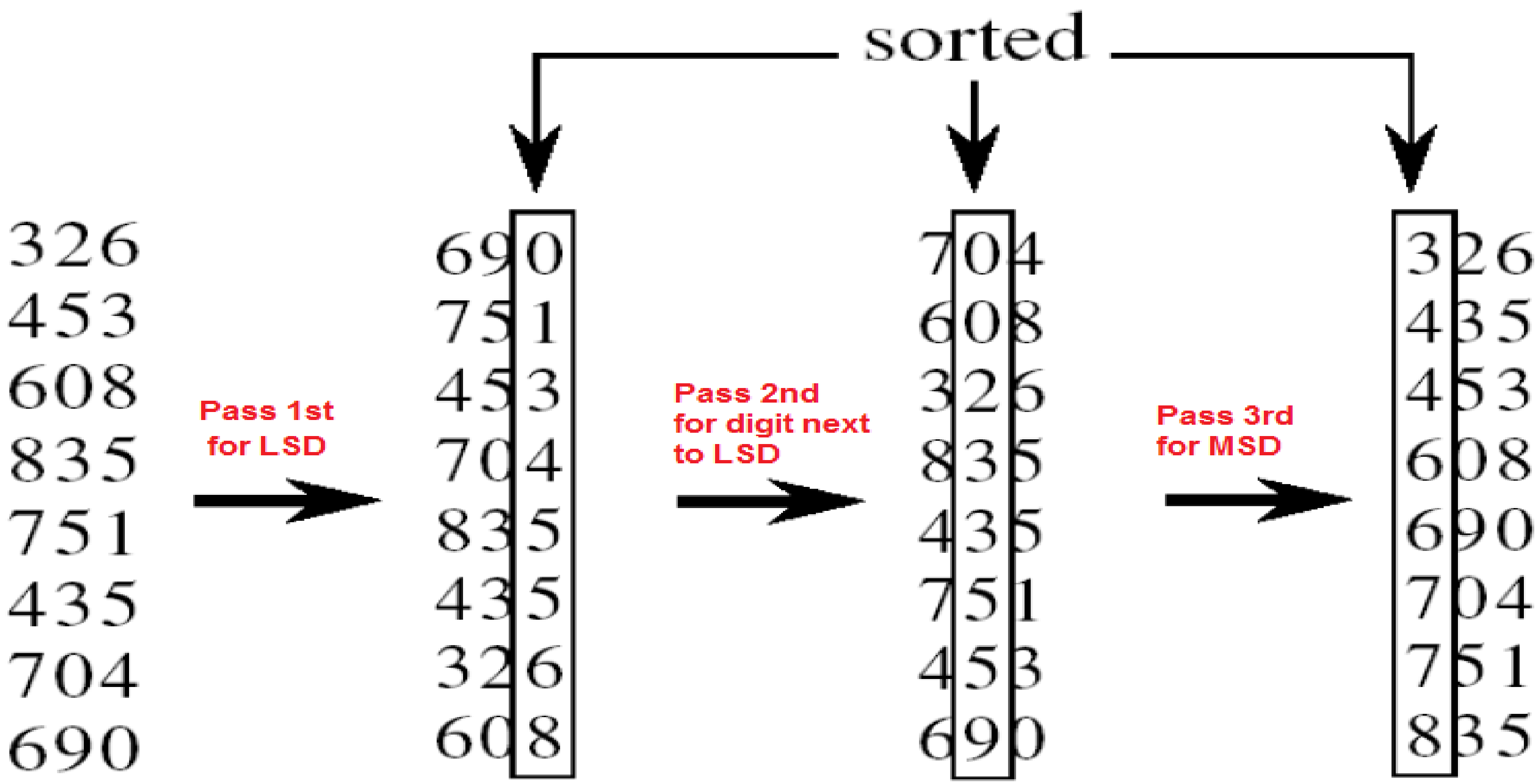
- First sort the numbers on the basis of least significant digit. Next time on the basis of 2^{nd} least significant digit. After it on the basis of 3^{rd} least significant digit. And so on. At the last, it sort on the basis of most significant digit.

Radix sorting

Example: Sort the following elements using radix sort

$326, 453, 608, 835, 751, 435, 704, 690.$

Solution: In these elements, number of digits is 3. Therefore, we have to used 3 iterations.



Radix sorting

```
radix_sort(A,d,k)
{
      for i=1 to d
            counting_sort(A,i,k)
}
```

```
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
```

Radix sorting

Time complexity of radix sort is

$$T(n) = \theta(d(n+k))$$

Note: When d is constant and $k=O(n)$, we can make radix sort run in linear time i.e $T(n) = \theta(n)$.

Bucket sorting

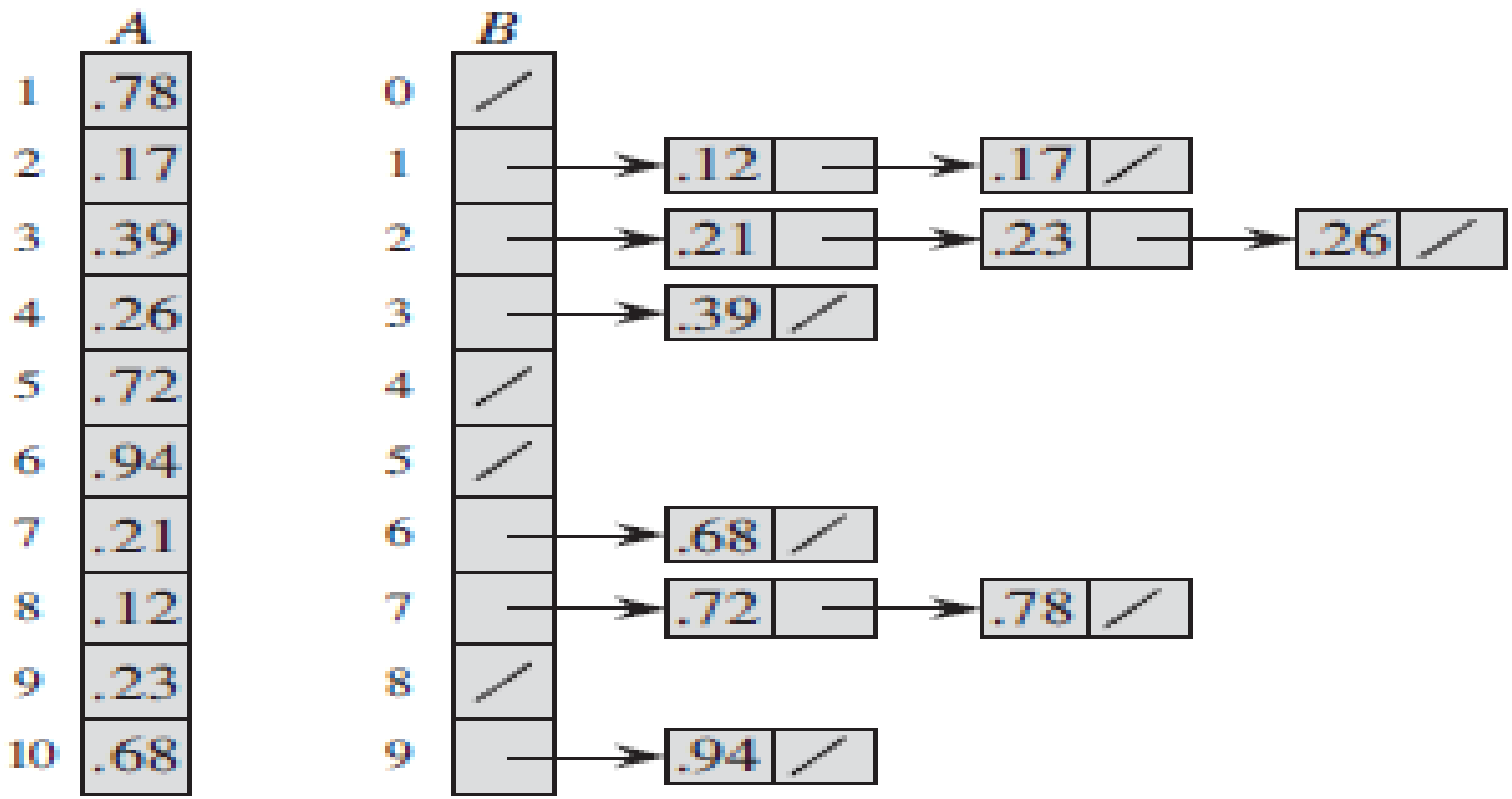
- Bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval [0,1).

- Bucket sort divides the interval [0,1) into n equal-sized subintervals, or ***buckets***, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over [0,1), we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

Bucket sorting

Example: Sort the following elements using bucket sort

0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68



(a) (b)

Bucket sorting

```
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
```

Time complexity $T(n) = \theta(n)$

Linear search

- It is a simple search algorithm that search a target value in an array or list.
- It sequentially scan the array, comparing each array item with the search element.
- If search element is found in an array, then the index of an element is returned.
- The time complexity of linear search is O(n).
- It can be applied to both sorted and unsorted array.

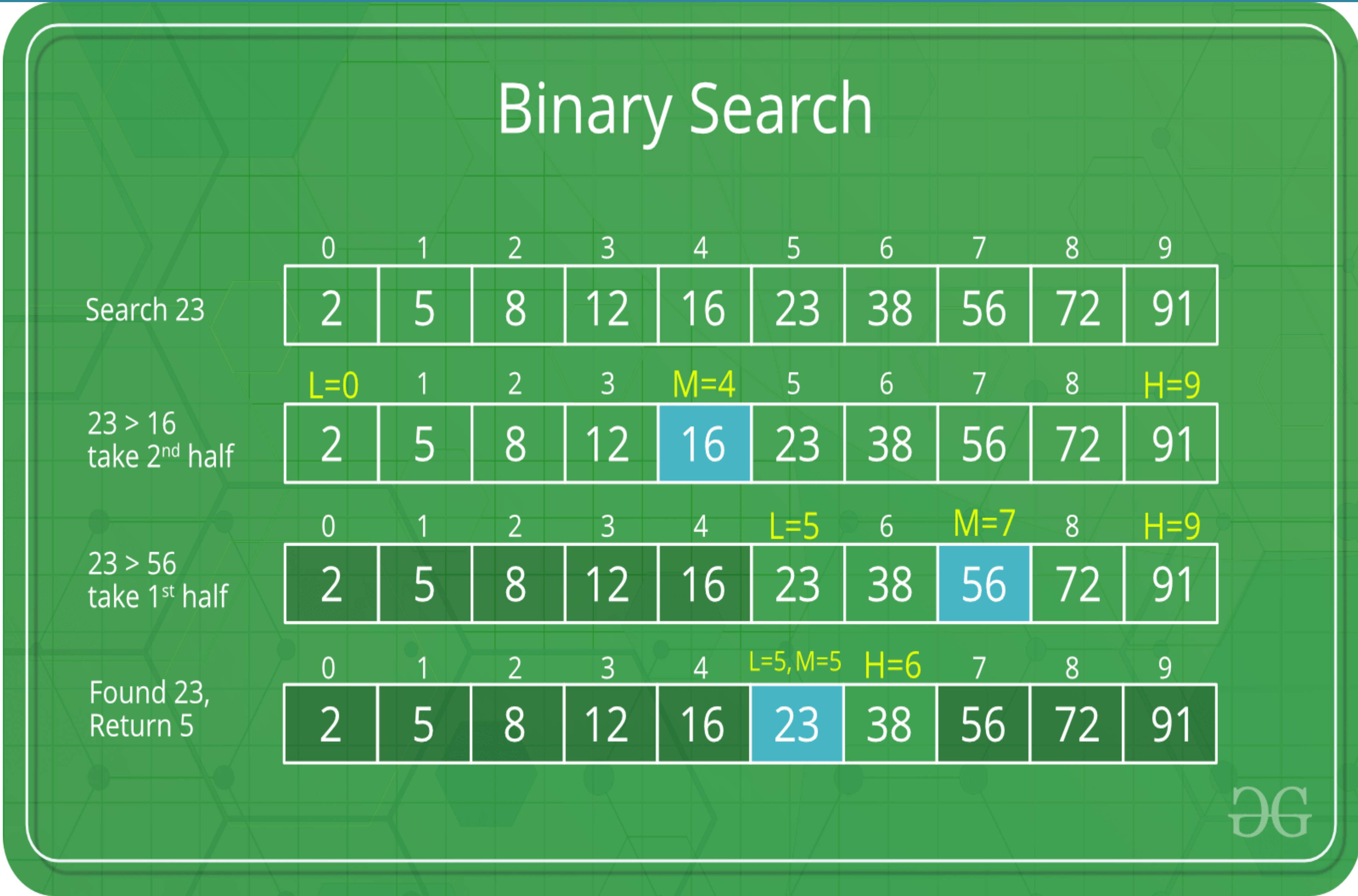
Binary search

- **Binary search** is the most popular Search algorithm. It is efficient and also one of the most commonly used techniques that is used to solve problems.

- Binary search works only on a sorted set of elements. To use binary search on a collection, the collection must first be sorted.

- When binary search is used to perform operations on a sorted set, the number of iterations can always be reduced on the basis of the value that is being searched.

Binary search process



Binary search

Binary-search(A, n, x)
$l = 1$
$r = n$
while $l \leq r$
do

    $m = \lfloor (l + r) / 2 \rfloor$
    if $A[m] < x$ **then**
        $l = m + 1$
    else if $A[m] > x$ **then**
        $r = m - 1$
    else
        return m

return unsuccessful

Time complexity **$T(n) = O(\lg n)$**

Thank you.