Algorithms & Data Structures

• Professor Reza Sedaghat

• COE428: Engineering Algorithms & Data Structures

•Email address: rsedagha@ee.ryerson.ca

• Course outline: www.ee.ryerson.ca/~courses/COE428/

• Course References:

1) **Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. Introduction to Algorithms, MIT, 2002, ISBN: 0-07-013151-1 (McGraw-Hill) (Course Text)**

Algorithms & Data Structures

- **Algorithm**
  - Steps to solve a problem
  - Well-defined computational procedure that takes a set of values as **inputs** and a set of values as **outputs**

  - In other words, an algorithm is a sequence of computational steps that transforms the inputs into outputs to solve a **problem**

  - The statement of a **problem** specifies the desired input/output relationship

Algorithms & Data Structures

- An algorithm is defined as follows:
  - **Finiteness**: An algorithm must terminate in finite time
  - **Definiteness**: Each step must be precisely defined
  - **Input:** A sequence of n numbers $a_1, a_2, \ldots, a_n$
  - **Output:** A permutation (reordering) of the input sequence such that $a_1 < a_2 < \ldots < a_n$

The problem of sorting

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Example:

Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Insertion sort

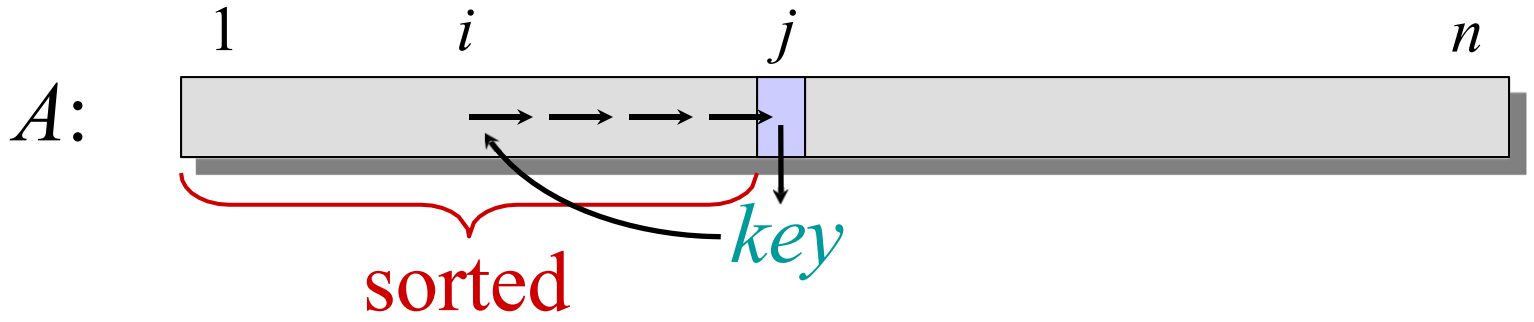
"pseudocode"

```
INSERTION-SORT (A, n)        ▷ A[1 . . n]
    for j ← 2 to n
        do key ← A[ j]
           i ← j – 1
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```

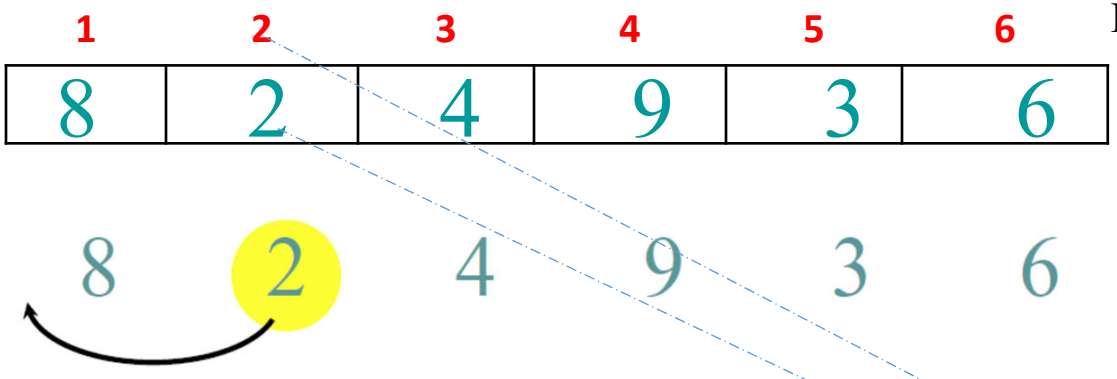
Insertion sort

"pseudocode"

```
INSERTION-SORT (A, n)        ▷ A[1 . . n]
    for j ← 2 to n
        do key ← A[ j]
           i ← j – 1
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```



Example of insertion sort



INSERTION-SORT (A, n) ▷ $A[1 \ldots n]$

for $j \leftarrow 2$ **to** n
  do $key \leftarrow A[j]$
  $i \leftarrow j - 1$
  while $i > 0$ and $A[i] > key$
    do $A[i+1] \leftarrow A[i]$
    $i \leftarrow i - 1$
  $A[i+1] = key$

INSERTION-SORT(A)

for $j \leftarrow 2$ **to** n
  do $key \leftarrow A[j]$
  ▷ Insert $A[j]$ into the sorted sequence $A[1 \ldots j-1]$
  $i \leftarrow j - 1$
  while $i > 0$ and $A[i] > key$
    do $A[i+1] \leftarrow A[i]$
    $i \leftarrow i - 1$
  $A[i+1] \leftarrow key$

j = 2
A[2] = 2, key = 2

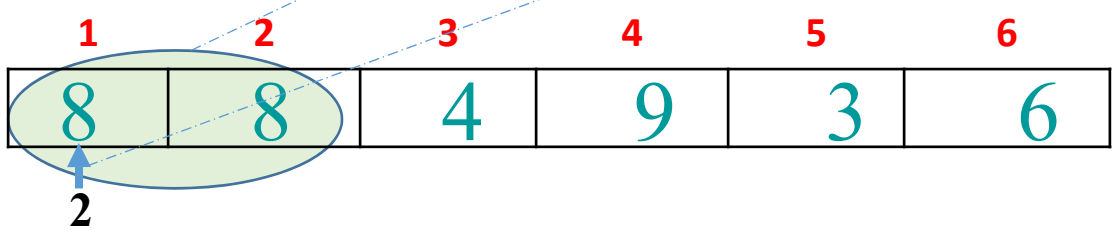
i = 1

A[1] = 8, Key=2 8 > 2

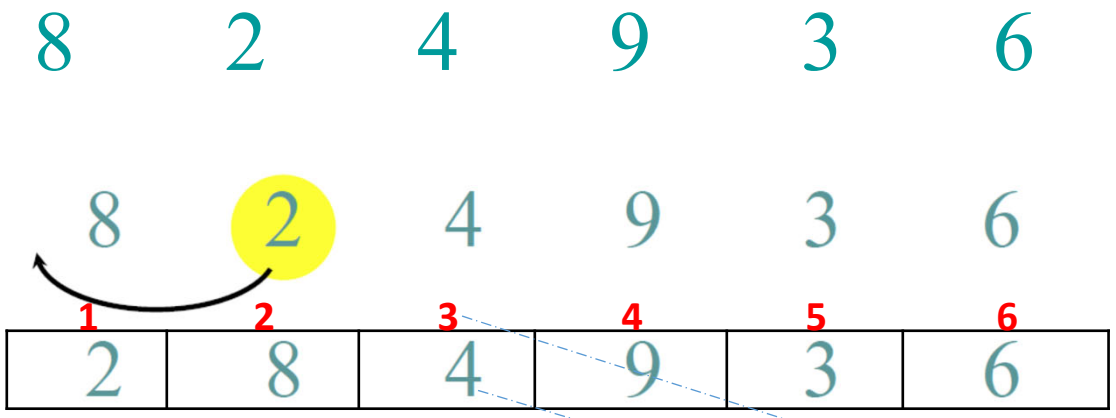
Change A[i], A[i+1]

i = 0

A[0+1]= 2, key = 2



Example of insertion sort

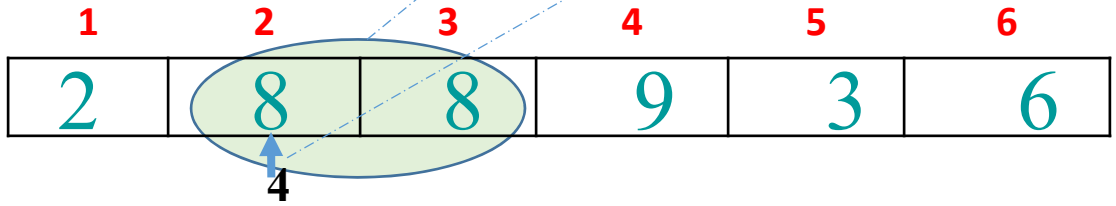


INSERTION-SORT (A, n) ▷ $A[1 \ldots n]$
for $j \leftarrow 2$ **to** n
do $key \leftarrow A[j]$
$i \leftarrow j - 1$
while $i > 0$ and $A[i] > key$
do $A[i+1] \leftarrow A[i]$
$i \leftarrow i - 1$
$A[i+1] = key$

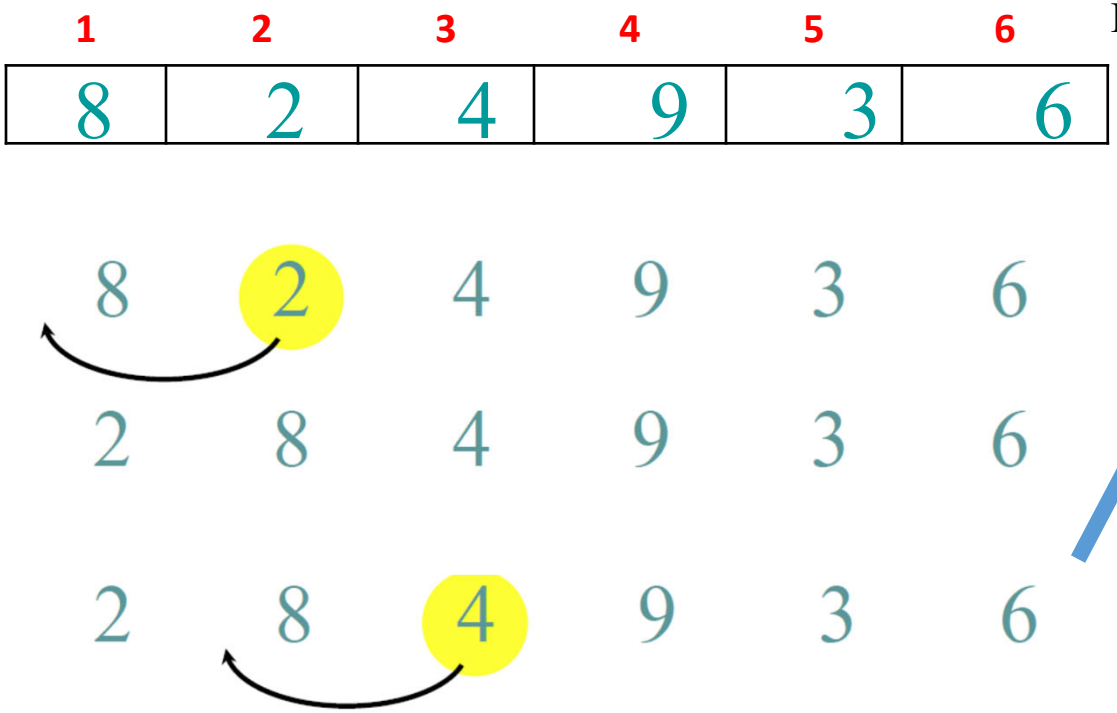
INSERTION-SORT(A)
for $j \leftarrow 2$ **to** n
do $key \leftarrow A[j]$
▷ Insert $A[j]$ into the sorted sequence $A[1 \ldots j-1]$
$i \leftarrow j - 1$
while $i > 0$ and $A[i] > key$
do $A[i+1] \leftarrow A[i]$
$i \leftarrow i - 1$
$A[i+1] \leftarrow key$

j = 3
A[3] = 4, key = 4

i = 2
A[2] = 8, Key=4 8 > 4
Change A[i], A[i+1]
i = 1
A[1+1]= 4, key = 4



Example of insertion sort

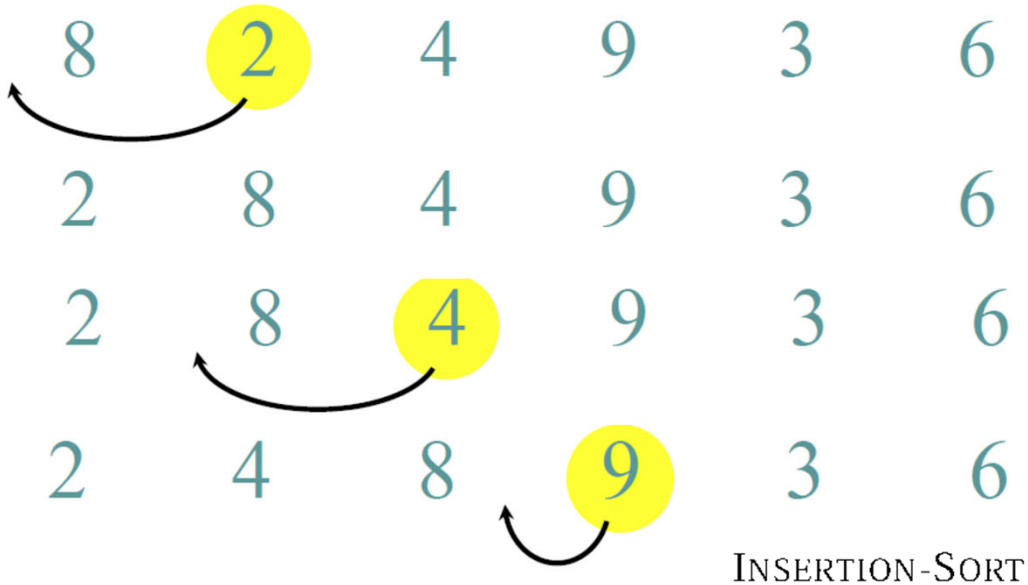


INSERTION-SORT (A, n) ▷ $A[1 \,..\, n]$
for $j \leftarrow 2$ **to** n
do $key \leftarrow A[j]$
$i \leftarrow j - 1$
while $i > 0$ and $A[i] > key$
do $A[i+1] \leftarrow A[i]$
$i \leftarrow i - 1$
$A[i+1] = key$

INSERTION-SORT(A)
for $j \leftarrow 2$ **to** n
do $key \leftarrow A[j]$
▷ Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$
$i \leftarrow j - 1$
while $i > 0$ and $A[i] > key$
do $A[i+1] \leftarrow A[i]$
$i \leftarrow i - 1$
$A[i+1] \leftarrow key$

Example of insertion sort

1	2	3	4	5	6
8	2	4	9	3	6

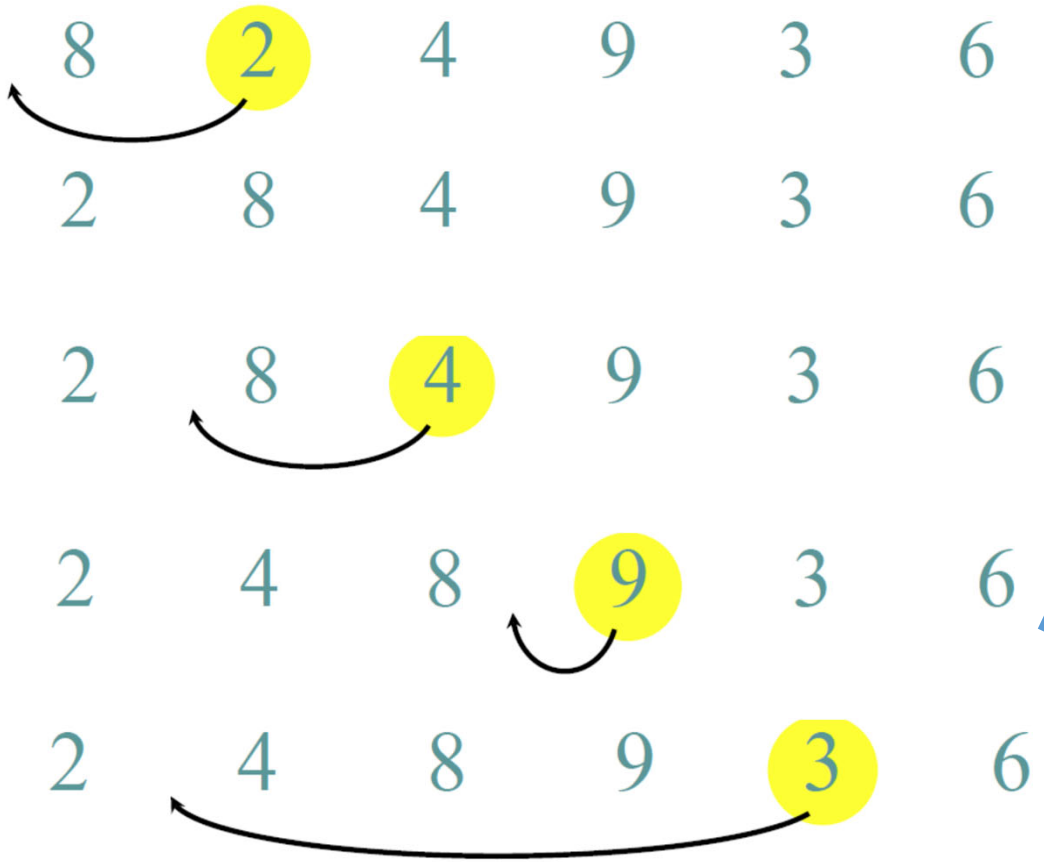


INSERTION-SORT (A, n) ▷ $A[1 \ldots n]$
for $j \leftarrow 2$ **to** n
do $key \leftarrow A[j]$
$i \leftarrow j - 1$
while $i > 0$ and $A[i] > key$
do $A[i+1] \leftarrow A[i]$
$i \leftarrow i - 1$
$A[i+1] = key$

INSERTION-SORT(A)
for $j \leftarrow 2$ **to** n
do $key \leftarrow A[j]$
▷ Insert $A[j]$ into the sorted sequence $A[1 \ldots j-1]$
$i \leftarrow j - 1$
while $i > 0$ and $A[i] > key$
do $A[i+1] \leftarrow A[i]$
$i \leftarrow i - 1$
$A[i+1] \leftarrow key$

Example of insertion sort

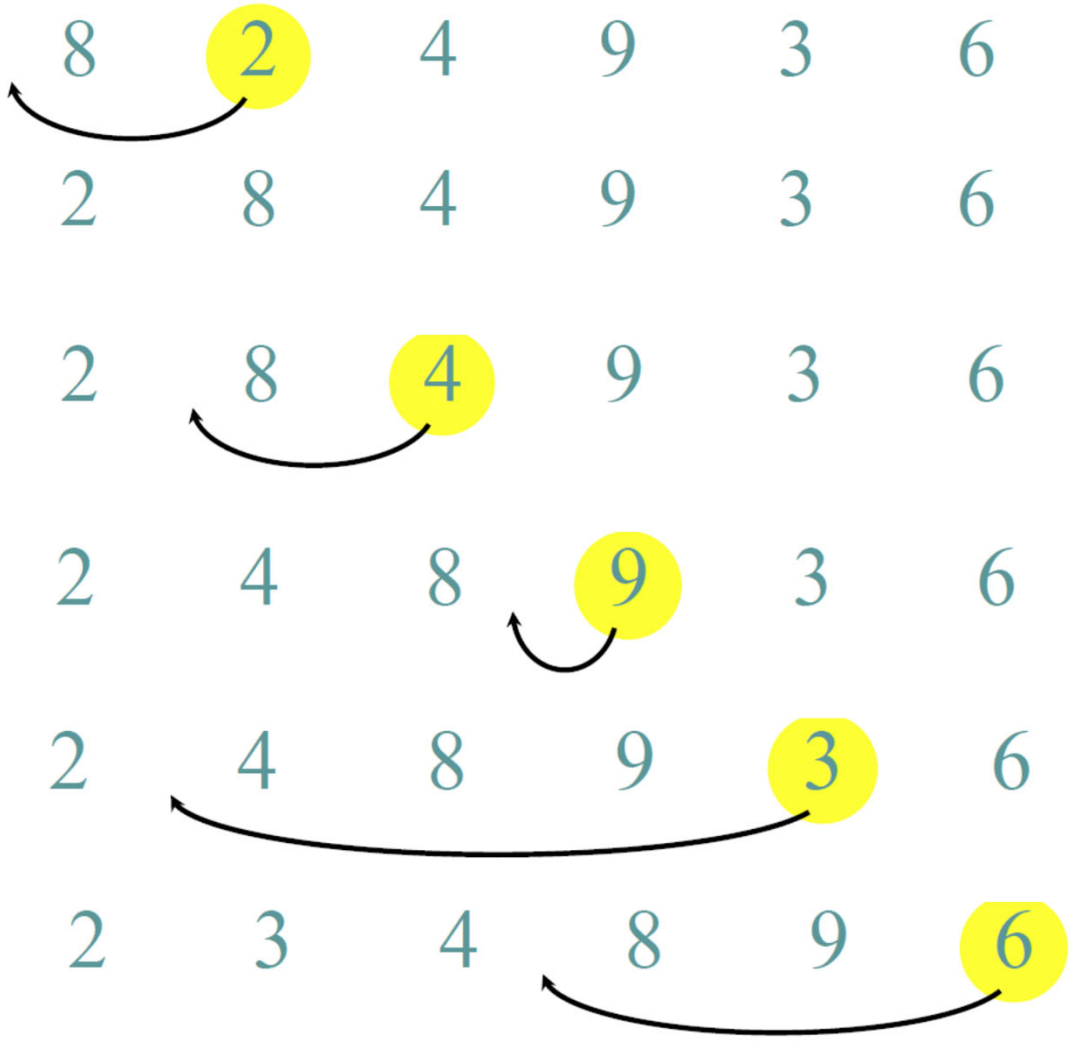
1	2	3	4	5	6
8	2	4	9	3	6




```
INSERTION-SORT (A, n)                ▷ A[1 . . n]
for j ← 2 to n
        do key ← A[ j]
        i ← j – 1
        while i > 0 and A[i] > key
                do A[i+1] ← A[i]
                i ← i – 1
        A[i+1] = key
```

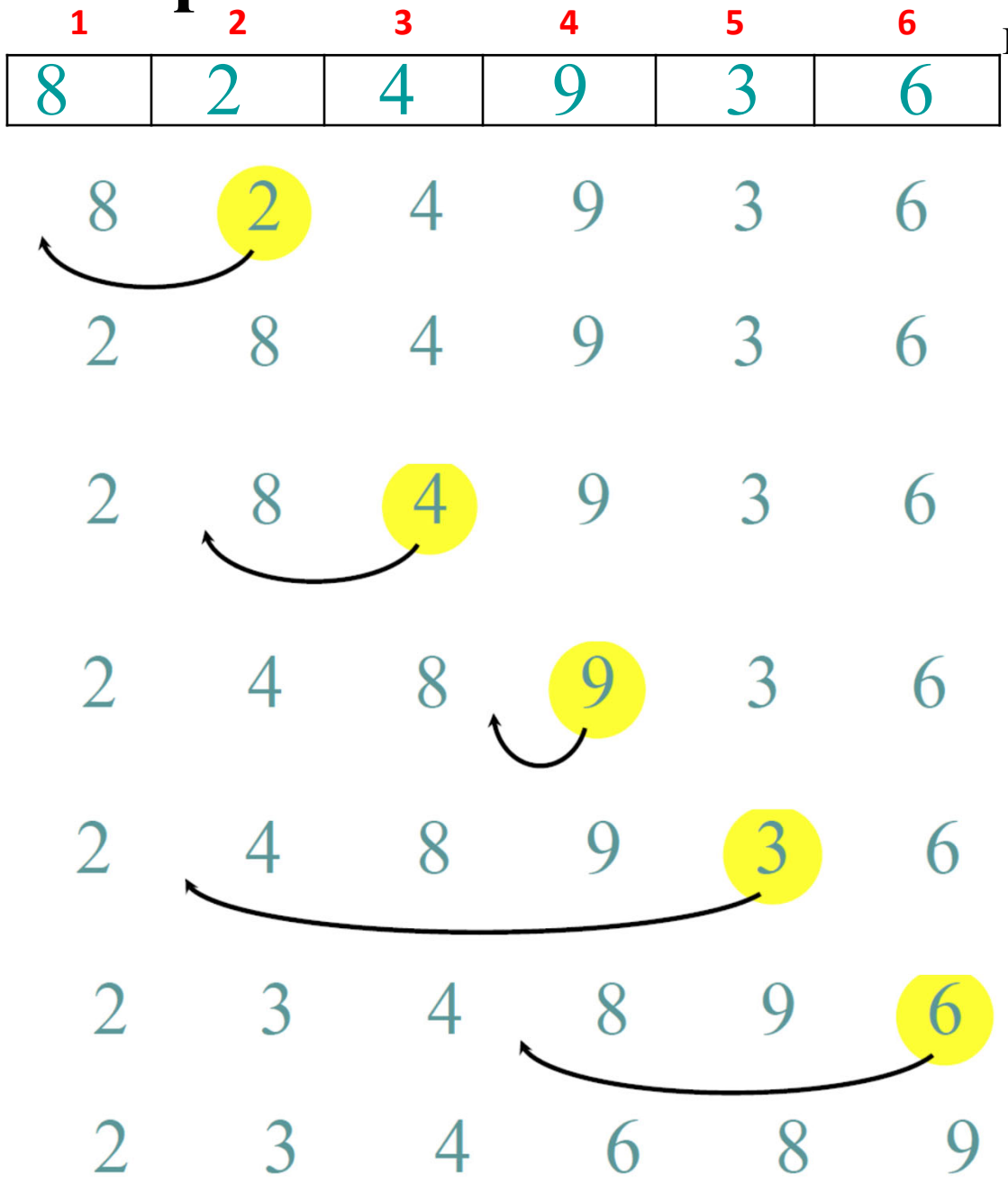
Example of insertion sort

1	2	3	4	5	6
8	2	4	9	3	6




```
INSERTION-SORT (A, n)                ▷ A[1 . . n]
for j ← 2 to n
        do key ← A[ j]
        i ← j – 1
        while i > 0 and A[i] > key
                do A[i+1] ← A[i]
                i ← i – 1
        A[i+1] = key
```

Example of insertion sort




```
INSERTION-SORT (A, n)                ▷ A[1 . . n]
for j ← 2 to n
    do key ← A[ j]
    i ← j – 1
    while i > 0 and A[i] > key
        do A[i+1] ← A[i]
        i ← i – 1
    A[i+1] = key
```

Algorithms & Data Structures

- Professor Reza Sedaghat

- COE428: Engineering Algorithms & Data Structures
- Email address: rsedagha@ee.ryerson.ca
- Course outline: www.ee.ryerson.ca/~courses/COE428/
- Course References:

  1) **Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. Introduction to Algorithms, MIT, 2002, ISBN: 0-07-013151-1 (McGraw-Hill) (Course Text)**

2 Running time

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

Timing analyses

Worst-case: (usually)

- $T(n)$ = maximum time of algorithm on any input of size n.

Average-case: (sometimes)

- $T(n)$ = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case: (bogus)

- Cheat with a slow algorithm that works fast on *some* input.

- **Analyzing algorithms**
  - Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.

  - Occasionally, resources such as memory, communication bandwidth, or logic gates are of primary concern

  - **Most often it is computational time that we want to measure**.

- **Analysis of insertion sort**
  - The time taken by the INSERTION-SORT procedure depends on the input: sorting a thousand numbers takes longer than sorting three numbers.

  - Moreover, INSERTION-SORT can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are.

  - In general, the time taken by an algorithm grows with the size of the input

  - In general, the running time of a program is described as a function of the size of its input.

- We need to define the terms "**running time**" and "**input size**“

- **Input size**
  - For many problems the most natural measure is the number of items for the input

  - We shall indicate which input size measure is being used with each problem we study.

- **Running time**
  - 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

  - One line may take a different amount of time than another line, but we shall aassume that each execution of the i-th line takes time c_i, where c_i is a constant.

2.2 Running time

• The INSERTION-SORT procedure with the time "cost" of each statement and the number of times each statement is executed.

• Example: For each j = 2, 3, . . . , n where n = length[A], we let t_j be the number of times the **"while"** loop in line 5 is executed for that value of j

• Note that when a **for** or **while** loop exits in the usual way, due to the test in the loop header, the test is executed one time more than the loop body.

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

• Running time

•The running time of the algorithm is the sum of running times for each statement executed;

• A statement that takes c_i steps to execute (cost) and is executed n times will contribute c_i . n to the total running time.

• To compute T(n), the running time of INSERTION-SORT, we sum the products of the cost and times columns, obtaining

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

- **Running time**

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

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

- **Running time**

```
INSERTION-SORT(A)                                          cost   times
for j ← 2 to n                                             c1     n
    do key ← A[j]                                          c2     n − 1
       ▷ Insert A[j] into the sorted sequence A[1 . . j − 1].  0     n − 1
       i ← j − 1                                           c4     n − 1
       while i > 0 and A[i] > key                          c5     ∑_{j=2}^{n} t_j
           do A[i + 1] ← A[i]                              c6     ∑_{j=2}^{n} (t_j − 1)
              i ← i − 1                                    c7     ∑_{j=2}^{n} (t_j − 1)
       A[i + 1] ← key                                      c8     n − 1
```

- **Best case:** The array is already sorted.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= \underbrace{(c_1 + c_2 + c_4 + c_5 + c_8)}n - \underbrace{(c_2 + c_4 + c_5 + c_8)} . \end{aligned}$$

This running time can be expressed as **an + b** for constants a and b that depend on the statement costs c_i; it is thus a **linear function of n**.

T (n) = an + b

- **Arithmetic series**
  - The summation

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n ,$$

  - Example: $n = 4$

$$\sum_{k=1}^{4} k = 1+2+3+4 = 10$$

  - Which came up when we analyzed insertion sort is an arithmetic series and has the value

$$\sum_{k=1}^{n} k = \frac{1}{2} n(n+1)$$

  - Example: $n = 4$

$$\sum_{k=1}^{4} k = \frac{1}{2} 4(4+1) = 10$$

• Running time

•Worst Case

• If the array is in reverse sorted order--that is, in decreasing order--the worst case results.

• We must compare each element A[j] with each element in the entire sorted subarray A[1. . j - 1], and so $t_j = j$ for j = 2,3, . . . , n.

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

• we find that in the worst case, the running time of INSERTION-SORT is

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

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

• This worst-case running time can be expressed as $\boldsymbol{an^2 + bn + c}$ for constants $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ that again depend on the statement costs c_i; it is thus a quadratic function of n

•Running time

• We usually concentrate on finding the **worst-case running time**: the longest running time for any input of size n.

•Reasons:

- The worst-case running time gives a guaranteed upper bound on the running time for any input.
- For some algorithms, the worst case occurs often.
- For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.

- **Order of growth**
- Another abstraction to ease analysis and focus on the important features.
  - Look only at the leading term of the formula for running time.
  - Drop lower-order terms.
  - Ignore the constant coefficient in the leading term.

- ***Example:*** For insertion sort, we already abstracted away the actual statement costs to conclude that the worst-case running time is $an^2 + bn + c$.
  - Drop lower-order terms => an^2 .
  - Ignore constant coefficient => n^2 .
  - But we cannot say that the worst-case running time $T(n)$ equals n^2 .
  - It grows like n^2 . But it doesn't *equal* n^2 .
  - We say that the running time is (n^2) to capture the notion that the order of growth is n^2 .

- We usually consider one algorithm to be more efficient than another if its worst case running time has a smaller order of growth.

Machine-independent time

What is insertion sort's worst-case time?

- It depends on the speed of our computer:
  - relative speed (on the same machine),
  - absolute speed (on different machines).

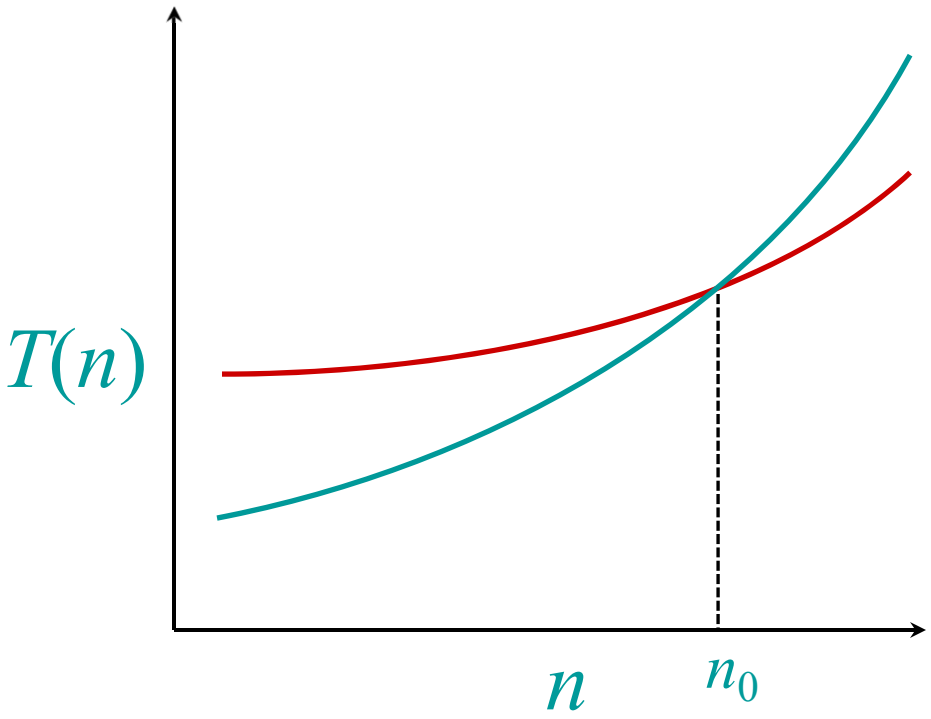
BIG IDEA:

- Ignore machine-dependent constants.
- Look at ***growth*** of $T(n)$ as $n \to \infty$.

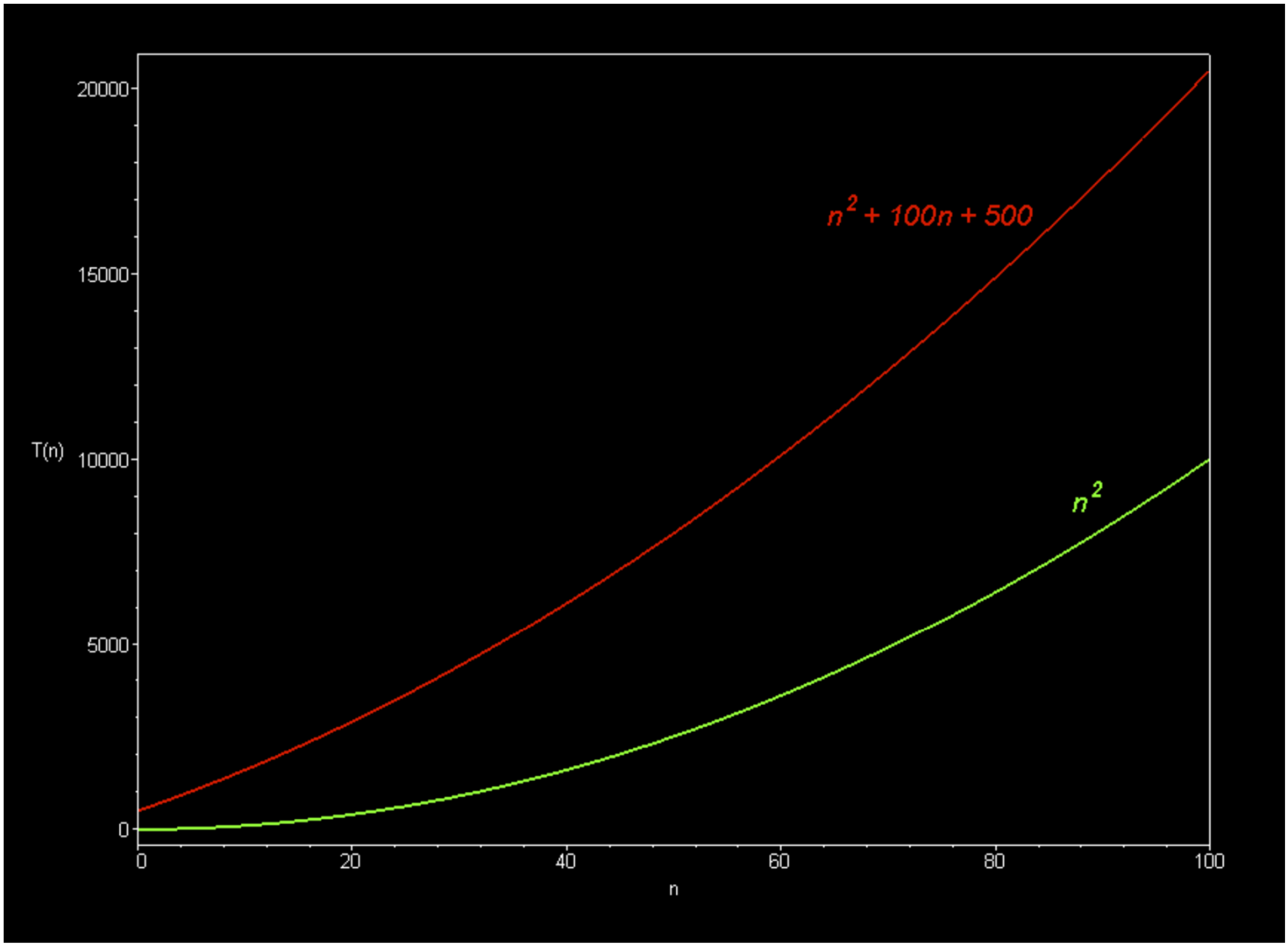
"Asymptotic Analysis"

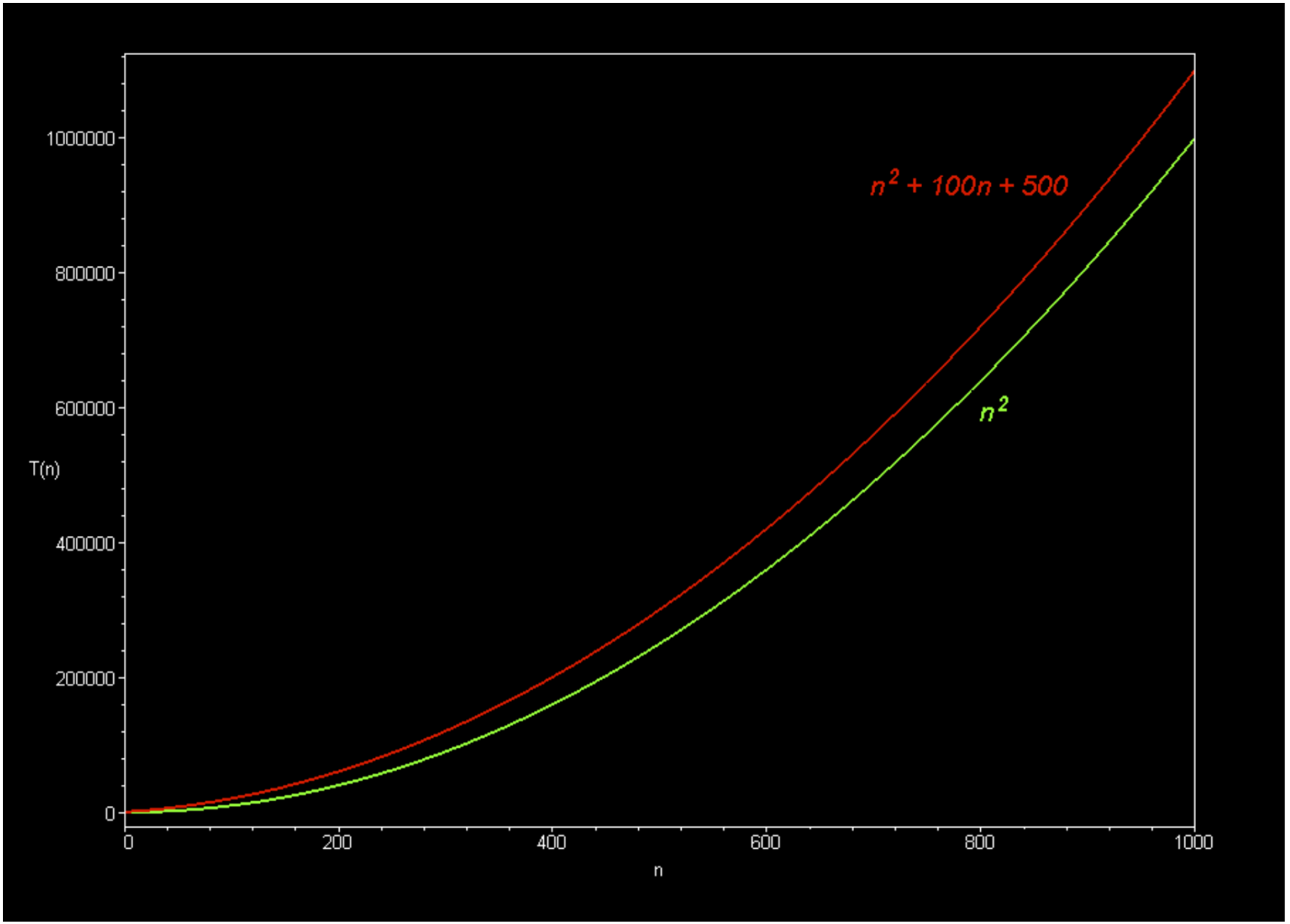
Asymptotic performance

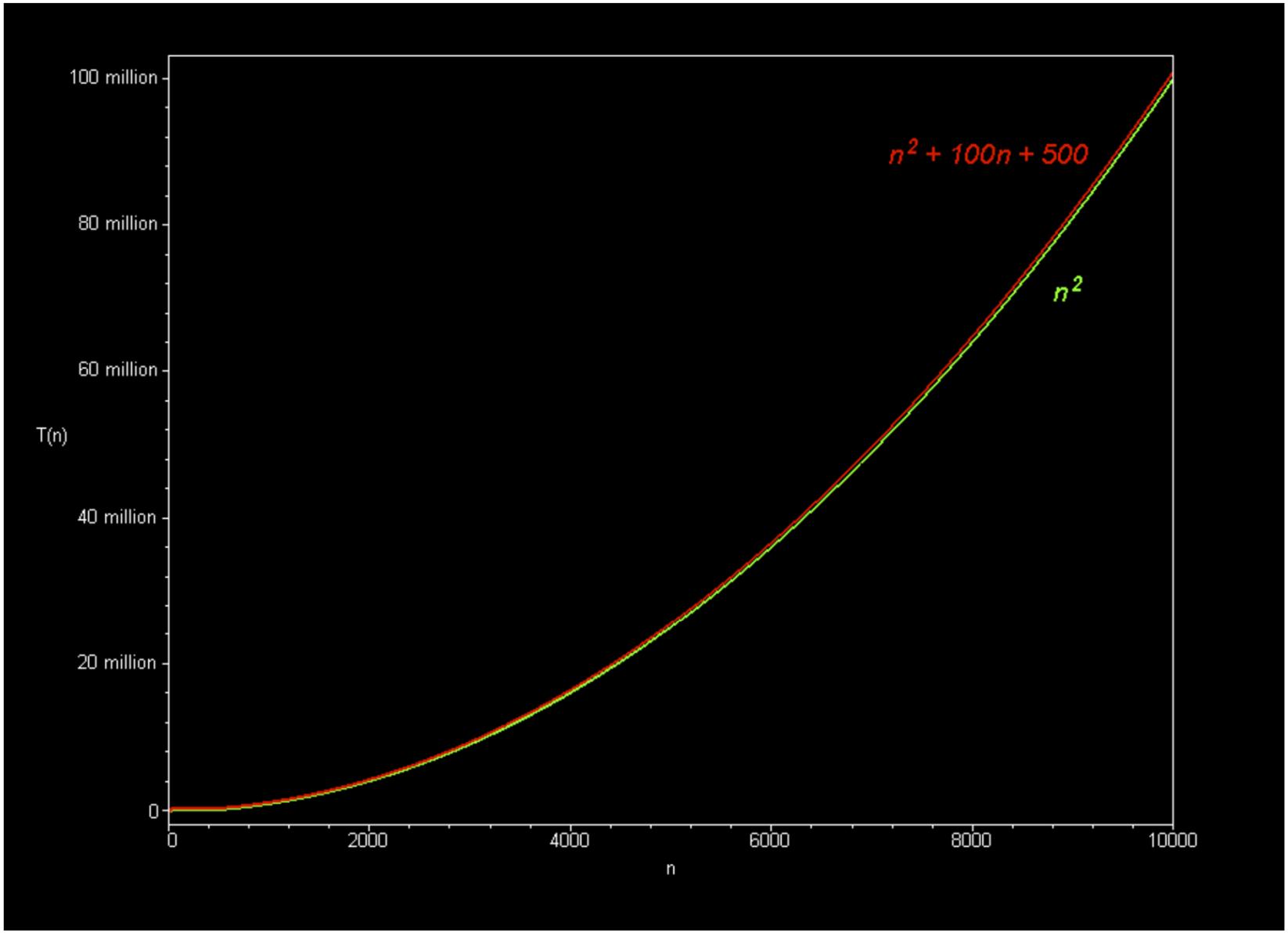
When n gets large enough, a $\Theta(n^2)$ algorithm *always* beats a $\Theta(n^3)$ algorithm.

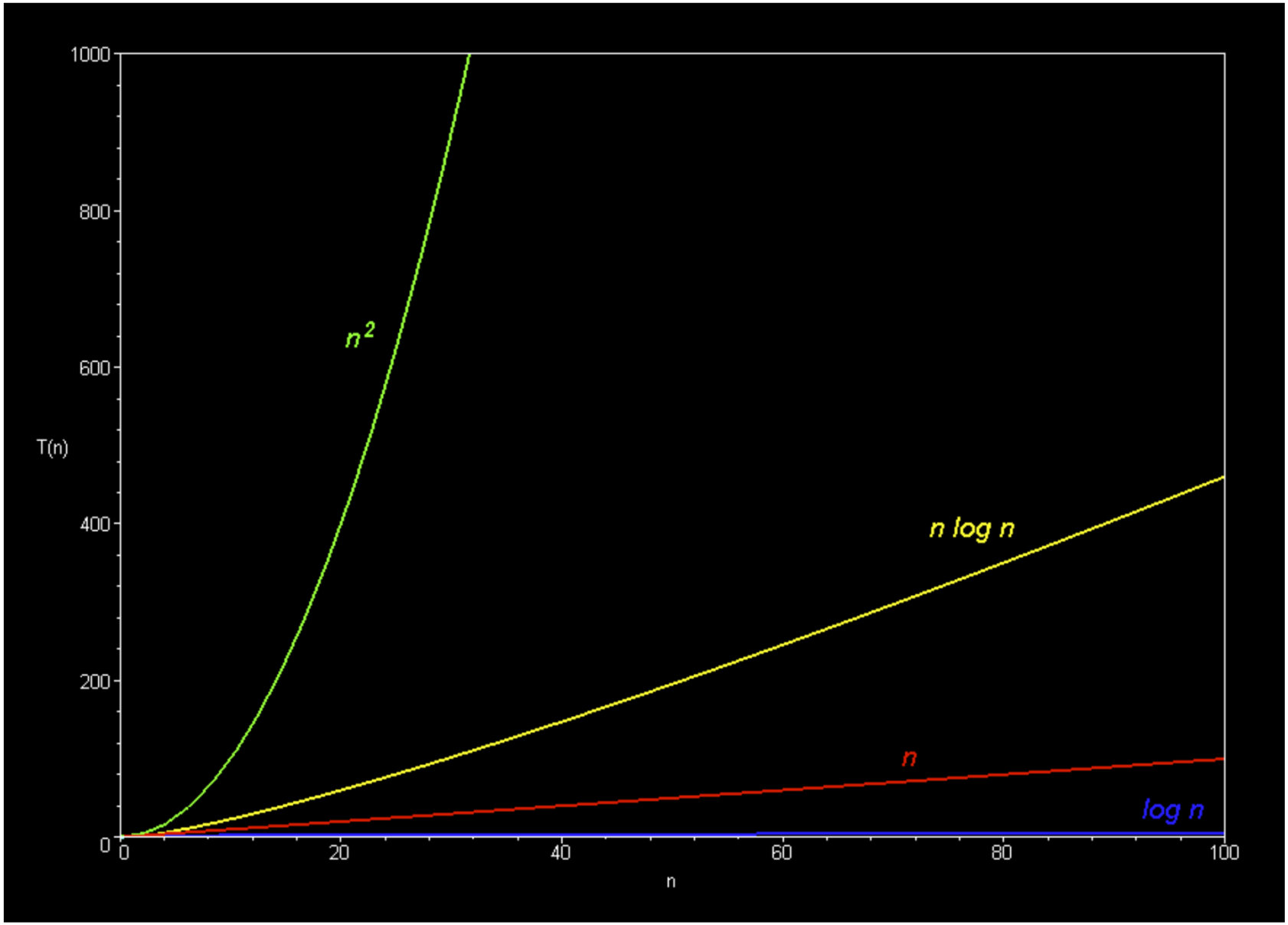


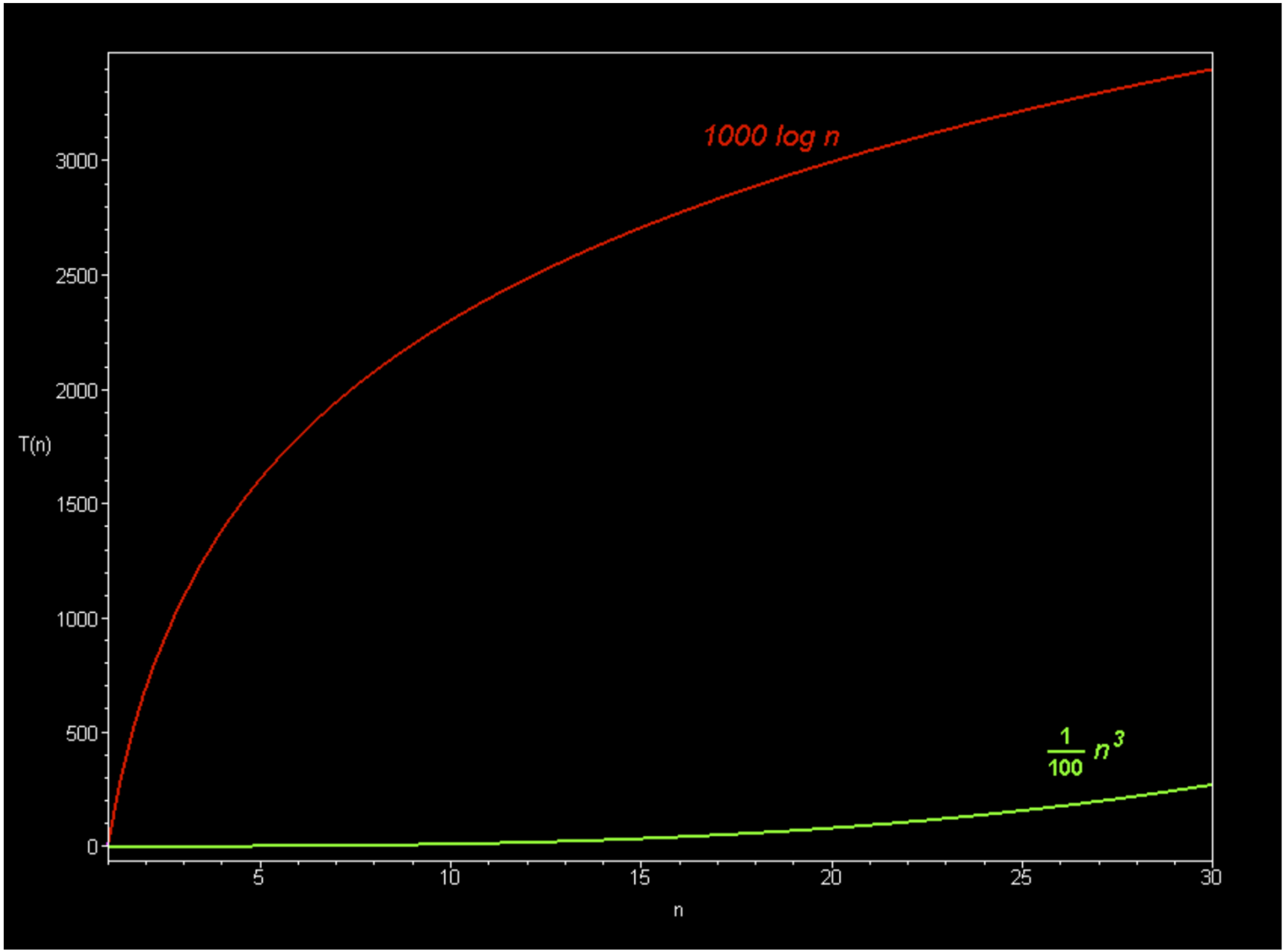
- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- Asymptotic analysis is a useful tool to help to structure our thinking.

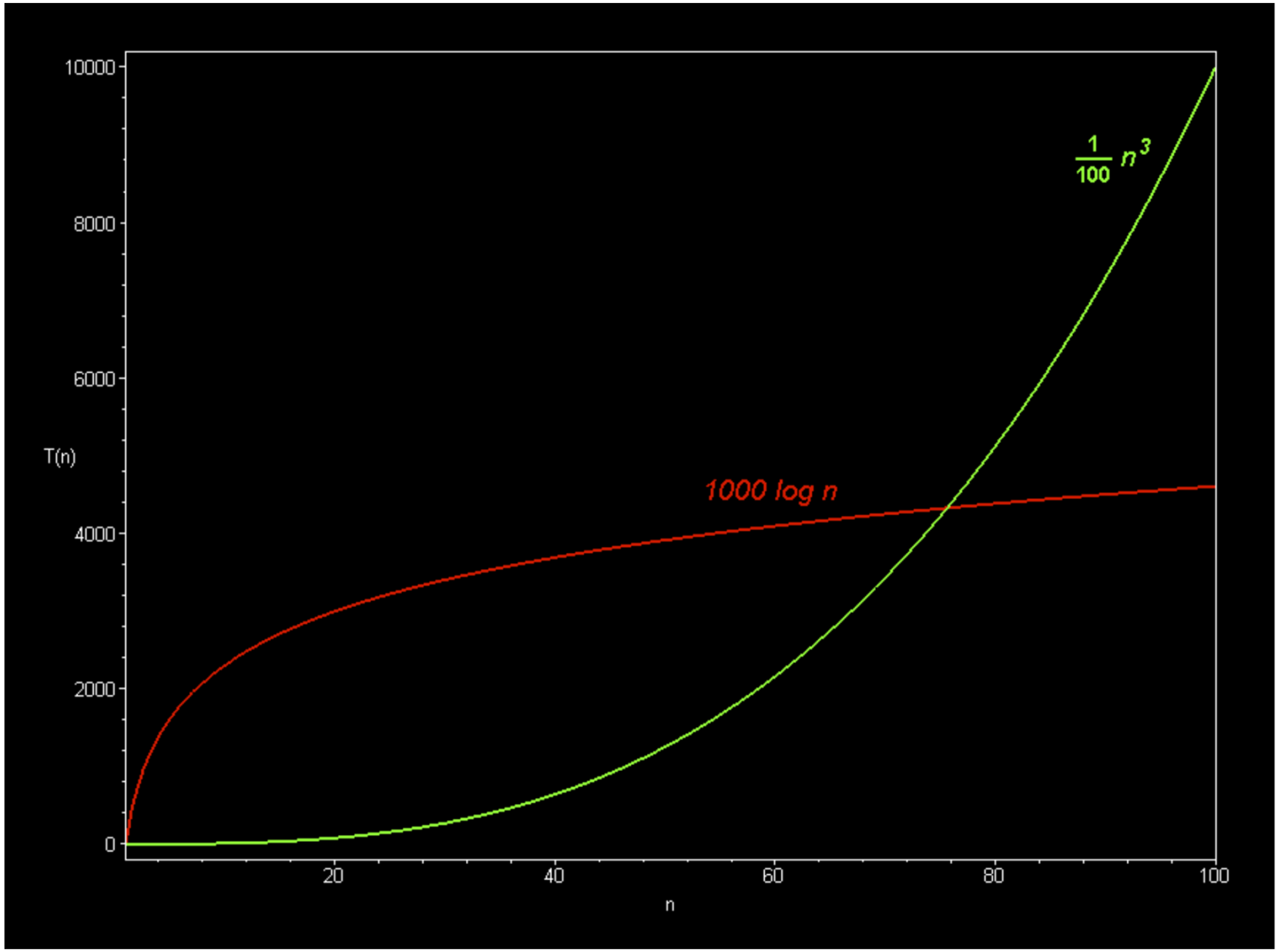

20000
15000
10000
5000
0
T(n)
0
20
40
60
80
100
n
$n^2 + 100n + 500$
n^2

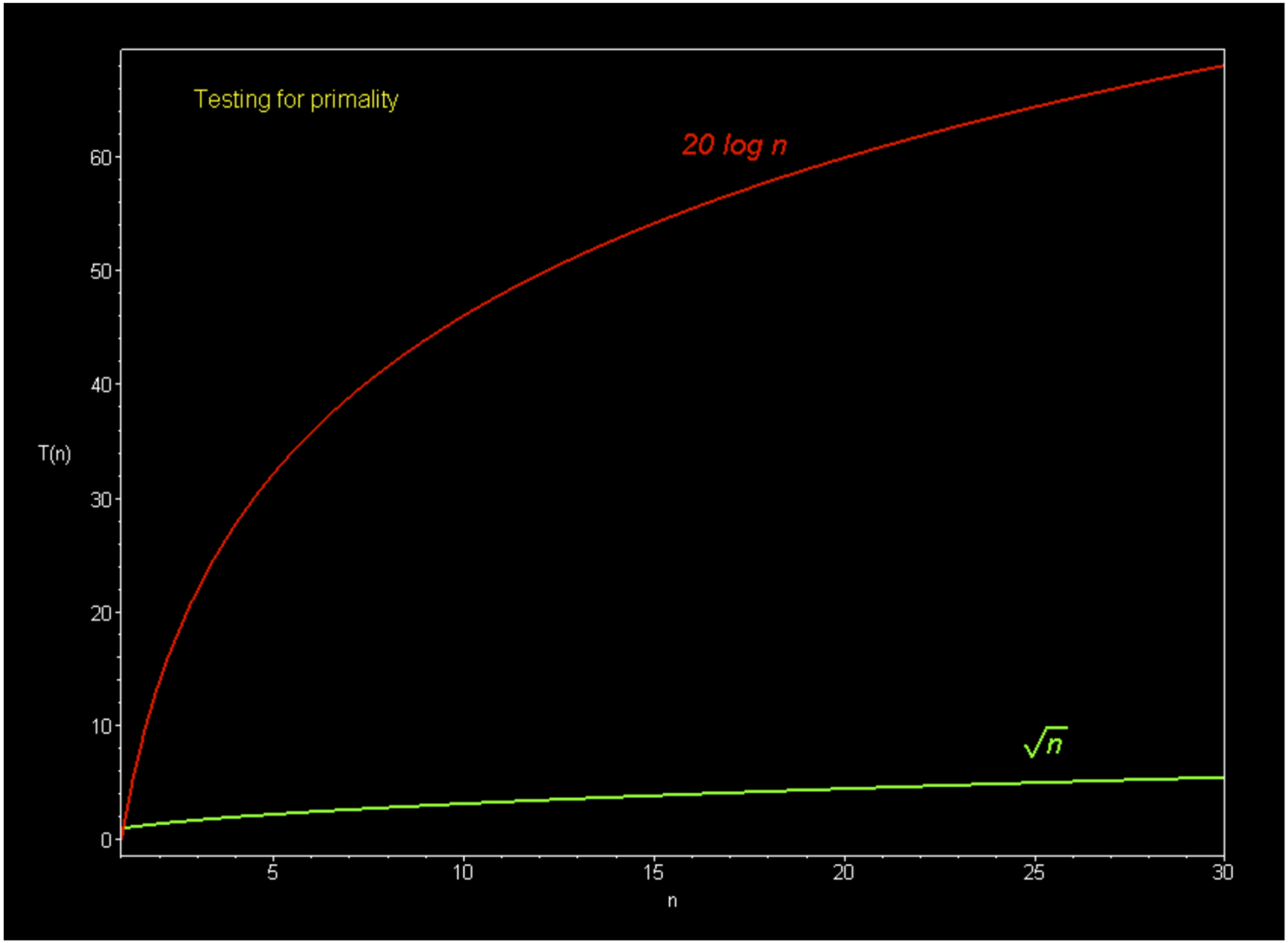

1000000
800000
600000
400000
200000
0
T(n)
0
200
400
600
800
1000
n
$n^2 + 100n + 500$
n^2

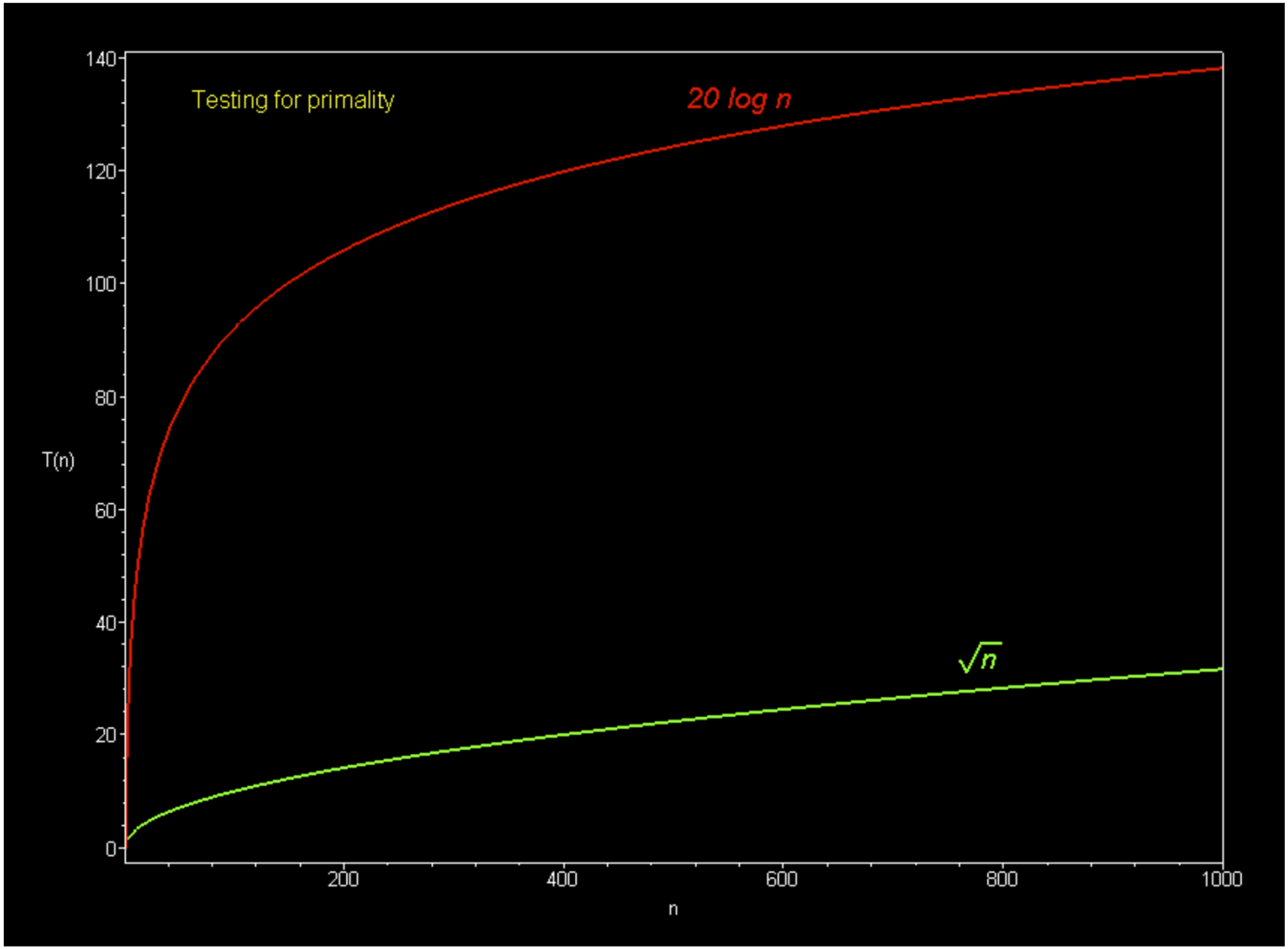

100 million
80 million
60 million
40 million
20 million
0
T(n)
0
2000
4000
6000
8000
10000
n
$n^2 + 100n + 500$
n^2

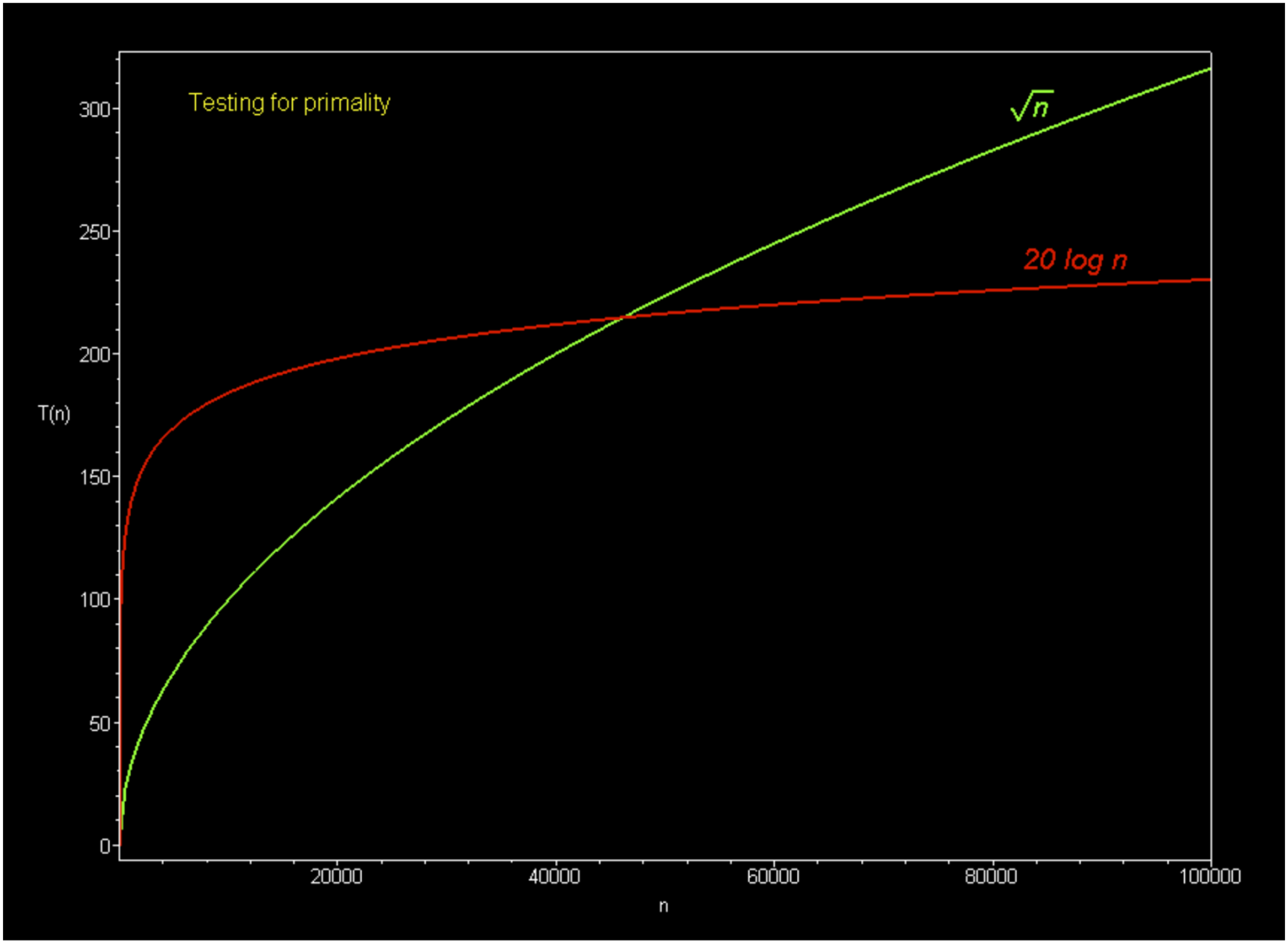

1000
800
600
400
200
0
T(n)
0
20
40
60
80
100
n
n 2
n log n
n
log n

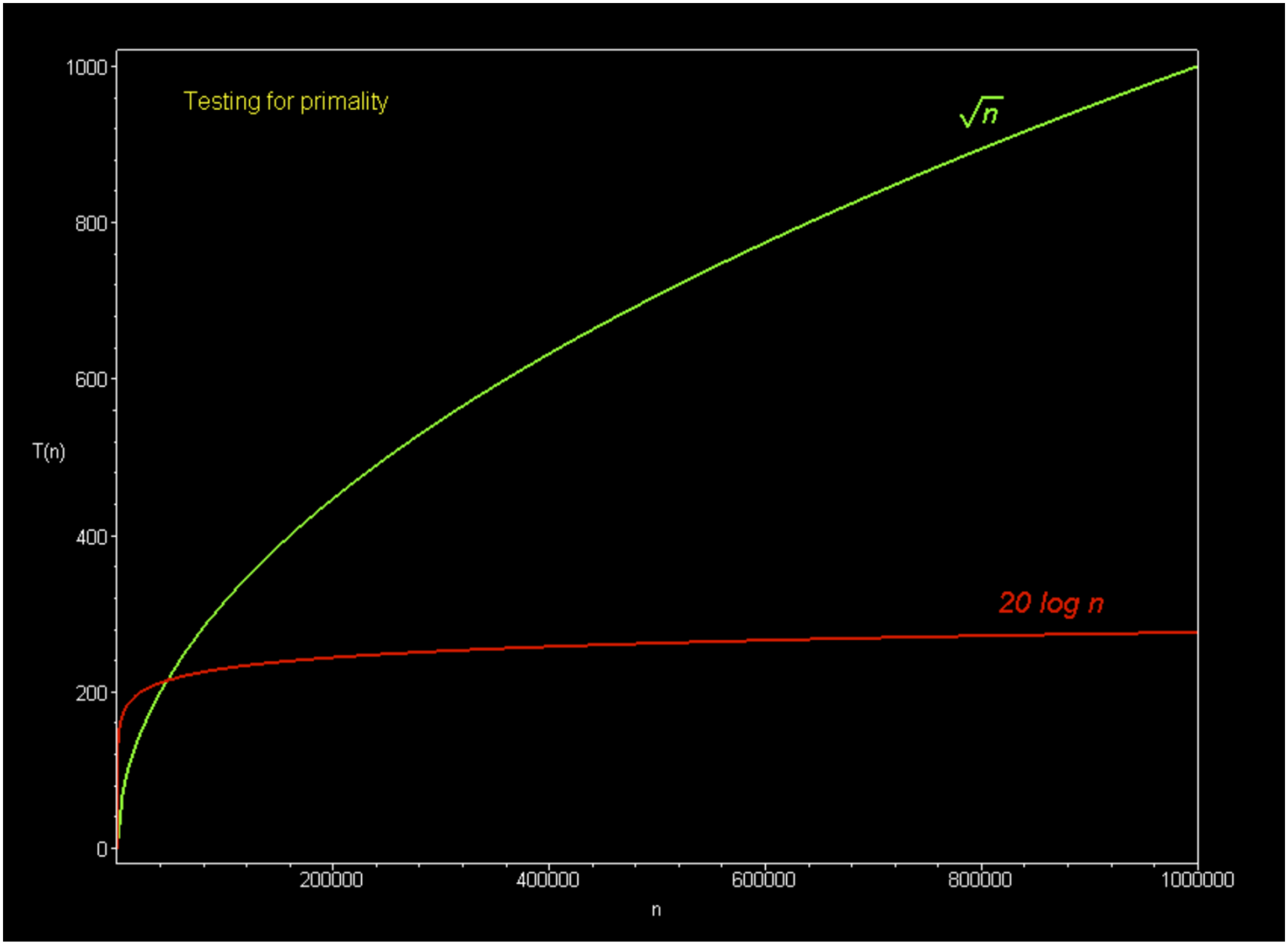

1000 log n
1/100 n^3
T(n)
n
0
500
1000
1500
2000
2500
3000
5
10
15
20
25
30


10000
8000
6000
4000
2000
0
T(n)
20
40
60
80
100
n
1000 log n
1/100 n^3


Testing for primality
20 log n
√n
T(n)
n
0
10
20
30
40
50
60
5
10
15
20
25
30


Testing for primality
20 log n
√n
T(n)
n
0
20
40
60
80
100
120
140
200
400
600
800
1000


Testing for primality
$\sqrt{n}$
20 log n
T(n)
300
250
200
150
100
50
0
20000
40000
60000
80000
100000
n


Testing for primality
$\sqrt{n}$
20 log n
T(n)
n
0
200
400
600
800
1000
200000
400000
600000
800000
1000000

2.3 Designing algorithms

• There are many ways to design algorithms.

• For example, insertion sort is ***incremental***: having sorted A[1 ..j -1], place A[j] correctly, so that A[1 ..j] is sorted.

• Another common approach is **"Divide and conquer" or "Recursion"**

Recur means; *return, happen again, be repeated*

• Recursive algorithms often follow a general pattern:

  • **Divide** the problem into a number of **subproblems**.

  • **Conquer** the subproblems by solving them **recursively**.

  • ***Base case:*** If the subproblems are small enough, just solve them by brute force.

  • **Combine** the subproblem solutions to give a solution to the original problem.

- **Recursion (example)**
  - **Merge sort**

• A sorting algorithm based on divide and conquer. Its worst-case running time has a lower order of growth than insertion sort.

•Because we are dealing with subproblems, we state each subproblem as sorting a subarray A[p ..r]. Initially, p =1 and $r = n$, but these values change as we recurse through subproblems.

• To sort A[p ..r]:

• **Divide** by splitting into two subarrays A[p ..q] and A[q +1 ..r], where q is the halfway point of A[p ..r].

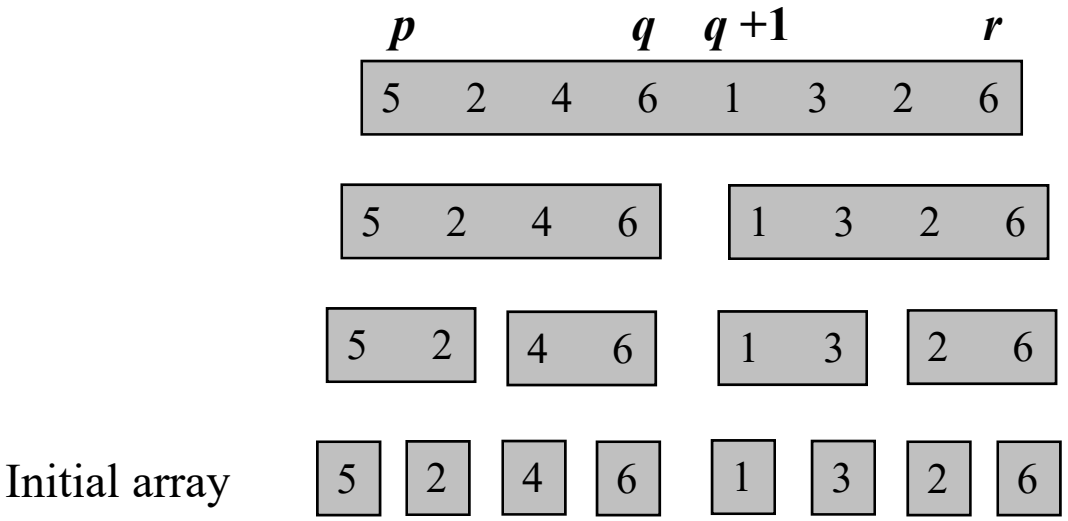
• **Conquer** by recursively sorting the two subarrays A[p ..q] and A[q +1 ..r].

• **Combine** by merging the two sorted subarrays A[p ..q] and A[q +1 ..r] to produce a single sorted subarray A[p ..r]. To accomplish this step, we'll define a procedure MERGE (A, p, q, r)

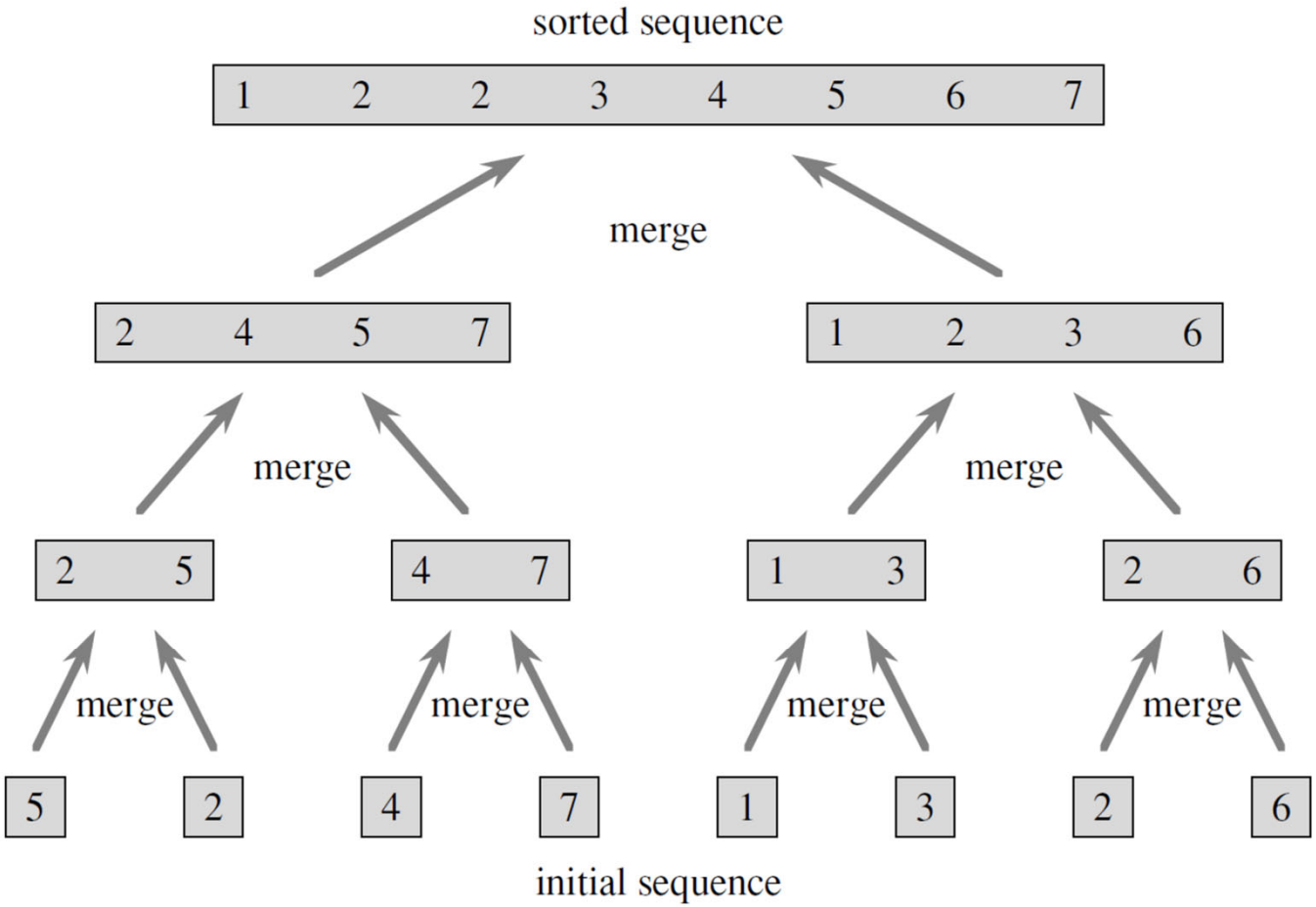
• The recursion bottoms out when the subarray has just 1 element, so that it's trivially sorted.

Merge sort

```
MERGE-SORT(A, p, r)
if p < r                                  ▷ Check for base case
  then q ← ⌊(p + r)/2⌋                    ▷ Divide
       MERGE-SORT(A, p, q)                ▷ Conquer
       MERGE-SORT(A, q + 1, r)            ▷ Conquer
       MERGE(A, p, q, r)                  ▷ Combine
```



Merge sort

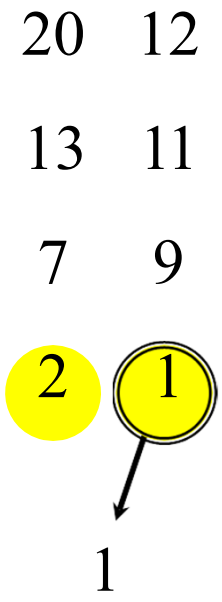


Time = $\Theta(n)$ to merge a total of n elements (linear time).

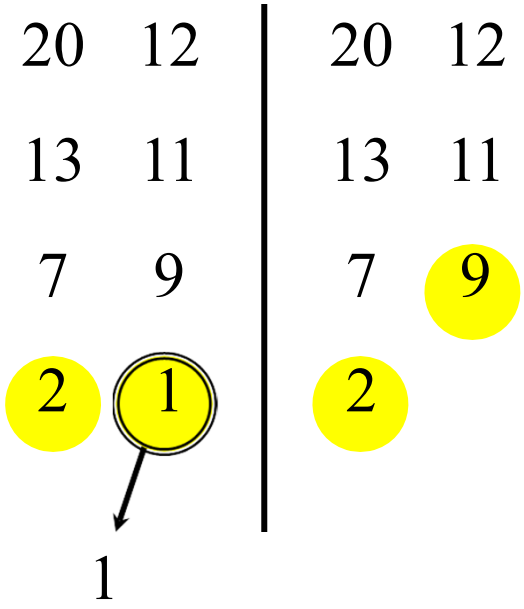
Merging two sorted arrays

20	12
13	11
7	9
2	1

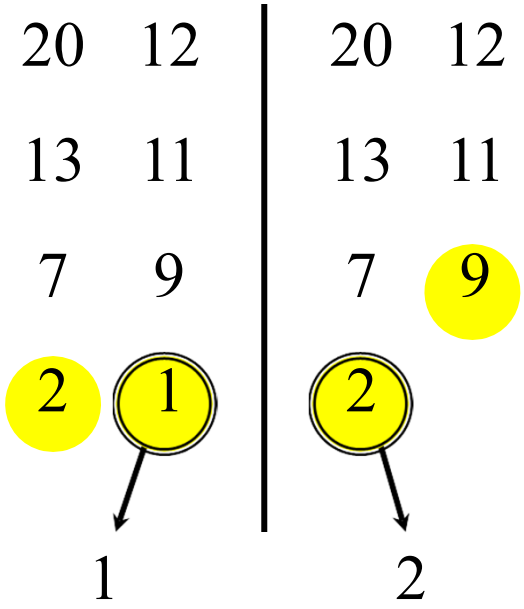
Merging two sorted arrays



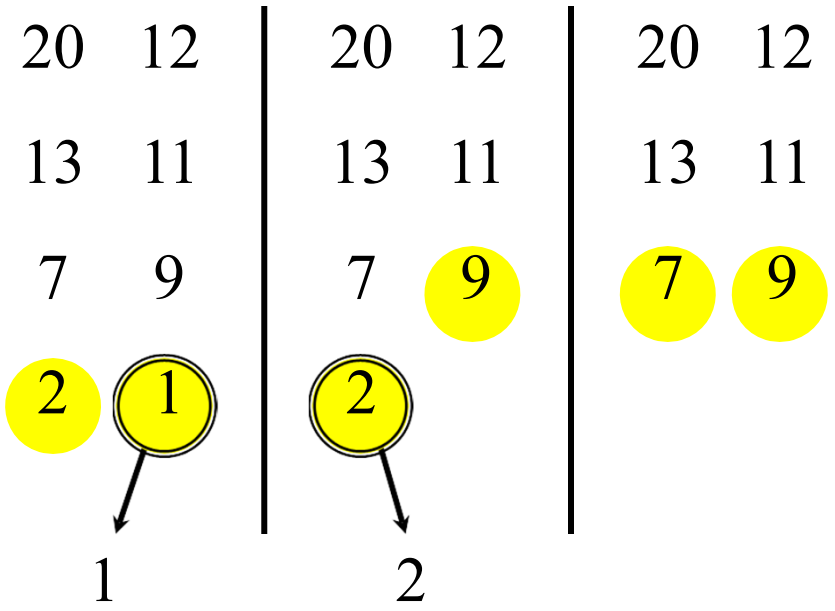
Merging two sorted arrays



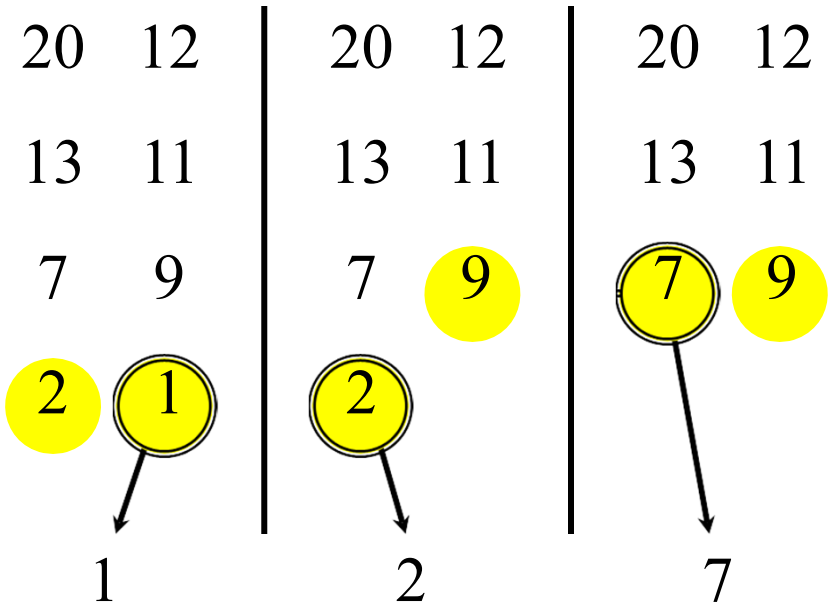
Merging two sorted arrays



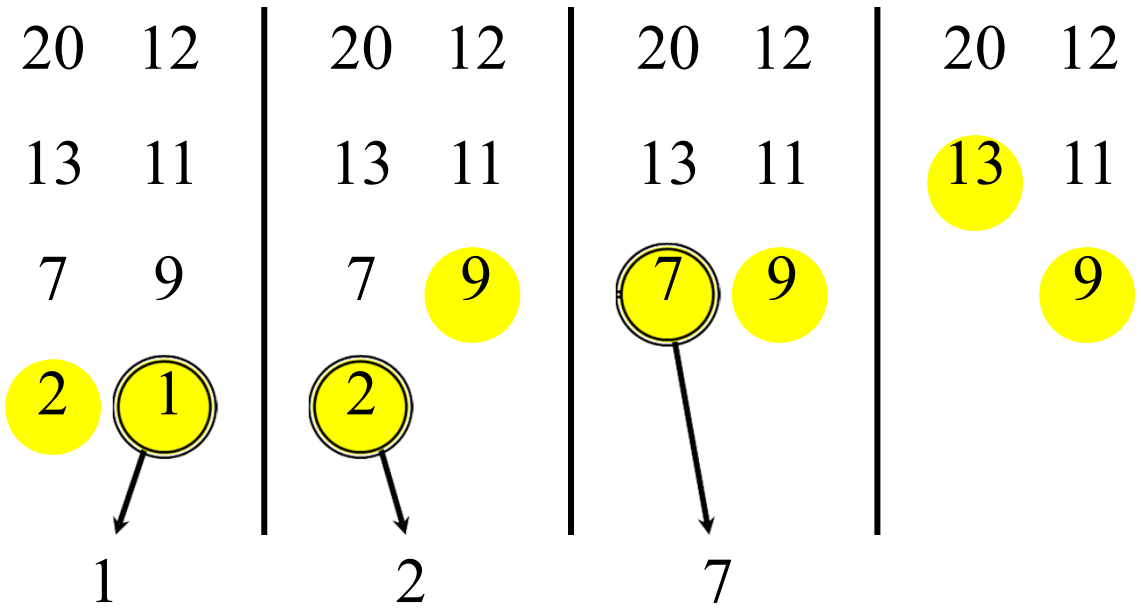
Merging two sorted arrays



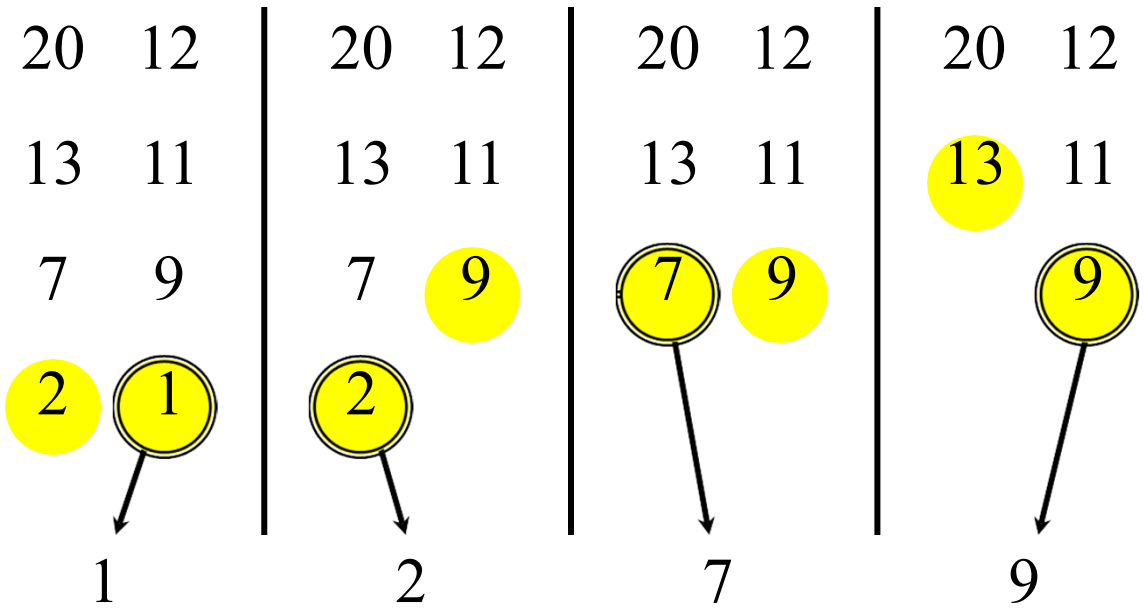
Merging two sorted arrays



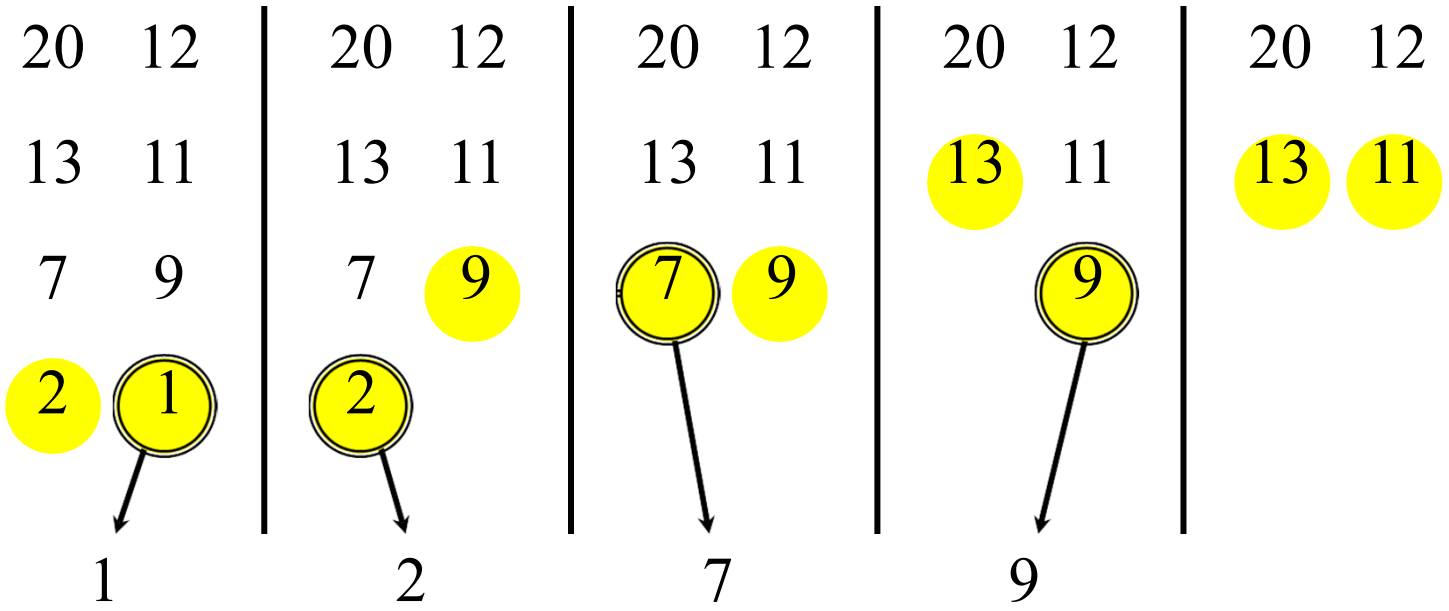
Merging two sorted arrays



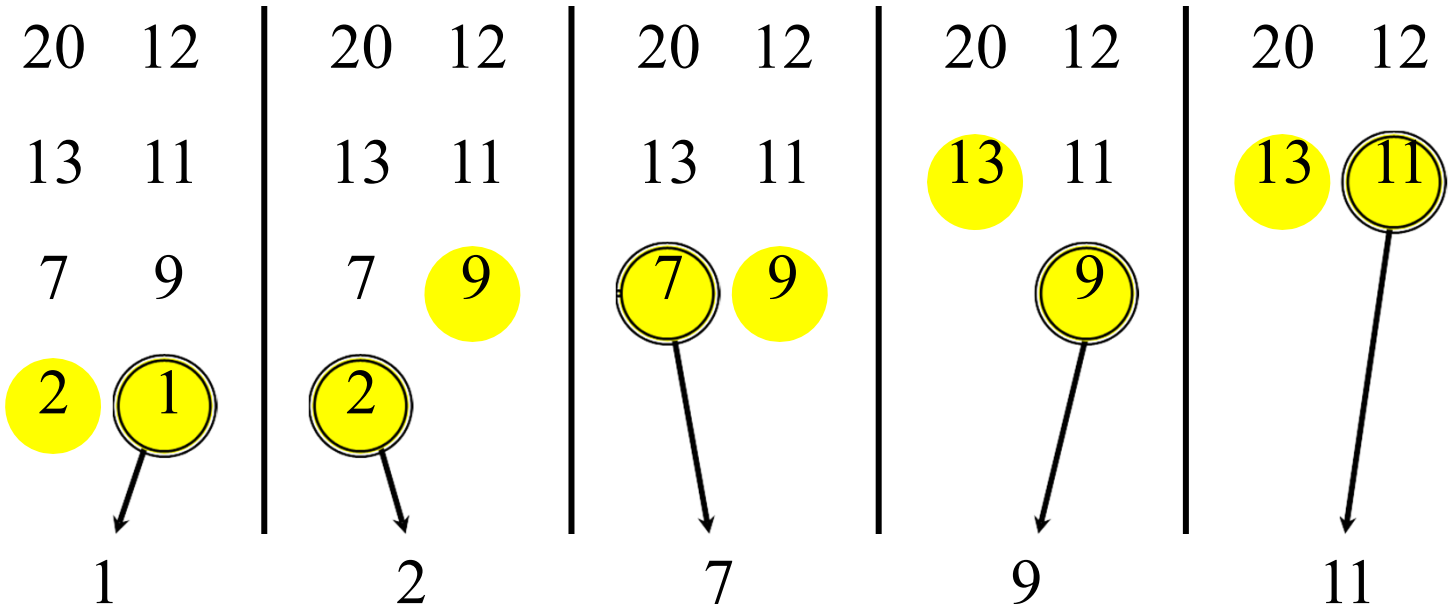
Merging two sorted arrays



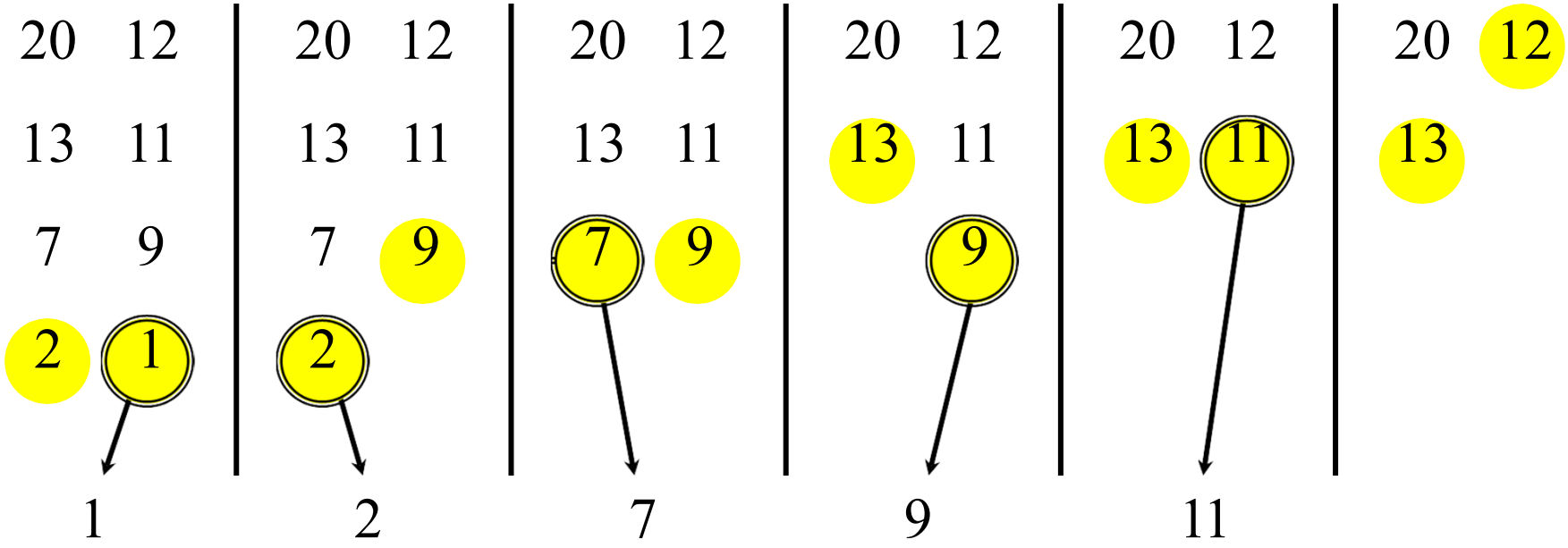
Merging two sorted arrays



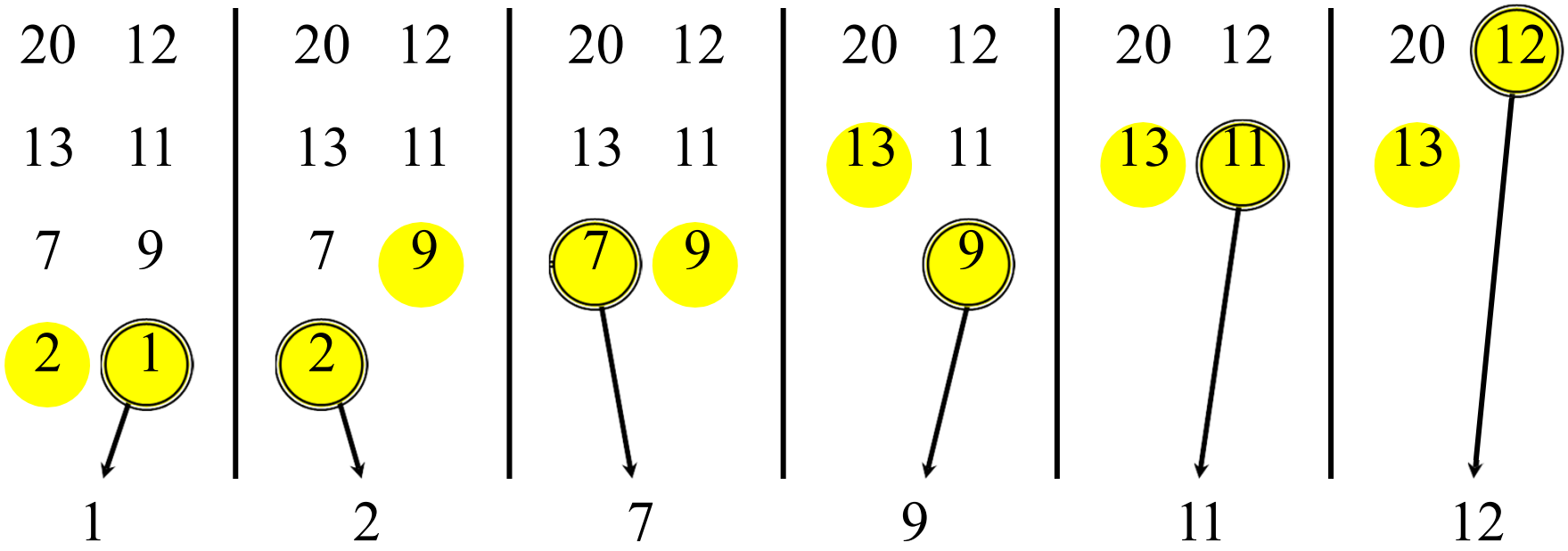
Merging two sorted arrays



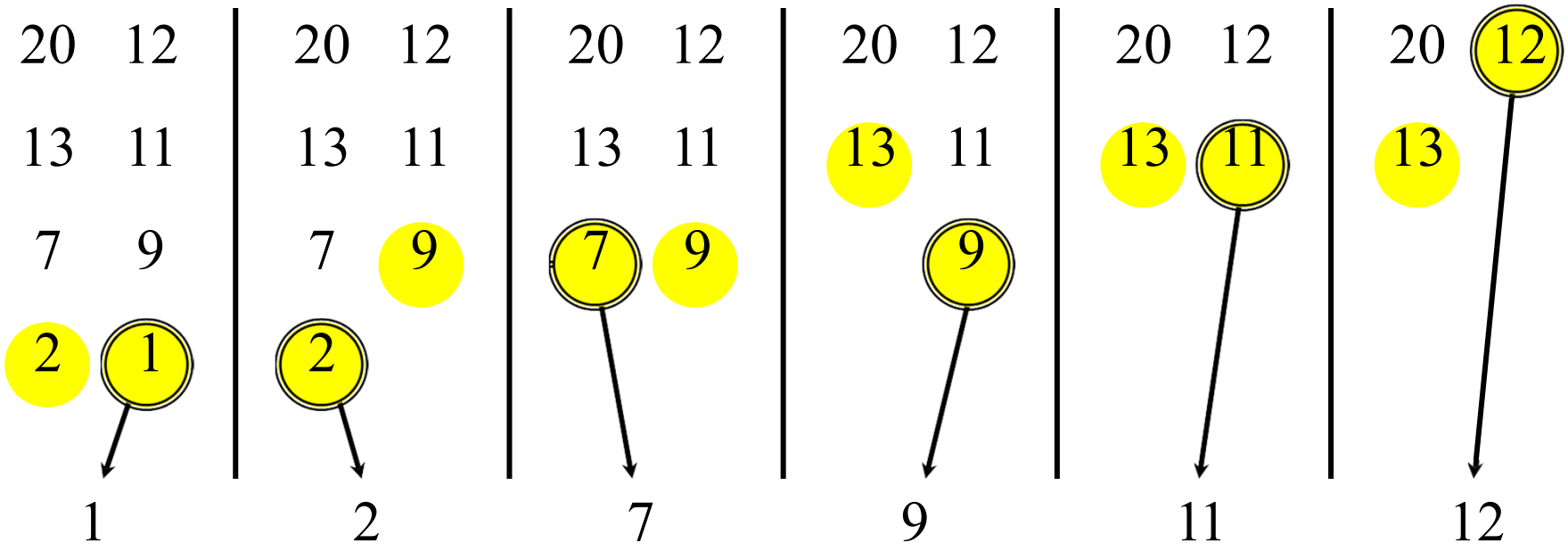
Merging two sorted arrays



Merging two sorted arrays



Merging two sorted arrays



Time = $\Theta(n)$ to merge a total of n elements (linear time).

Algorithms & Data Structures

- Professor Reza Sedaghat

- COE428: Engineering Algorithms & Data Structures
- Email address: rsedagha@ee.ryerson.ca
- Course outline: www.ee.ryerson.ca/~courses/COE428/
- Course References:

  1) **Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. Introduction to Algorithms, MIT, 2002, ISBN: 0-07-013151-1 (McGraw-Hill) (Course Text)**

2.3.2 Analysis of Merge Sort Algorithm

- How can we determine the time required to perform this algorithm?
  - Time to sort n numbers

T(n) = Time to divide the number +	T_{divide}
Time to sort left side (size = $n/2$) +	$T(n/2)$
Time to sort right side (size = $n/2$) +	$T(n/2)$
Time to merge (total size = $n/2 + n/2 = n$)	$c_1 n$

- Assumptions:
  - Let T(n) represent the time to sort n numbers
  - Let T_{divide} be the time to divide the numbers and assume $T_{divide} = 0$
  - Let $c_1 n + c_0$ be the time to merge n numbers, which is a linear algorithm and assume and $c_0 = 0$

$$T(n) = T_{divide} + T(n/2) + T(n/2) + c_1 n + c_0 = 2T(n/2) + n$$

Analysis of Merge Sort Algorithm

$$T(n) = T_{divide} + T(n/2) + T(n/2) + c_1 n + c_0 = 2T(n/2) + cn$$

• Let ***c*** be a constant that describes the running time for the base case and also is the time per array element for the divide and merge steps.

• We rewrite the recurrence as $$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(n/2) + cn & \text{if } n > 1. \end{cases}$$

Growth?

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

T(*n/2*): ?

n: Linear

Recurrence for merge sort

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

Recursion tree

- Shows successive expansions of the recurrence.

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

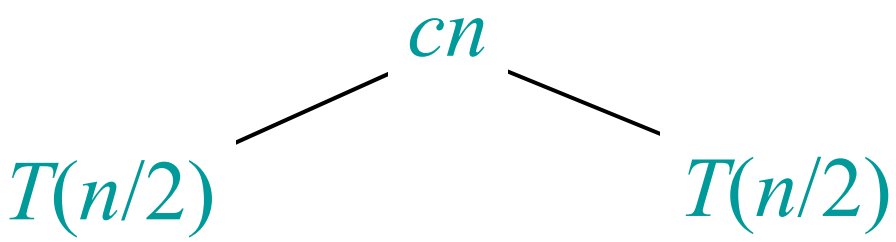
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$T(n)$

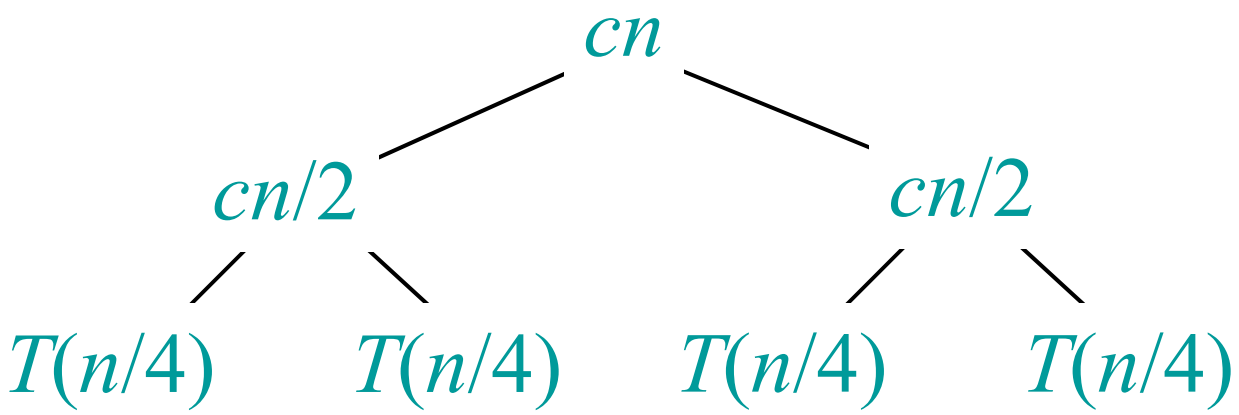
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



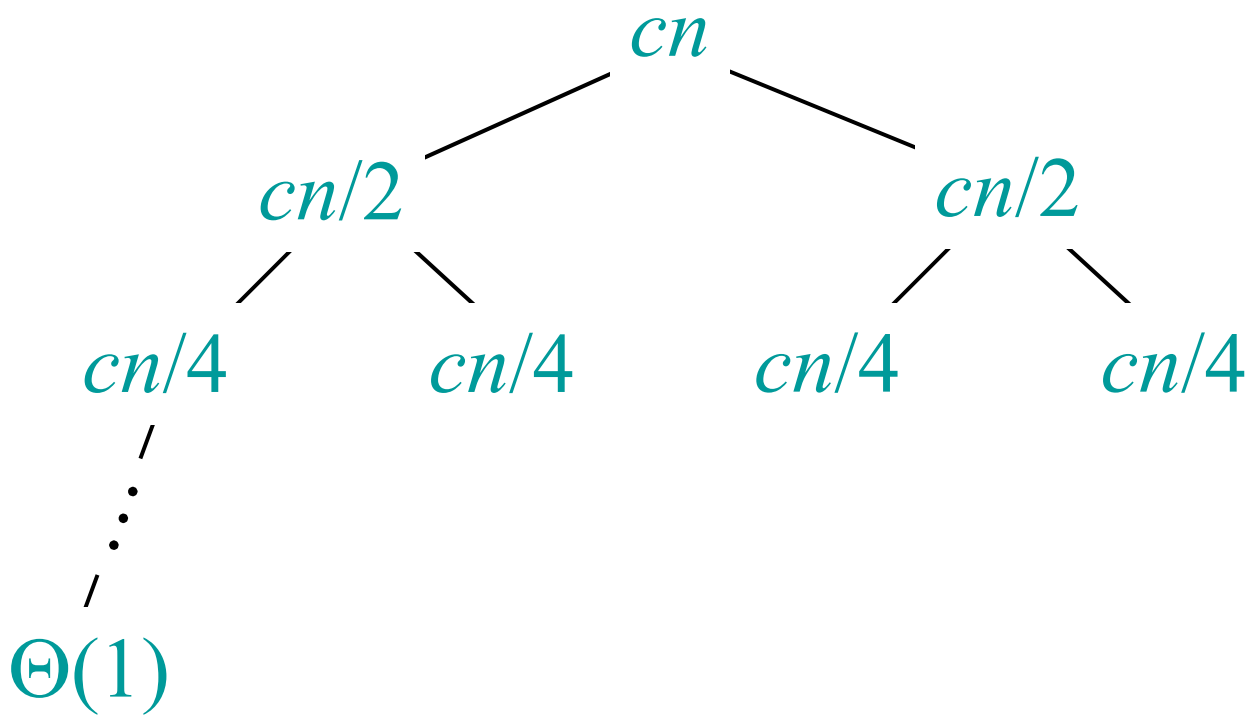
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



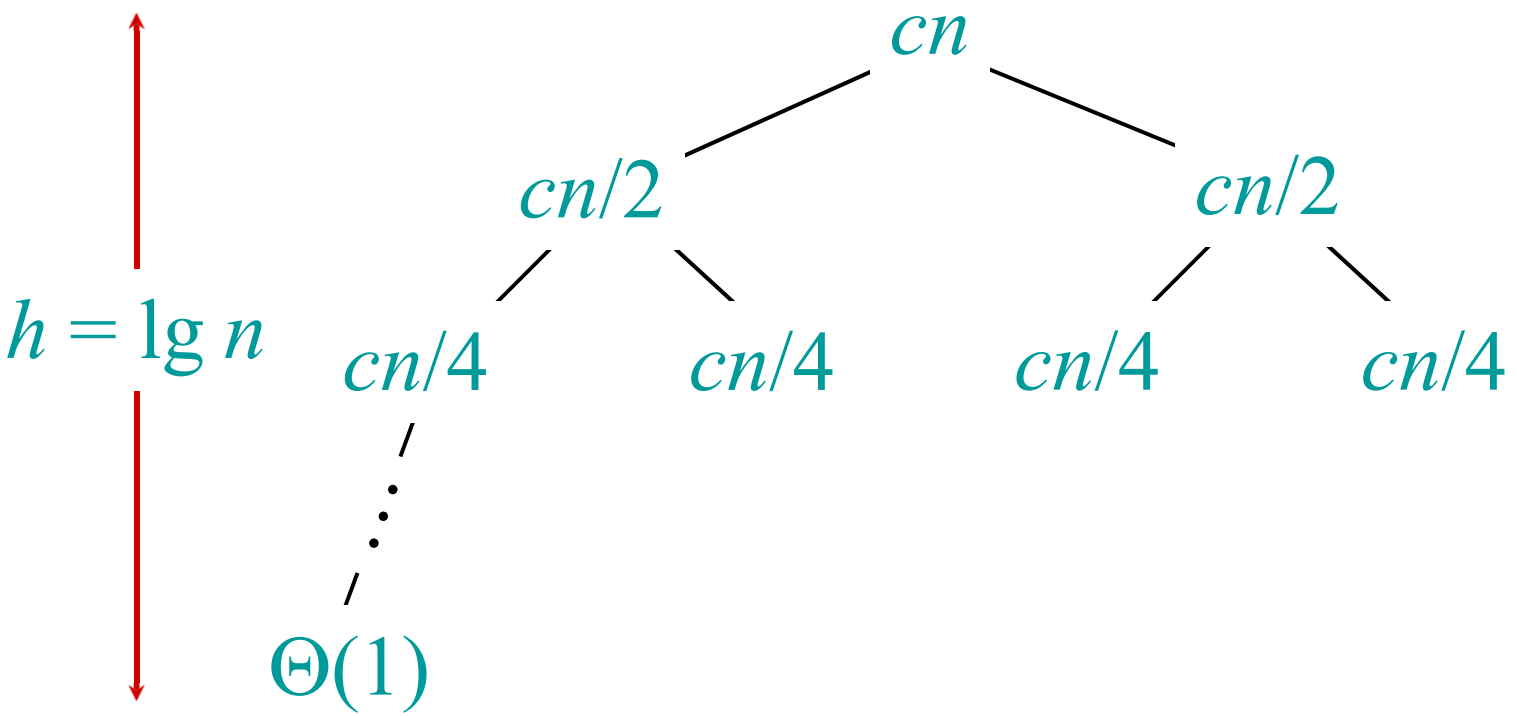
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



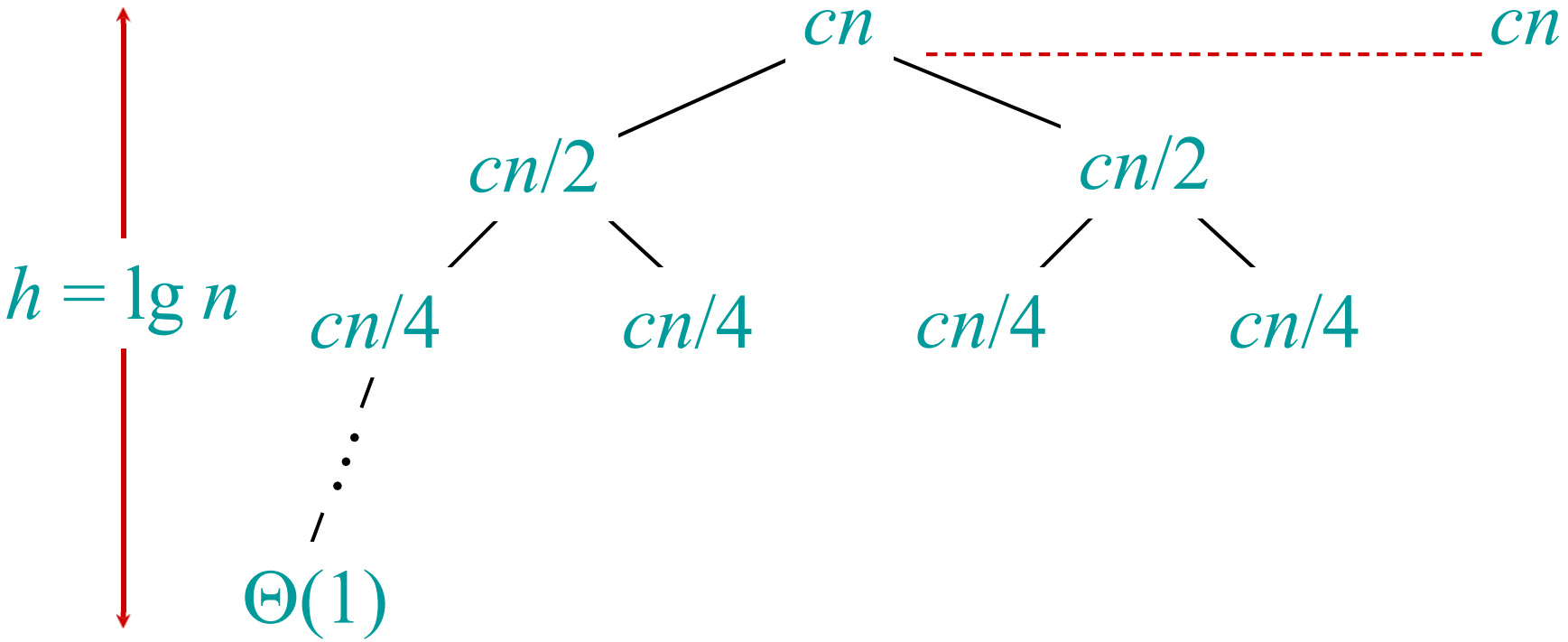
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



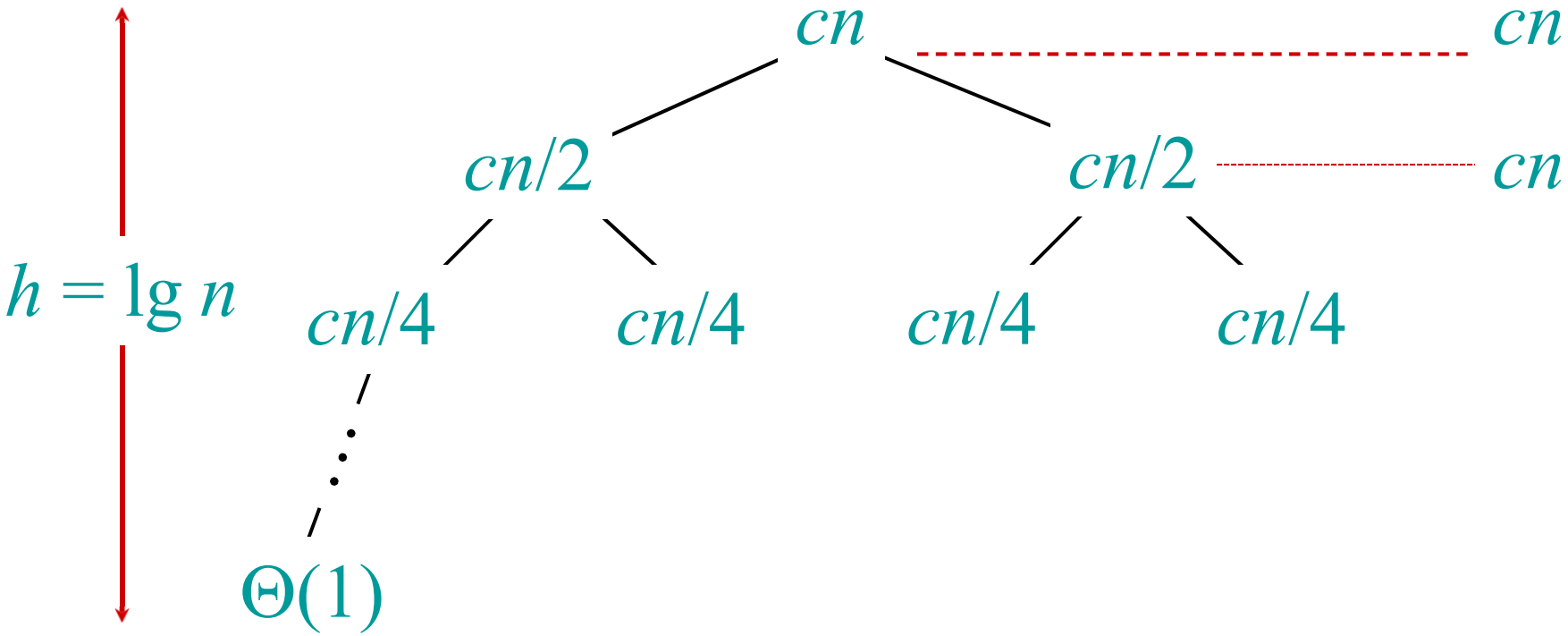
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



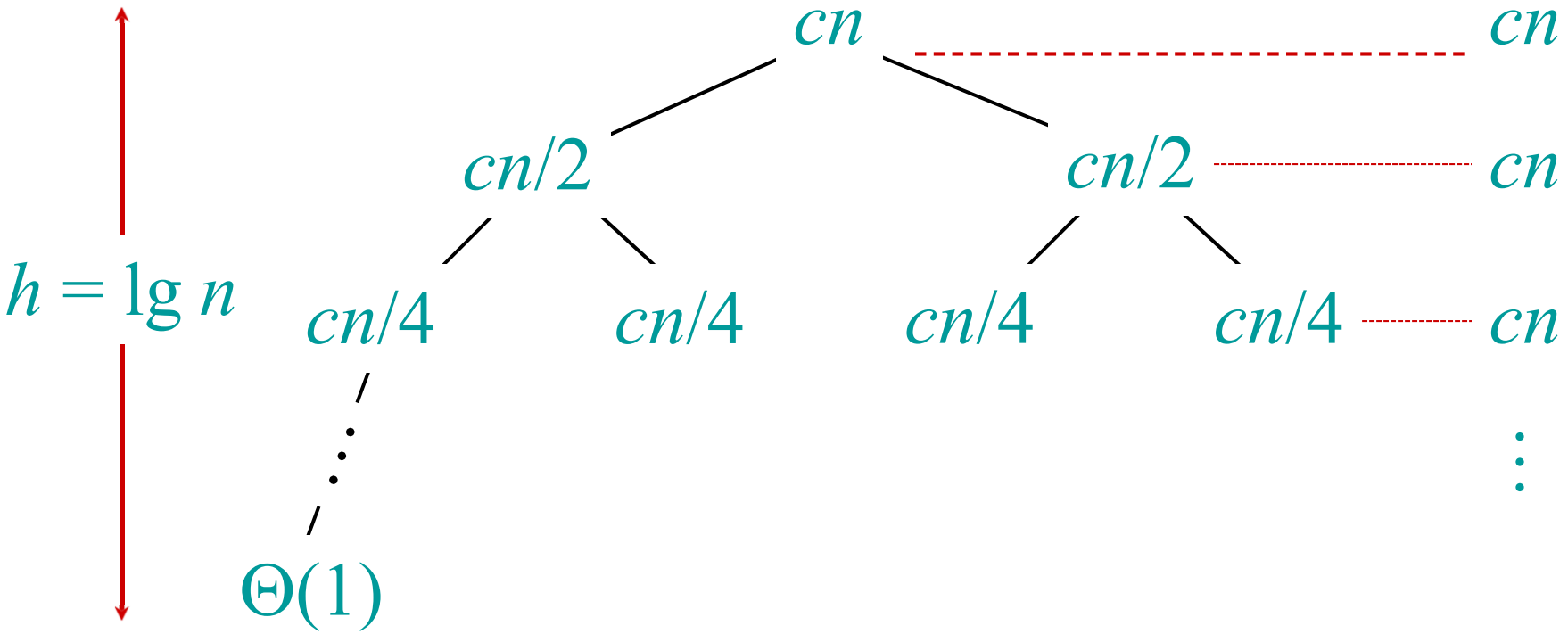
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



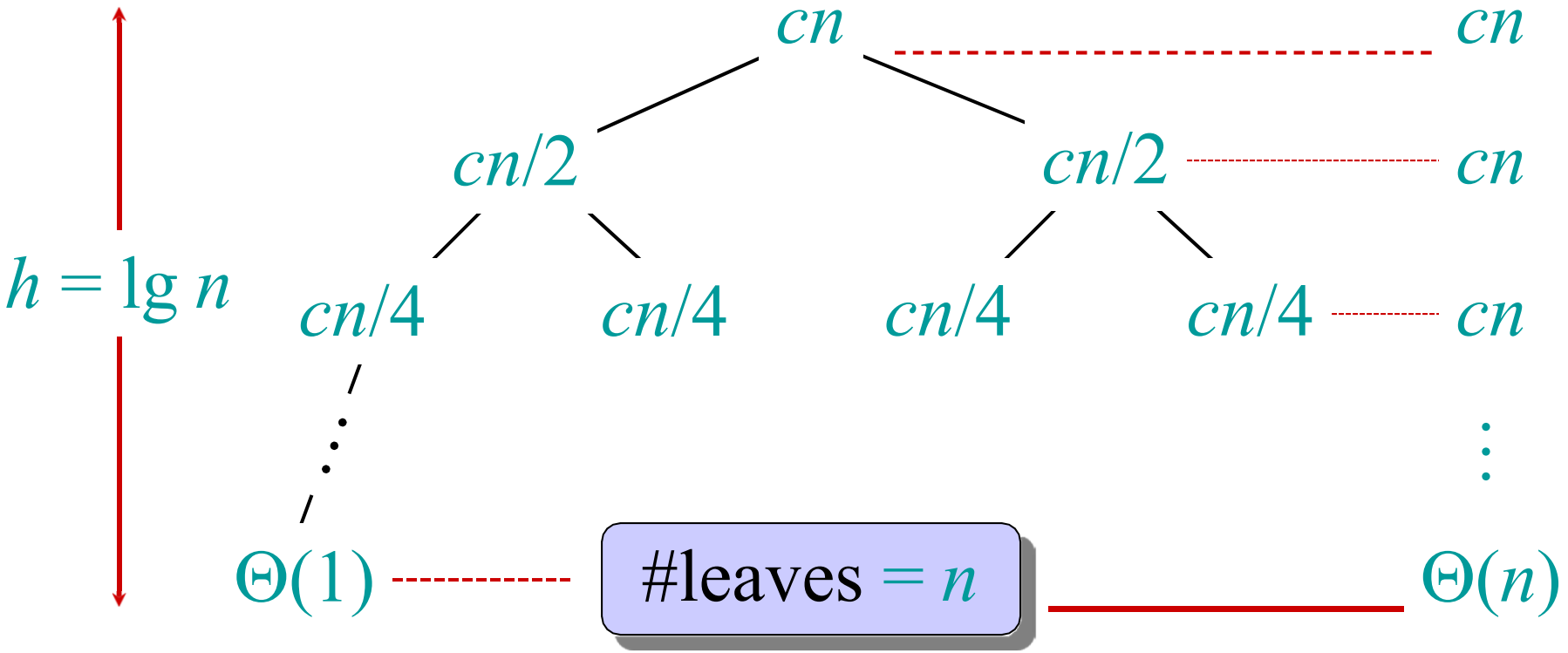
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



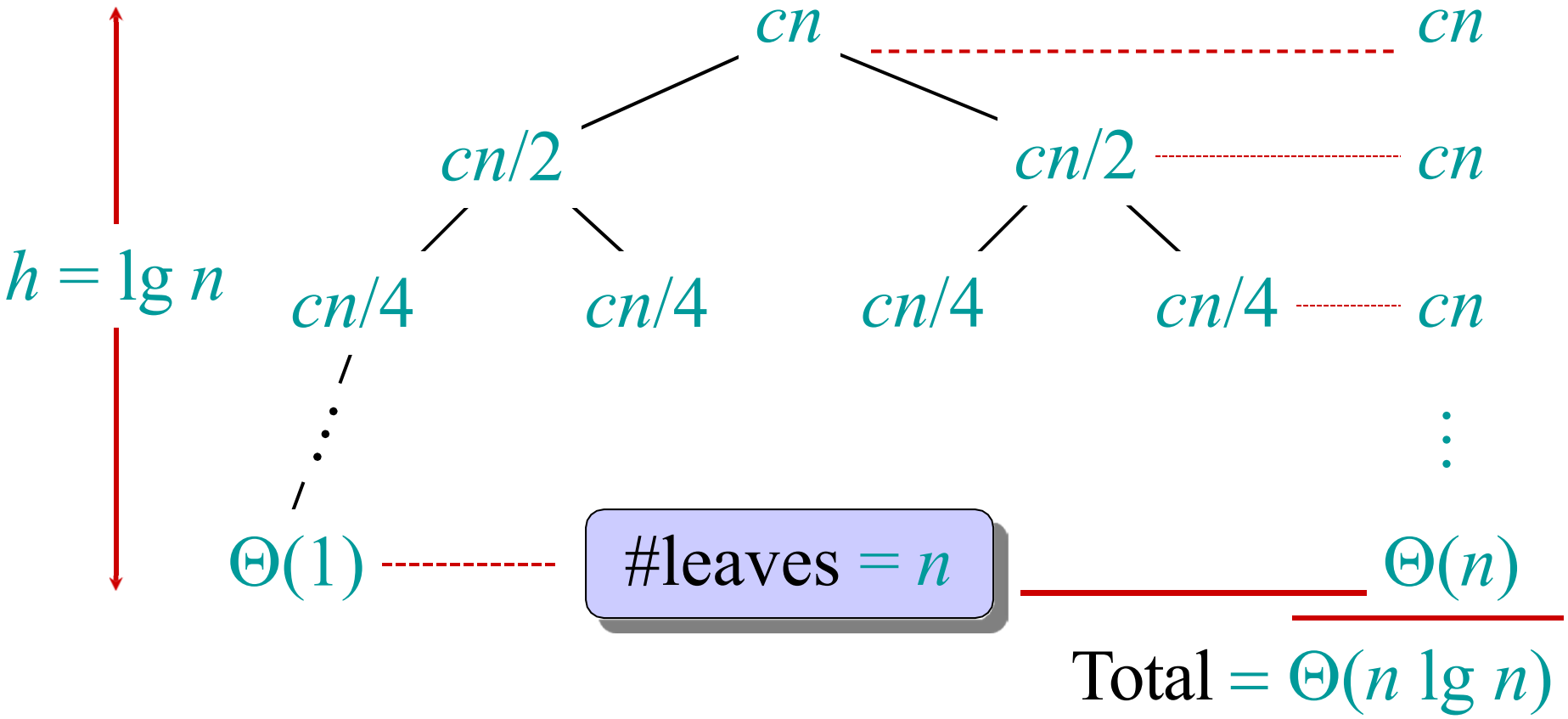
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



• **Analysis of Merge Sort Alg.**

• Continue expanding until the problem sizes get down to 1:

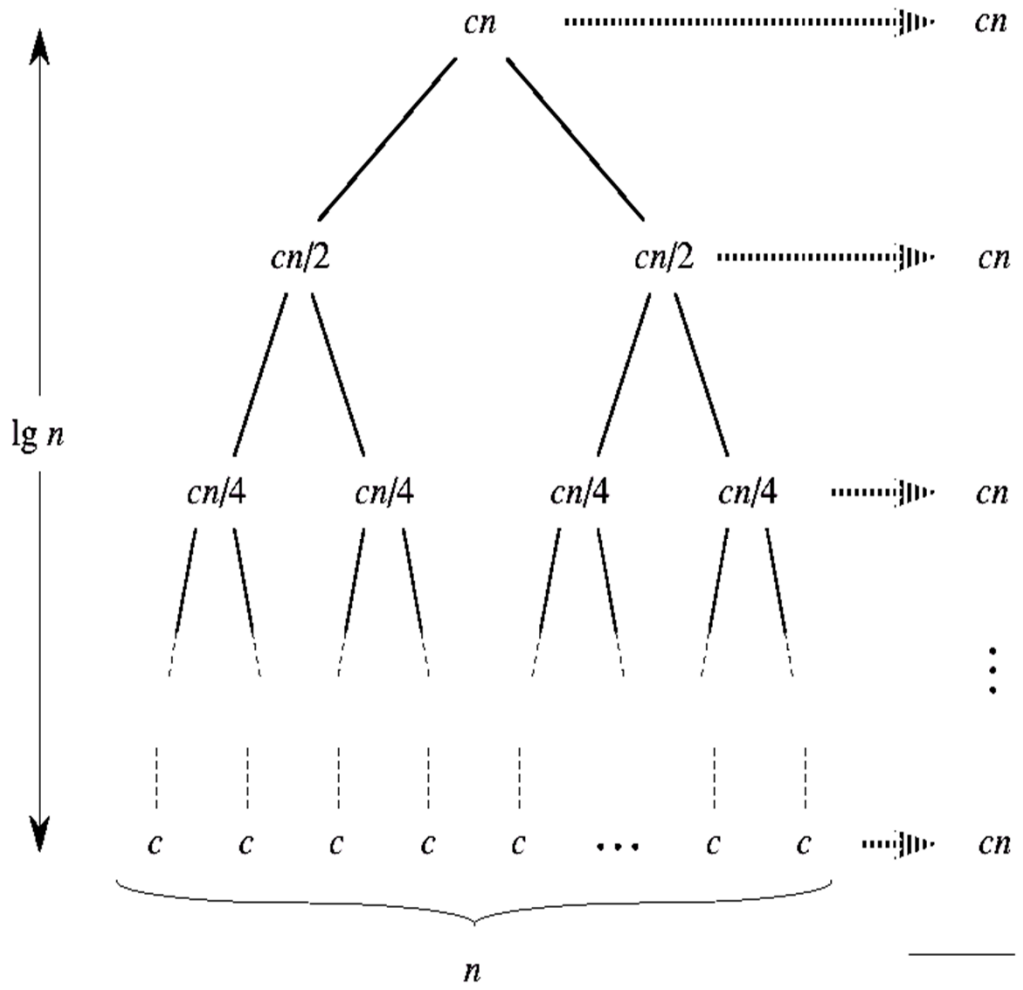
• **Each level has cost *cn***.

•The top level has cost *cn*.

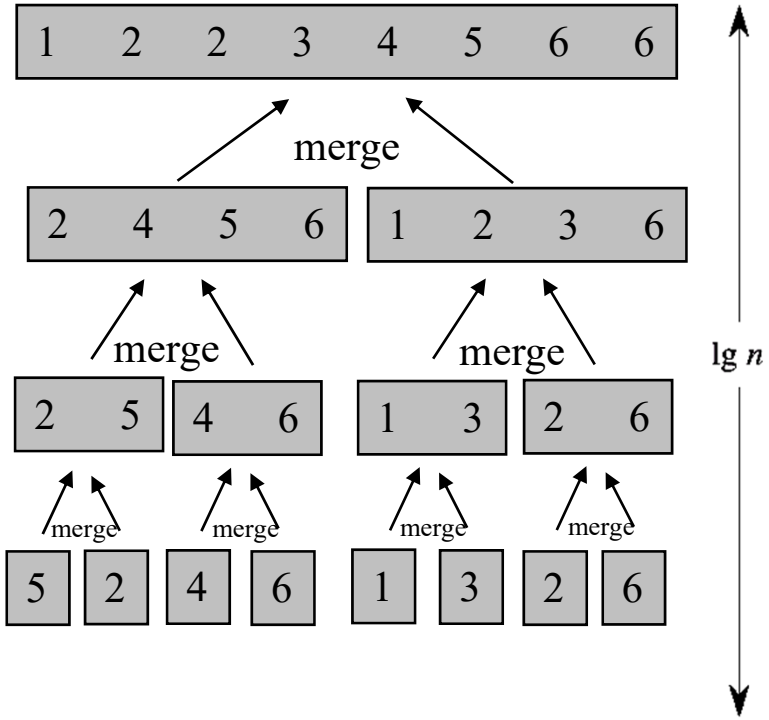
• The next level down has 2 subproblems, each contributing cost *cn*/2.

• The next level has 4 subproblems, each contributing cost *cn*/4.

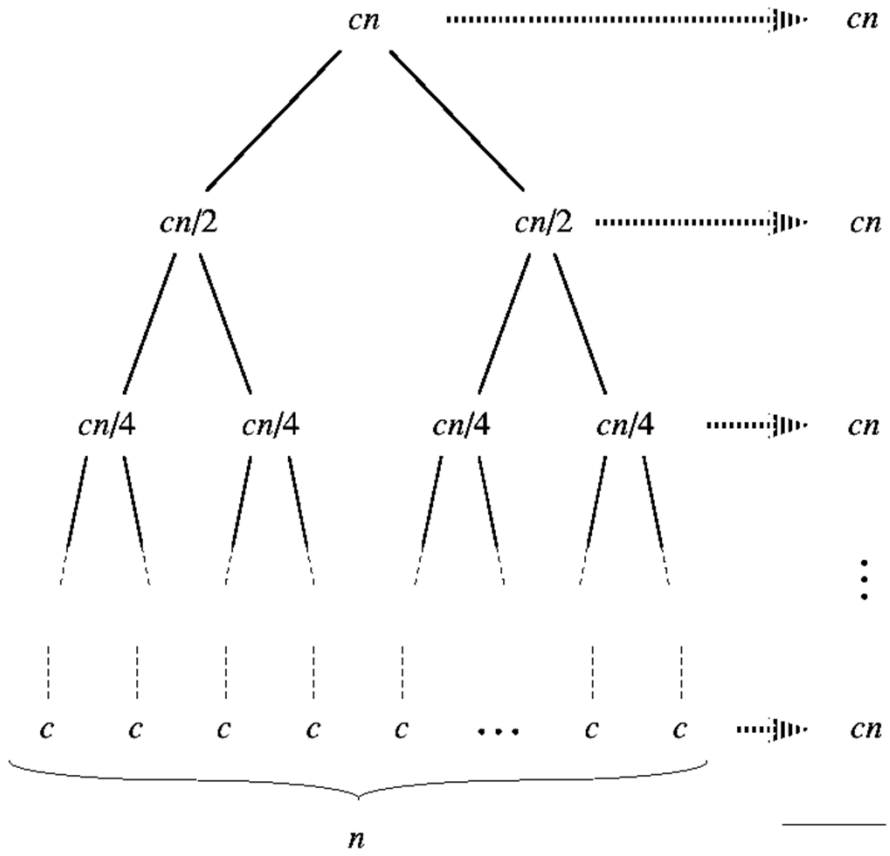
• Each time we go down one level, the number of subproblems doubles but the **cost per subproblem halves** cost per level stays the same.



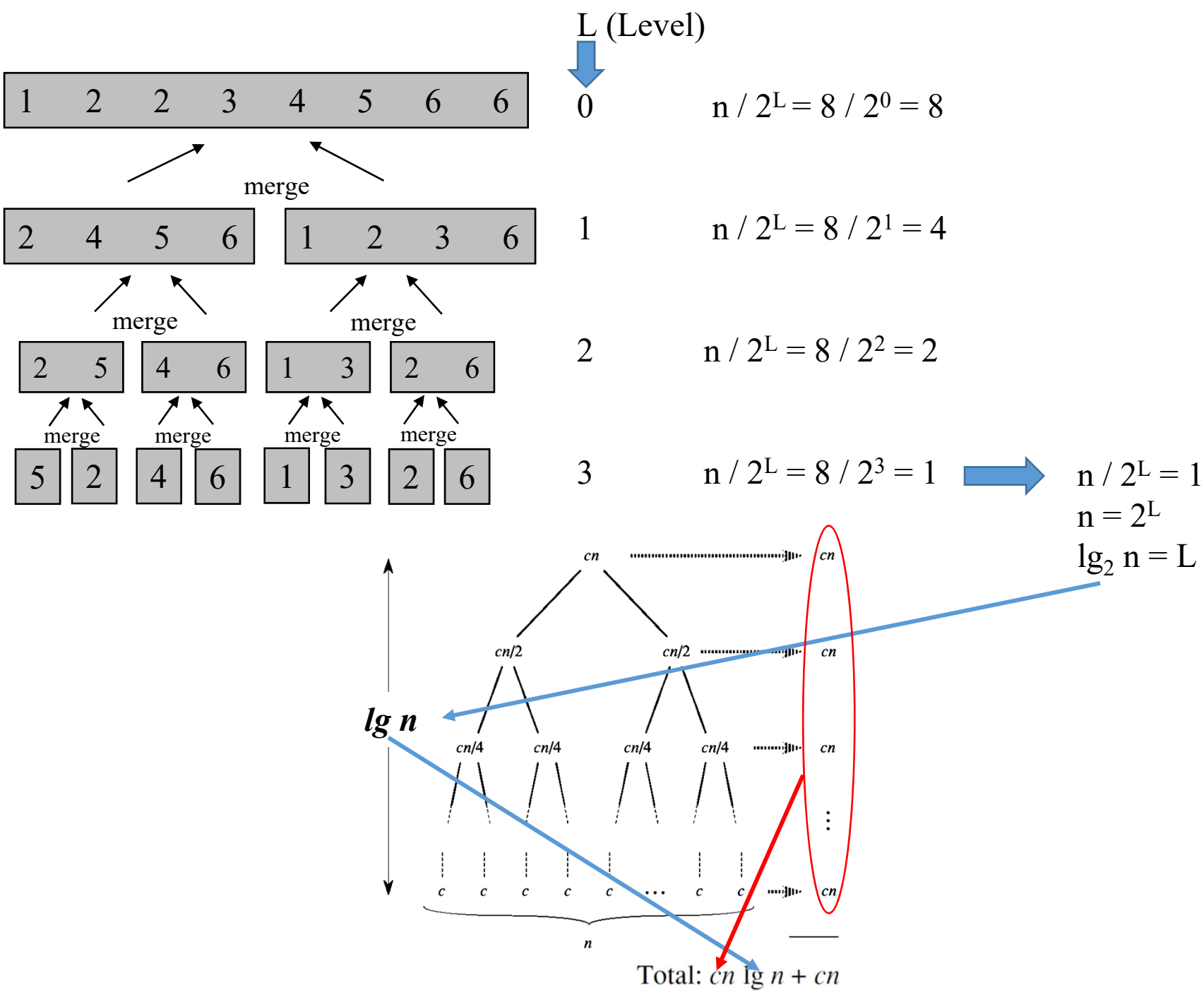
Total: $cn \lg n + cn$



lg *n*



Total: $cn \lg n + cn$


L (Level)
1 2 2 3 4 5 6 6
0
n / 2^L = 8 / 2^0 = 8
merge
2 4 5 6
1 2 3 6
1
n / 2^L = 8 / 2^1 = 4
merge
merge
merge
2 5
4 6
1 3
2 6
2
n / 2^L = 8 / 2^2 = 2
merge
merge
merge
merge
5 2
4 6
1 3
2 6
3
n / 2^L = 8 / 2^3 = 1
n / 2^L = 1
n = 2^L
lg2 n = L
cn
cn/2
cn/4
lg n
c
n
Total: cn lg n + cn

- **Complexity**

  - Algorithms can be classified as linear, quadratic, logarithmic, etc. depending on the time they take as a function of a number denoting the size of the problem.

  - The insertion sort algorithm (worst-case) was quadratic with a merge sort of $n \lg n$ complexity

  - The task is to compare the time or space complexity of different algorithms for large sizes of inputs

  - Assume that the time (or space) required for performing the algorithm is some function of n, which is the size of the input

  - The evaluation of the comparative performance of the two algorithms for large problem sizes can be calculated as follows:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

- **Complexity**

1) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

2) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

3) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C$

1) The limit is ∞. In this case, we say that *f(n)* grows asymptotically much faster than *g(n)* (or, equivalently, that *g(n)* grows much slower than *f(n)*

2) The limit is 0. This is the opposite of the first case; *f(n)* grows much slower than *g(n)* (or, equivalently, that *g(n)* grows much faster)

3) The limit is some constant. In this case, both *f(n)* and *g(n)* grow at roughly the same rate (asymptotically)

- **Complexity**
  - Definition: Θ-Notation
    - For a given function $g(n)$ **the set of function** is denoted by $\Theta(g(n))$
    - $\Theta(g(n)) = \{ f(n)$: positive constants c_1, c_2, and n_0 exist such that

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

  - A function $f(n)$ belongs to the set $\Theta(g(n))$ if positive constants c_1 and c_2 exist such that it can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n.

  - Because $\Theta(g(n))$ is a set, we could write "$f(n) \in \Theta(g(n))$" to indicate that $f(n)$ is a member of $\Theta(g(n))$

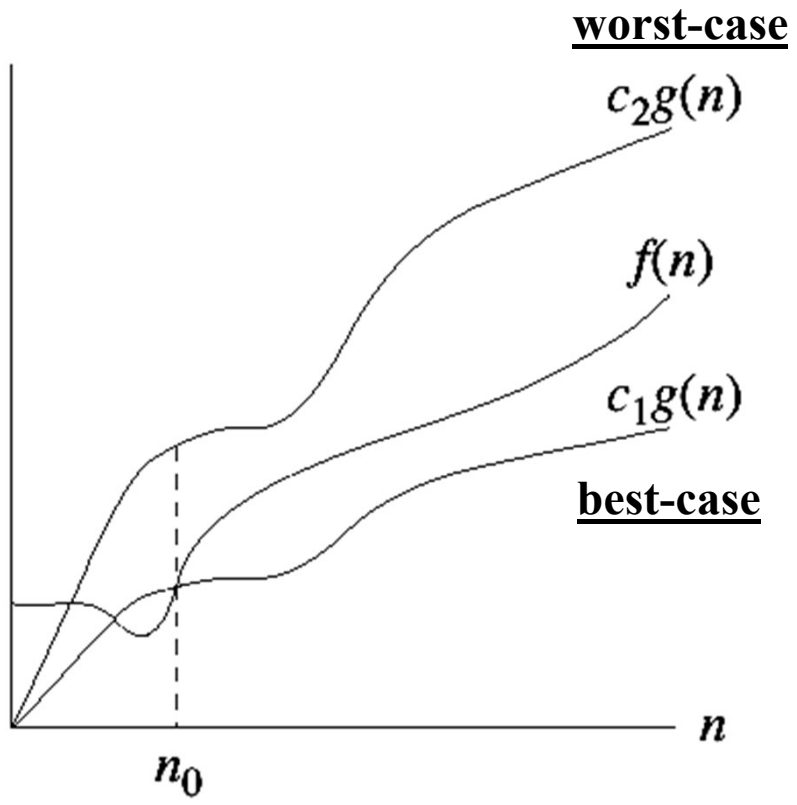
- **Complexity, Θ-Notation**

  - Instead, "*f(n)* = Θ(*g(n)*)" is used to express the same notion

  - The figure gives an intuitive diagram of functions *f(n)* and *g*(*n*) for *f(n)* = Θ(*g(n)*)"

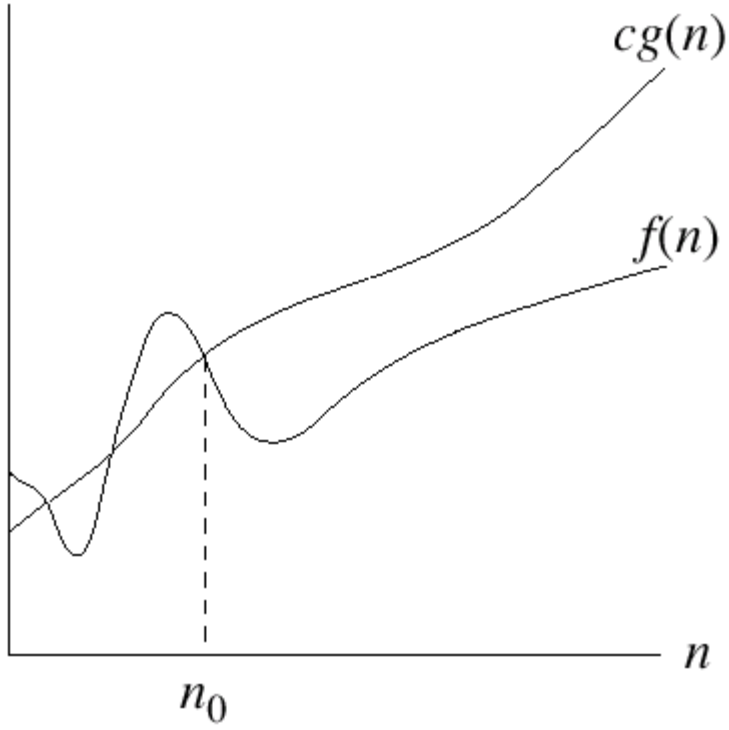
  - For all values of *n* to the right of n_0, the value of *f(n)* lies at or above c_1 *g(n)* and at or c_1 *g(n)*

  - In other words, for all $n \geq n_0$, the function *f(n)* is equal to *g(n)* to within a constant factor. The *g(n)* is an **asymptotically tight bound** for *f(n)*



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

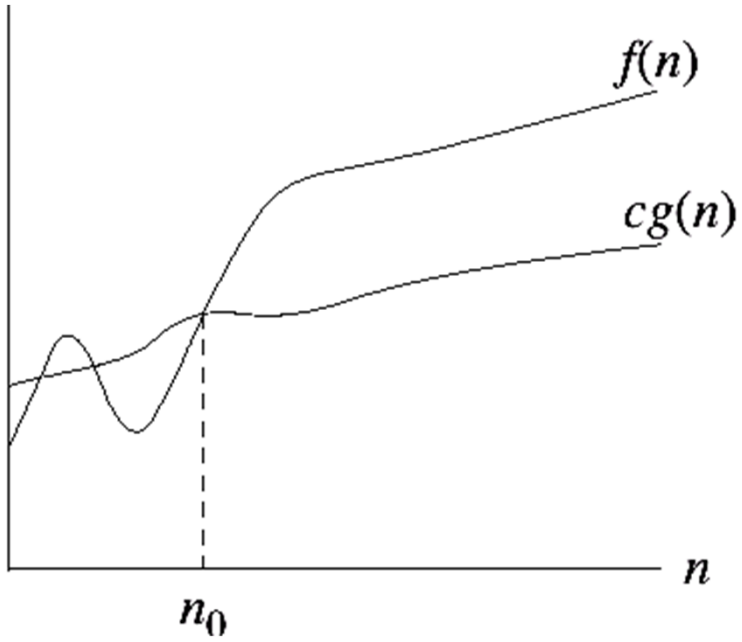
- **Complexity**
  - Definition: O-Notation
  - Remember that the Θ-notation asymptotically bounds a function from above & below
  - The O-Notation is an **asymptotic upper bound** as illustrated
  - For a given function $g(n)$ the set of function is denoted by **O(*g*(*n*))**



$O(g(n)) = \{ f(n)$: the positive constants c, and n_0 exist such that

$0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0 \}$

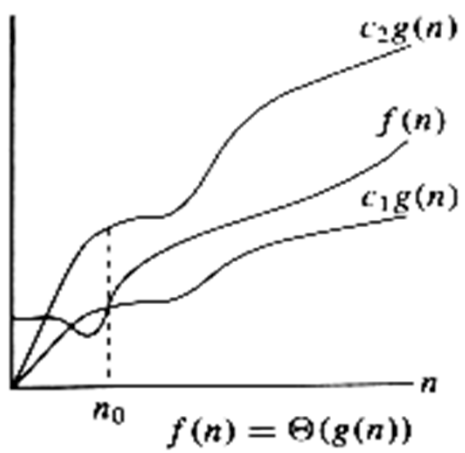
- **Complexity**
  - Definition: Ω-Notation
  - Remember that the O-notation provides an asymptotic upper bound on a function
  - The Ω-Notation is an **asymptotic lower bound** as illustrated below
  - For a given function g(*n*) **the set of function** is denoted by Ω(g(*n*))

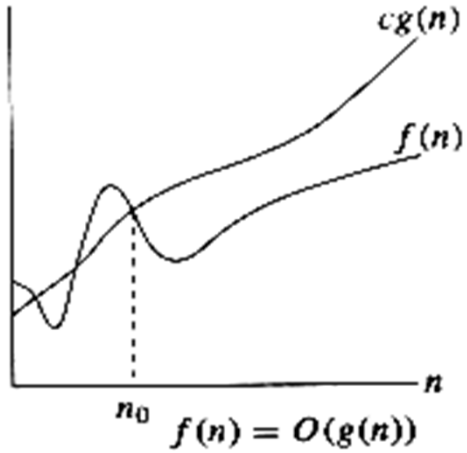


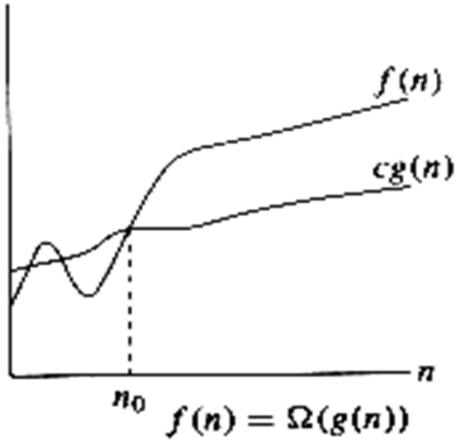
$\Omega(g(n)) = \{ f(n)$: the positive constants c, and n_0 exist such that

$0 \leq c\, g(n) \leq f(n)$ for all $n \geq n_0 \}$

- **Complexity**
  - Summary:







$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

$0 \le f(n) \le c\, g(n)$ for all $n \ge n_0$

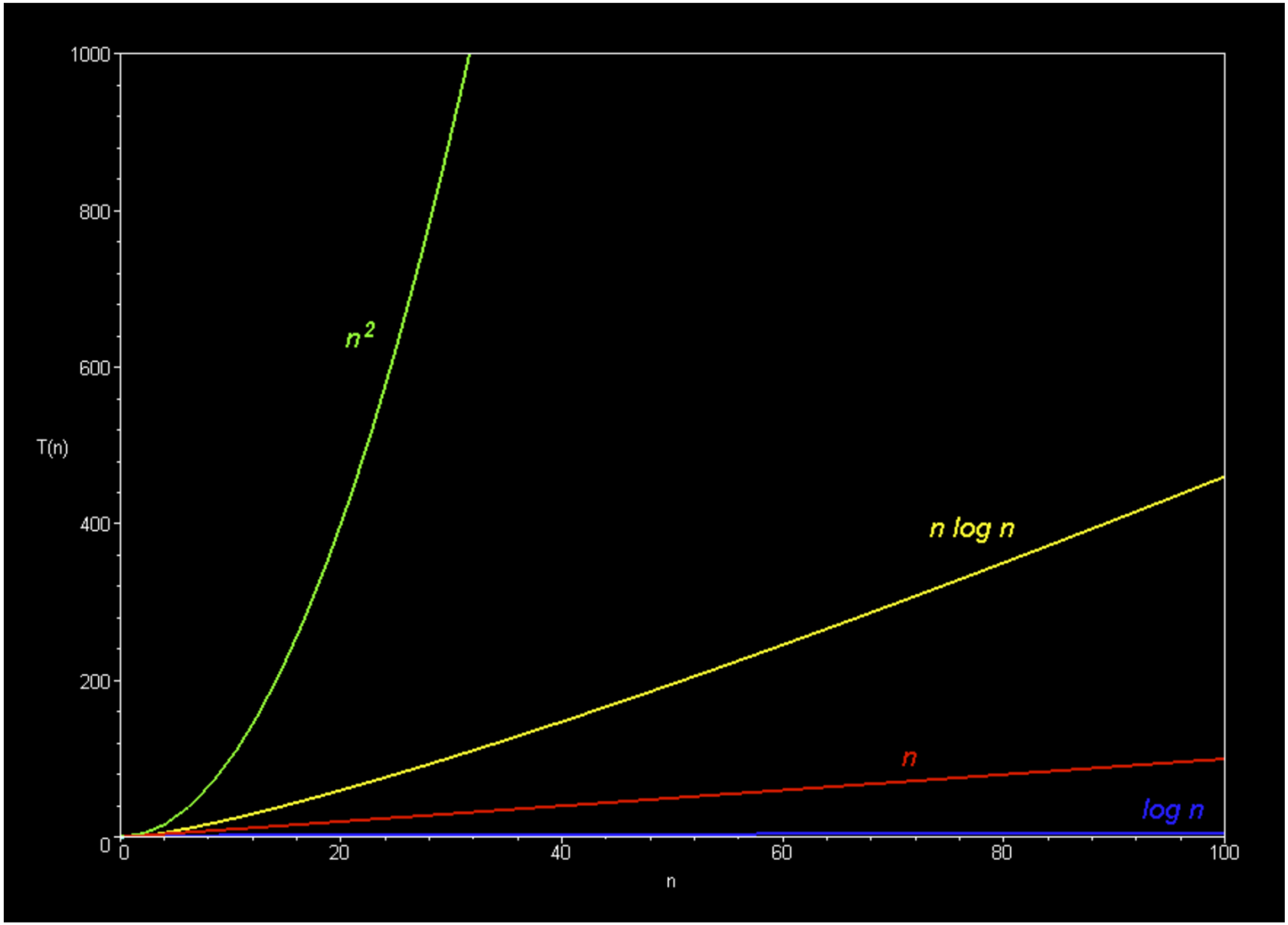
$0 \le c\, g(n) \le f(n)$ for all $n \ge n_0$

Algorithms & Data Structures

- Professor Reza Sedaghat

- COE428: Engineering Algorithms & Data Structures
- Email address: rsedagha@ee.ryerson.ca
- Course outline: www.ee.ryerson.ca/~courses/COE428/
- Course References:

  1) **Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. Introduction to Algorithms, MIT, 2002, ISBN: 0-07-013151-1 (McGraw-Hill) (Course Text)**


1000
800
600
400
200
0
T(n)
0
20
40
60
80
100
n
n 2
n log n
n
log n

Ordering by asymptotic growth rates

n^5
n
$n \log n$
n^2
n^n
2^n
$\log n$
3^{3n}
$\sqrt{n}$
C (Constant)

$$c \text{ (constant)} < \log n < \sqrt{n} < n < n \log n < n^2 < n^5 < 2^n < 3^{3n} < n^n$$

Rank the following functions by order of growth ?

$3 \log 2$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$\sqrt{n}$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	n^{54}
$\ln \ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$\log n$	e^n	$n \log n$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2\sqrt{2 \lg n}$	$\sqrt{4}$	2^n	$n \lg n$	$2^{2^{n+1}}$

Assignment

Calculate and assign a Θ-notation to each expression bellow.

Expression	Θ(?)
a) $2n^2+1$	$\Theta(n^2)$
$6n^3+12n^2+1$	
$2\lg n+4n+3n\lg n$	
$6n^6+n+4$	
$(6n+4)(1+\lg n)$	
$2+4+6+8+\ldots+2n$	
$\frac{(n+1)(n+3)}{n+2}$	
$\frac{(n^2+\lg n)(n+1)}{n+n^2}$	
$1+2+3+4+\ldots+n$	

Assignment

Let $f(n) = (n^3)$

Circle True or False for each statement below.

True or false: $f(n) = O(n^2)$
True or false: $f(n) = O(n^3 \, lgn^3)$
True or false: $f(n) = \Theta(2^{\lg n})$
True or false: $f(n) = \mathrm{O}(n^{1.5})$
True or false: $f(n) = \Omega(n^3)$
True or false: $f(n) = \Omega(n \, lgn^3)$
True or false: $f(n) = \Theta(n^2 \, lgn^2)$
True or false: $f(n) = O(2^{\lg n})$
True or false: $f(n) = \Omega(n^{1.5})$
True or false: $f(n) = O(n^3)$

- **Running time**

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

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

• we find that in the worst case, the running time of INSERTION-SORT is

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

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

• This worst-case running time can be expressed as $\boldsymbol{an^2 + bn + c}$ for constants $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ that again depend on the statement costs c_i; it is thus a quadratic function of n

2.3.2 Analysis of Merge Sort Algorithm

- How can we determine the time required to perform this algorithm?
  - Time to sort n numbers

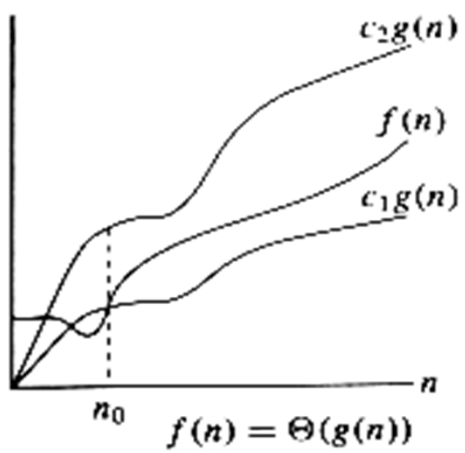
T(n) = Time to divide the number +	T_{divide}
Time to sort left side (size = $n/2$) +	$T(n/2)$
Time to sort right side (size = $n/2$) +	$T(n/2)$
Time to merge (total size = $n/2 + n/2 = n$)	$c_1 n$

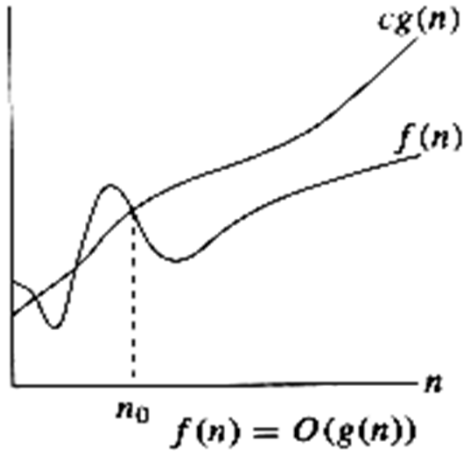
- Assumptions:
  - Let T(n) represent the time to sort n numbers
  - Let T_{divide} be the time to divide the numbers and assume $T_{divide} = 0$
  - Let $c_1 n + c_0$ be the time to merge n numbers, which is a linear algorithm and assume and $c_0 = 0$

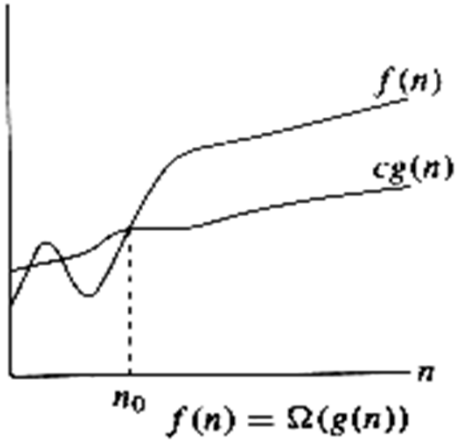
$$T(n) = T_{divide} + T(n/2) + T(n/2) + c_1 n + c_0$$

$$\mathbf{T(n) = 2T(n/2) + n}$$

- **Complexity**
  - Summary:







$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

$0 \le f(n) \le c\, g(n)$ for all $n \ge n_0$

$0 \le c\, g(n) \le f(n)$ for all $n \ge n_0$

Example

$f(n) \overset{!}{=} \Theta(g(n))$ $\quad 0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

$f(n) = 2n^2 + 3n + 1 \overset{!}{=} O(g(n))$ Determine c and n_0

To be proven: Is $g(n)$ is an asymptotically upper bound for $f(n)$

$$0 \le \boldsymbol{f(n) \le c\, g(n)} \text{ for all } n \ge n_0$$

$$0 \le 2n^2 + 3n + 1 \le cn^2$$

Lets assume $g(n)$ for highest order-term of $f(n)$

$$\boldsymbol{f(n)} \qquad \boldsymbol{g(n)}$$

$$2n^2 + 3n + 1 \le 2n^2 + 3n^2 + 1n^2$$

$$2n^2 + 3n + 1 \le 6n^2$$

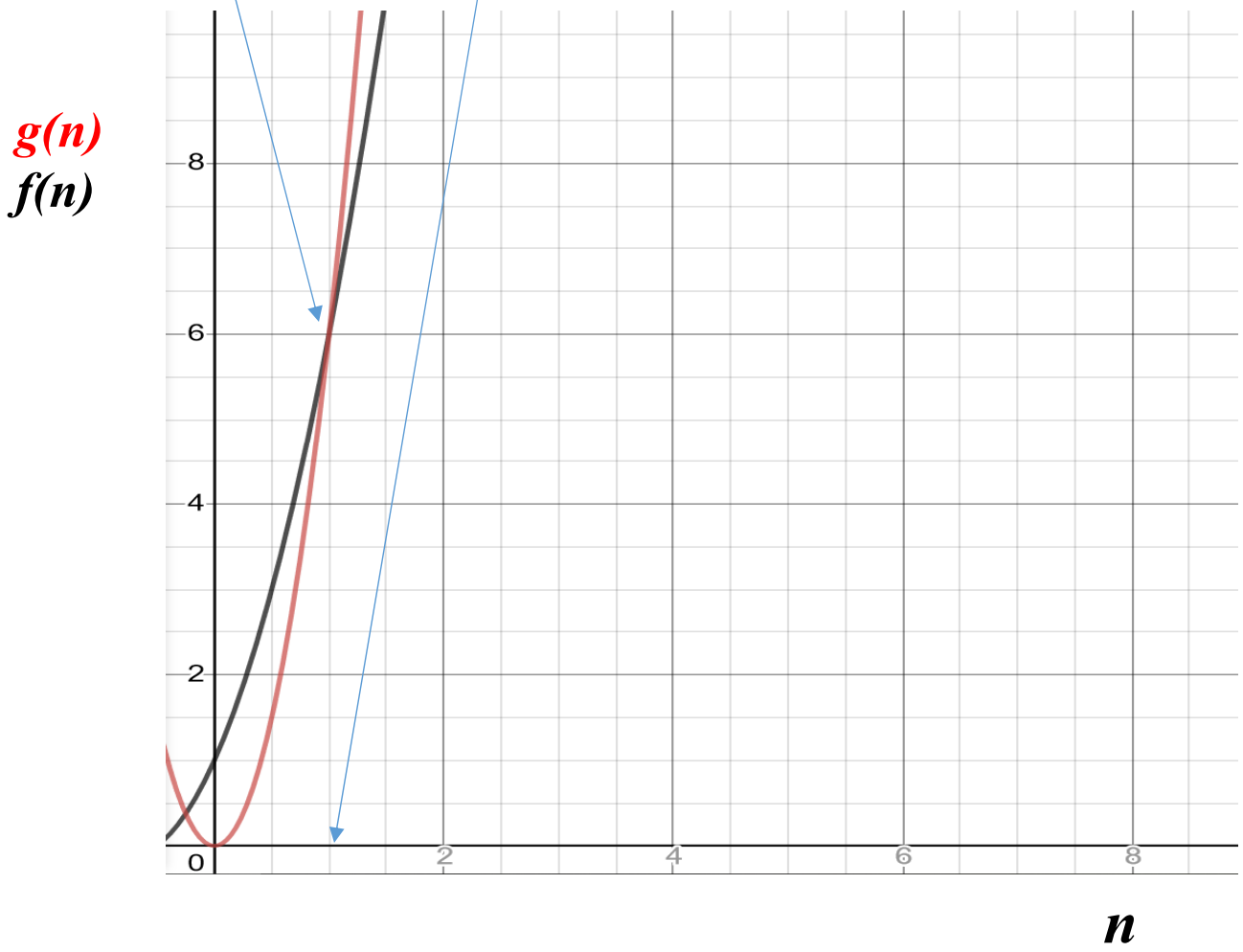
$$\boldsymbol{f(n) \le c\, g(n),}$$

For $c = 6$ *and* $n \ge n_0 = 1$ $\quad \boldsymbol{f(n) \le O(n^2)}$

Example from slide 86: $c\ (const) < \log n < \sqrt{n} < n < n \log n < n^2 < n^5 < 2^n < 3^{3n} < n^n$

Lower bound $\boldsymbol{\Omega}(g(n))$ — tight bound $\Theta(g(n))$ — upper bound $O(g(n))$

$c = 6$ *and* $n \geq n_0 = 1$ $\quad f(n) \leq O(n^2)$



Example

$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

$f(n) \overset{!}{=} \Theta(g(n))$

$f(n) = 2n^2 + 3n + 1 \overset{!}{=} \boldsymbol{\Omega}(g(n))$ Determine c and n_0

To be proven: Is $g(n)$ is an asymptotically upper bound for $f(n)$

$0 \le \boldsymbol{c\, g(n) \le f(n)}$ for all $n \ge n_0$

$0 \le cn^2 \le 2n^2 + 3n + 1$

Lets assume $g(n)$ for highest order-term of $f(n)$ *with its constant*

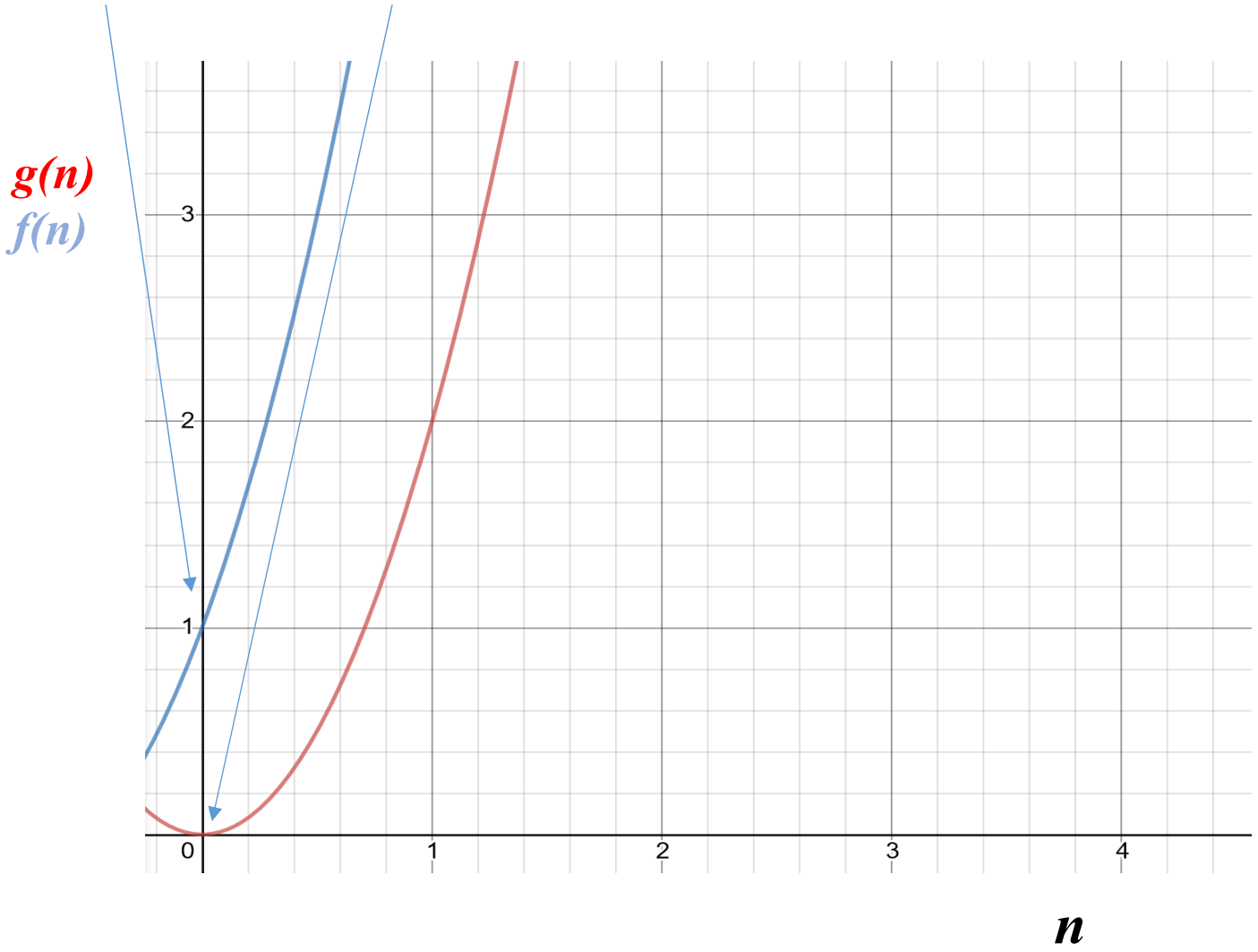
	g(n)	***f(n)***
	$2n^2 \le$	$2n^2 + 3n + 1$
c must be smaller than 2: $c < 2$	$1n^2 \le$	$2n^2 + 3n + 1$

$\boldsymbol{c\, g(n) \le f(n)}$

For $c \le 2$ *and* $n \ge n_0 > 0$ $\boldsymbol{f(n) \le \Omega(n^2)}$

Example from slide 86: $\underbrace{c\ (const) < \log n < \sqrt{n} < n < n \log n}_{\text{Lower bound } \boldsymbol{\Omega}(g(n))} \underbrace{< n^2 <}_{\text{tight bound } \Theta(g(n))} \underbrace{n^5 < 2^n < 3^{3n} < n^n}_{\text{upper bound } O(g(n))}$

$c \leq 1$ *and* $n \geq n_0 > 0$ $\quad f(n) \leq \Omega(n^2)$



Example

$$f(n) = \frac{1}{2}n^2 - 3n \overset{!}{=} \Theta(n^2)$$

- Determine c_1, c_2, and n_0
- To be proven: Is $g(n)$ is an asymptotically tight bound for $f(n)$

$$0 \le \boldsymbol{c_1 g(n) \le f(n) \le c_2 g(n)} \text{ for all } n \ge n_0$$

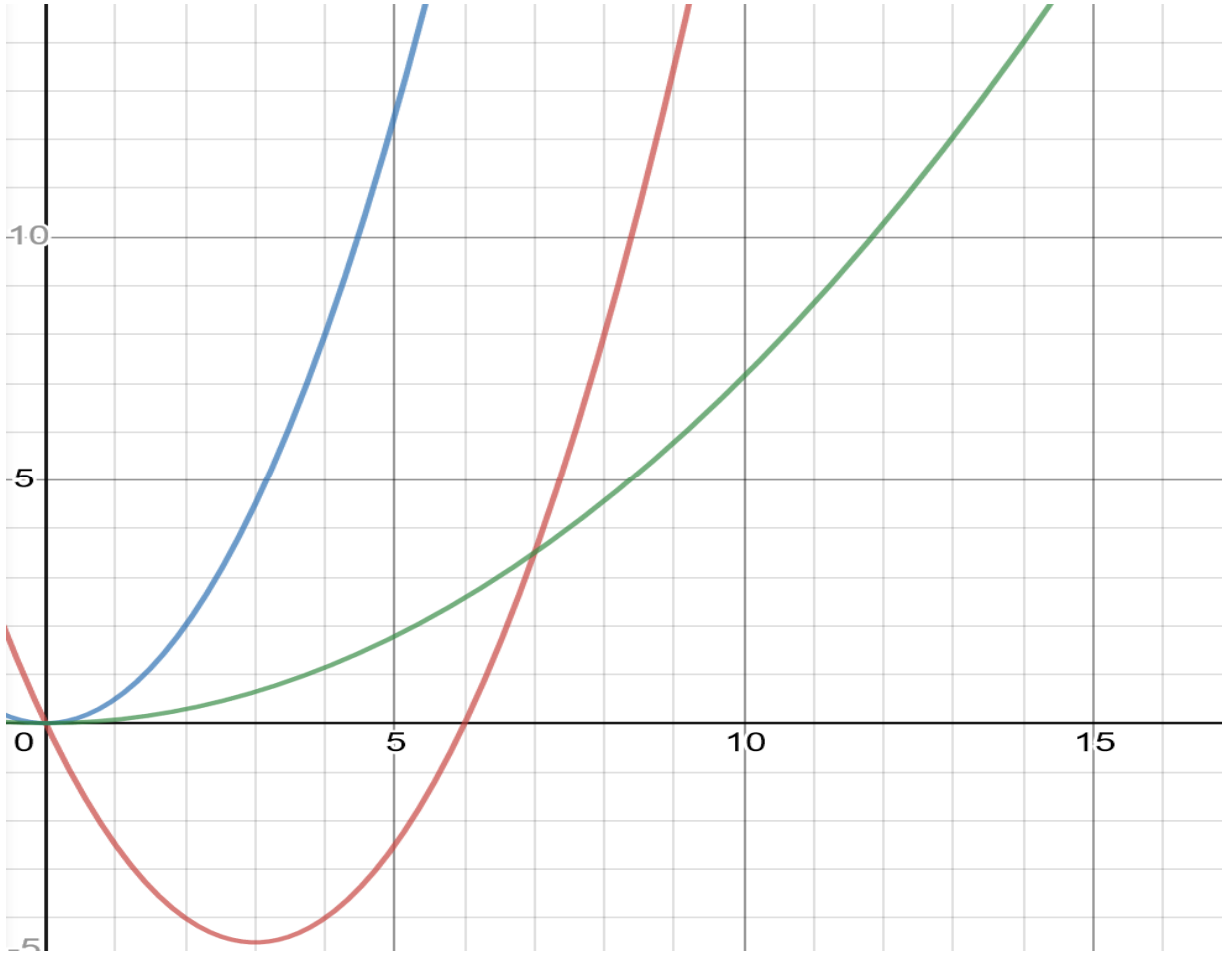
$$c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2$$

- Remember that in the function the **largest** component gives the running time, here we drop lower term 3n

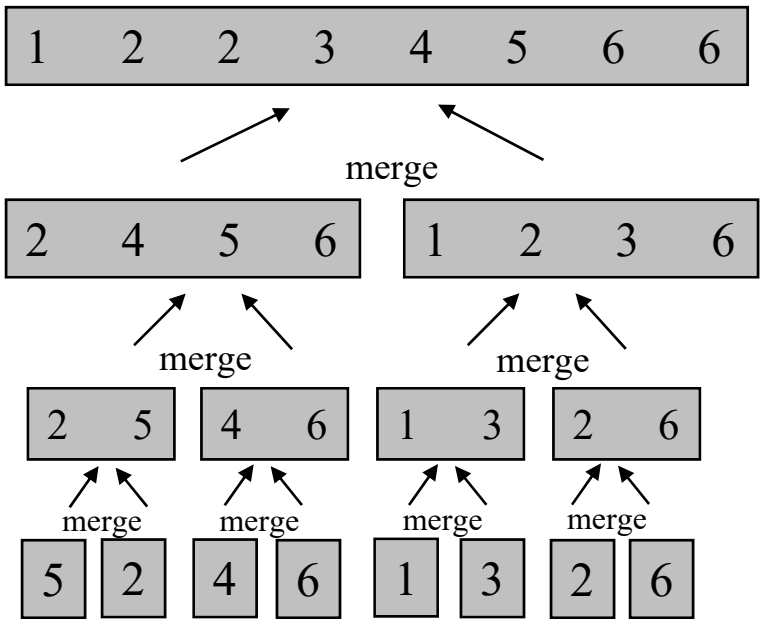
$\frac{1}{2}\boldsymbol{n^2} \le \boldsymbol{c_2 n^2}$, assuming if $n_0 \ge 1$ on the right side **(upper bound)** gives $c_2 \ge \frac{1}{2}$

- Dividing by n^2 gives $c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$
- On the left side, **lower bound**, $(c_1 \le \frac{1}{2} - \frac{3}{n})$ if $n = 6 \Rightarrow c_1 = 0$, which is invalid
- So, n must be $n_0 \ge 7$ and results in $c_1 \ge \frac{1}{14}$

Lower bound, $n_0 \geq 7$, in $c_1 \geq 1/14$



Merge Sort



L (Level)

0 $n / 2^L = 8 / 2^0 = 8$

1 $n / 2^L = 8 / 2^1 = 4$

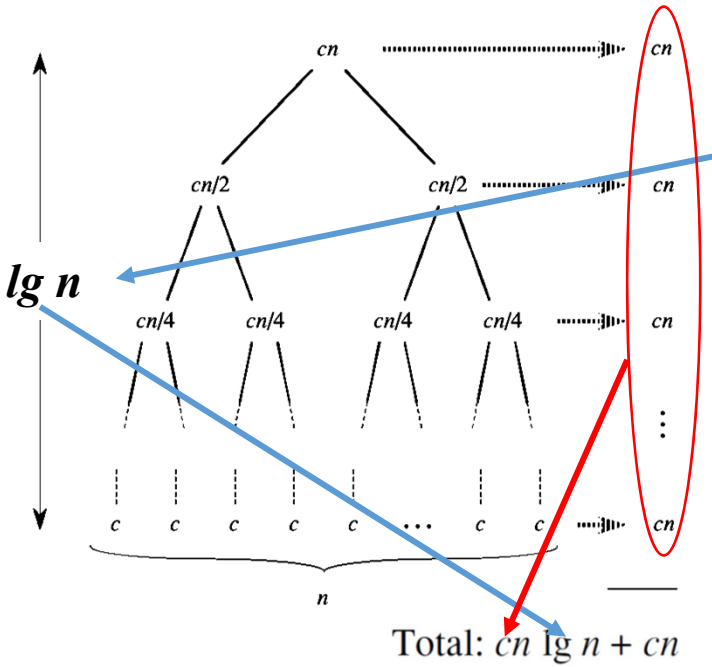
2 $n / 2^L = 8 / 2^2 = 2$

3 $n / 2^L = 8 / 2^3 = 1$

$n / 2^L = 1$

$n = 2^L$

$\lg_2 n = L$



Complexity: Recurrence

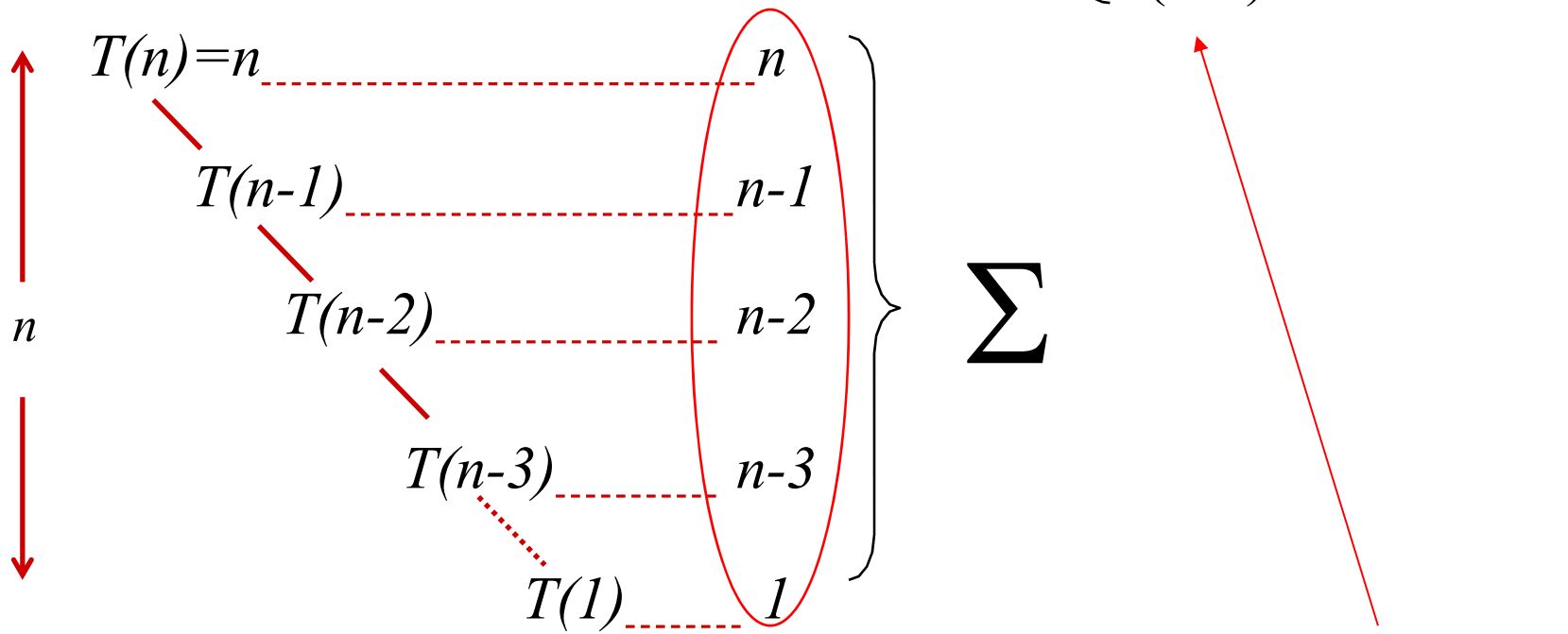
- Recursion Tree Method

- Substitution Method

Complexity: Recurrence

- Recursion Tree Method

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ T(n\text{-}1) + n \text{ if } n > 1. \end{cases}$$



$$Total = n+(n-1)+(n-2)+(n-3)+ \ldots\ldots +1 = \frac{n(n+1)}{2}$$

The growth of $T(n)$ is the higher growing term

$$\frac{n(n+1)}{2} = n^2 \Rightarrow \Theta(n^2)$$

Complexity: Recurrence

- Substitution Method

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ T(n-1) + n \text{ if } n > 1. \end{cases}$$

$T(n)=T(n-1)+n$ Then $T(n-1)=T(n-2)+(n-1)$

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

$T(n-3)=T(n-4)+(n-3)$

and so on ……

note: don't add n.

Substitute T(n-1): $T(n)=\left[T(n-2)+(n-1)\right]+n=T(n-2)+(n-1)+n$ *Generate a sequence*

Substitute T(n-2): $T(n)=\left[T(n-3)+(n-2)\right]+(n-1)+n=T(n-3)+(n-2)+(n-1)+n$

If it is developed for m times then $T(n)=T(n-m)+(n-(m-1))+(n-(m-2))+\ldots+(n-1)+n$

If n-m=0 then n=m, $T(n)=T(n-n)+(n-n+1)+(n-n+2)+\ldots+(n-1)+n$

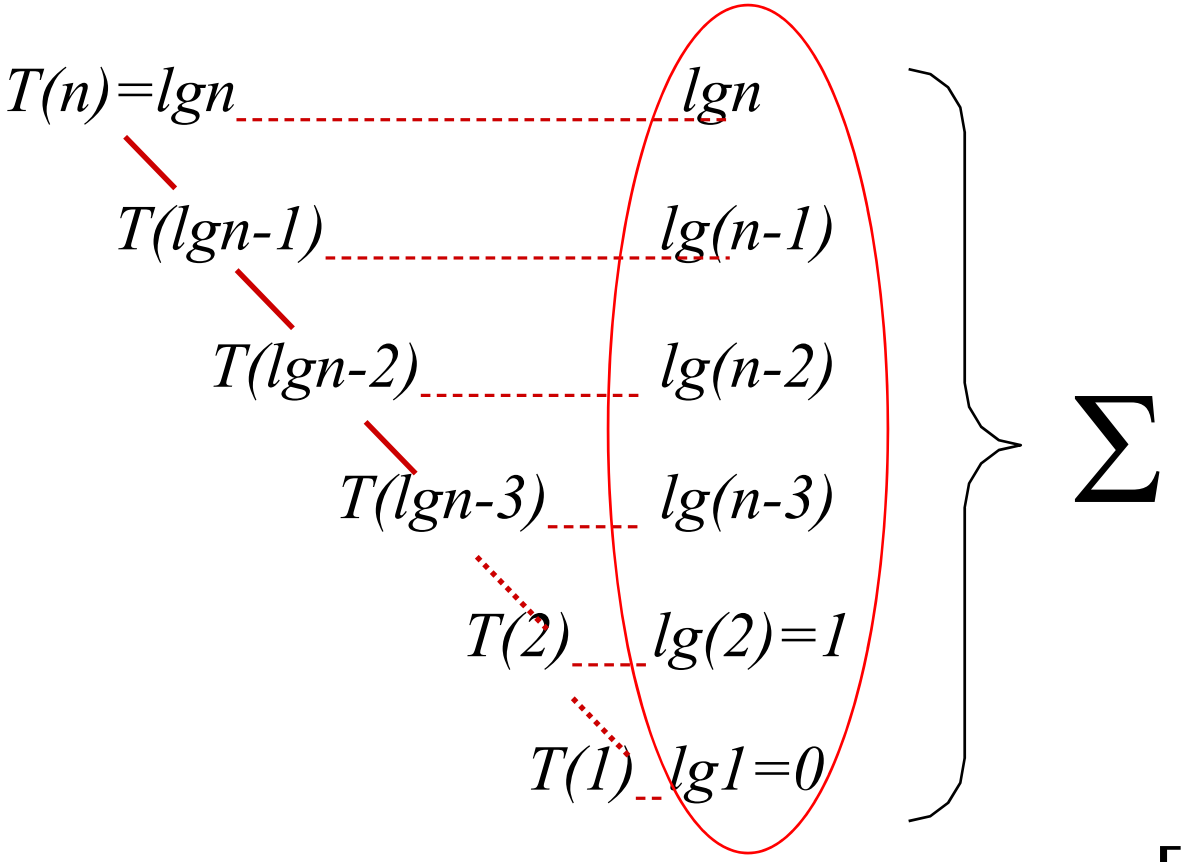
$$= T(0) + \underbrace{(1) + (2) + \ldots + (n-1) + n}$$

$\sum$ *is* $= 1 + \frac{n(n+1)}{2} = n^2 \Rightarrow \Theta(n^2)$

Complexity: Recurrence

- Recursion Tree Method

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ T(n\text{-}1) + \lg n \text{ if } n > 1. \end{cases}$$

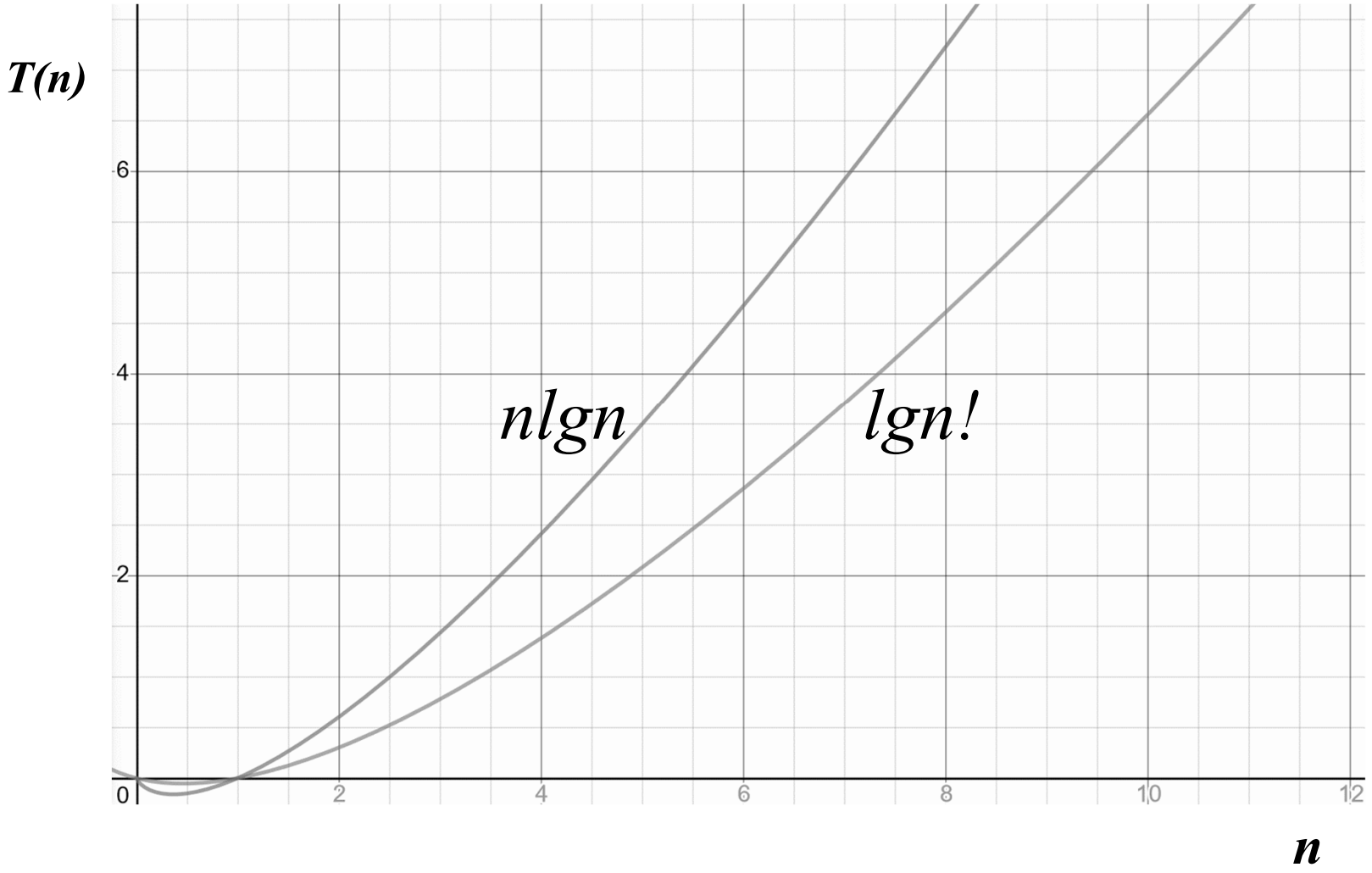


$$Total = lgn + lg(n-1) + lg(n-2) + \ldots + lg2 + lg1 = lg\left[n*(n-1)*(n-2)*\ldots 2*1\right]$$

$$= lg\ n!$$

$$= lgn! < nlgn \qquad O(nlgn)$$

Growth: lgn! < nlgn *O(nlgn)*



Complexity: Recurrence

- Substitution Method

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ T(n-1) + \lg n \text{ if } n > 1. \end{cases}$$

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

$T(n-2)=T(n-3)+lg(n-2)$

$T(n-3)=T(n-4)+lg(n-3)$

and so on ……

Substitute T(n-1): $T(n)=\left[T(n-2)+lg(n-1)\right]+lgn = T(n-2)+lg(n-1)+lgn$

Substitute T(n-2): $T(n)=\left[T(n-3)+lg(n-2)\right]+lg(n-1)+lgn = T(n-3)+lg(n-2)+lg(n-1)+lgn$

If it is developed for m times then;

$$T(n)=T(n-m)+lg(n-(m-1))+lg(n-(m-2))+\ldots+lg(n-1)+lgn$$

If n-m=0 then n=m;

$$=T(n-n)+lg(n-(n-1))+lg(n-(n-2))+\ldots+lg(n-1)+lgn$$

$$=T(n-n)+\underbrace{lg1 + lg2 +\ldots+lg(n-1)+lgn}$$

$$= T(0) + lg\left[n*(n-1)*(n-2)*\ldots 2*1\right]$$

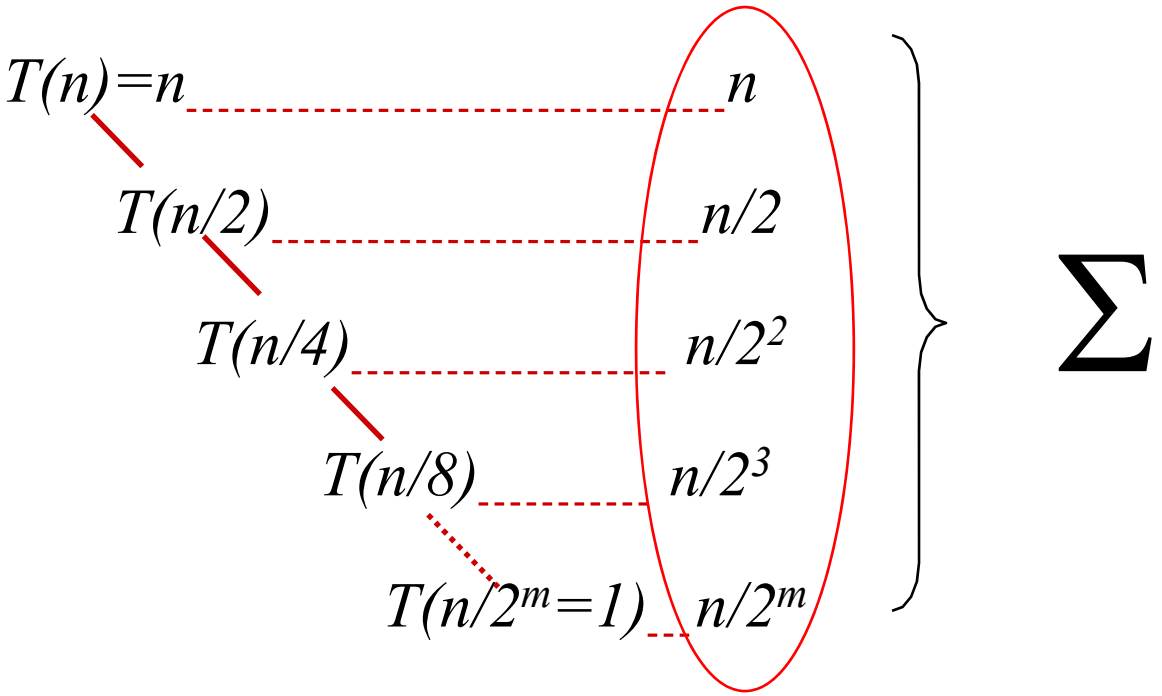
$$= 1 + lgn!$$

$lgn! < nlgn$ $O(nlgn)$

Complexity: Recurrence

- Recursion Tree Method

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



$$Total = n+(n/2)+(n/2^2)+(n/2^3)+ \ldots +(n/2^m)$$

$$= n\left[1+(1/2)+(1/2^2)+(1/2^3)+ \ldots +(1/2^m)\right] = n\sum_{i=0}^{m}\frac{1}{2^i}$$

$$= \Theta(n)$$

Complexity: Recurrence

- Substitution Method

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

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

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

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

and so on ……

Substitute T(n/2): $T(n)=\left[T(n/2^2)+n/2\right]+n = T(n/2^2)+n/2+n$

Substitute $T(n/2^2)$*:* $T(n)=\left[T(n/2^3)+n/2^2\right]+n/2+n=T(n/2^3)+n/2^2+n/2+n$

If it is developed for m times then;

$$T(n)= T(n/2^m)+ n/2^{m-1}+n/2^{m-2}+....+n/2+n$$

If $n/2^m=1$ *then* $n=2^m$ *and* $m=\lg n$

$$= 1 + n\left[1/2^{m-1}+1/2^{m-2}+....+1/2+1\right]$$

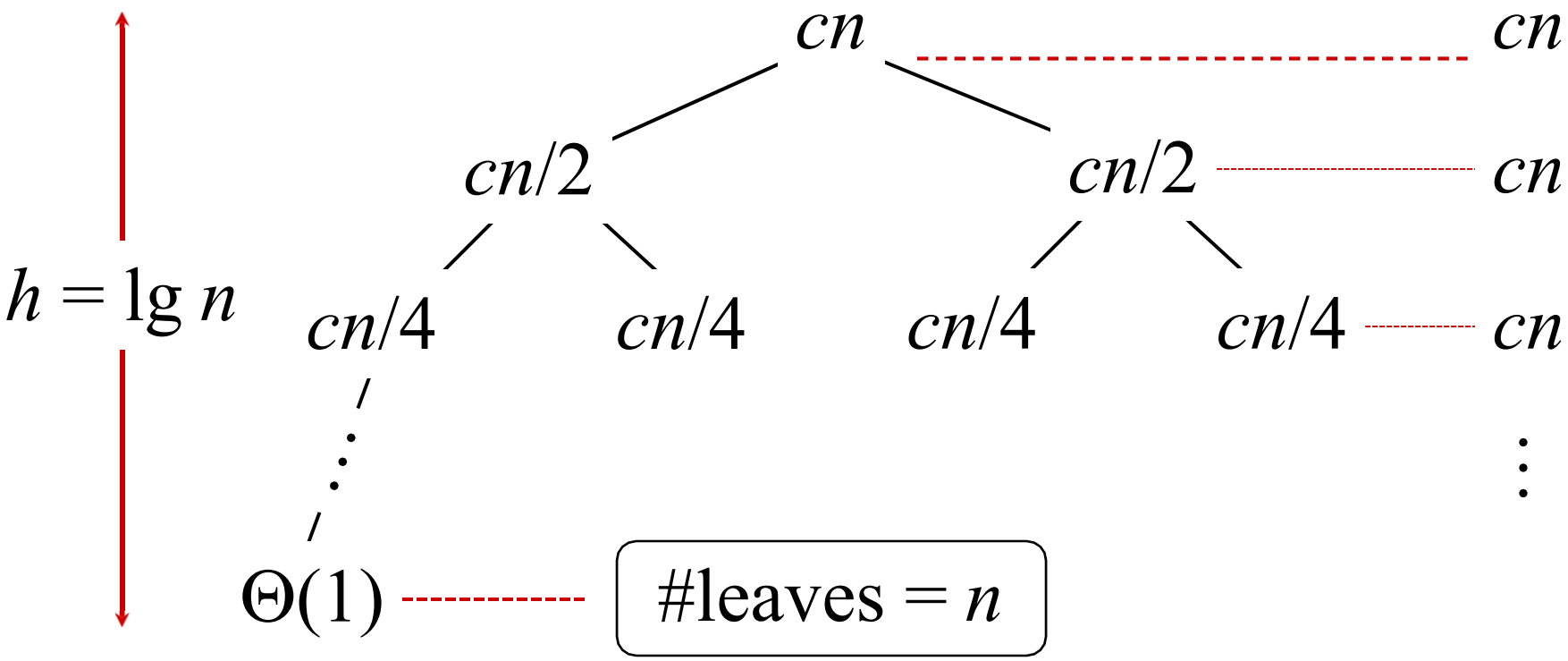
$$= 1 + n\left[\quad 1 \quad +1\right]$$

$$= 1 + 2n$$

$$= O(n)$$

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Total = $\Theta(\boldsymbol{n \lg n})$

$T(n) = 2T(n/2) + \Theta(n)$, Guess: $T(n) \leq cn \lg n$

Upper bound:

$$T(n) \leq 2T(n/2) + n$$

$$= 2\left(c\frac{n}{2}\lg\frac{n}{2}\right) + n$$

$$= 2\left(c\frac{n}{2}\lg\frac{n}{2}\right) + n$$

$$= cn\lg n - cn\lg 2 + n \quad (\lg 2 = 1)$$

$$\leq cn\lg n - cn + n$$

$$-cn + n \leq 0 \qquad \text{if } n = n_0 = 1$$

$$-c + 1 \leq 0$$

$$1 \leq c$$

$T(n) = 2T(n/2) + \Theta(n)$, Guess: $T(n) \geq cn \lg n$

Lower bound:

$T(n) \geq 2T(n/2) + n$

$= 2\left(c\frac{n}{2}\lg\frac{n}{2}\right) + n$

$= 2\left(c\frac{n}{2}\lg\frac{n}{2}\right) + n$

$= cn \lg n - cn \lg 2 + n$ ($\lg 2 = 1$)

$\geq cn \lg n - cn + n$

$-cn + n \geq 0$ if $n = n_0 = 1$

$-c + 1 \geq 0$

$1 \geq c > 0$

$T(n) = 2T(n/2) + \Theta(n)$, Guess: $T(n) \le cn \lg n$

Upper bound:

$$T(n) \le 2T(n/2) + c_2 n$$

$$= 2\left(c\frac{n}{2}\lg\frac{n}{2}\right) + c_2 n$$

$$= 2\left(c\frac{n}{2}\lg\frac{n}{2}\right) + c_2 n$$

$$= cn\lg n - cn\lg 2 + c_2 n \quad (\lg 2 = 1)$$

$$\le cn\lg n - cn + c_2 n$$

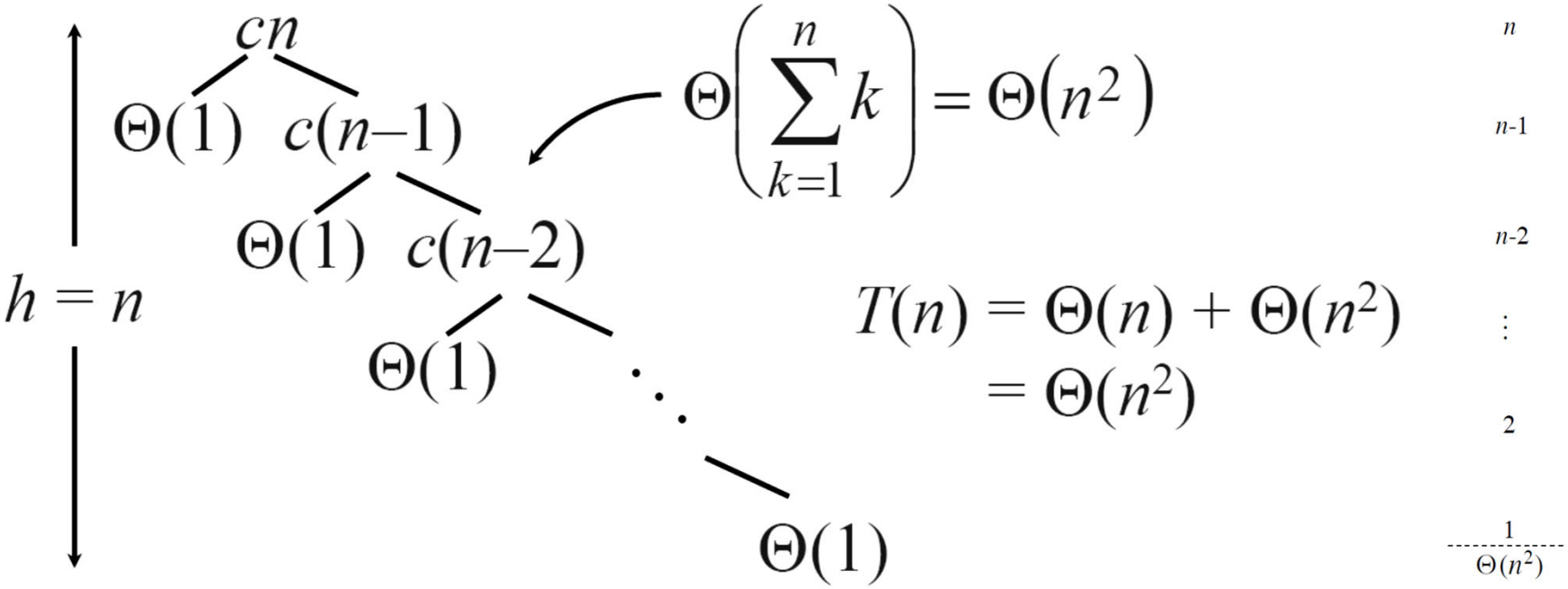
$$-cn + c_2 n \le 0$$

$$c_2 n \le cn \quad \text{if } n = n_0 = 1$$

$$c_2 \le c$$

Assignment, Recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$$

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

$T(n) = T(n-1) + \Theta(n)$, Guess: $T(n) \leq n^2$

Upper bound:

$T(n) \leq T(n-1) + n$

$= c\,(n-1)^2 + n$

$\leq cn^2 - 2cn + c + n$

$-2cn + c + n \leq 0$

$n(1-2c) + c \leq 0$ if $n=n_0=1$

$1-2c+c \leq 0$

$1 \leq c$

$T(n) = T(n-1) + \Theta(n)$, Guess: $T(n) \geq n^2$

lower bound:

$T(n) \geq T(n-1) + n$

$= c\,(n-1)^2 + n$

$\geq cn^2 - 2cn + c + n$

$-2cn + c + n \geq 0$

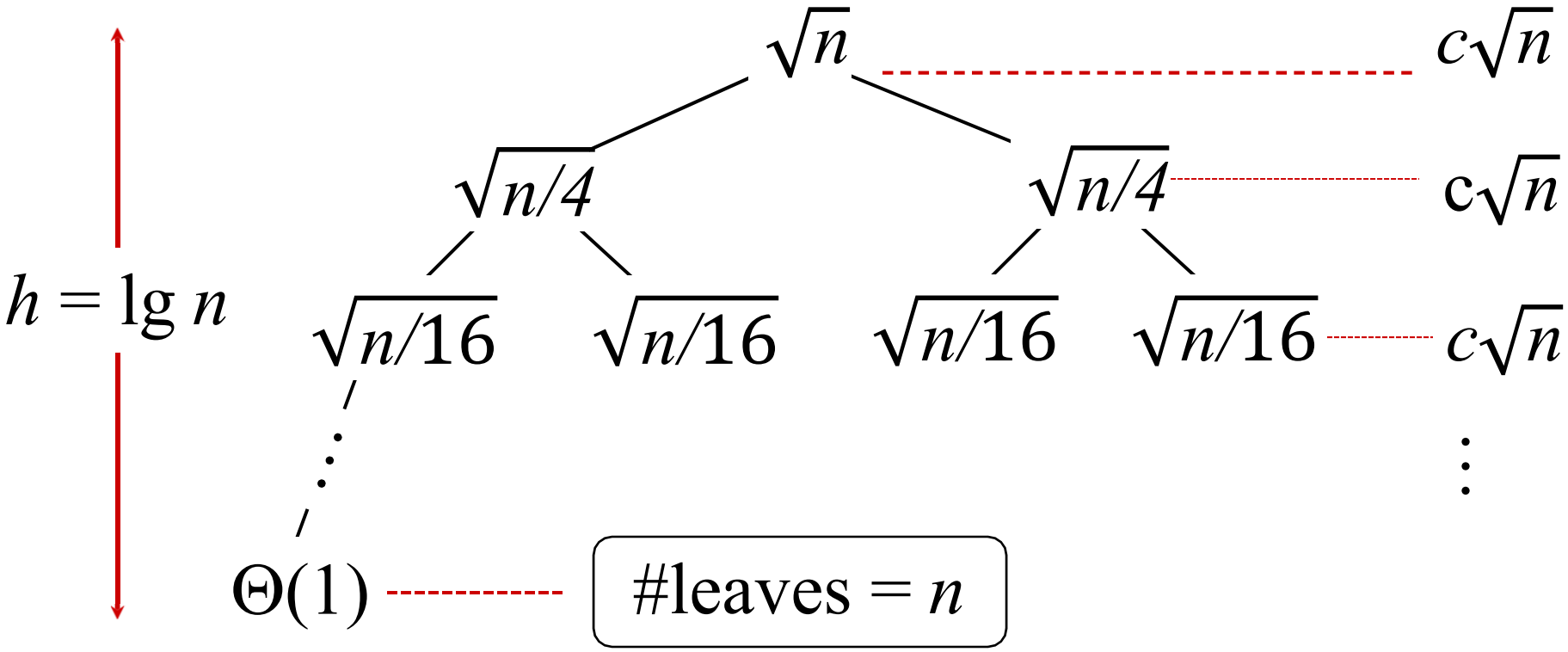
$n(1-2c) + c \geq 0$ if $n = n_0 = 1$

$1-2c+c \geq 0$

$1 \geq c > 0$

Recursion tree

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



Total = $\Theta(\sqrt{n} \lg n)$

$T(n) = 2T(n/4) + \sqrt{n}$, Guess: $T(n) \leq \sqrt{n} \lg n$

Upper bound:

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

$$= 2c\frac{\sqrt{n}}{2} \lg \frac{n}{4} + \sqrt{n}$$

$$= \cancel{2}\, c\frac{\sqrt{n}}{\cancel{2}} \lg n - c\sqrt{n} \lg 4 + \sqrt{n}$$

$$\leq c\sqrt{n} \lg n - c\sqrt{n} \lg 4 + \sqrt{n}$$

$$-c\sqrt{n} \lg 4 + \sqrt{n} \leq 0 \qquad \text{if } n = n_0 = 1$$

$$-c \lg 4 + 1 \leq 0$$

$$1 \leq c$$

$T(n) = 2T(n/4) + \sqrt{n}$, Guess: $T(n) \geq \sqrt{n} \lg n$

Lower bound:

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

$$= 2c\frac{\sqrt{n}}{2} \lg \frac{n}{4} + \sqrt{n}$$

$$= \cancel{2}\, c\frac{\sqrt{n}}{\cancel{2}} \lg n - c\sqrt{n} \lg 4 + \sqrt{n}$$

$$\geq c\sqrt{n} \lg n - c\sqrt{n} \lg 4 + \sqrt{n}$$

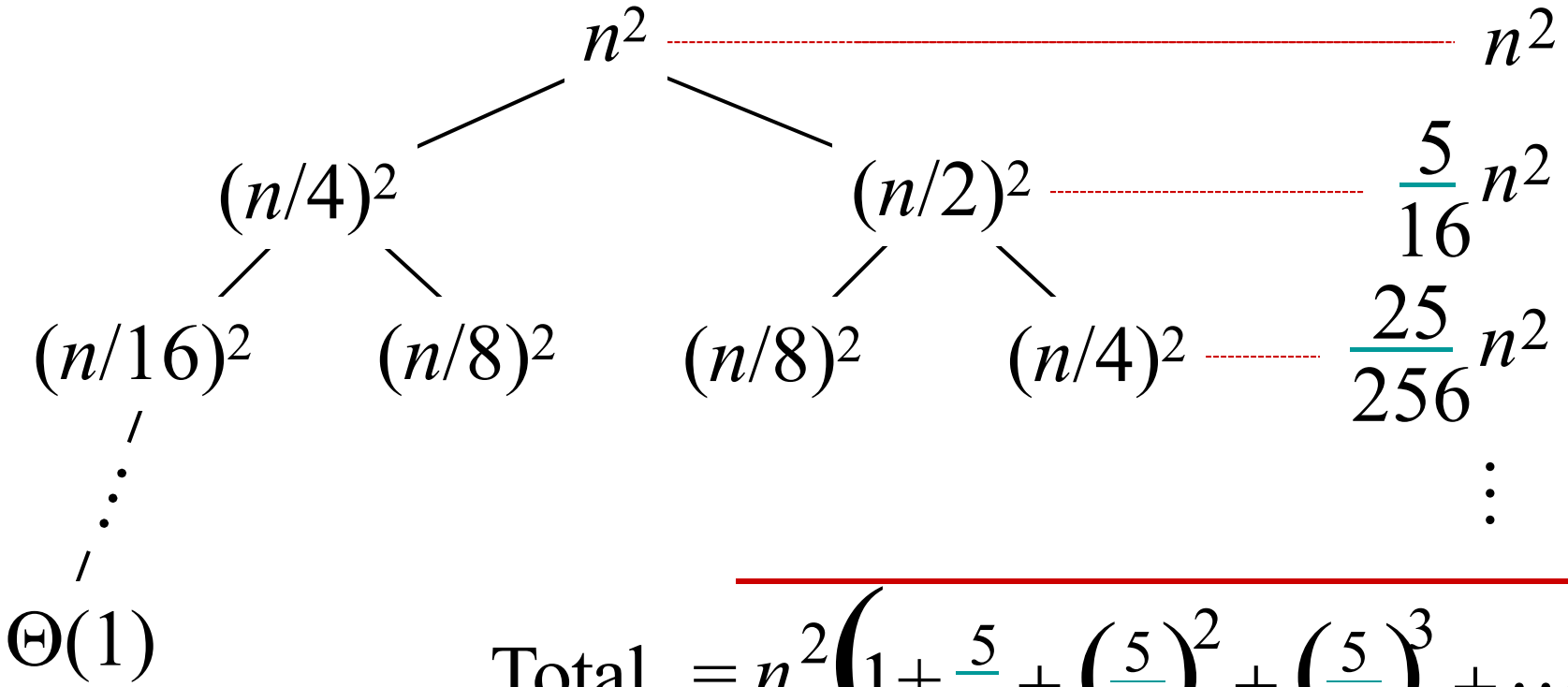
$$-c\sqrt{n} \lg 4 + \sqrt{n} \geq 0 \qquad \text{if } n = n_0 = 1$$

$$-c \lg 4 + 1 \geq 0$$

$$1 \geq c > 0$$

Assignment, Recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



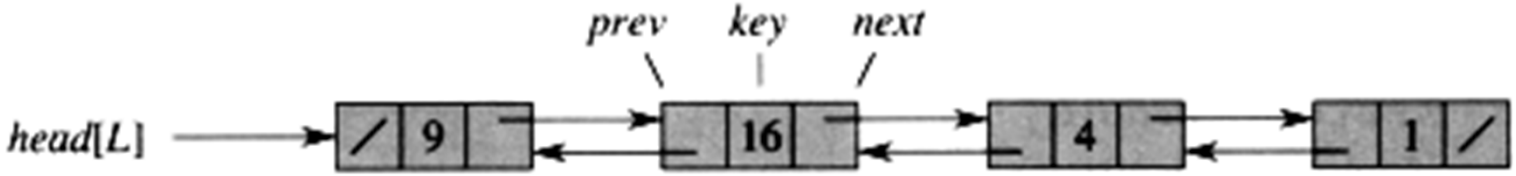
$$\text{Total} = n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)$$

$$= \Theta(n^2) \quad \textit{geometric series}$$

- **Data Structure, Memory and Pointers**

  - Operation of sets
    - Search (S, k)
    - Insert (S, x)
    - Delete (S, x)
    - Minimum (S)
    - Maximum (S)
    - Successor (S, x)
    - Predecessor (S, x)

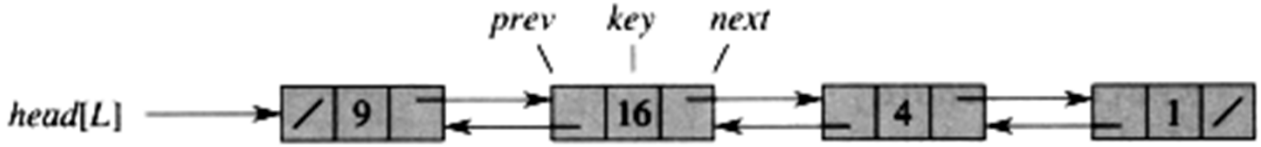
- **Data Structure, Memory and Pointers**
  - **Linked list**
    - A linked list is a data structure in which the objects are arranged in a linear order
    - Unlike an array, though, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object



    - Linked lists provide a simple, flexible representation for sets, supporting all the operations listed

• Data Structure, Memory and Pointers

• Linked list



• As illustrated, each element of a **linked list** *L* is an object with a *key* field and two other pointer fields: *next* and *prev*

• Given an element *x* in the list, *next*[*x*] points to its successor in the linked list, and *prev*[*x*] points to its predecessor

• If prev[*x*] =NULL, the element *x* has no predecessor and is therefore the first element, or **head**, of the list.

• If *next*[*x*] =NULL, the element *x* has no successor and is therefore the last element, or **tail**, of the list

- **Data Structure, Memory and Pointers**

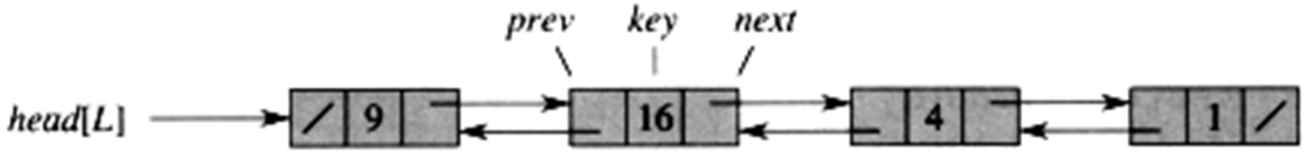
  - Searching a linked list

```
LIST-SEARCH(L,k)
    x ← head[L]
    while x ≠ NULL and key[L] ≠ k
          do  x ← next[x]
    return x
```

The procedure LIST-SEARCH(L,k) finds the first element with key k in list L by a simple linear search, returning a pointer to this element

- If no object with key k appears in the list, then NULL is returned

- For the linked list here, the call LIST-SEARCH ($L,4$) returns a pointer to the third element, and the call LIST-SEARCH($L,7$) returns NULL



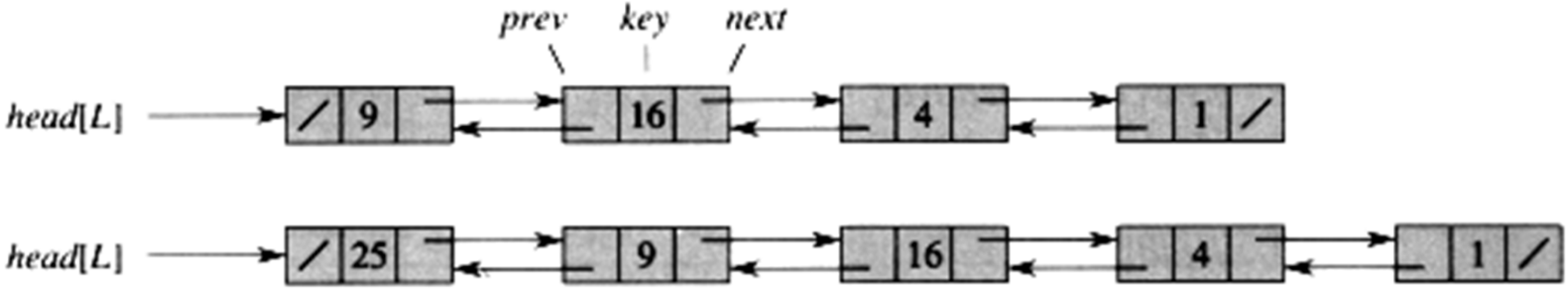
• Data Structure, Memory and Pointers

• Inserting into a linked list

LIST-INSERT(L,x)

$next[x] \leftarrow head[L]$
if $head[L] \neq$ NULL
then $prev[head[L]] \leftarrow x$
$head[L] \leftarrow x$
$prev[x] \leftarrow$ NULL

• Given an element x whose *key* field has already been set, the LIST-INSERT procedure "splices" x onto the front of the linked list, as shown below

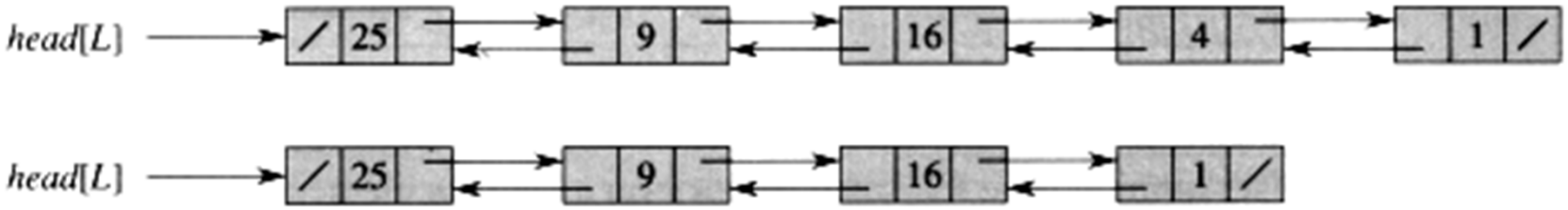


• Data Structure, Memory and Pointers

• Deleting into a linked list

```
LIST-INSERT(L,x)
    if prev[x] ≠ NULL
       then next[prev[x]] ← next[x]
       else head[L] ← next[x]
    if next[x] ≠ NULL
       then prev[next[x]] ← prev[x]
```

• The procedure LIST-DELETE removes an element x from a linked list L



• It must be given a pointer to x, and it then "splices" x out of the list by updating pointers.

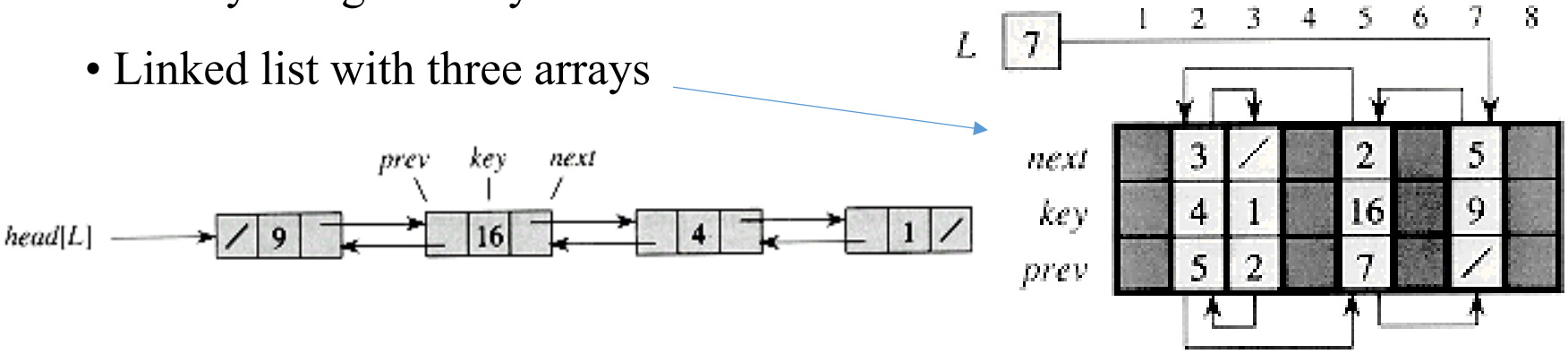
• Before deleting an element, it must first call LIST-SEARCH to retrieve a pointer to the element

• Implementing pointers and objects

• Multiple-array representation of objects

• The figure below represents a collection of objects that have the same fields by using an array for each field

• Linked list with three arrays



• The array *key* holds the values of the keys currently in the dynamic set, and the pointers are stored in the arrays *next* and *prev*

• For a given array index x, *key*[x], *next*[x], and *prev*[x] represent an object in the linked list

•Under this interpretation, a pointer x is simply a common index into the *key*, *next*, and *prev* arrays.

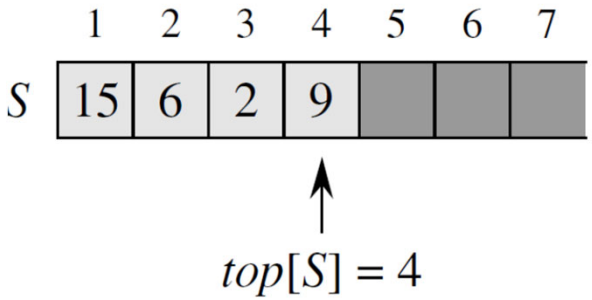
Stack & Queue

• Data Structure, Memory and Pointers

Stacks

• The INSERT operation on a stack is often called PUSH, and the DELETE operation is called POP

• In a stack, the element deleted from the set is the one most recently inserted: the stack implements a **last-in, first-out**, or **LIFO**.



• The array has an attribute *top*[*S*] that indexes the most recently inserted element

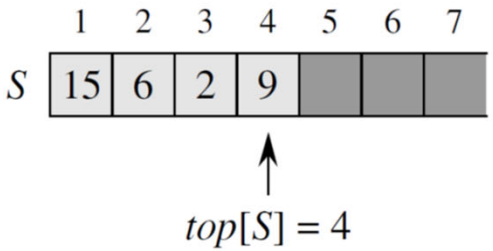
• The stack consists of elements *S*[1.. *top*[*S*]], where *S*[1] is the element at the bottom of the stack and *S*[*top*[*S*] is the element at the top

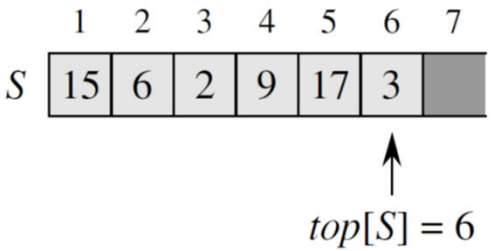
•. When *top*[*S*] = 0, the stack contains no elements and is empty

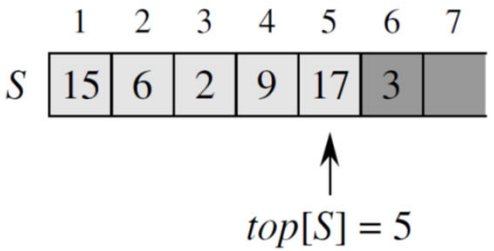
- **Data Structure, Memory and Pointers**
  - **Stacks (LIFO)**

PUSH(S,x)
  $top[S] \leftarrow top[S] + 1$
  $S[top[S]] \leftarrow x$

POP(S)
  if STACK EMPTY(S)
    then error
  else $top[S] \leftarrow top[S] - 1$
    return $S[top[S] + 1]$







Queues (FIFO)

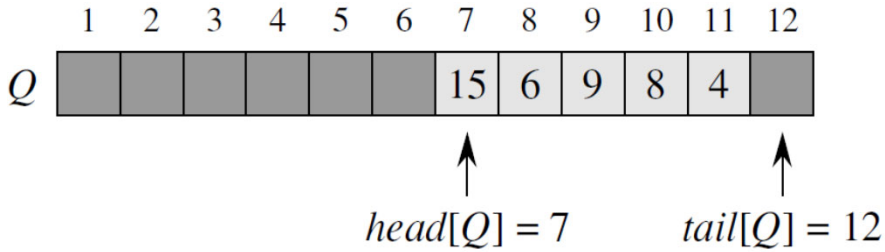
• The INSERT operation on a queue is called ENQUEUE, and we call the DELETE operation DEQUEUE

• The queue has a **head** and a **tail**. When an element is *enqueued*, it takes its place at the tail of the queue

• The element *dequeued* is always the one at the head of the queue

• In a queue , the element deleted is always the one that has been in the set for the longest time: the queue implements a **first-in, first-out**, or **FIFO**.

• The figure below shows one way to implement a queue of at most n-1 elements using an array $Q[1..n]$



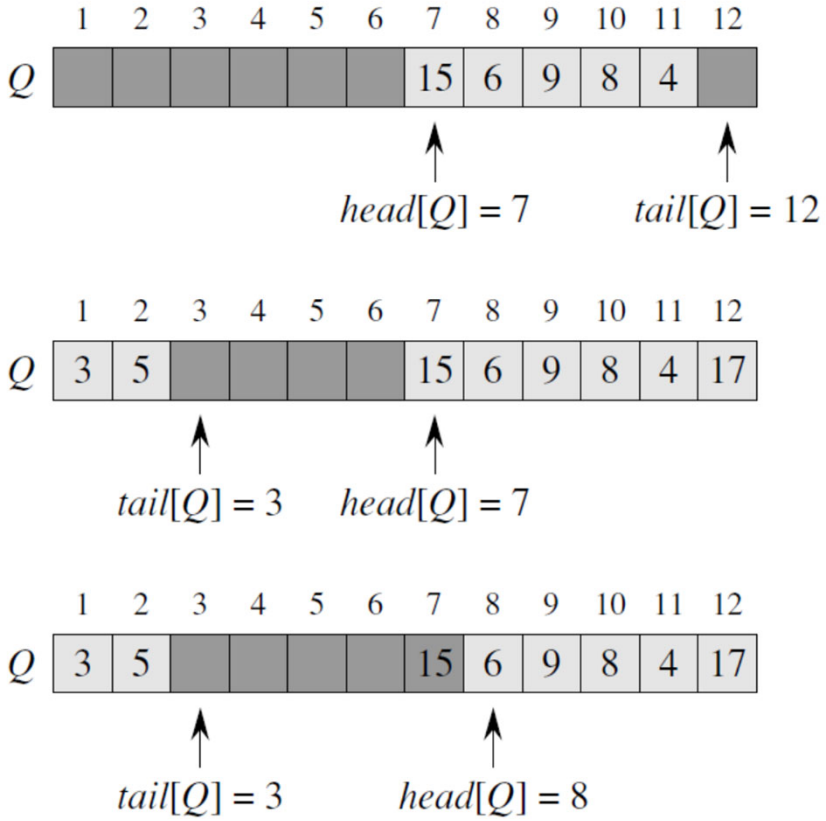
• The queue has an attribute $head[Q]$ that indexes, or points to, its head.

• The attribute $tail[Q]$ indexes the next location at which a newly arriving element will be inserted into the queue

- **Data Structure, Memory and Pointers**
  - **Queues**

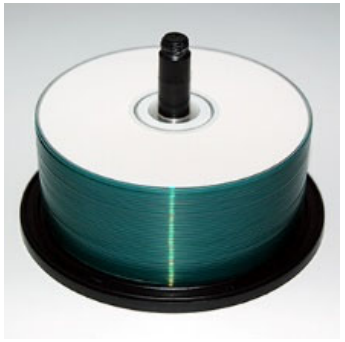
```
ENQUEUE(Q, x)
  Q[tail[Q]] ← x
  if tail[Q] = length[Q]
        then tail[Q] ← 1
        else tail[Q] ← tail[Q] + 1
```

```
DEQUEUE(Q, x)
x ← Q[head[Q]]
 if head[Q] = length[Q]
       then head[Q] ← 1
       else head[Q] ← head[Q] + 1
 return x
```



Stack Overflow, Stack Underflow

A stack can also be implemented to have a maximum capacity. If the stack is full and does not contain enough slots to accept new entities, it is said to be an *overflow* – hence the phrase "stack overflow". Likewise, if a pop operation is attempted on an empty stack then a "stack underflow" occurs.



Push - it specifies adding an element to the Stack. If we try to insert an element when the Stack is full, then it is said to be Stack **Overflow** condition

Pop - it specifies removing an element from the Stack. Elements are always removed from top of Stack. If we try to perform pop operation on an empty Stack, then it is said to be Stack **Underflow** condition.

Queue Overflow, Queue Underflow

- Queue overflow happens by trying to add a key to a full queue.
- Queue underflow happens when trying to remove an element from an empty queue

Summery

Stack and Queue can be implemented using both, arrays and linked list. The limitation in case of array is that we need to define the size at the beginning of the implementation. This makes the Stack or Queue static. It can also result in "Stack or Queue overflow" if we try to add a key after the array is full. **The alleviate to this problem is, we use dynamic memory allocation of Stack and Queue.**

Heap sort

- O *(n* lg *n*) worst case—like merge sort.
- Sorts in place—like insertion sort.
- Combines the best of both algorithms.

Heap sort algorithm

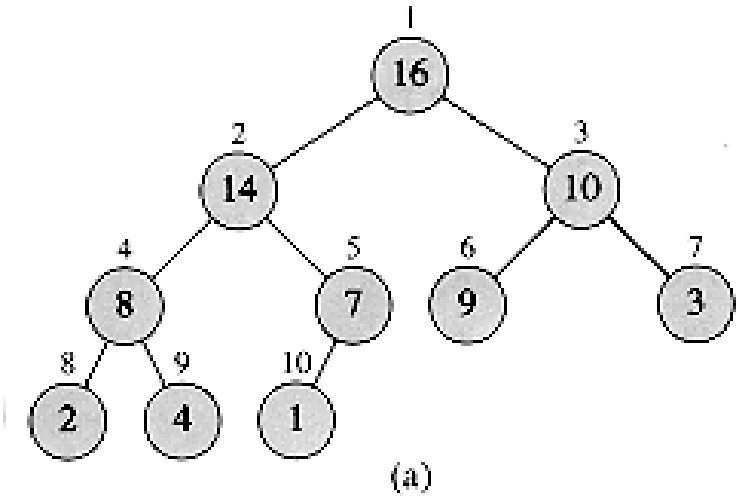
Step 1: Build Heap

Step 2: MAX or MIN HEAPIFY

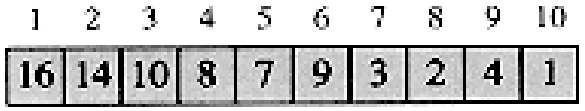
Step 3: Sorting

Heap data structure

- Heap A is a nearly complete binary tree.
- ***Height*** of node = # of connections on a longest path from the node down to a leaf.
- ***Height*** of heap = height of root = ($\lg n$).
- A heap can be stored as an array A.
- Root of tree is A[1].
- Parent of A[i] =A[$\lfloor i/2 \rfloor$].
- Left child of A[i] =A[$2i$].
- Right child of A[i] =A[$2i$ +1].



(a)



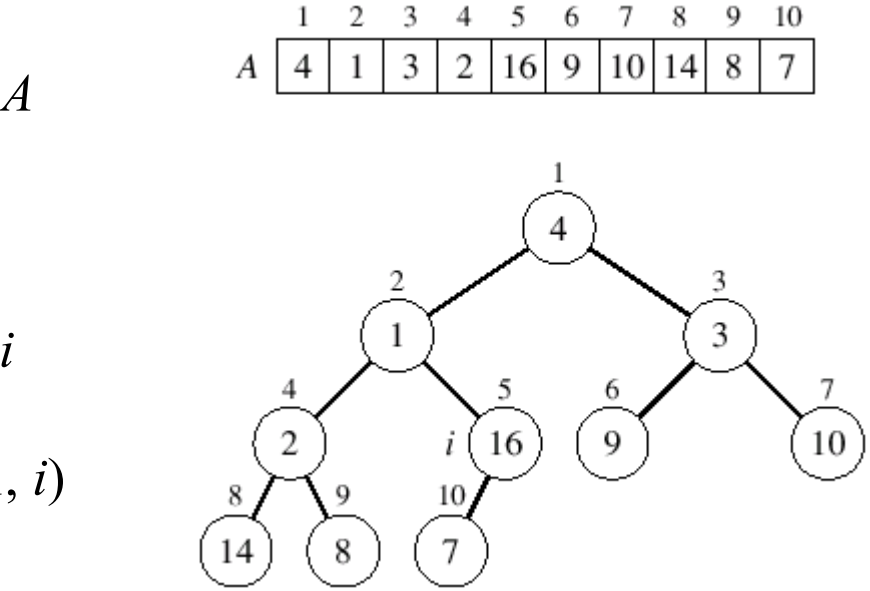
(b)

- **Maintaining the heap property**
  - **Heap property**
  - For max-heaps (largest element at root), ***max-heap property:*** for all nodes *i*, excluding the root, A[PARENT *i*)] =A[*i*].
  - For min-heaps (smallest element at root), ***min-heap property:*** for all nodes *i*, excluding the root, A[PARENT *i*)] =A[*i*].
  - By induction and transitivity of =, the max-heap property guarantees that the maximum element of a max-heap is at the root. Similar argument for min-heaps.
  - HEAPIFY is an important subroutine for manipulating heaps. Its inputs are an array A and an index i into the array.
  - When HEAPIFY is called, it is assumed that the binary trees rooted at LEFT(i) and RIGHT(i) are heaps, but that A[i] may be smaller than its children, thus violating the heap property
  - . The function of HEAPIFY is to let the value at A[i] "float down" in the heap so that the subtree rooted at index i becomes a heap.

- **Heap sort**
  - MAX-HEAPIFY is important for manipulating max-heaps. It is used to maintain the max-heap property.
  - Before MAX-HEAPIFY, A[*i*] may be smaller than its children.
  - Assume left and right subtrees of *i* are max-heaps.
  - After MAX-HEAPIFY, subtree rooted at *i* is a max-heap.
  - **The way MAX-HEAPIFY works:**
    - Compare A[*i*], A[LEFT *i*)], and A[RIGHT *i*)].
    - If necessary, swap A[*i*] with the larger of the two children to preserve heap property.
    - Continue this process of comparing and swapping down the heap, until subtree rooted at *i* is max-heap.
    - If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.

• Building a heap

• A 10-element input array A and the binary tree it represents



• Build the binary tree from the Array *A*

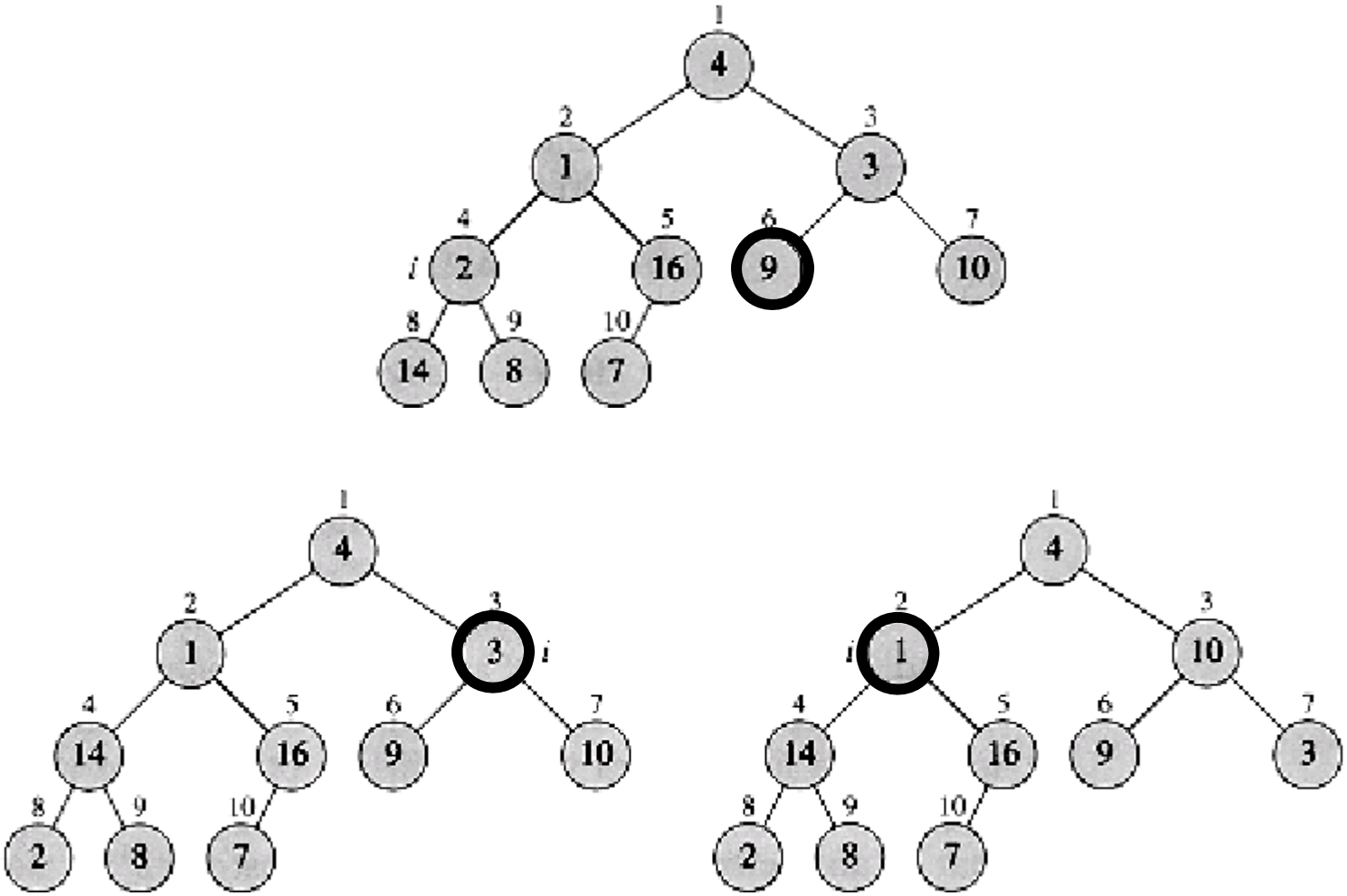
• Since $i = \lfloor n / 2 \rfloor$ before the first iteration, the invariant is initially true

• The figure shows that the loop index *i* points to node 5 , before the call HEAPIFY(*A*, *i*)

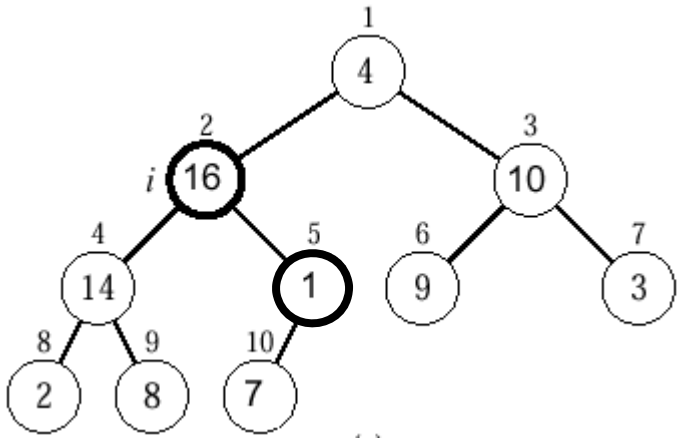
• *i* starts off as 5.

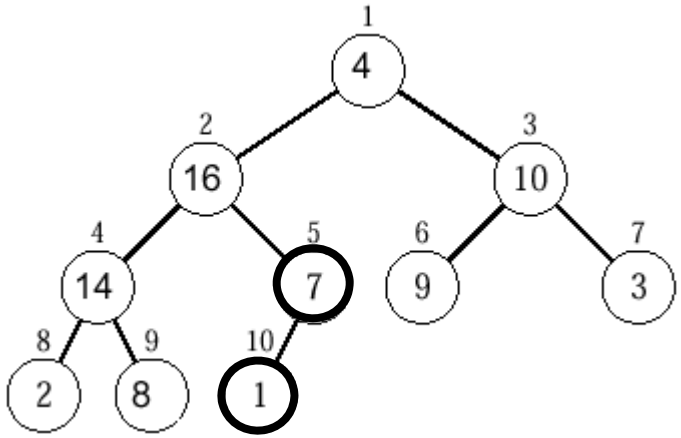
• MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.

MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.
Begin from index 6.

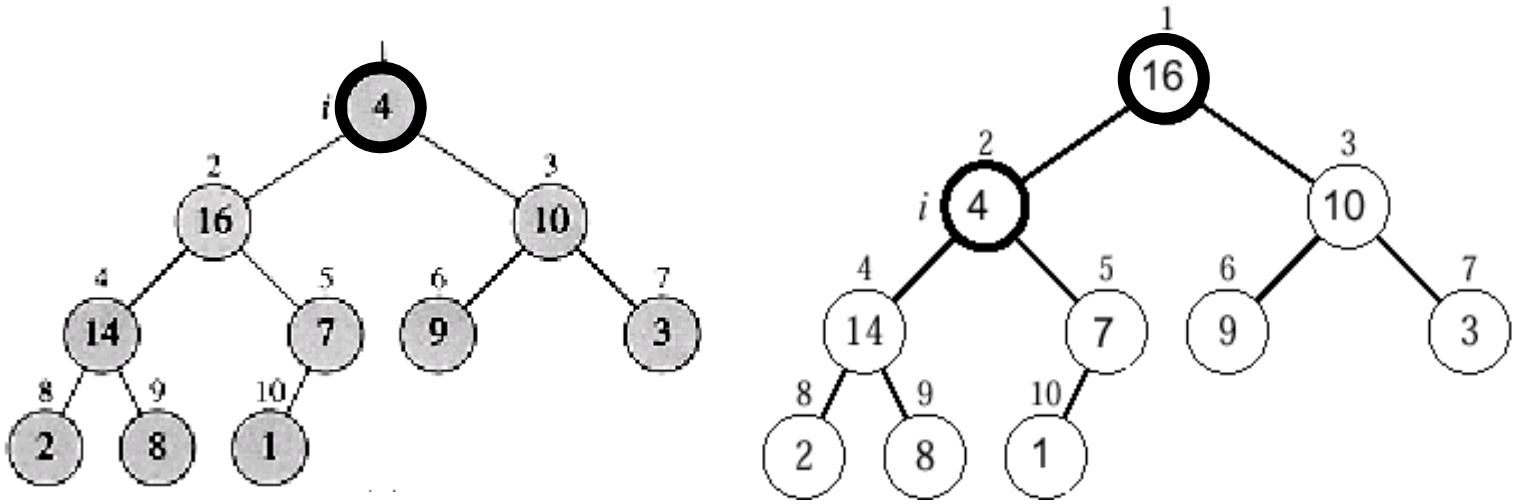


MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.

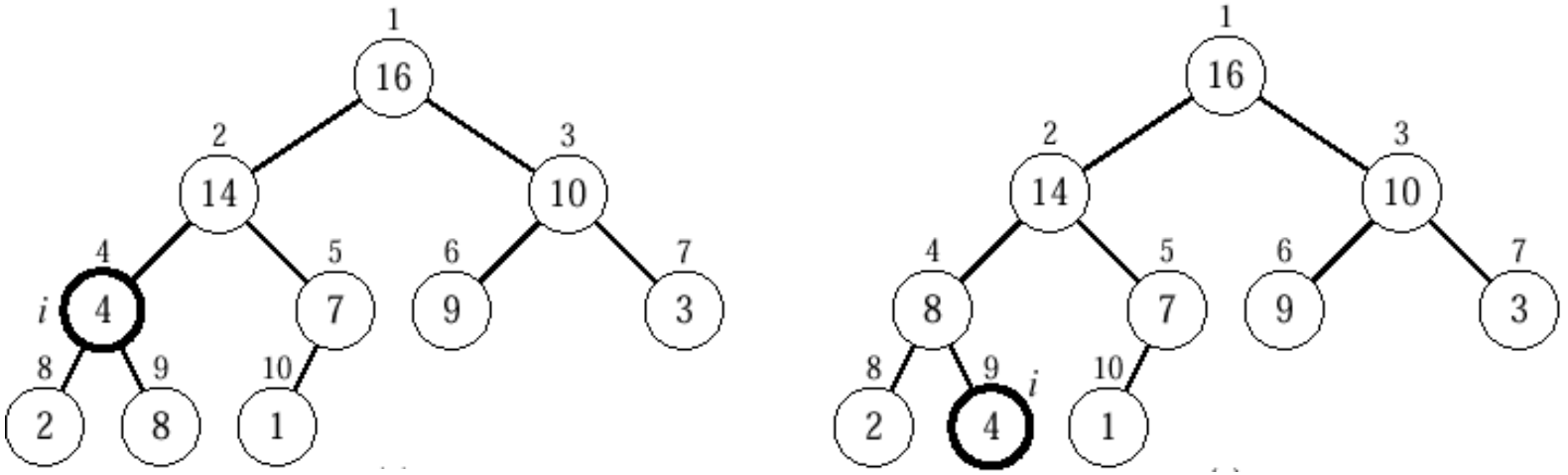




MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.



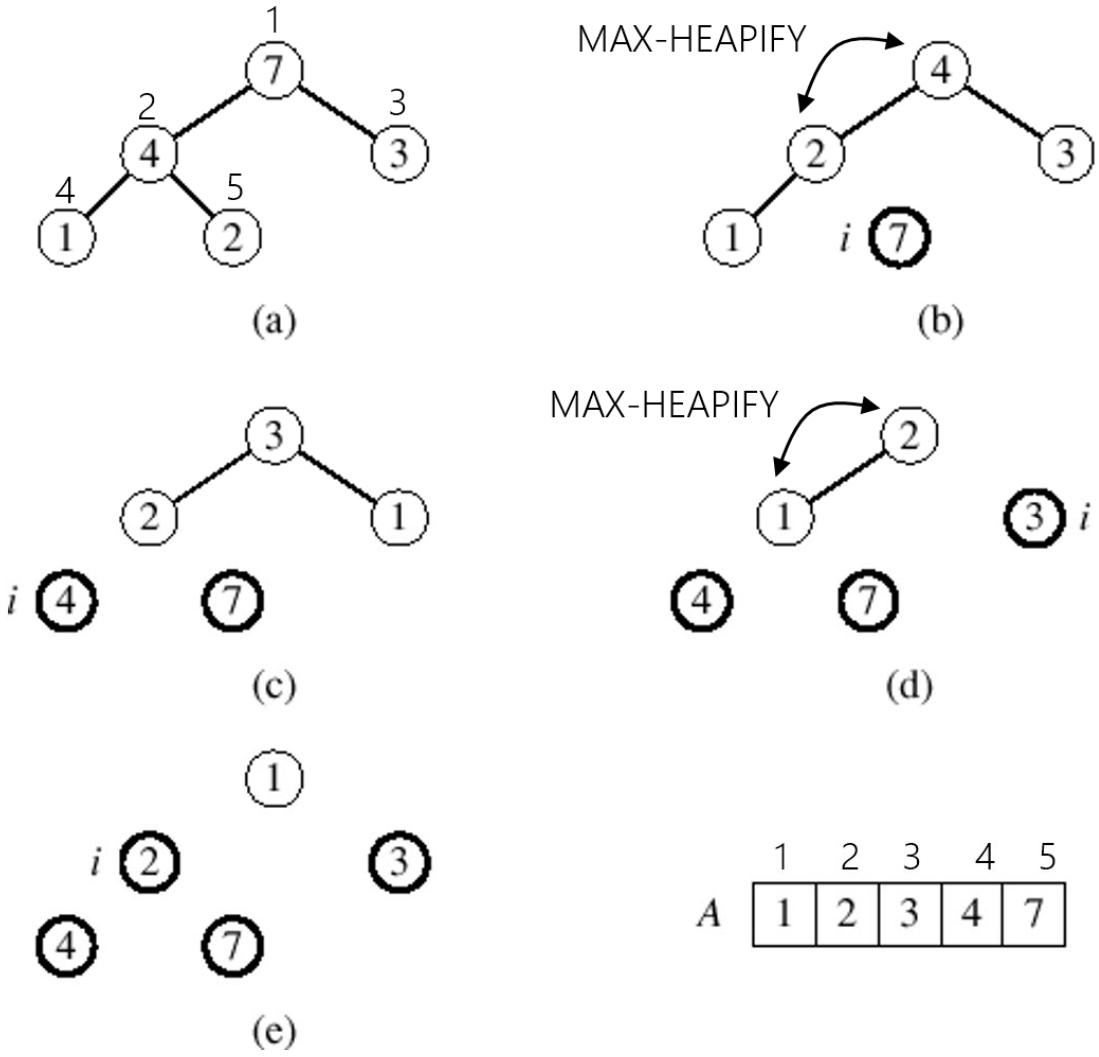
MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.



• The heapsort algorithm

- Given an input array, the heapsort algorithm acts as follows:
  - Builds a max- or min- heap from the array.
  - Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.
  - "Discard" this last node (knowing that it is in its correct place) by decreasing the heap size, and calling MAX-HEAPIFY on the new (possibly incorrectly-placed) root.
  - Repeat this "Discarding" process until only one node (the smallest element) remains, and therefore is in the correct place in the array.

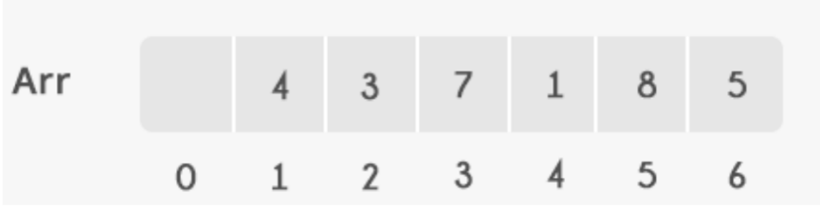
- **The heapsort algorithm**

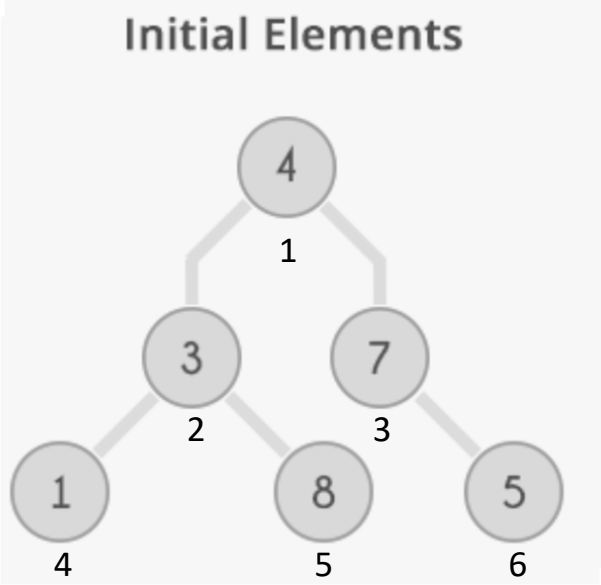


Example

https://www.hackerearth.com/practice/algorithms/sorting/heap-sort/tutorial

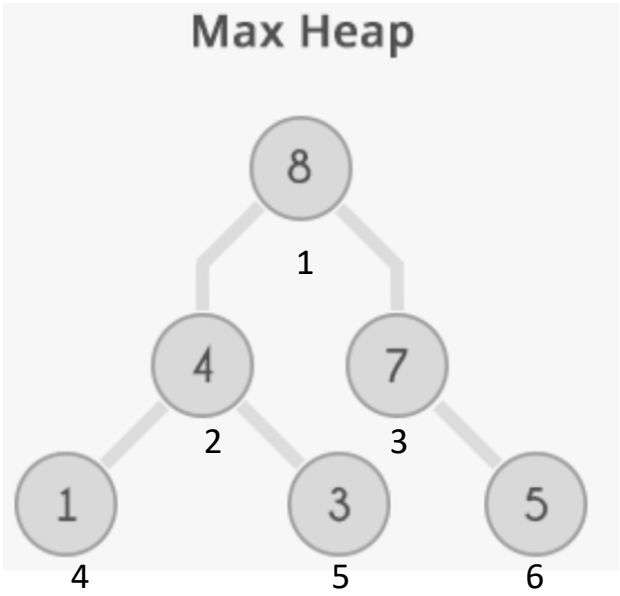
- Build Heap





Example

- HEAPIFY: Max Heap

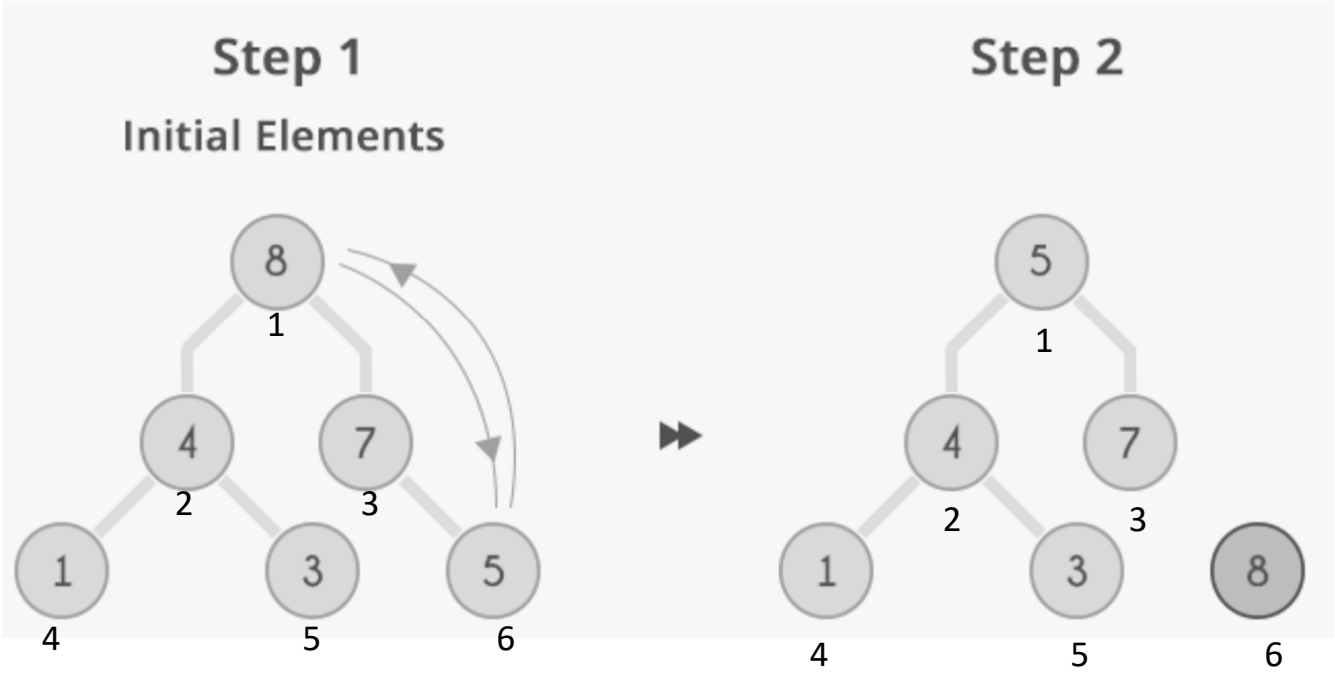


Arr		8	4	7	1	3	5
	0	1	2	3	4	5	6

- **Heapsort algorithm**

Step 1: 8 is swapped with 5.

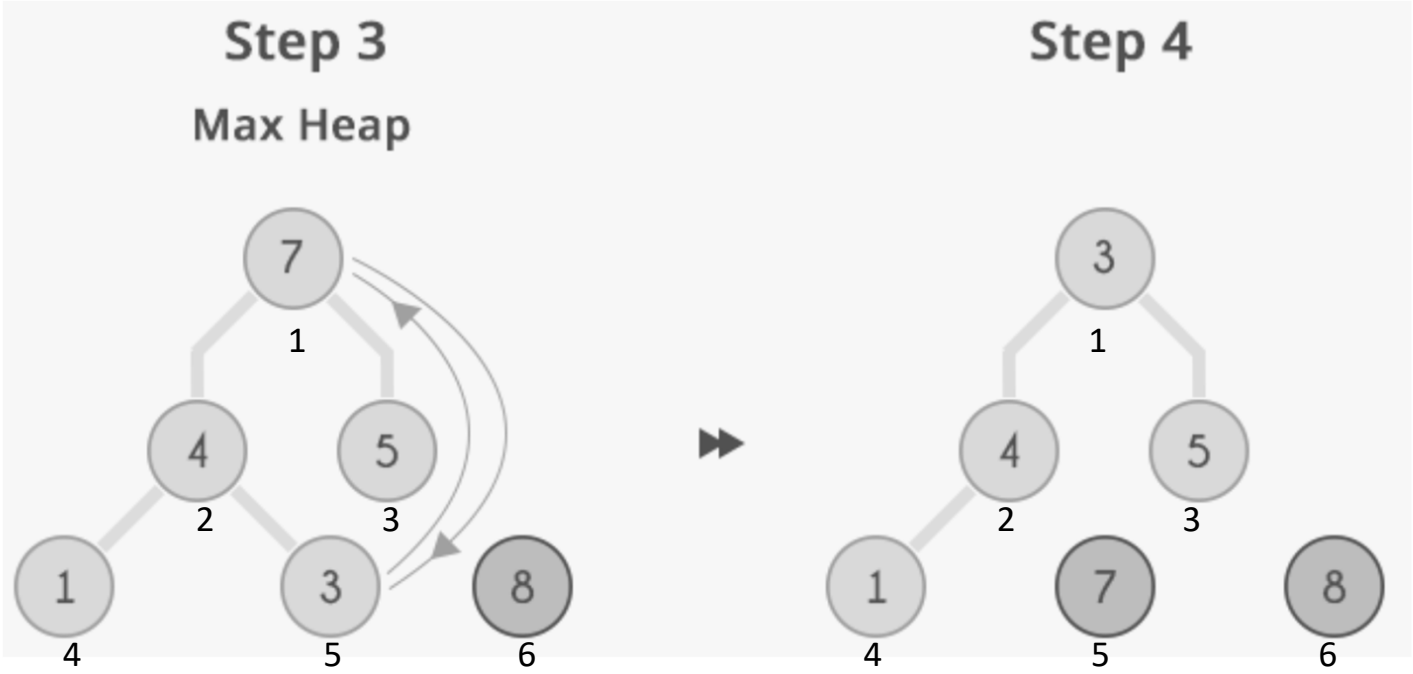
Step 2: 8 is disconnected from heap as 8 is in correct position now and.



- **Heapsort algorithm**

Step 3: Max-heap is created and 7 is swapped with 3.

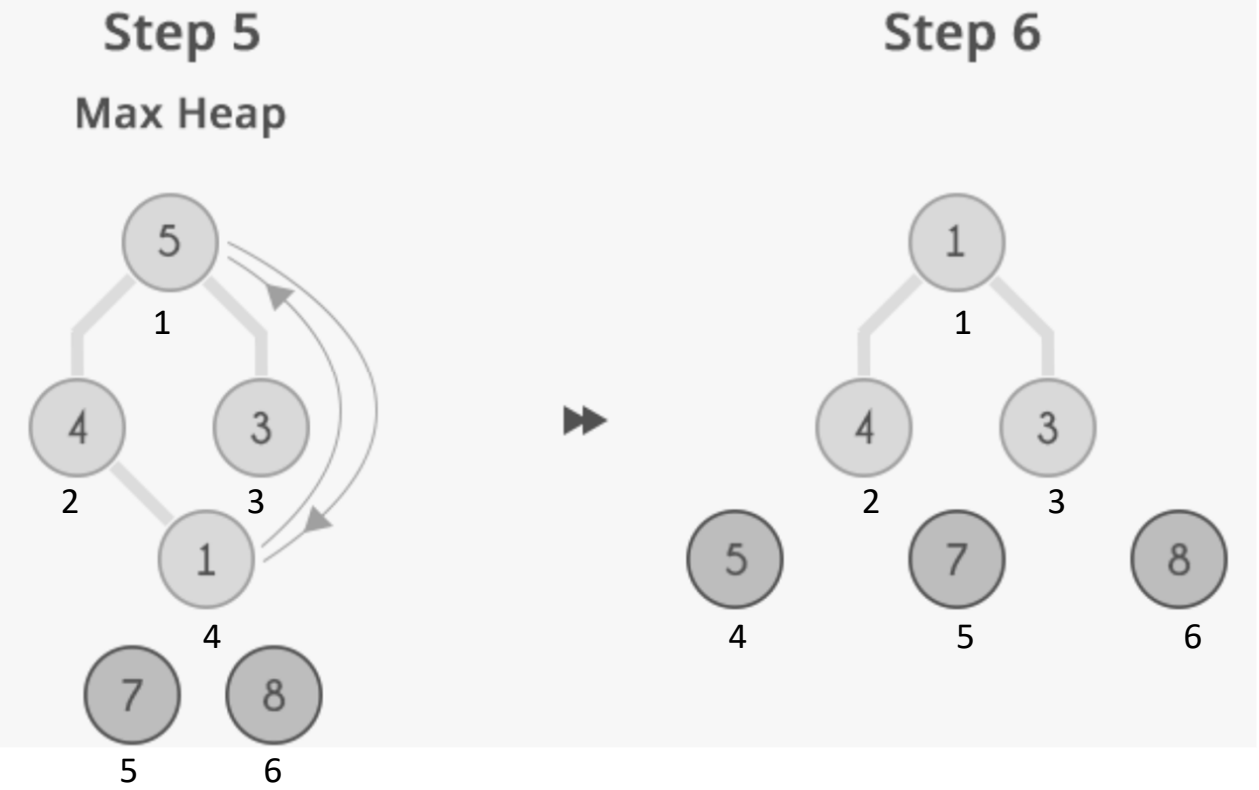
Step 4: 7 is disconnected from heap.



- **Heapsort algorithm**

Step 5: Max heap is created and 5 is swapped with 1.

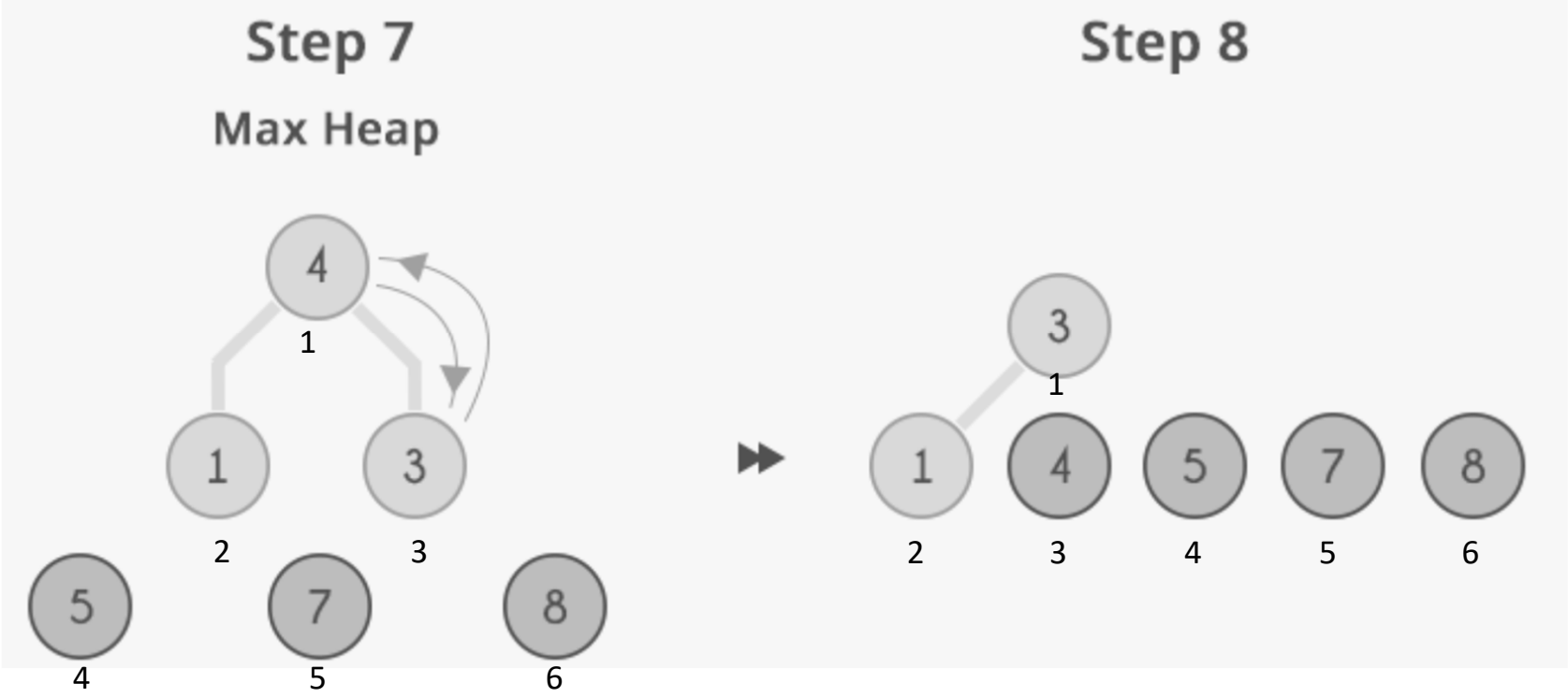
Step 6: 5 is disconnected from heap.



- **Heapsort algorithm**

Step 7: Max heap is created and 4 is swapped with 3.

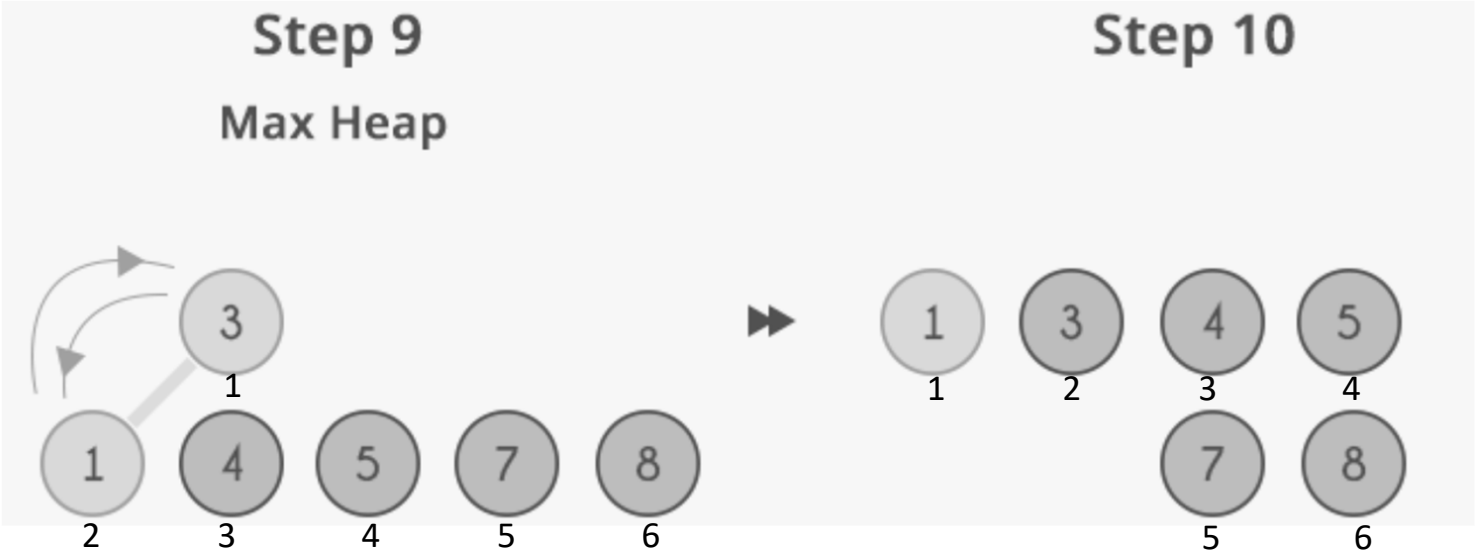
Step 8: 4 is disconnected from heap.



- **Heapsort algorithm**

Step 9: Max heap is created and 3 is swapped with 1.

Step 10: 3 is disconnected.



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

Algorithms & Data Structures

- Professor Reza Sedaghat

  COE428: Engineering Algorithms & Data Structures

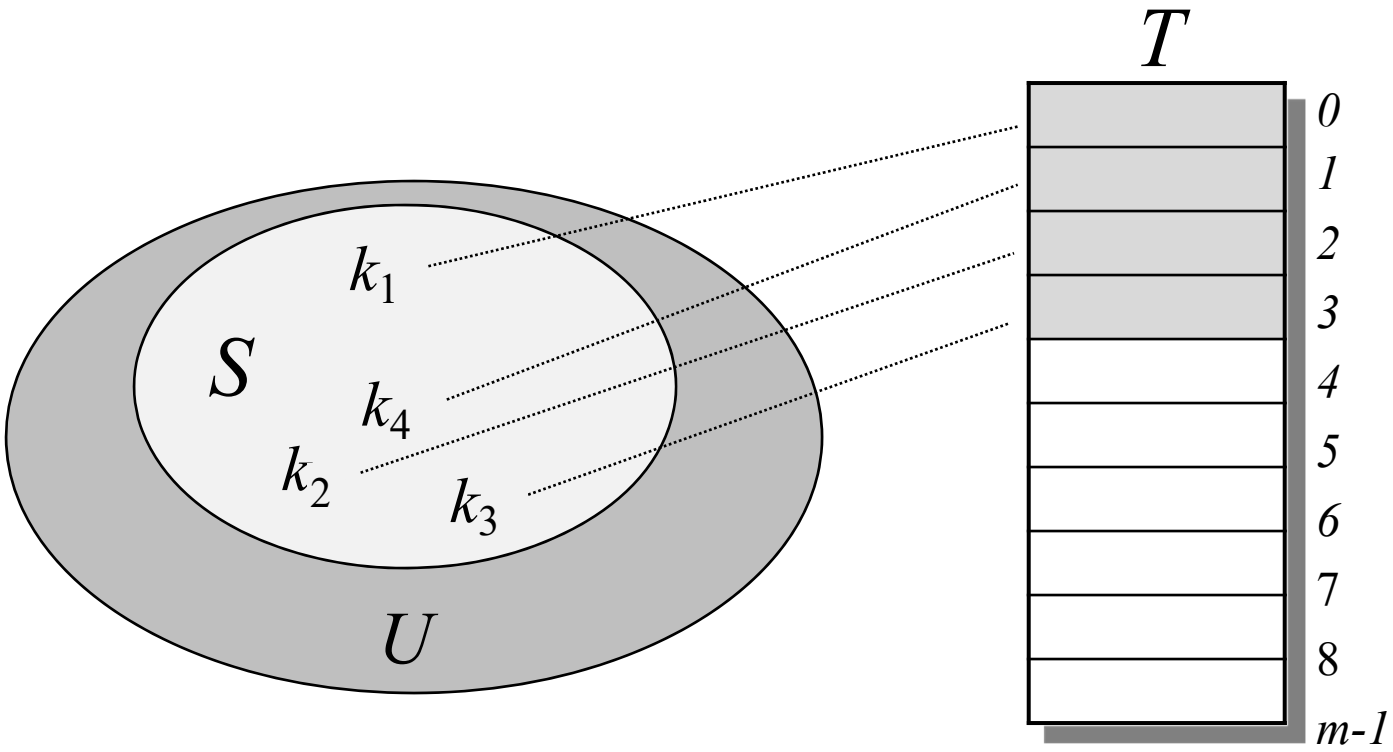
- Email address: rsedagha@ee.ryerson.ca
- Course outline: www.ee.ryerson.ca/~courses/COE428/
- Course References:

  1) **Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. Introduction to Algorithms, MIT, ISBN: 0-07-013151-1 (McGraw-Hill) (Course Text)**

Hash Table

• Many applications require a dynamic set that supports only the dictionary operations `INSERT`, `SEARCH`, and `DELETE`.

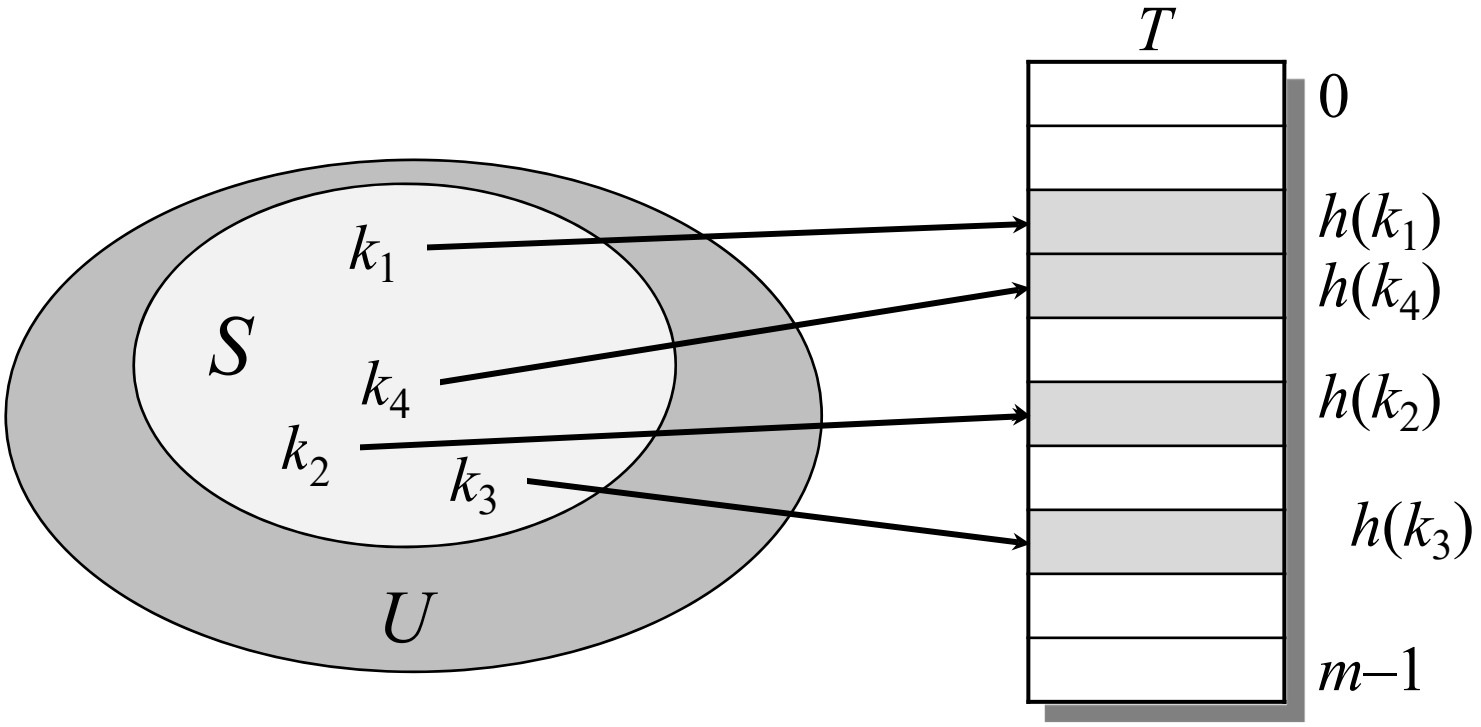
• Although searching for an element in a hash table can take as searching for an element in an array in worst case, linear, O(n)



Hashing, Hash function

• An element with key k **hashes** to slot h(k); we also say that h(k) is the **hash value** of key k

• Figure bellow illustrates the basic idea. The point of the hash function is to reduce the range of array indices that need to be handled. Instead of |U| values, we need to handle only m values



Hash Table

• Although searching for an element in a hash table can take as searching for an element in an array in worst case, linear, O(n)

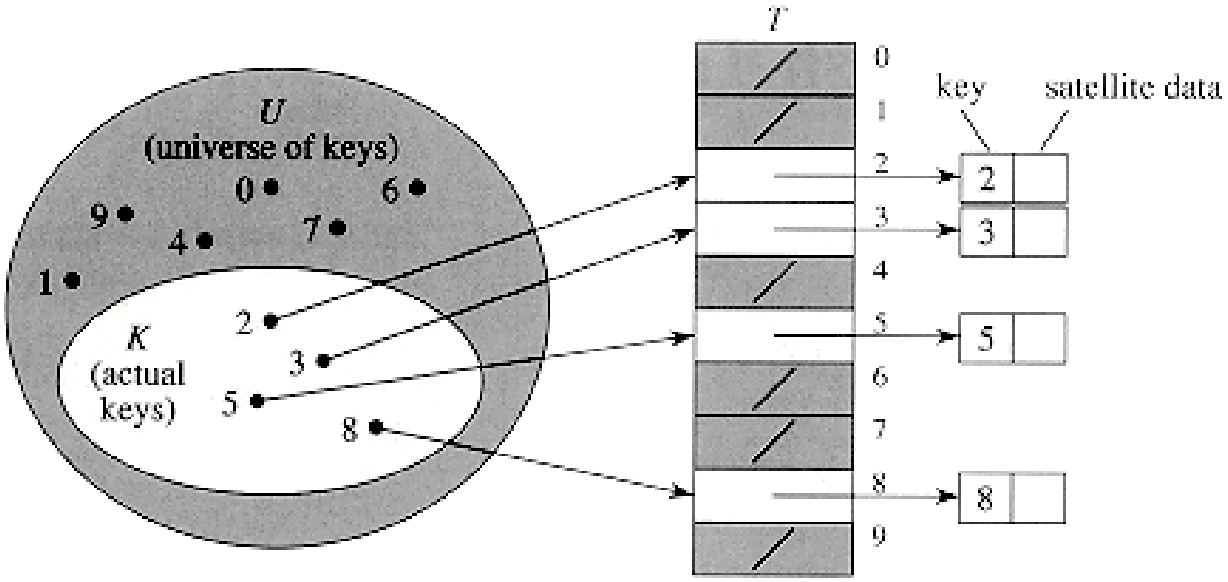
• **Can we reduce the worse case time?**

• A hash table is a generalization of the simpler notion of an ordinary array. Directly addressing into an ordinary array makes effective use of our ability to examine an arbitrary position in an array in **O (1)** time.

• **Direct addressing** is applicable when we can afford to allocate an array that has one position for every possible key.

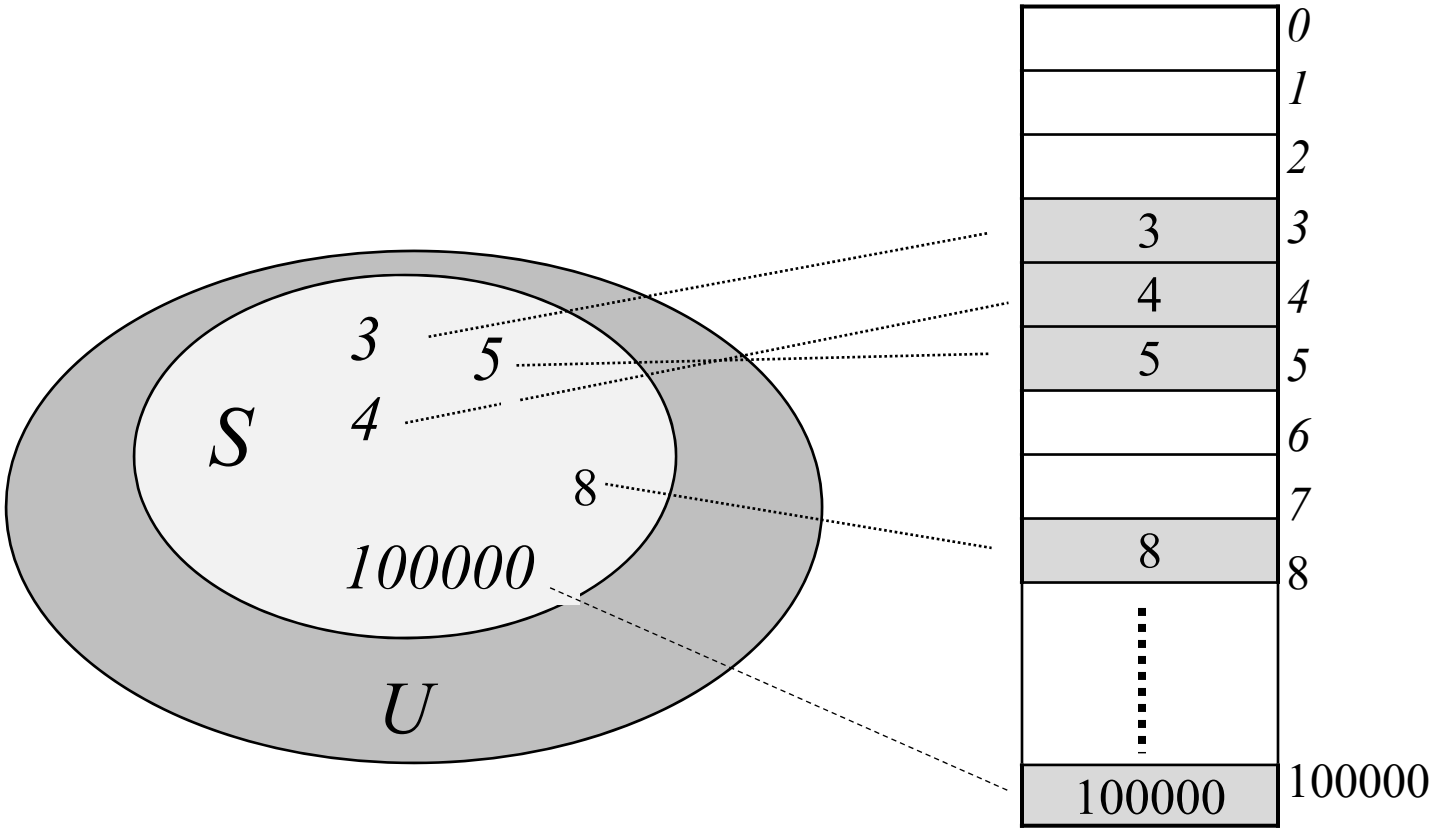
Hash Table

- **Direct-address tables**
  - •Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.
  - •Suppose that an application needs a dynamic set in which each element has a key drawn from the universe U = {0,1, . . . , m - 1}, where m is not too large.



Hash Table

- **Problems of Direct-address tables**
  If there is a big range between the key, the memory will be wasted to fill all keys in the table.
  Example: given is the set of keys, {3, 4, 5, 8, 100000}

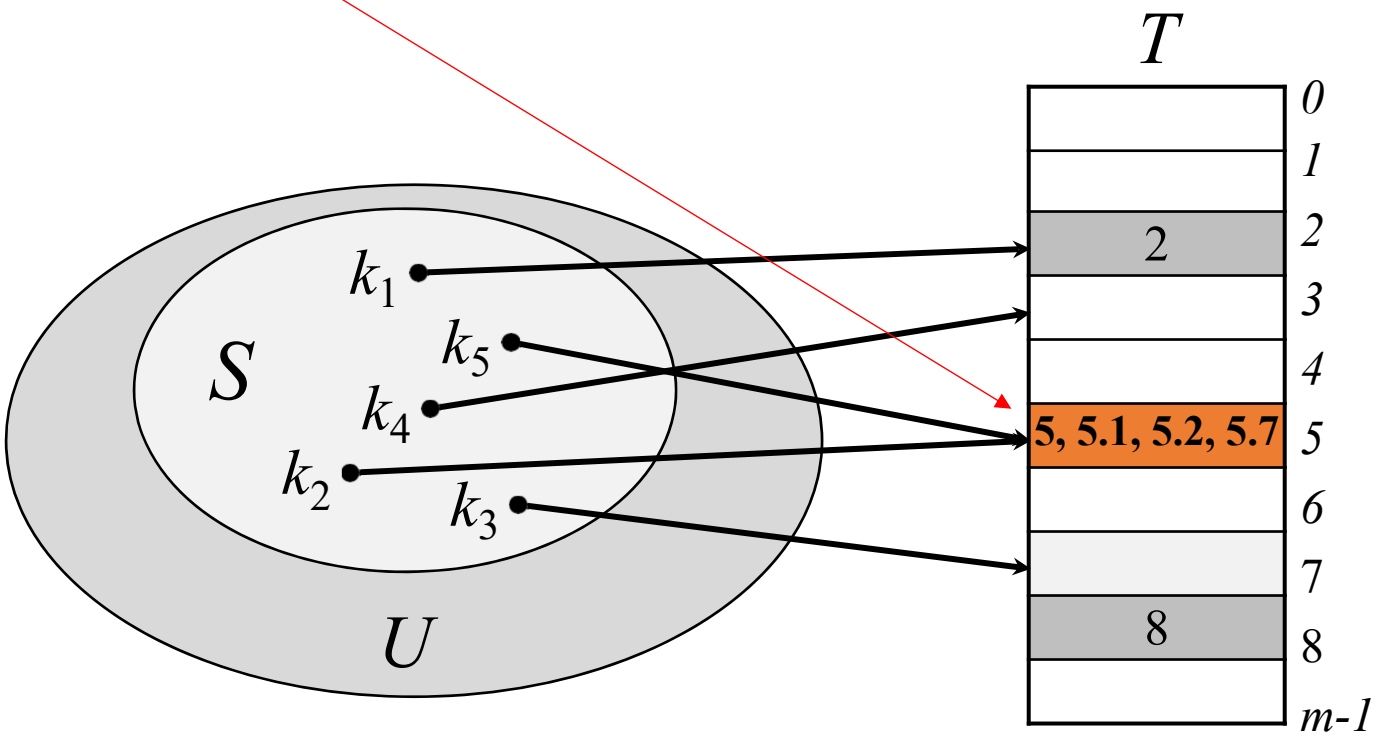


Hash Table

- **Problems of Direct-addressing tables**

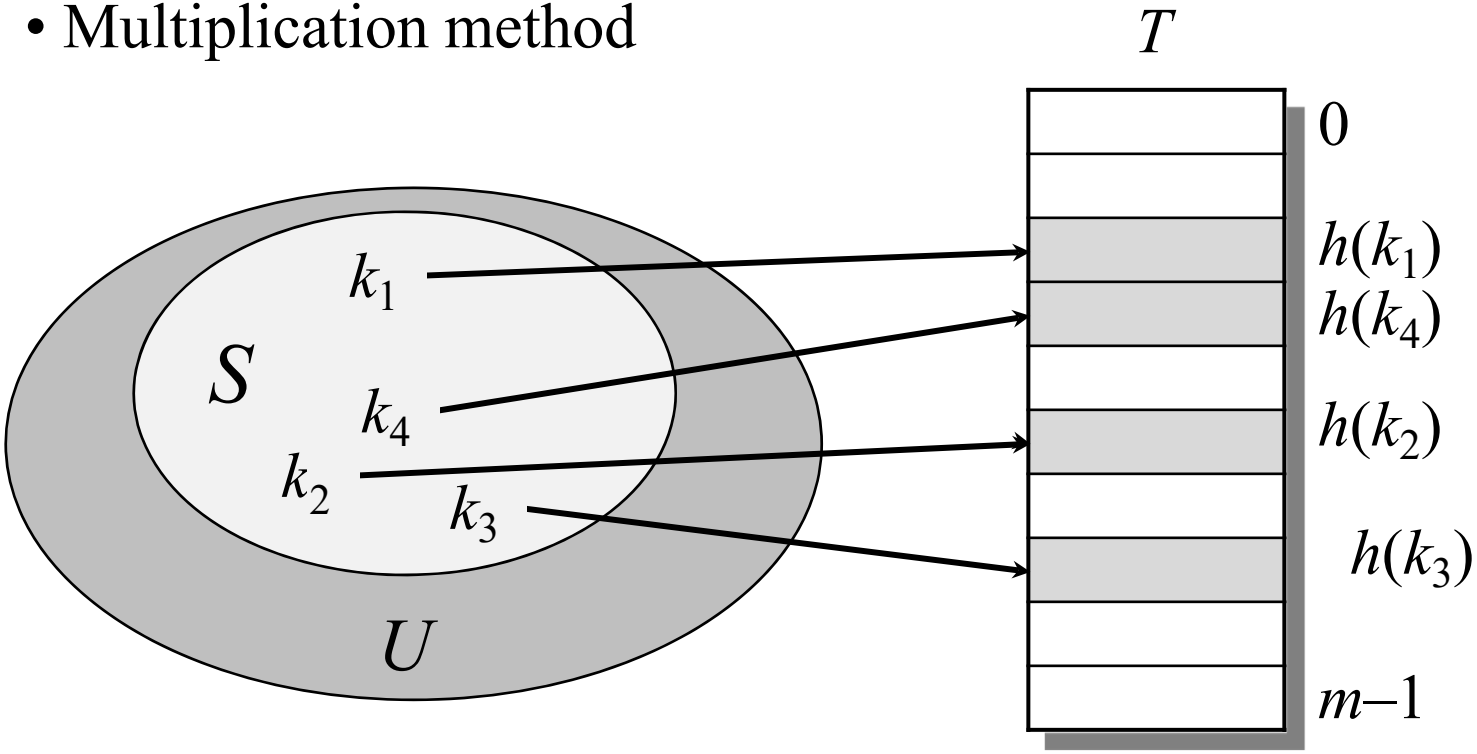
When a key to be inserted to an already occupied slot in T, a ***collision*** occurs.

Example: Similar keys, e.g. {2, **5, 5.1, 5.2, 5.7**, 8}



Hashing, Hash function

- A **hash function** h is used to compute the slot (**hash value**) from the key k, T[0 . . m-1]: h: U {0,1, . . . , m - 1}
- **Some hash functions are:**
  - Division method
  - Multiplication method



Hashing, Hash function

- For the creation of hashing exist three methods:
  - Hashing by division, hashing by multiplication, and universal hashing
- **The division method**
  - In the **division method** for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m.
  - The hash function is

$$h(k) = k \bmod m = Mod\ (k, m)$$

  - Example, if the hash table has size m = 10 and the key is k = 12, then the hash value is h(k)=12 mod 10 = 2
  - Since it requires only a single division operation, hashing by division is quite fast

Hashing, Hash function

- **Multiplication method**

  • The **multiplication method** for creating hash functions operates in two steps.

  • First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. A is a real-valued constant. Although any value of A gives the hash function, but some values of A are better than others. According to *Knuth*, we can use the golden ratio for A, So A will be

$$A = \frac{\sqrt{5} - 1}{2} = 0.61803398$$

  • Second, we multiply this value by m and take the floor of the result. In short, the hash function is

$$h(k) = \lfloor m\,(k\,A \bmod 1)\rfloor\ ,$$

Hashing, Hash function

- **The multiplication method**

- If $0 < A < 1$, $h(k) = \lfloor m\,(kA \bmod 1)\rfloor = \lfloor m\,(kA - \lfloor kA \rfloor)\rfloor$,
  where $kA \bmod 1$ means the fractional part of kA, i.e., $kA - \lfloor kA \rfloor$.

- Disadvantage: Slower than the division method.
- Advantage: Value of m is not critical.

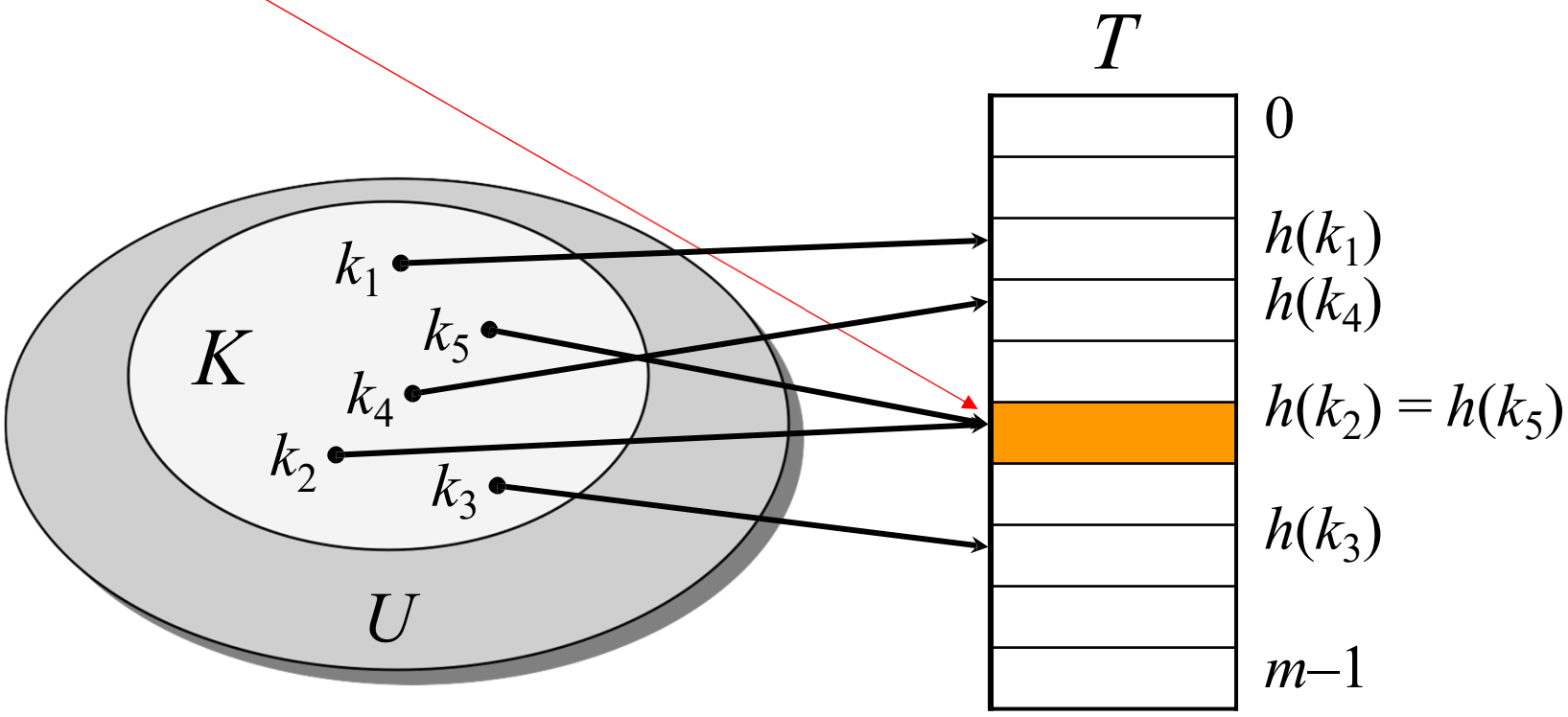
- Example: $m = 1000$, $k = 123$, $A \sim 0.6180339887\ldots$

$$h(k) = \lfloor 1000(\,123 \cdot 0.61803\,39887 \bmod 1\,)\rfloor$$
$$= \lfloor 1000 \cdot 0.018169\ldots \rfloor = 18.$$

Hash Table

- **Problems of Hash Functions**

When a key to be inserted to an already occupied slot in T, a ***collision*** occurs, e.g. if hash values of $h(k_2) = h(k_5)$



Hashing, Hash function

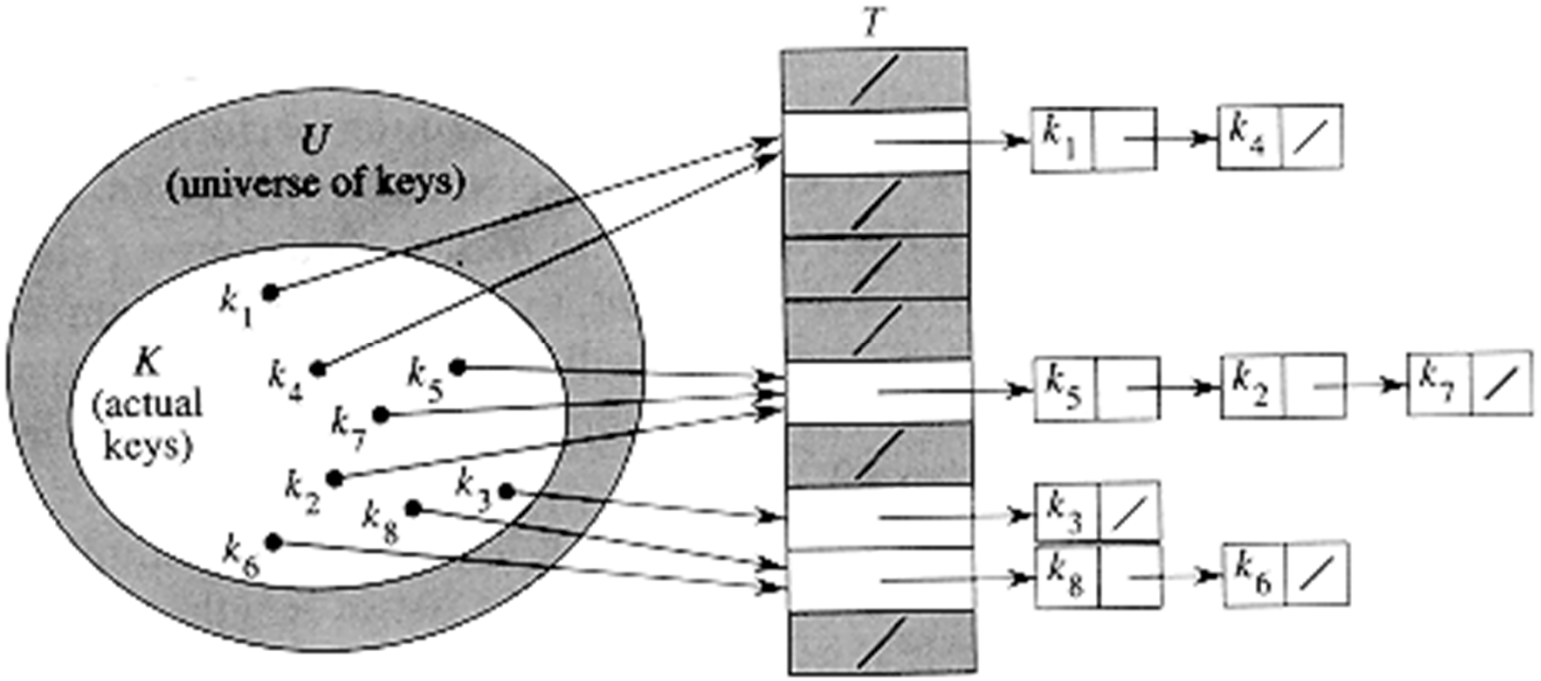
- Since |U| > m, there must be two keys that have the same **hash value**; avoiding collisions altogether is therefore impossible.
- **A well-designed, "random"- looking hash function can minimize the number of collisions.**
- The simplest collision resolution technique are **chaining, Linear probing, Quadratic probing, and double hashing**, etc.

Hashing, Chaining

• Collision resolution by chaining. Each hash-table slot T[j] contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$

• In **chaining**, we put all the elements that hash to the same slot in a linked list, as shown in Figure bellow

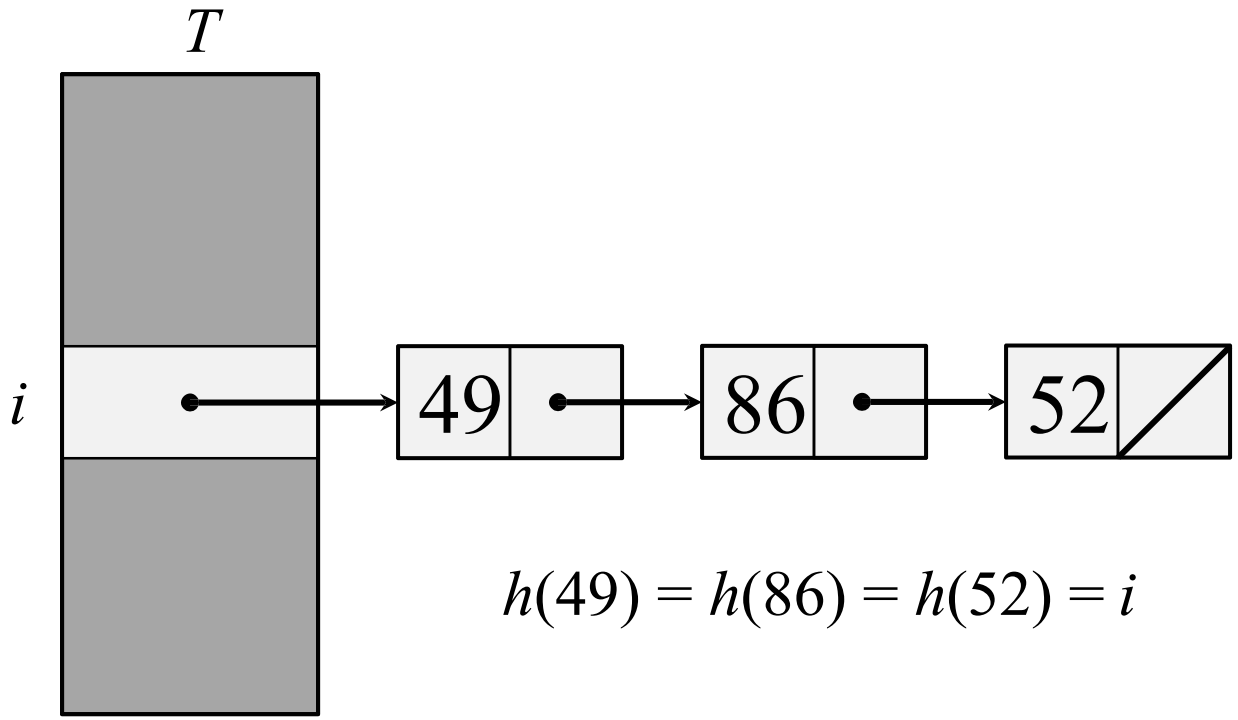
• Slot j contains a pointer to the head of the list of all stored elements that hash to j, if there are no such elements, slot j contains `NIL`.



Resolving collisions by chaining

- Worse case time of Chaining is linear **O(n)**
- Records in the same slot are linked into a list.

We want to minimize or eliminate chaining



Resolving collisions by Open Addressing

• Open Addressing is done with following approaches:

a) ***Linear Probing:*** In linear probing, we linearly probe for next slot. For example, the typical gap between two probes is 1

b) ***Quadratic Probing***

We look for i^2'th slot in i'th iteration

c) ***Double Hashing***

We use another hash function hash2(x) and look for i*hash2(x) slot in i'th rotation

Resolving collisions by Open Addressing

Linear Probing

Let *hash(k)* be the slot index computed using a hash function and *F(i)* be the table index *i* and table size *m*

If slot [*hash(k) + F(i)] mod m* is occupied, then we try
(hash(k) + 1) + *F(i)] mod m*

If [(hash(k) + 1) + *F(i)] mod m* is also occupied, then we try
[(hash(k) + 2) + *F(i)] mod m*

If [(hash(k) + 2) + *F(i)] mod m* is also occupied, then we try
[(hash(k) + 3) + *F(i)] mod m*

Until $i = m$

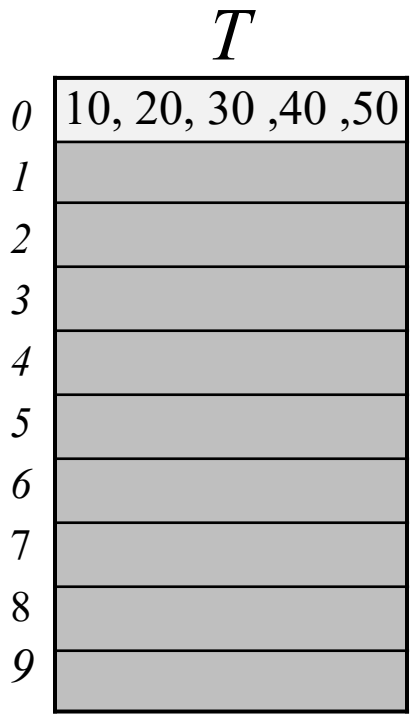
Resolving collisions by Open Addressing

Linear Probing

Example: Let's take the hash function $h(k) = k \bmod m$

$k = (10, 20, 30, 40, 50)$. This set crates collusion on slot $i = 0$

Hash Values:
10 mod 10 = 0
20 mod 10 = 0
30 mod 10 = 0
40 mod 10 = 0
50 mod 10 = 0



Collision

Resolving collisions by Open Addressing

Linear Probing

Example: Let's take the hash function $h(k) = k \bmod m$

$k = (10, 20, 30, 40, 50)$

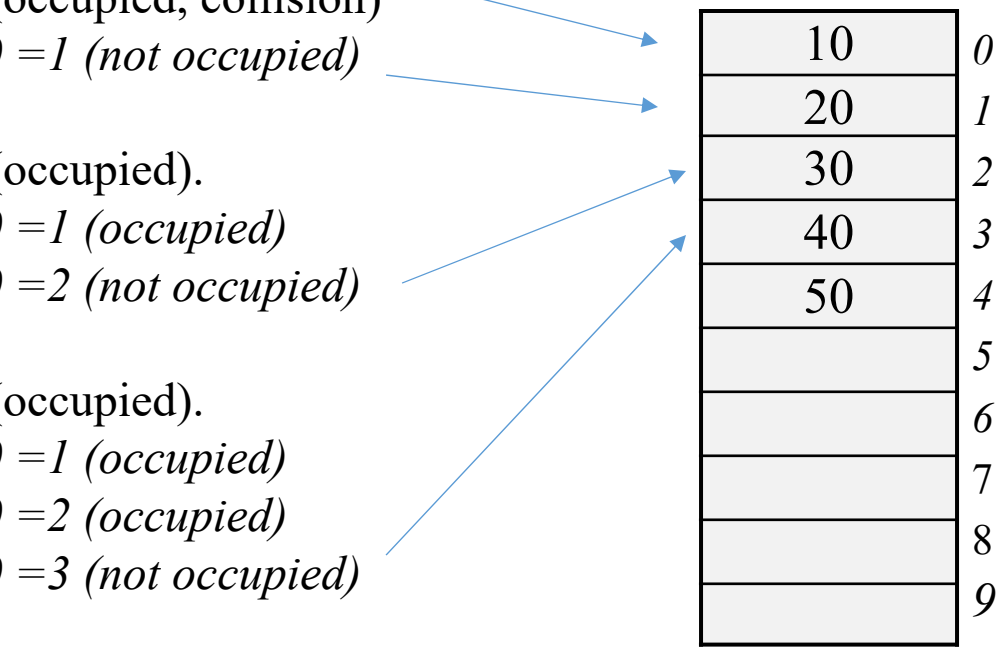
Hash Values:

$hash(k) = 10 \bmod 10 = 0$

$hash(k) = 20 \bmod 10 = 0$ (occupied, collision)
Try $[hash(k) + 1] \bmod 10 = 1$ *(not occupied)*

$hash(k) = 30 \bmod 10 = 0$ (occupied).
Try $[hash(k) + 1] \bmod 10 = 1$ *(occupied)*
Try $[hash(k) + 2] \bmod 10 = 2$ *(not occupied)*

$hash(k) = 40 \bmod 10 = 0$ (occupied).
Try $[hash(k) + 1] \bmod 10 = 1$ *(occupied)*
Try $[hash(k) + 2] \bmod 10 = 2$ *(occupied)*
Try $[hash(k) + 3] \bmod 10 = 3$ *(not occupied)*



Use same approach for key 50.

Resolving collisions by Open Addressing

Linear Probing Problem

Linear probing creates clusters (groups) in hash table

Searching in a hash table with linear probing stops if a slot is empty.

For example, searching for a not existing key, e.g. 70, linear probing stops after reaching slot *i=5*.

10	*0*
20	*1*
30	*2*
40	*3*
50	*4*
	5
	6
	7
58	*8*
68	*9*

hash(k)= 70 *mod 10* = 0, which is not 70.
Linear probing tries slot *i=1*, which isn't 70,
It tries until reaches slot *i=5*,which is empty and stops searching
The problem is that the hash table is not completely searched.

Resolving collisions by Open Addressing

Quadratic Probing

We look for i^2'th slot in i'th iteration.

If slot [*hash(k) + F(i)] mod m* is occupied, then we try
(hash(k) + 1) + $F(i^2)$ *] mod m*

If slot *hash(k)* mod m is full, then we try *(hash(k) + 1*1) mod m*

If *(hash(k) + 1*1*) mod m is also full, then we try
*(hash(k) + 2*2) mod m*

If *(hash(k) + 2*2)* mod m is also full, then we try
*(hash(k) + 3*3) mod m*

and so on!

Resolving collisions by Open Addressing

Quadratic Probing

Example: Let's take the hash function $h(k) = k \bmod m$

$k = (3, 5, 13, 24, 33)$

Hash Values:

hash(k) = 3 *mod 10* = 3

hash(k) = 5 *mod 10* = 5

hash(k) = 13 *mod 10* = 3, *collision*

Using QP; [hash(k) + F(i²)] mod m

*hash(k) = [3 + 1*1)] mod 10 = 4*

hash(k) = 24 *mod 10* = 4, *collision*

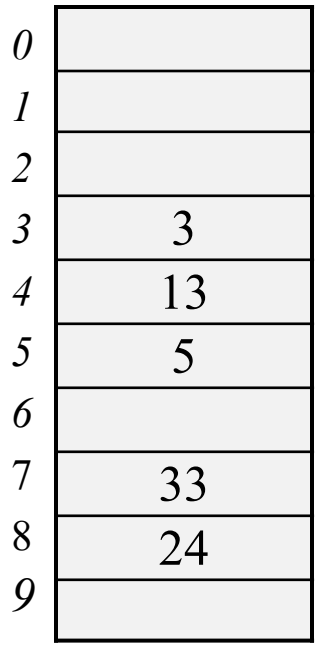
*hash(k) = [4 + 1*1)] mod 10 = 5, collision*

*hash(k) = [4 + 2*2)] mod 10 = 8*

hash(k) = 33 *mod 10* = 3, *collision*

*hash(k) = [3 + 1*1)] mod 10 = 4, collision*

*hash(k) = [3 + 2*2)] mod 10 = 7*



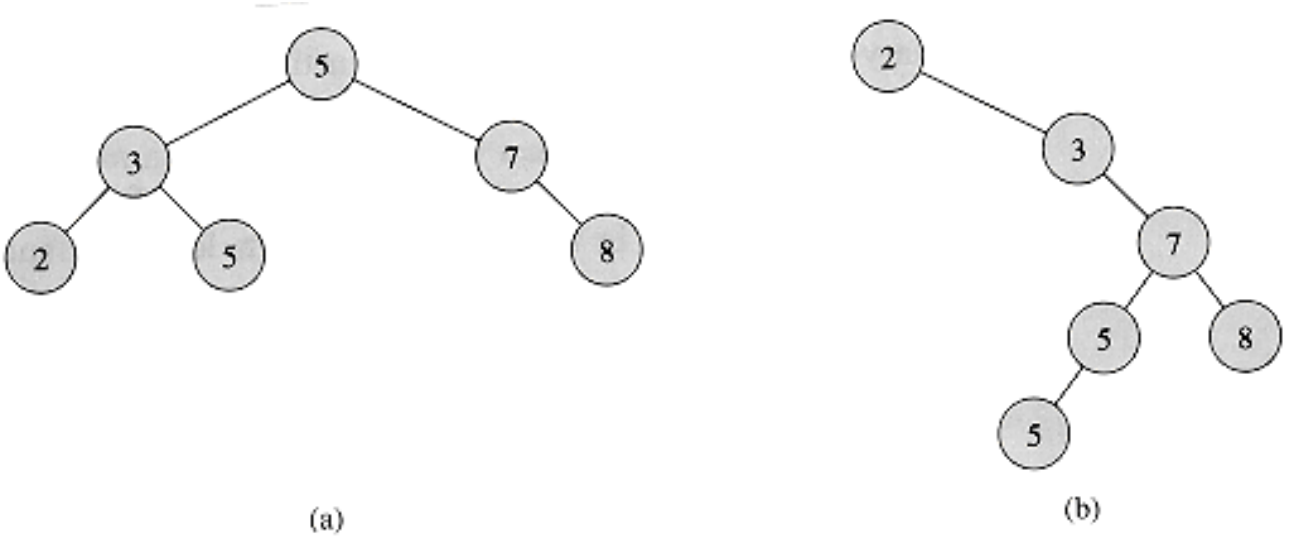
Binary Search Tree

•A binary search tree is organized, as the name suggests, in a binary tree

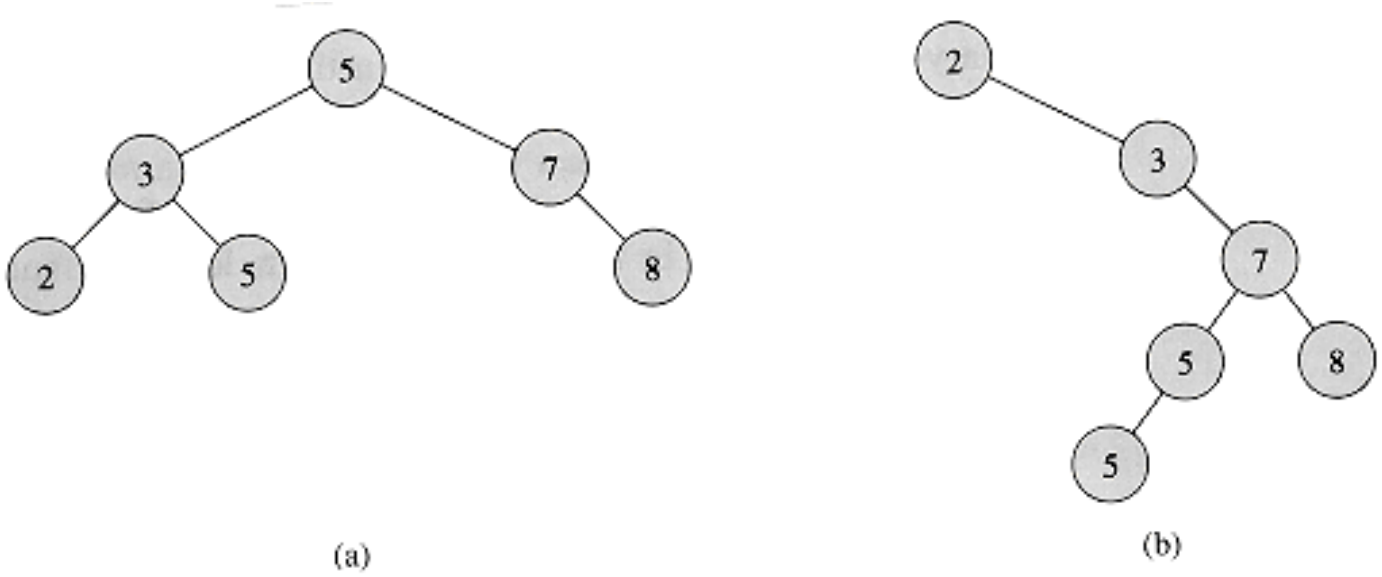
• Such a tree can be represented by a linked data structure in which each node is an object

• In addition to a *key* field, each node contains fields *left, right,* and *p* that point to the nodes corresponding to its left child, its right child, and its parent, respectively

• If a child or the parent is missing, the appropriate field contains the value `NIL`. The root node is the only node in the tree whose parent field is `NIL`.



Binary Search Tree

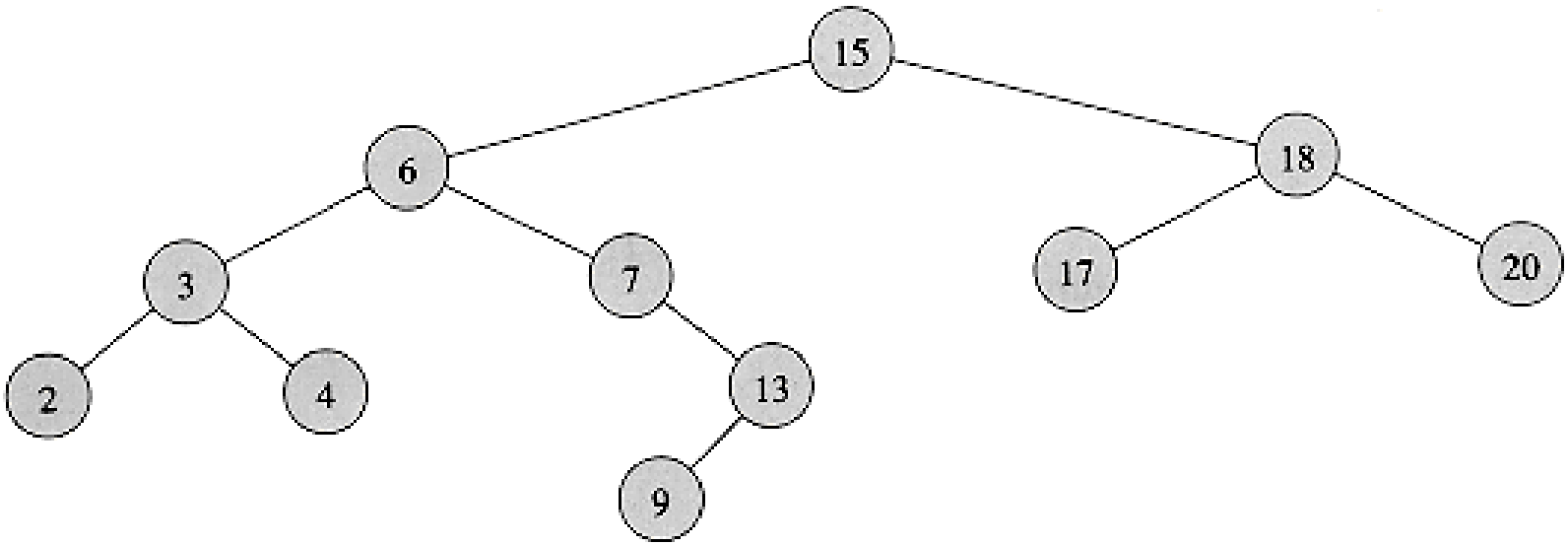


• In Figure above, the key of the root is 5, the keys 2, 3, and 5 in its left subtree are no larger than 5, and the keys 7 and 8 in its right subtree are no smaller than 5

• The same property holds for every node in the tree. For example in (b), the key 3 is no smaller than the key 2 in its left subtree and no larger than the key 5 in its right subtree

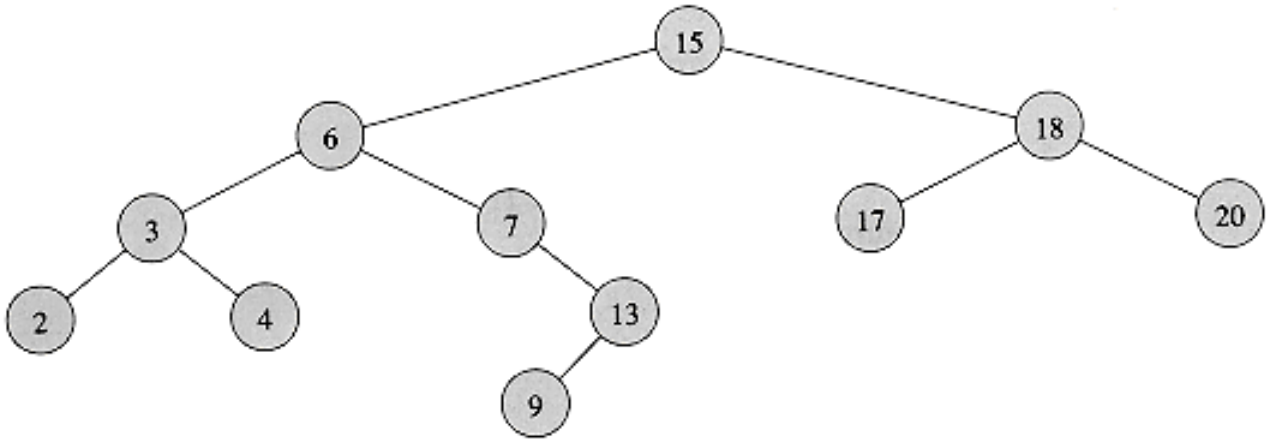
Binary Search Tree

- The most common operation performed on a binary search tree is searching for a key stored in the tree
- Besides the `SEARCH` operation, binary search trees can support such queries as `MINIMUM`, `MAXIMUM`, `SUCCESSOR`



Binary Search Tree

- **Searching**



•To search for the key 13 in the tree, the path 15 => 6 => 7 => 13 is followed from the root

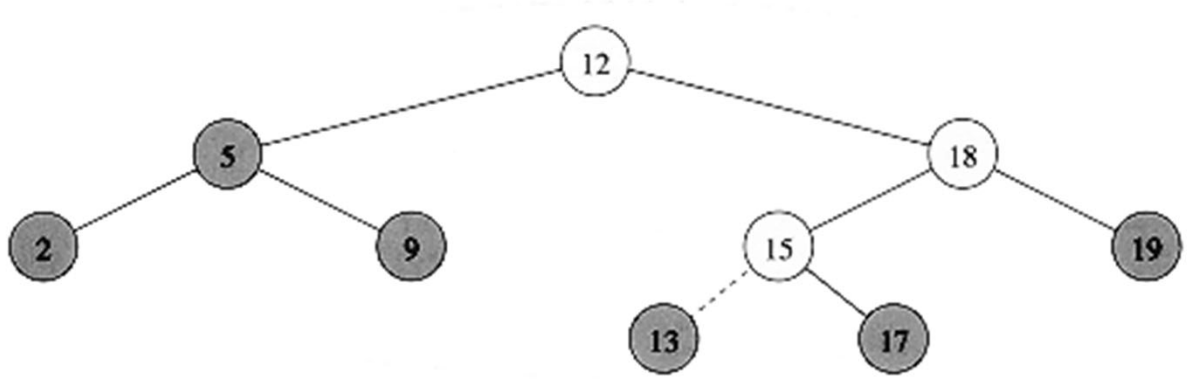
• The **minimum** key in the tree is 2, which can be found by following left pointers from the root

• The **maximum** key 20 is found by following right pointers from the root

• The **successor** of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15

Binary Search Tree

- **Insertion**

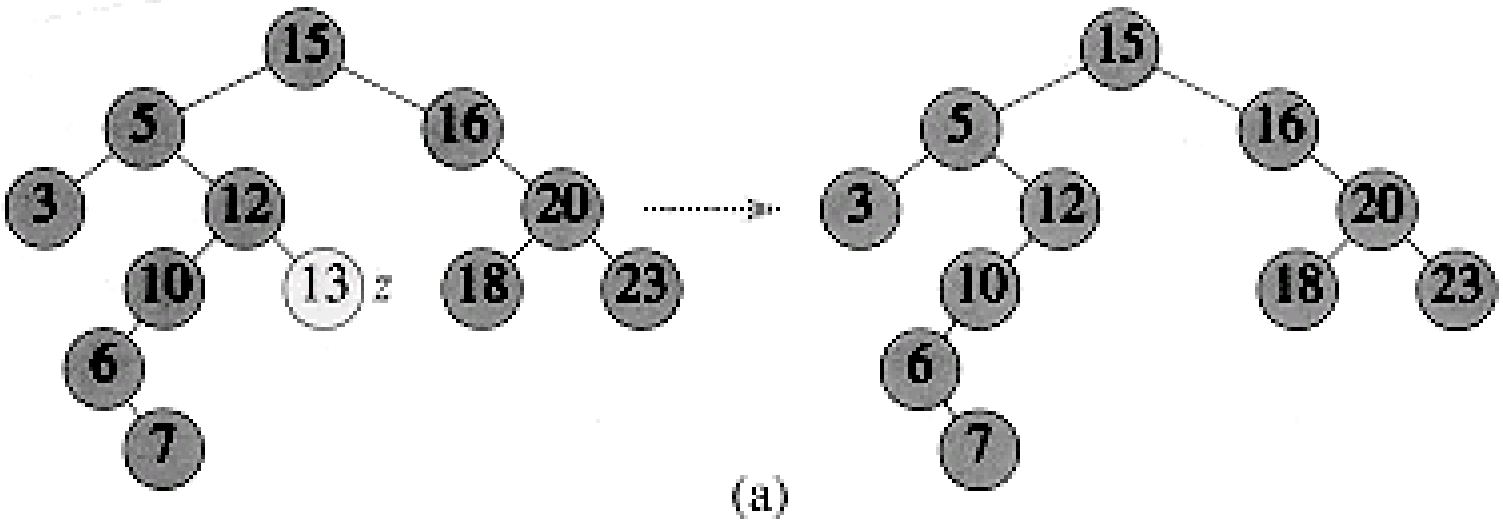


- The pointer x traces the path, and the pointer y is maintained as the parent of x
- After initialization, the **while** loop in lines 3-7 causes these two pointers to move down the tree, going left or right depending on the comparison of $key[z]$ with $key[x]$, until x is set to `NIL`
- This `NIL` occupies the position where we wish to place the input item z. Lines 8-13 set the pointers that cause z to be inserted

```
TREE-INSERT(T,z)
1    y ← NIL
2    x ← root[T]
3       while x ≠ NIL
4       do y ← x
5  if key[z] < key[x]
6       then x ← left[x]
7       else x ← right[x]
8       p[z] ← y
9  if y = NIL
10         then root[T] ← z
11         else if key[z] < key[y]
12         then left[y] ← z
13         else right[y] ← z
```

Binary Search Tree

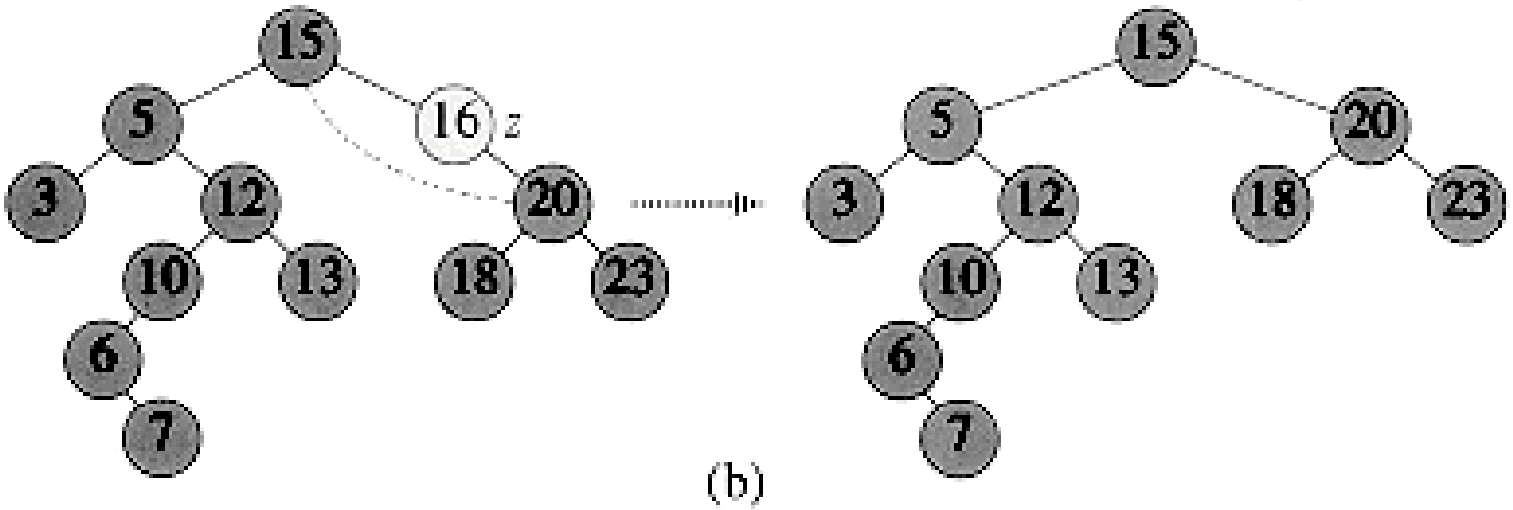
- **Deletion**
  - Deleting a node z from a binary search tree. the node actually removed is lightly shaded
  - 1) If z has no children, we just remove it



(a)

Binary Search Tree

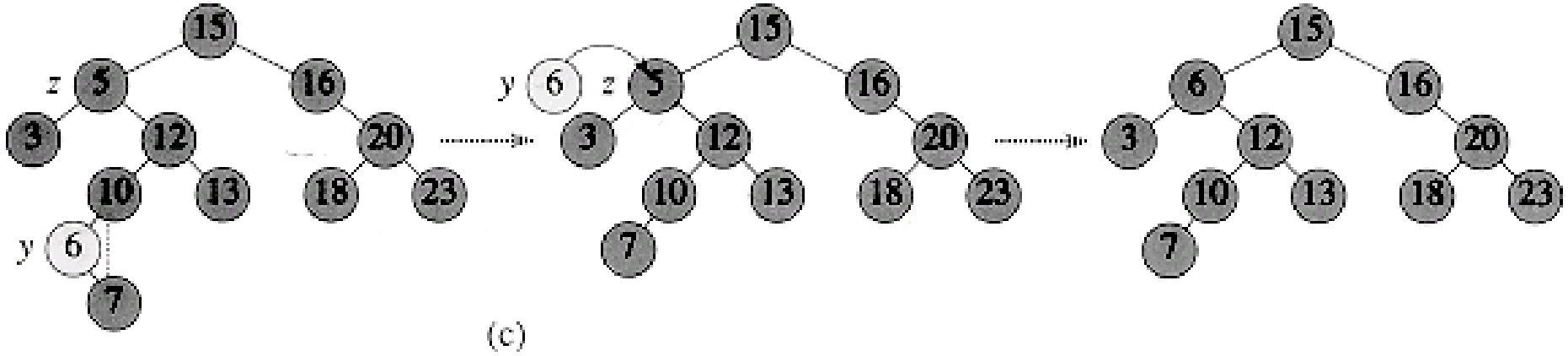
- **Deletion**
  - Deleting a node z from a binary search tree. the node actually removed is lightly shaded
  - 2) If z has only one child, we splice out z



(b)

Binary Search Tree

- **Deletion**
  - Deleting a node z from a binary search tree. the node actually removed is lightly shaded
  - 3) If z has two children, we splice out its successor y, which has at most one child, and then replace the contents of z with the contents of y

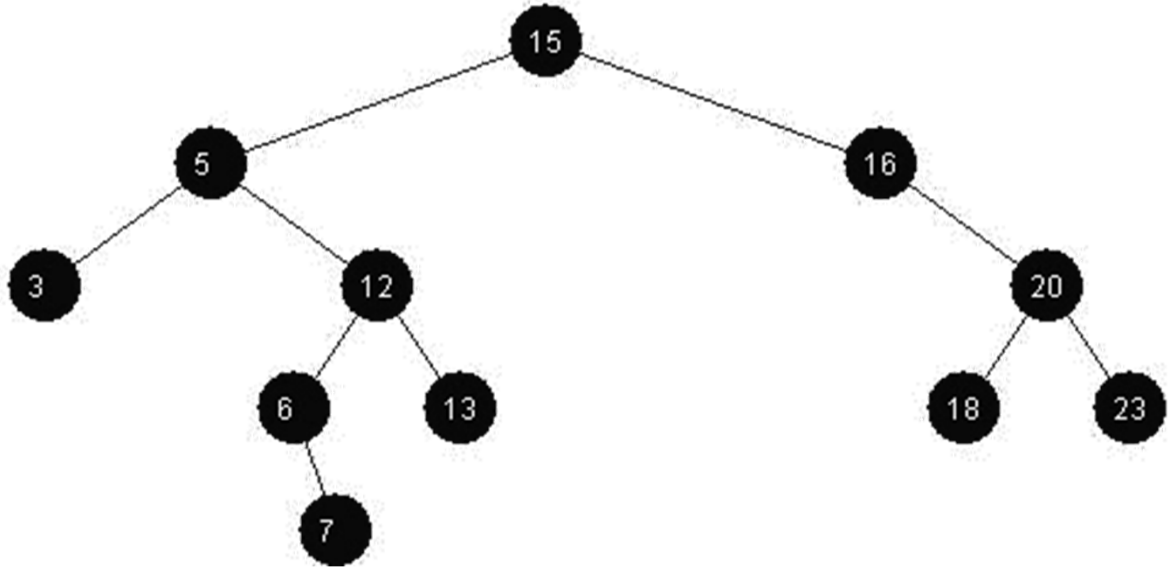


(c)

Assignment

•Binary Search Tree

- Build a BST with the given array; 15, 5, 16, 3, 12, 20, 6, 13, 18, 23, 7



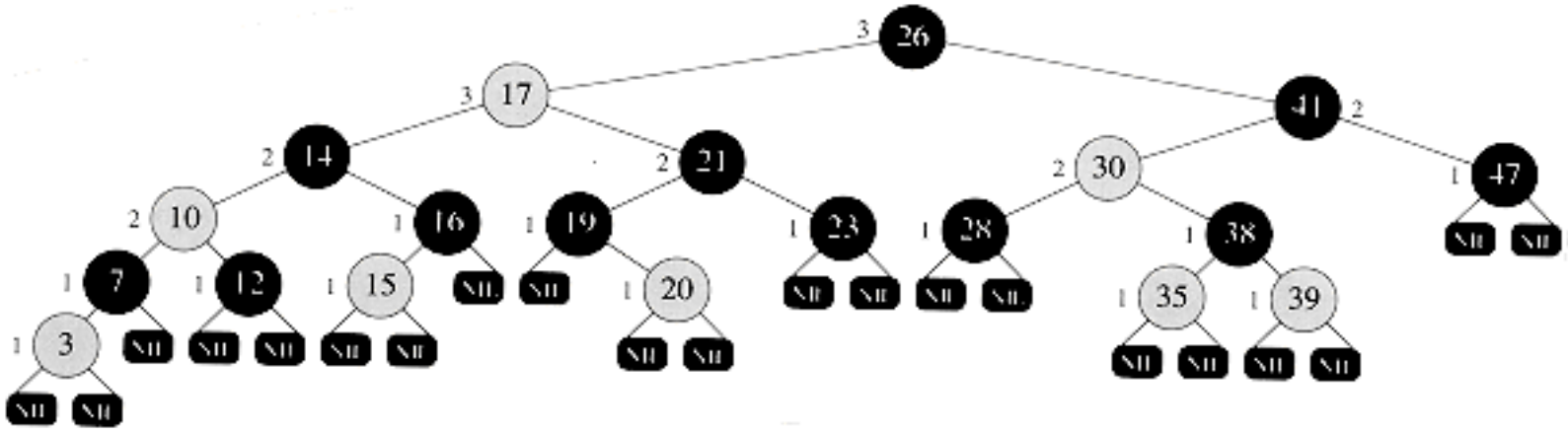
Red and Black Tree

• A **red-black tree** is a binary search tree with one extra bit of storage per node: its **color**, which can be either RED or BLACK
• By constraining the way nodes can be colored on any path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately **balanced**
• height is O(lg n) ,where n is the number of nodes.
• Operations will take O(lg n) time in the worst case.
• Each node of the tree now contains the fields color, key, left, right, and p
• If a child or the parent of a node does not exist, the corresponding pointer field of the node contains the value NIL
• We shall regard these NIL'S as being pointers to external nodes (leaves) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree

Red and Black Tree

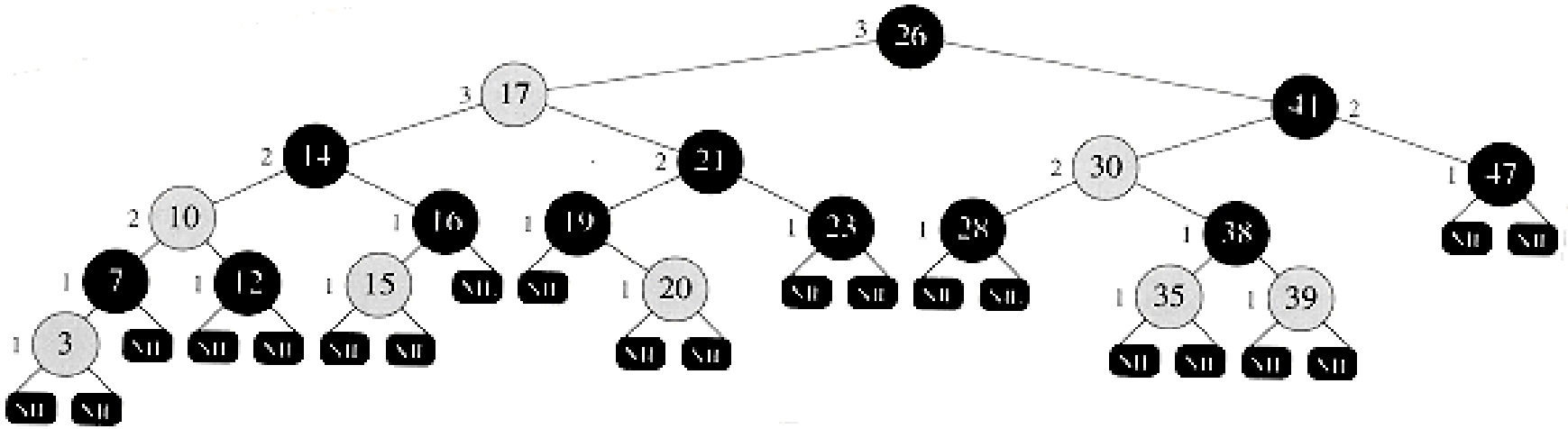
• A binary search tree is a red-black tree if it satisfies the following **red-black properties:**

1. Every node is either red or black.
2. Every leaf (`NIL`) is black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes
5. Nodes with leafs can be any color



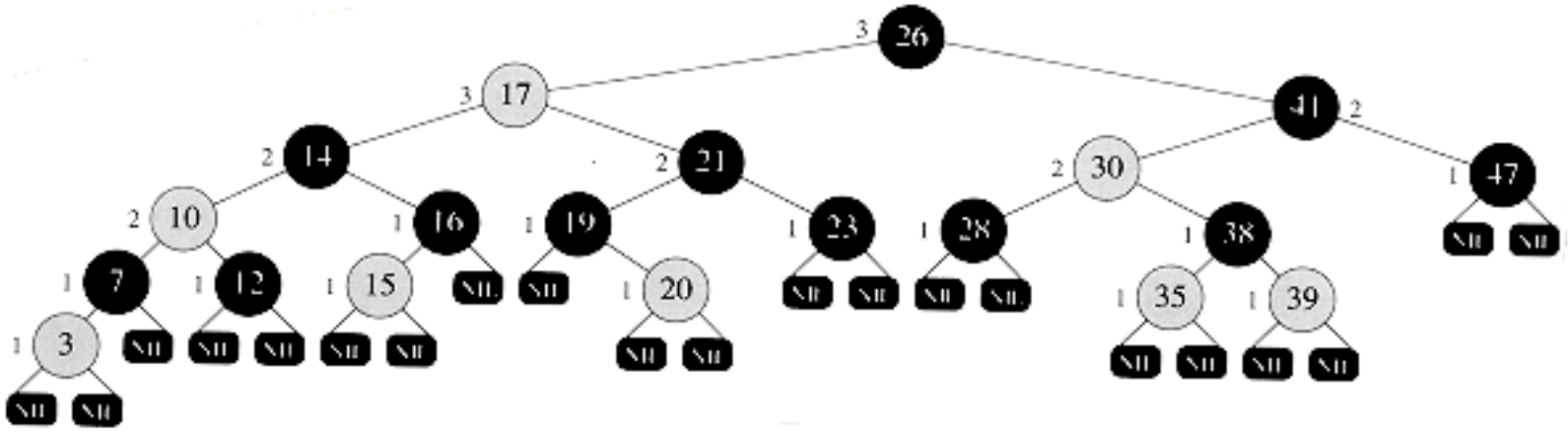
Red and Black Tree

- We call the number of black nodes on any path from, but not including, a node x to a leaf the black-height of the node, denoted bh(x).
- By property 4, the notion of black-height is well defined, since all descending paths from the node have the same number of black nodes.
- We define the black-height of a red-black tree to be the black-height of its root



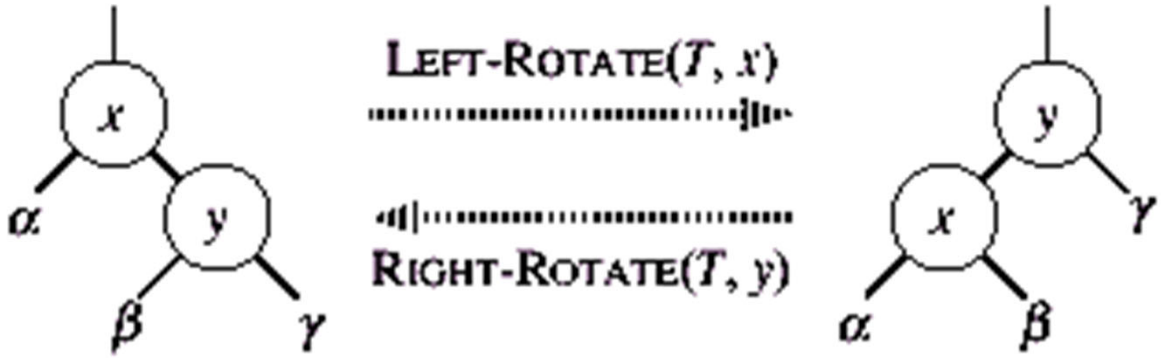
Operations on Red and Black Trees

- The non-modifying binary-search-tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in O height time.
- They take O (lg n) time on red-black trees.
- Insertion and deletion are not so easy.
- If we insert, what colour to make the new node?
- Red? Might violate property 4.
- Black? Might violate property 5

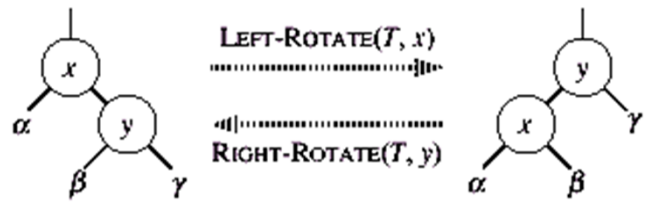


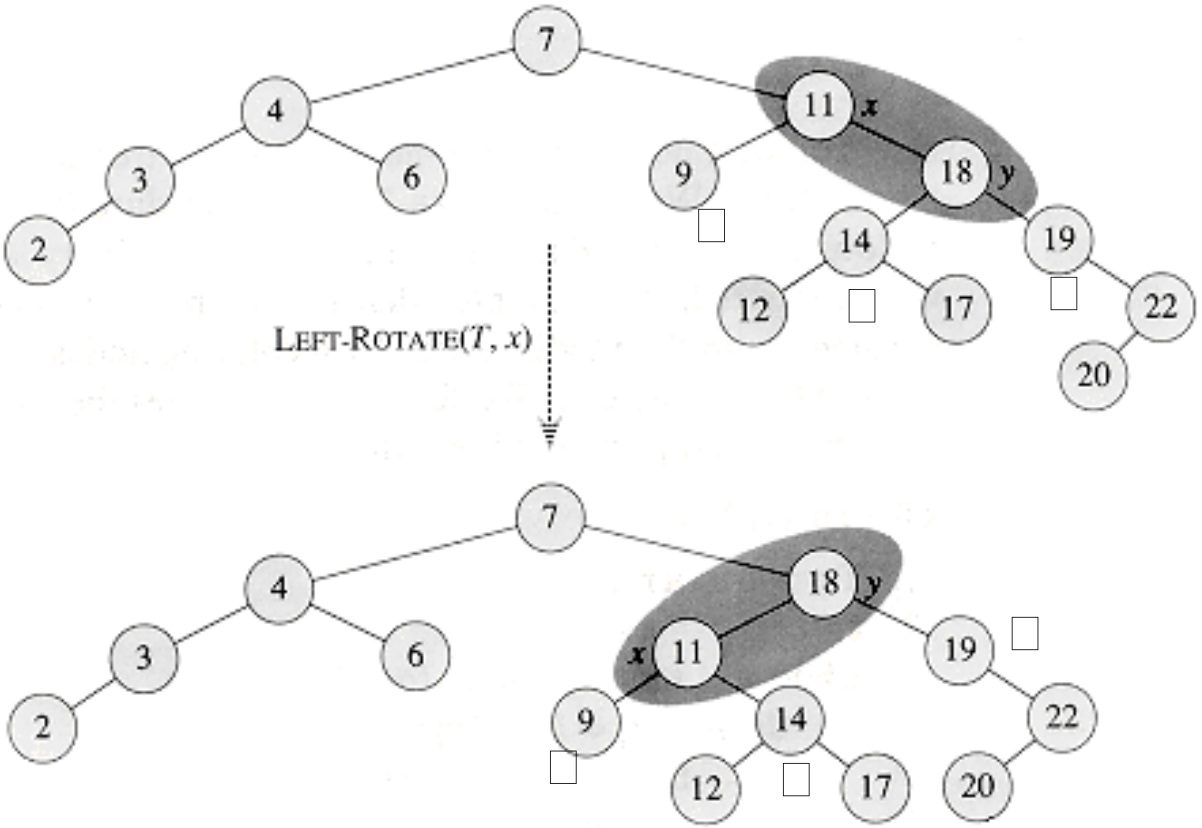
Rotations

- The basic tree-restructuring operation.
- Needed to maintain red-black trees as balanced binary search trees.
- Changes the local pointer structure. (Only pointers are changed.)
- Won't upset the binary-search-tree property.
- Have both left rotation and right rotation. They are inverses of each other.
- A rotation takes a red-black-tree and a node within the tree.



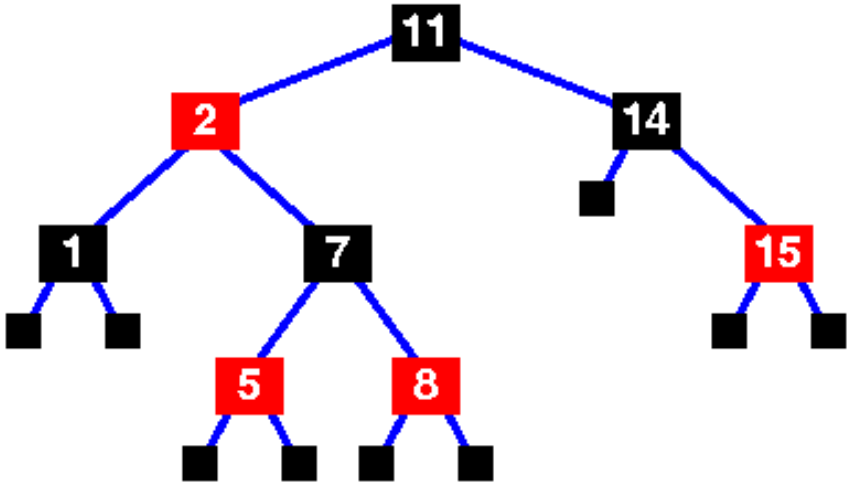
Rotations

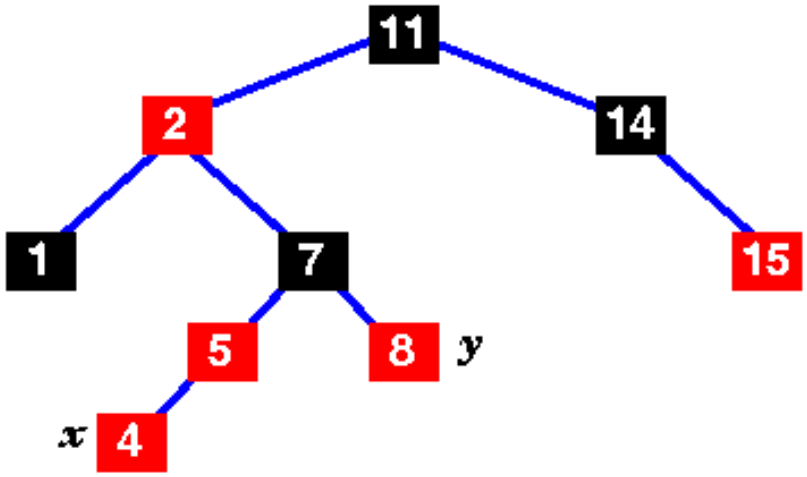

LEFT-ROTATE(T, x)
RIGHT-ROTATE(T, y)
x
y
α
β
γ


LEFT-ROTATE(T, x)
7
4
11
x
3
6
9
18
y
2
14
19
12
17
22
20
7
4
18
y
3
6
x
11
19
2
9
14
22
12
17
20

Insertion

• The tree insert routine has just been called to insert node "4" into the tree



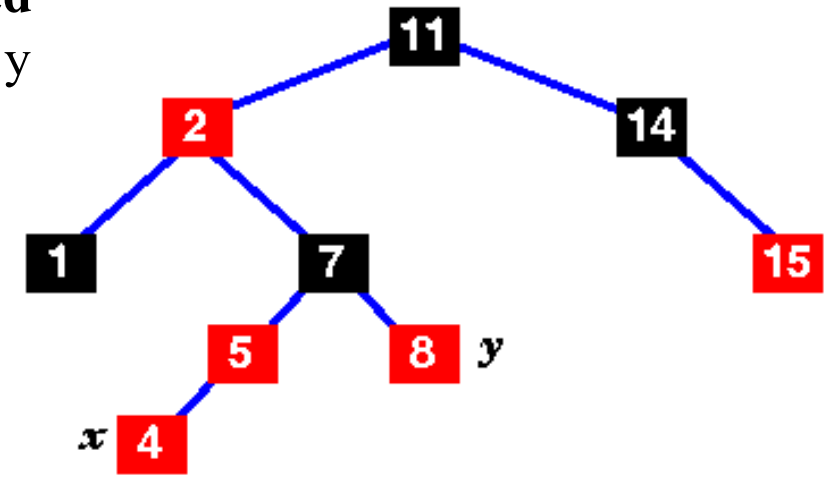


• This is no longer a red-black tree - there are two successive red nodes 5-4 VIOLATION

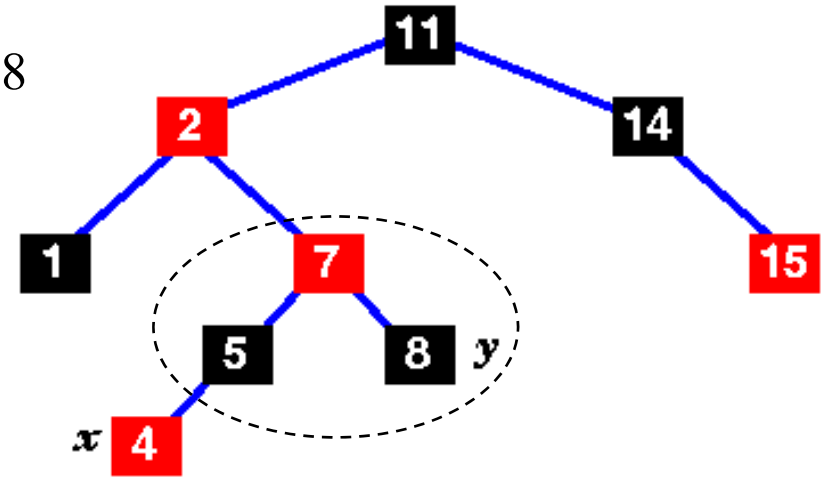
Insertion

- We have three cases

- **Case 1, y (uncle of inserted node x) is red**
  - Mark the new node, x, and it's **uncle**, y
  - y is **red**, VIOLATION

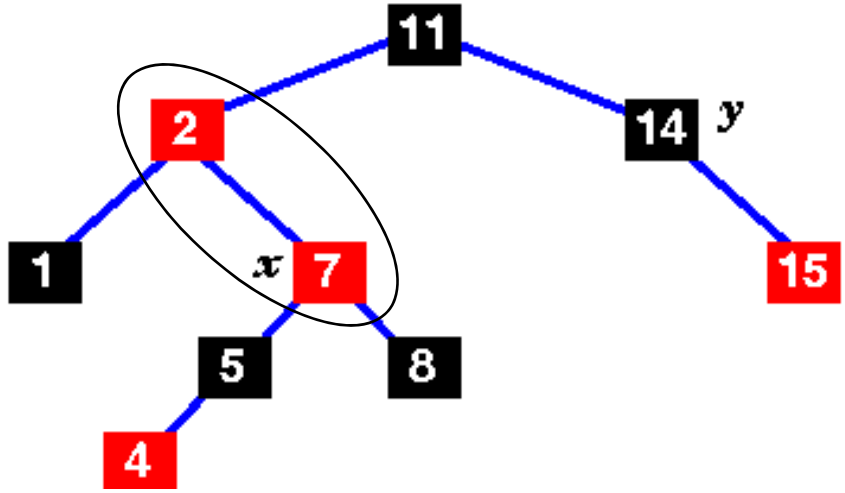


- **Change the colors** of nodes 5, 7 and 8
- (Parent, grandparent, and uncle)

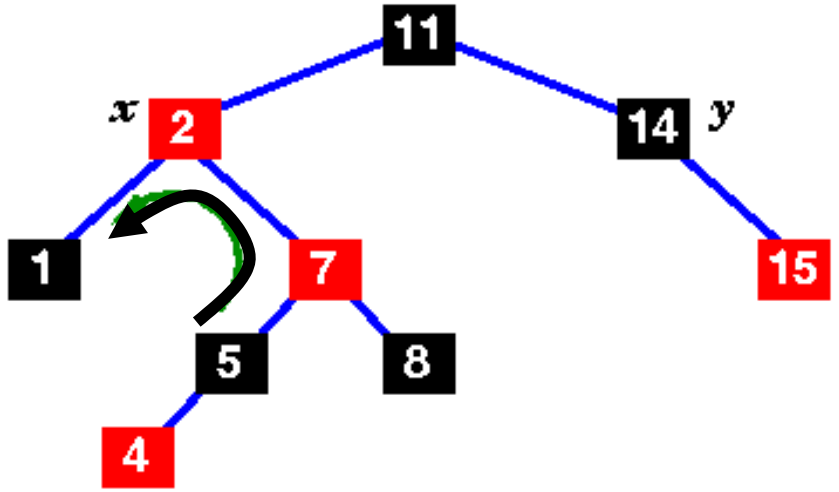


Insertion

- Move x up to its grandparent, 7
- there are two successive red nodes 2 – 7
- VIOLATION
- Find the uncle y, 14
- **Case 2,** y (uncle) is black

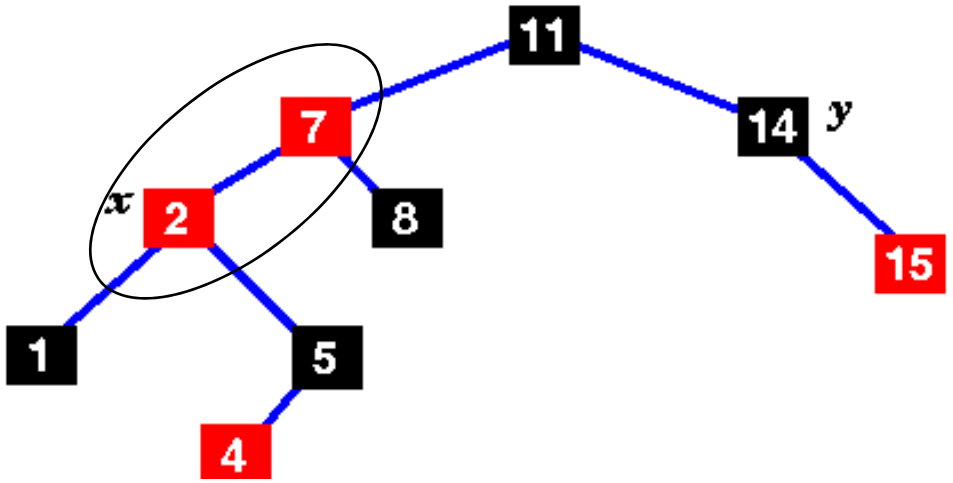


- Move x up and **rotate left**

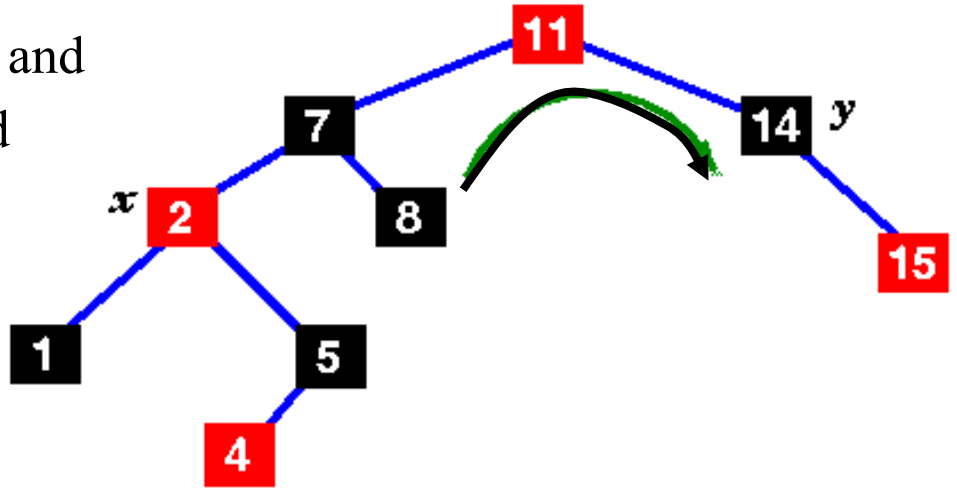


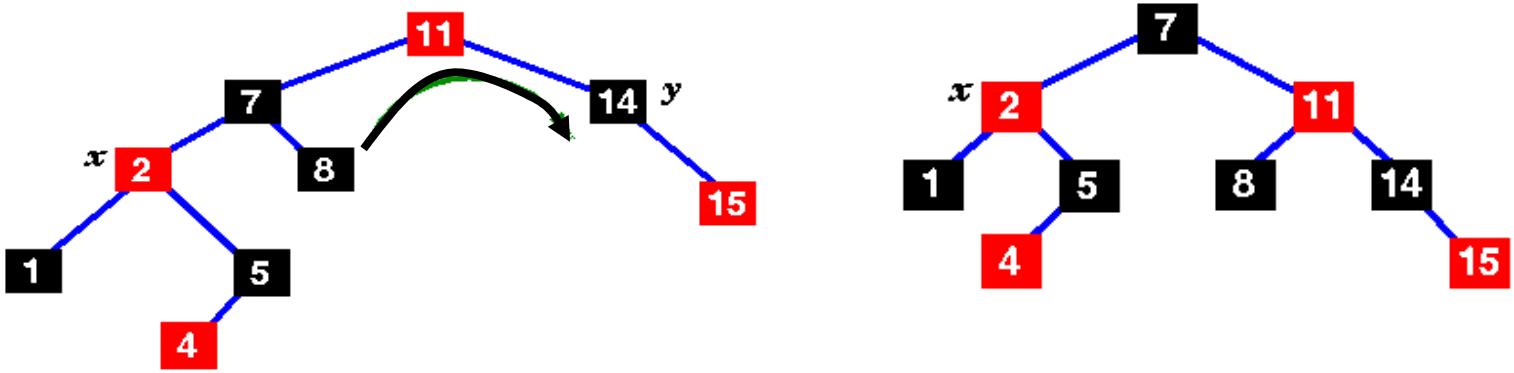
Insertion

- y is black, x is a left child, **case 3**
- VIOLATION



- **Change the colours** of 7 and 11(Parent and grandparent) and **rotate right** around the grandparent





Insertion Summary

- **Case 1 and 1A:** y (uncle of inserted node x) is red > Change the colors of nodes (Parent, grandparent, and uncle)
- **Case 2:** y is black and x is a right child > rotate left
- **Case 2A:** y is black and x is a left child > rotate right
- **Case 3:** y is black and x is a left child > Change the colours of Parent and grandparent and rotate right
- **Case 3A:** y is black and x is a right child > Change the colours of Parent and grandparent and rotate left

Delete of “ordinary” BST

- If node to be deleted is a leaf, just delete it.
- If node to be deleted has just one child, replace it with that child
- If node to be deleted has two children, replace the **value** in the node by its in-order predecessor/successor’s value then delete the in-order predecessor/successor (a recursive step)

Bottom-Up Deletion

- Do ordinary BST deletion. Eventually a “case 1” or “case 2” deletion will be done (leaf or just one child).

- If deleted node, U, is a leaf, think of deletion as replacing U with the NULL pointer, V.

- If U had one child, V, think of deletion as replacing U with V.

Which RB Property may be violated after deletion?

- If U is Red?

  Not a problem – no RB properties violated

- If U is Black?

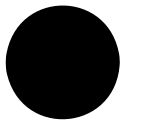
  If U is not the root, deleting it will change the black-height along some path

Fixing the problem

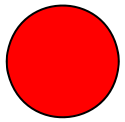
- There are four cases – our examples and "rules" assume that V is a left child. There are symmetric cases for V as a right child.

Terminology

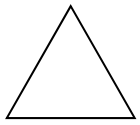
- The node just deleted was U
- The node that replaces it is V
- The parent of V is P
- The sibling of V is S



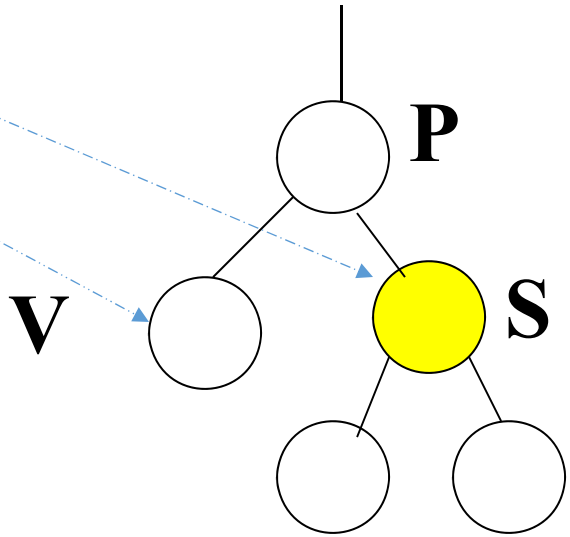
Black Node



Red Node



Red or Black
that we don't care

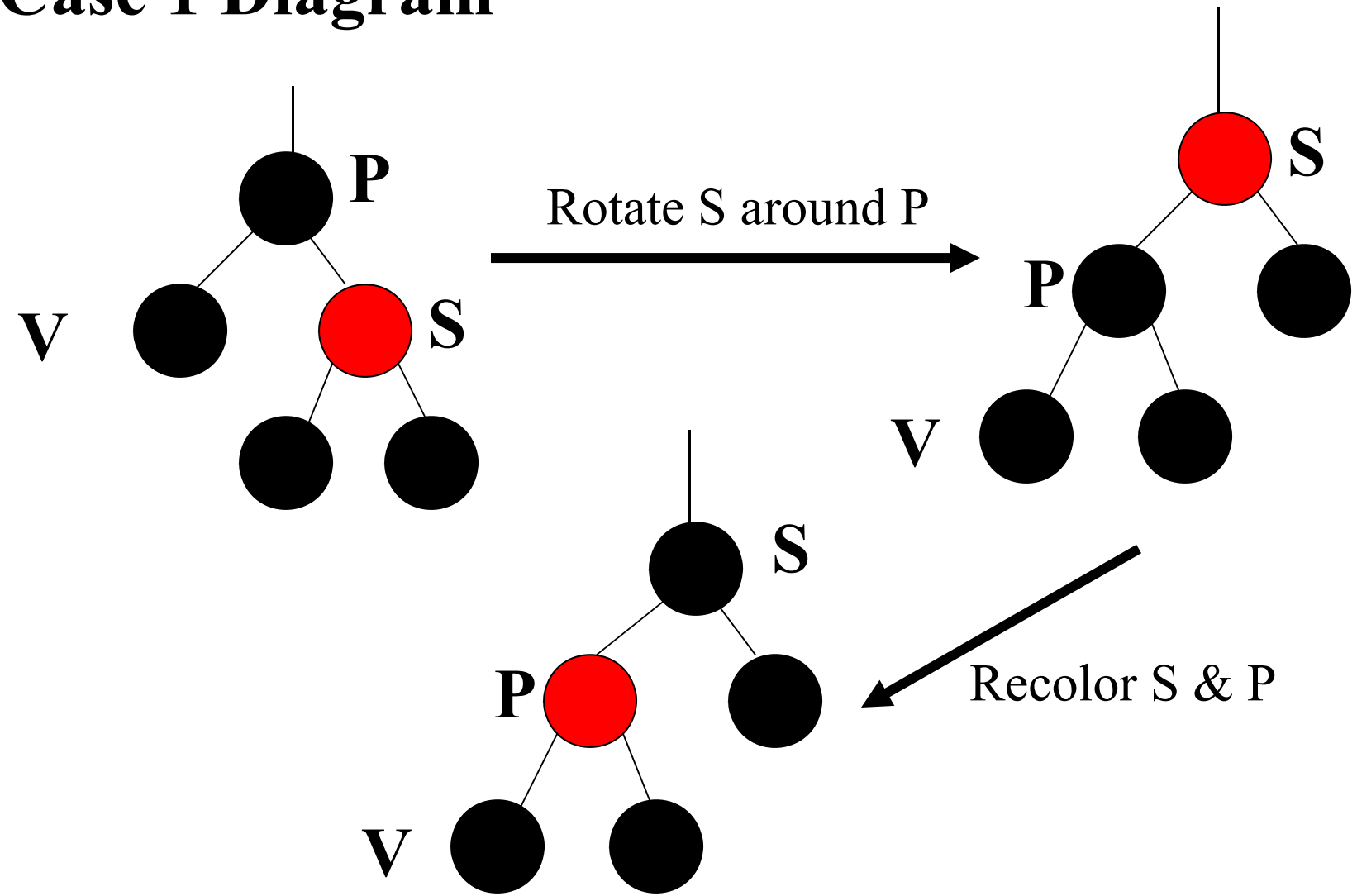


Bottom-Up Deletion Case 1

- V's sibling, S, is Red
  - Rotate S around P and recolor S & P

- NOT a terminal case – One of the other cases will now apply
- All other cases apply when S is Black

Case 1 Diagram

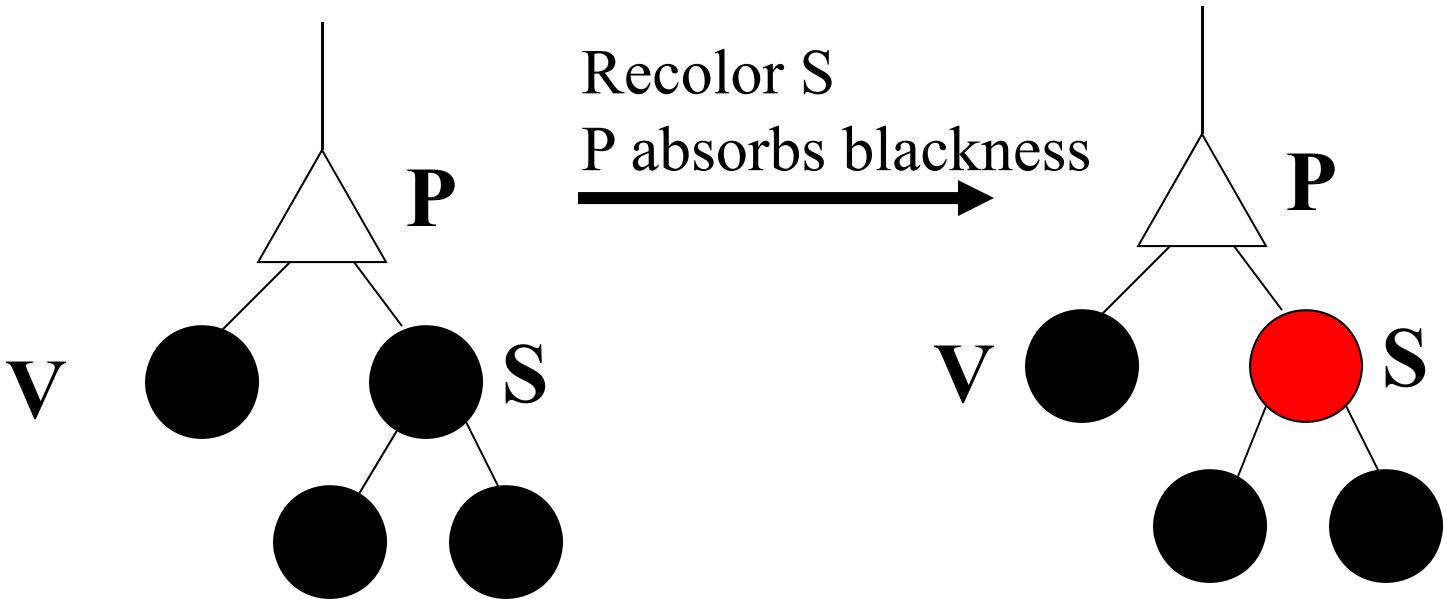


Bottom-Up Deletion Case 2

- V's sibling, S, is Black and has two Black children.
  - Recolor S to be Red

    - If P is Red, we're done
    - If P is Black, stays black

Case 2 diagram

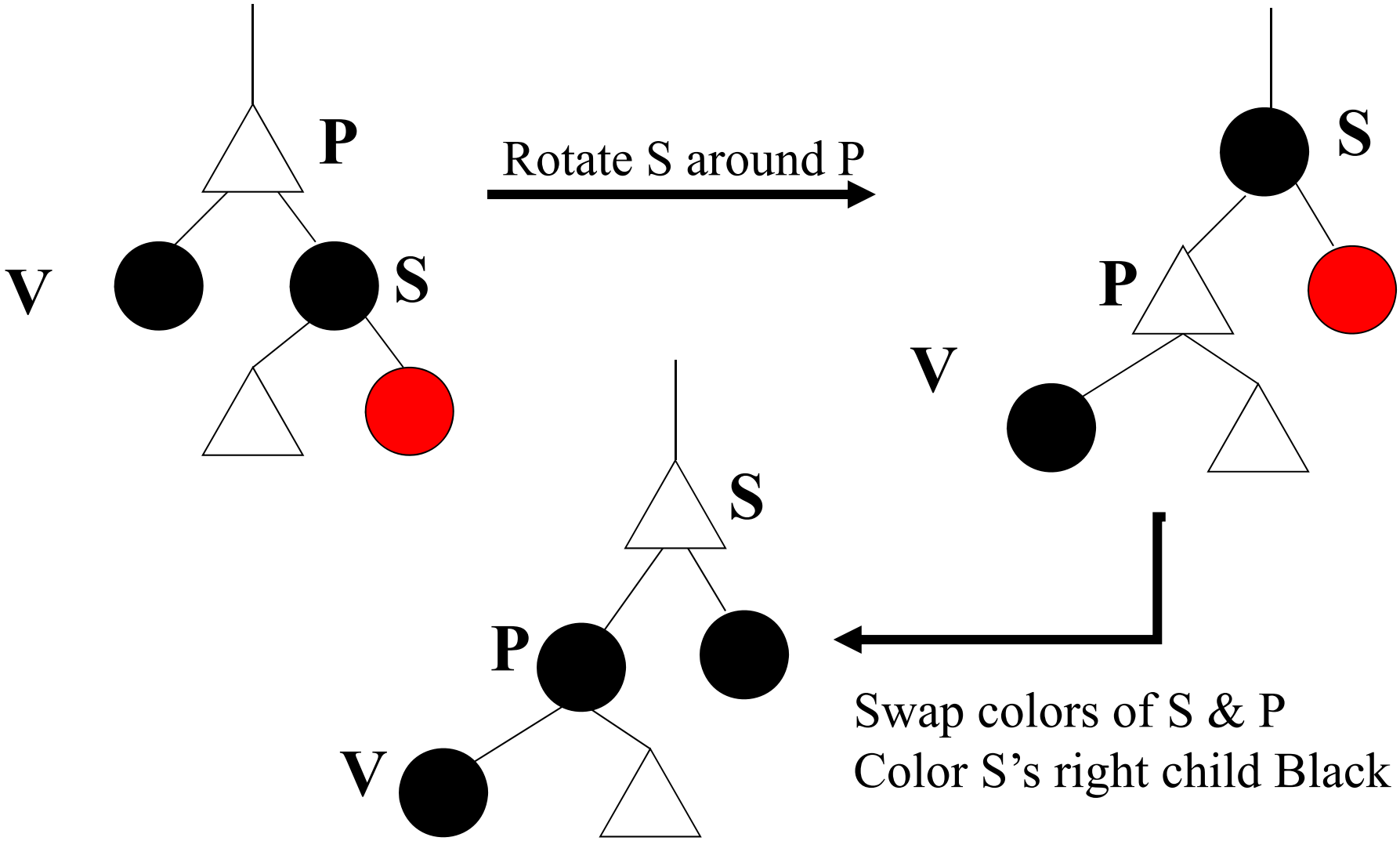


Bottom-Up Deletion Case 3

- S is Black
- S's right child is RED (Left child either color)
  - Rotate S around P
  - Swap colors of S and P,
    and color S's right child Black

- This is the terminal case – we're done

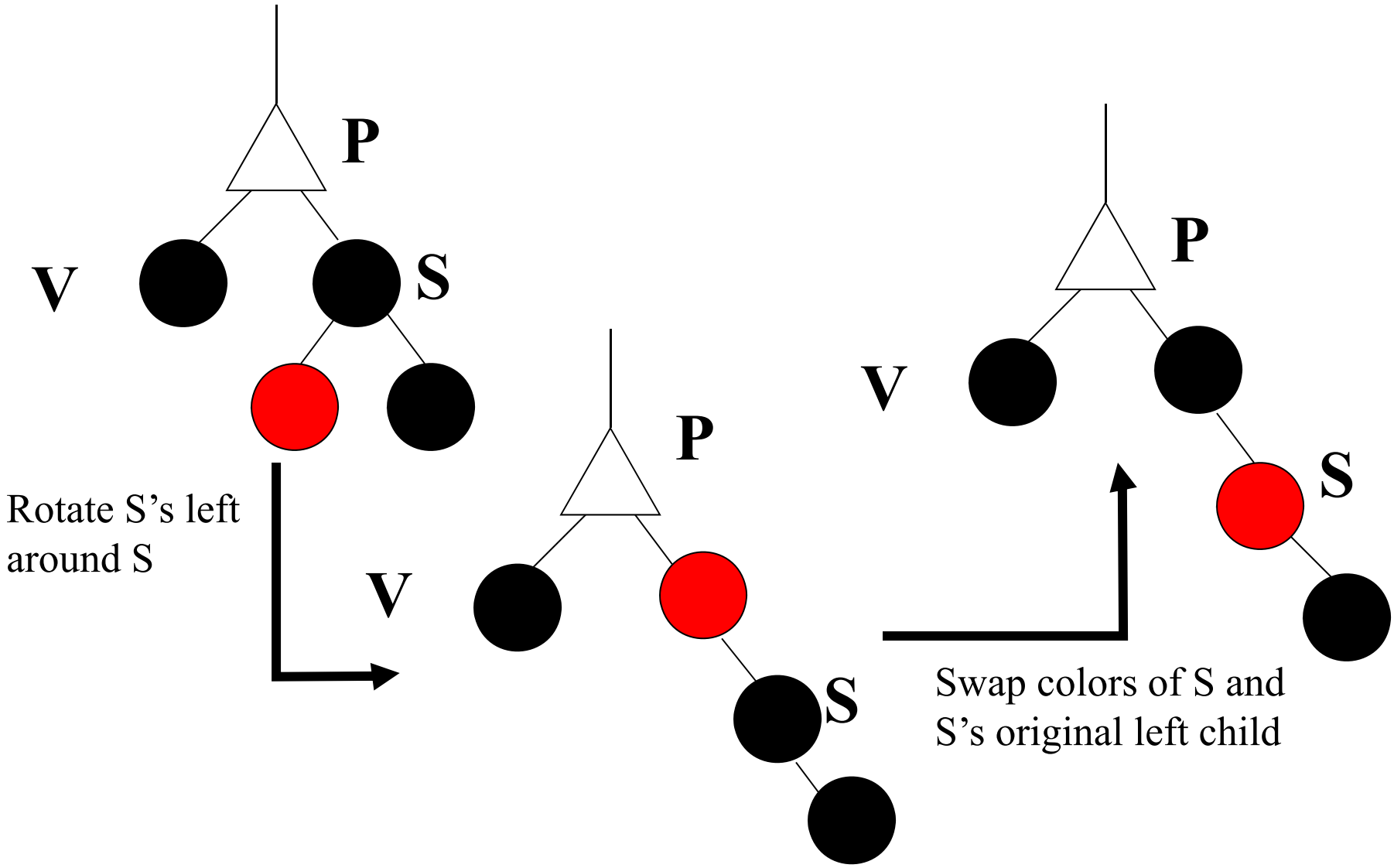
Case 3 diagrams



Bottom-Up Deletion Case 4

- S is Black, S's right child is Black and S's left child is Red
  - Rotate S's left child around S
  - Swap color of S and S's left child
  - Now sibling of V is changed and has red right child (case 3)

Case 4 Diagrams



Assignment for Red&Black Tree, Insertion

Write all the steps and cases for
50, 40, 30, 20, 70, 90, 80, 10, 60, 40, 65, 45, 5

ELEMENTARY GRAPH ALGORITHMS

•Definitions

•A graph, G, consists of two sets V and E. V is a finite non-empty set of ***vertices***. E is a set of pairs of vertices, these pairs are called ***edges***.

• $V(G)$ and $E(G)$ will represent the sets of vertices and edges of graph G. We will also write $G = (V,E)$ to represent a graph.

• In an *undirected graph* the pair of vertices representing any edge is unordered

• Thus, the pairs (v_1, v_2) and (v_2, v_1) represent the same edge

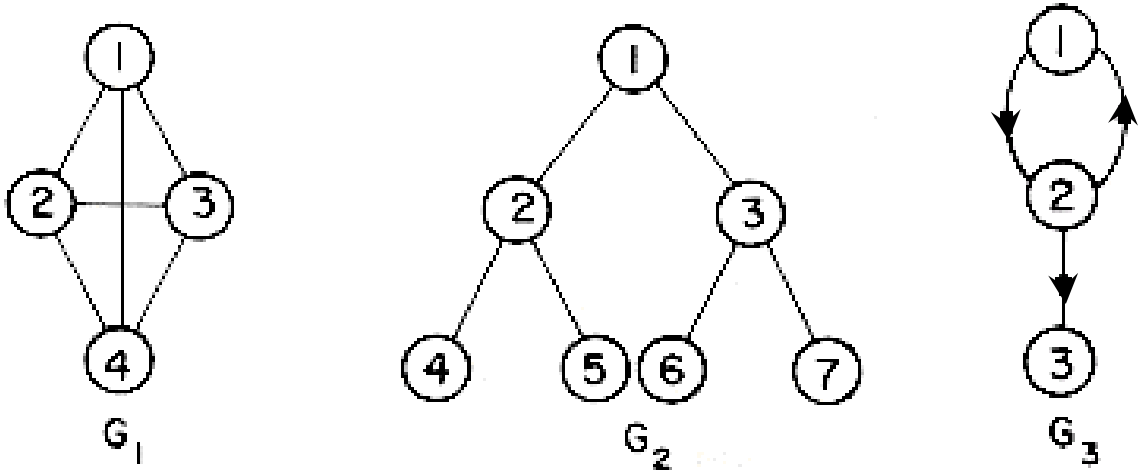
• In a *directed graph* each edge is represented by a directed pair (v_1, v_2), v_1 is the *tail* and v_2 the *head* of the edge

• Therefore $<v_2, v_1>$ and $<v_1, v_2>$ represent two different edges

ELEMENTARY GRAPH ALGORITHMS

•Definitions

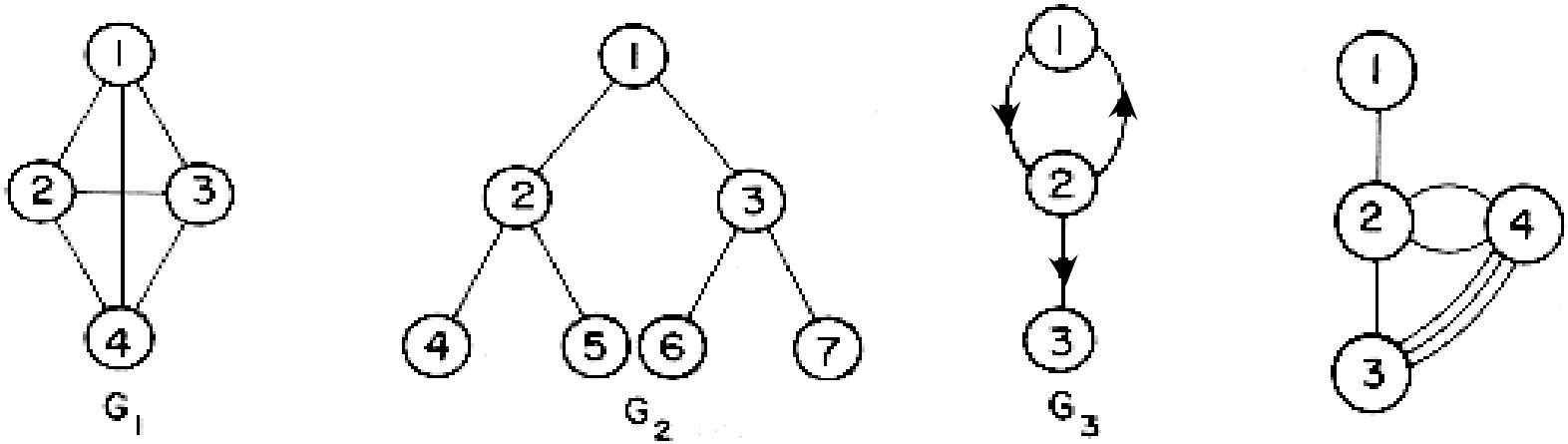
• Figure below shows three graphs G_1, G_2 and G_3



• The graphs G_1 and G_2 are undirected. G_3 is a directed graph, Note that the edges of a directed graph are drawn with an arrow from the tail to the head.

- $V(G_1) = \{1,2,3,4\}$; $E(G_1) = \{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$
- $V(G_2) = \{1,2,3,4,5,6,7\}$; $E(G_2) = \{(1,2),(1,3),(2,4),(2,5),(3,6),(3,7)\}$
- $V(G_3) = \{1,2,3\}$; $E(G_3) = \{<1,2>, <2,1>, <2,3>\}$.

ELEMENTARY GRAPH ALGORITHMS



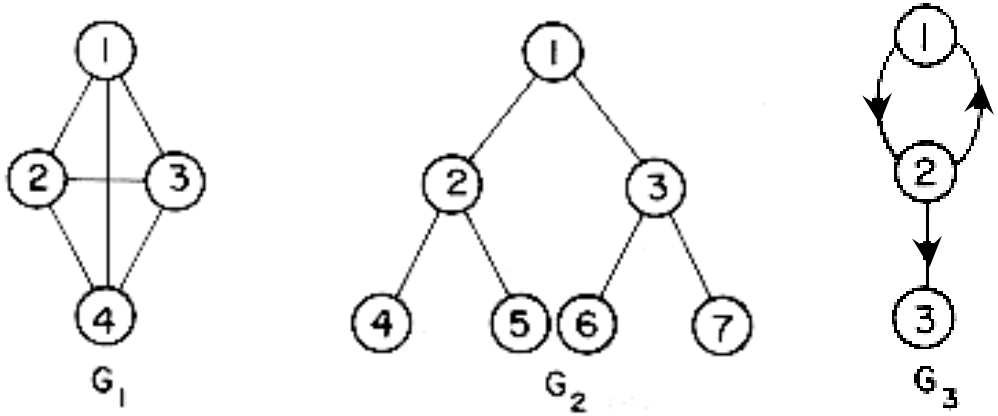
- The graph G_2 is also a tree while the graphs G_1 and G_3 are not.
- Trees can be defined as a special case of graphs
- If (v_1,v_2) or $<v_1,v_2>$ is an edge in $E(G)$, then we require $v_1 \neq v_2$. In addition, since $E(G)$ is a set, a graph may not have multiple occurrences of the same edge.

ELEMENTARY GRAPH ALGORITHMS

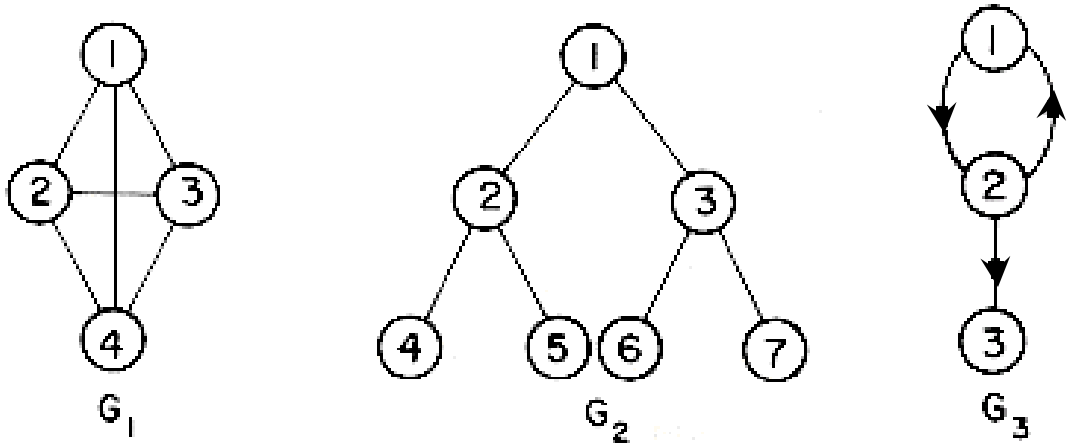
• The number edges (v_i,v_j) with $v_i \neq v_j$ in a graph with n vertices is n(n - 1)/2. This is the maximum number of edges in any n vertex **undirected graph**.

•An n vertex undirected graph with exactly n(n - 1)/2 edges is said to be **complete**.

• G_1 is the complete graph on 4 vertices while G_2 and G_3 are not complete graphs. In the case of a **directed graph** on n vertices the maximum number of edges is n(n - 1).



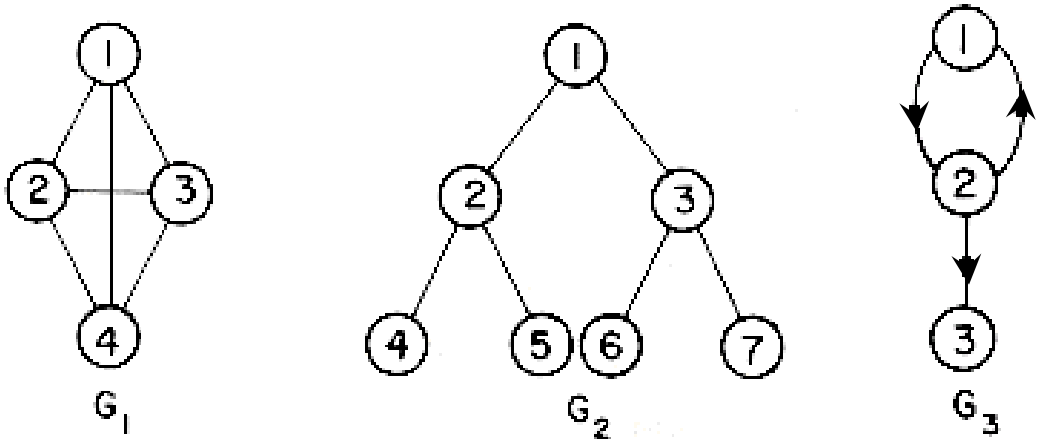
ELEMENTARY GRAPH ALGORITHMS

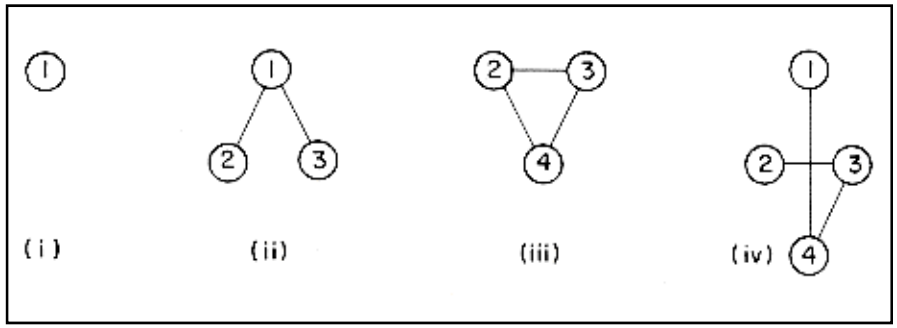


- If (v_1,v_2) is an edge in $E(G)$, then we shall say the vertices v_1 and v_2 are ***adjacent*** and that the edge (v_1,v_2) is ***incident*** on vertices v_1 and v_2
- The vertices adjacent to vertex 2 in G_2 are 4, 5 and 1
- The edges incident on vertex 3 in G_2 are (1,3), (3,6) and (3,7), ….
- If $<v_1,v_2>$ is a directed edge, then vertex v_1 will be said to be *adjacent* ***to*** v_2 while v_2 is *adjacent* ***from*** v_1
- The edge $<v_1,v_2>$ is incident to $v1$ and v_2. In $G3$ the edges incident to vertex 2 are <1,2>, <2,1> and <2,3>

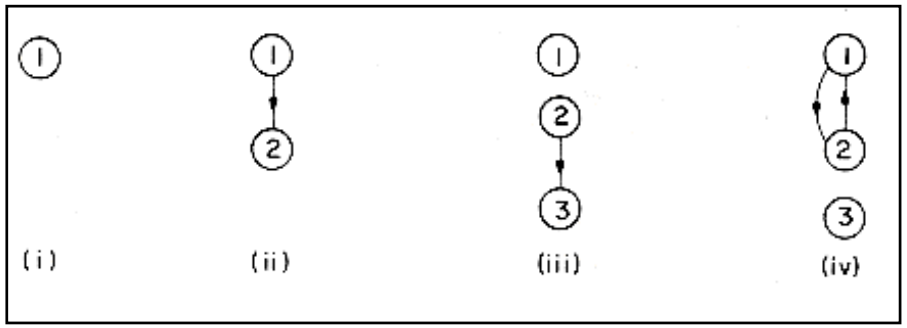
ELEMENTARY GRAPH ALGORITHMS

• A *subgraph* of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. Figure below shows some of the subgraphs of G_1 and G_3





Some of the subgraphs of G_1



Some of the subgraphs of G_3

ELEMENTARY GRAPH ALGORITHMS

• A **path** from vertex v_p to vertex v_q in graph G is a sequence of vertices $v_p, v_{i1}, v_{i2}, \ldots, v_{in}, v_q$ such that $(v_p,v_{i1}),(v_{i1},v_{i2}), \ldots,(v_{in},v_q)$ are edges in E(G)

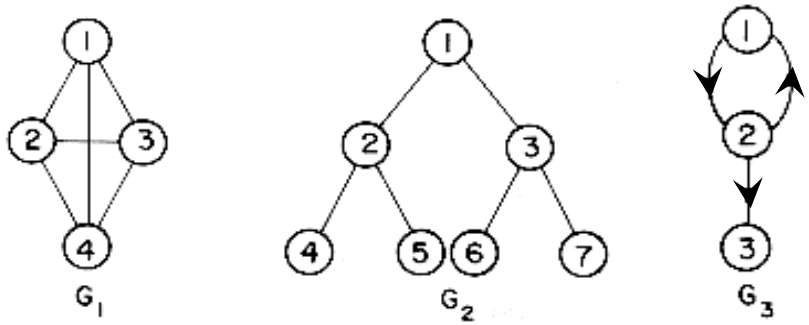
• If G' is directed then the path consists of $\langle v_p,v_{i1}\rangle, \langle v_i,v_{i2}\rangle, \ldots, \langle v_{in},v_q\rangle$, edges in E(G') the length of a path is the number of edges on it.

• A simple path is a path in which all vertices except possibly the first and last are distinct.

• A path such as (1,2) (2,4) (4,3) we write as 1,2,4,3. Paths 1,2,4,3 and 1,2,4,2 are both of length 3 in G_1. The 1,2,4,3 is a simple path while the 1,2,4,2 is not

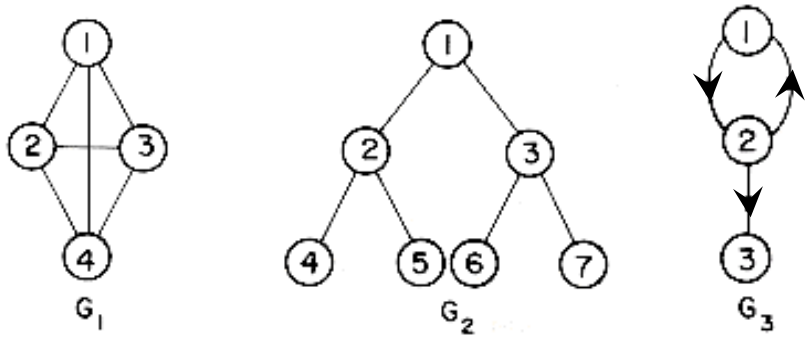
• 1,2,3 is a simple directed path in G_3. 1,2,3,2 is not a path in G_3 as the edge $\langle 3,2\rangle$ is not in $E(G_3)$.

•A **cycle** is a simple path in which the first and last vertices are the same. 1,2,3,1 is a cycle in G_1. 1,2,1 is a cycle in G_3



ELEMENTARY GRAPH ALGORITHMS

- A directed graph G is said to be **strongly connected** if for every pair of distinct vertices v_i, v_j in V(G) there is a directed path from v_i to v_j and also from v_j to v_i.
- The graph G_3 is not strongly connected as there is no path from v_3 to v_2



- The **degree** of a vertex is the number of edges incident to that vertex.
- The degree of vertex v1 in G_1 is 3. In case G is a directed graph, we define the ***in-degree*** of a vertex v to be the number of edges for which v is the head.
- The ***out-degree*** is defined to be the number of edges for which v is the tail. Vertex 2 of G_3 has in-degree 1, out-degree 2 and degree 3.
- A directed graph is a digraph. An undirected graph will sometimes be referred to simply as a graph

ELEMENTARY GRAPH ALGORITHMS

- **Graph Representations**
  - Three most commonly used Representations are:

    adjacency matrices, adjacency lists and adjacency multi-lists

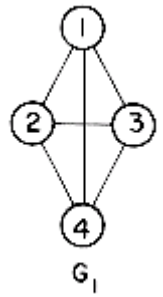
- **Adjacency Matrix**

• Let G = (*V*,*E*) be a graph with *n* vertices, *n* > 1

• The adjacency matrix of *G* is a 2-dimensional n x n array, say *A*, with the property that *A*(*i*,*j*) = 1 if the edge (v_i,v_j) (<v_i,v_j> for a directed graph) is in E(G).

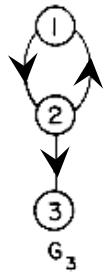
• A(i,j) = 0 if there is no such edge in G.

	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0



G_1

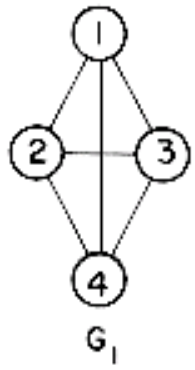
	1	2	3
1	0	1	0
2	1	0	1
3	0	0	0



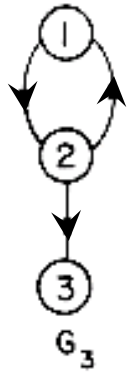
G_3

ELEMENTARY GRAPH ALGORITHMS

• The adjacency matrix for an undirected graph is symmetric as the edge (v_i,v_j) is in E(G) if the edge (v_j,v_i) is also in E(G).



	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0



	1	2	3
1	0	1	0
2	1	0	1
3	0	0	0

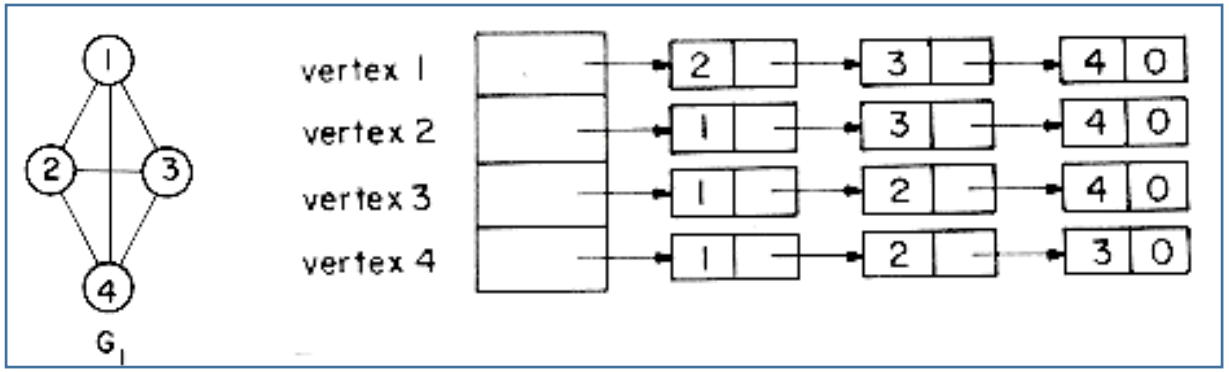
• The adjacency matrix for a directed graph need not be symmetric (as is the case for G_3).

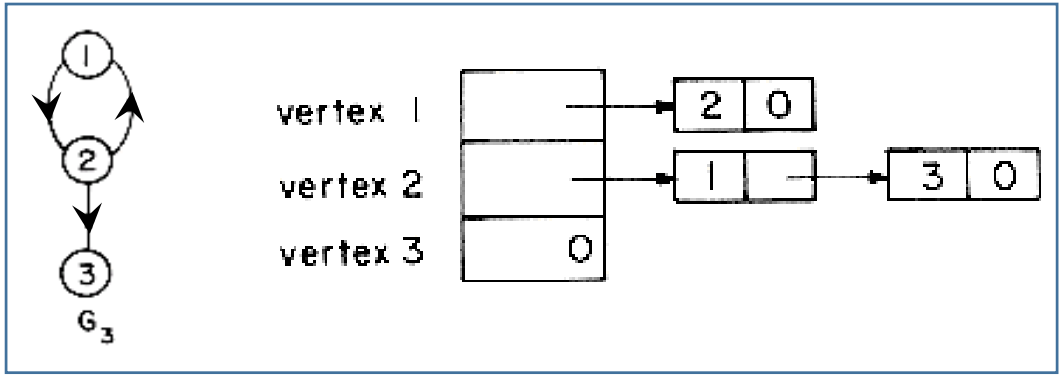
• The space needed to represent a graph using its adjacency matrix is n^2 bits

• About half this space can be saved in the case of undirected graphs by storing only the upper or lower triangle of the matrix.

ELEMENTARY GRAPH ALGORITHMS

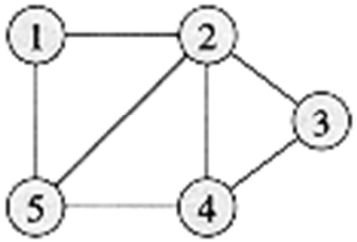
- The adjacency matrix can be represented as *n* linked lists.
- There is one list for each vertex in *G*.
- The nodes in list *i* represent the vertices that are adjacent from vertex *i*.
- Each node has at least two fields: VERTEX and LINK.
- The VERTEX fields contain the indices of the vertices adjacent to vertex *i*



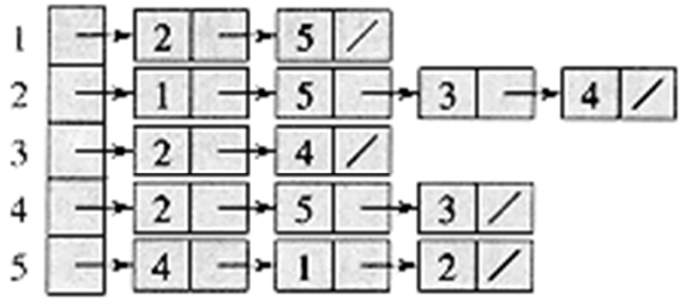


ELEMENTARY GRAPH ALGORITHMS

- **Graph Representations**
- Graph



(a)

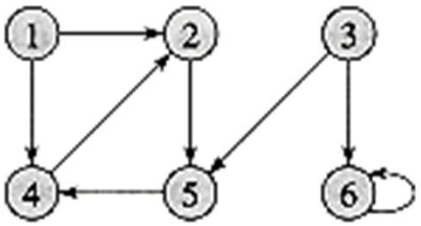


(b)

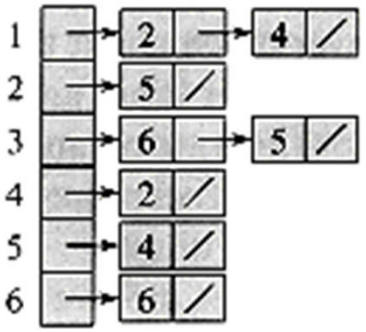
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

- Digraph



(a)

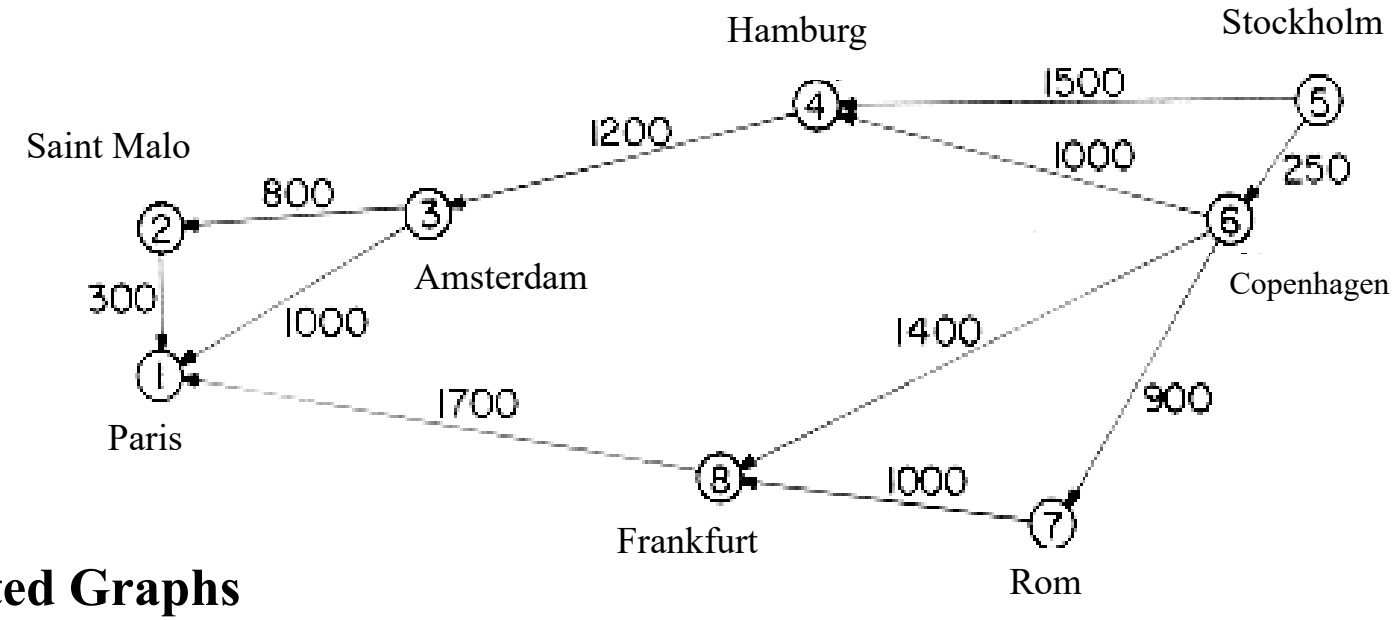


(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

ELEMENTARY GRAPH ALGORITHMS



- **Weighted Graphs**
  - In any practical situation, the edges will be assigned weights.
  - These weights might represent the cost of construction, the length of the link, etc.

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	1000	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

ELEMENTARY GRAPH ALGORITHMS

• Depth First Search and Breadth First Search

• One of the most common things one wishes to do is to traverse the graph and visit the vetices in some order.

• Given an undirected graph $G = (V,E)$ and a vertex v in $V(G)$ we are interested in visiting all vertices in G that are reachable from v (i.e., all vertices connected to v).

• There are two ways of doing this: Depth First Search and Breadth First Search

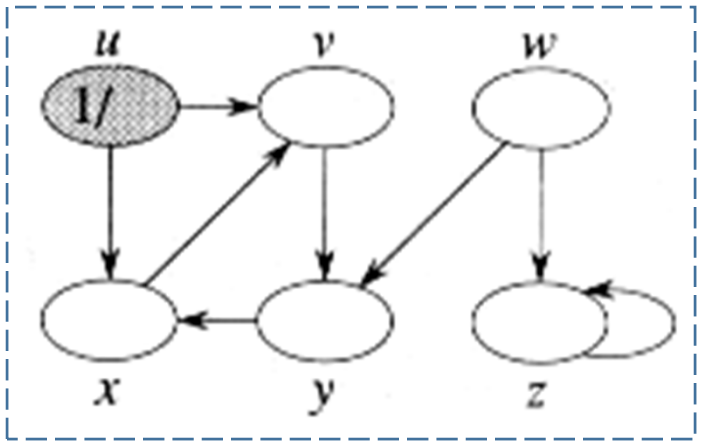
• Depth First Search (DFS)

• The start vertex v is visited. Next an unvisited vertex w adjacent to v is selected and a depth first search from w initiated.

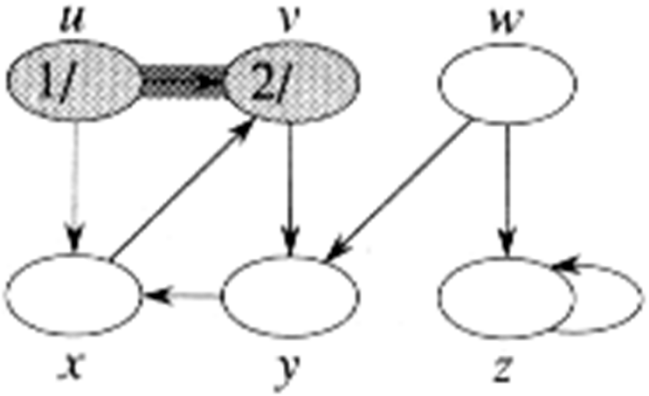
• When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited which has an unvisited vertex w adjacent to it and initiate a depth first seaerch from w.

• The search terminates when no unvisited vertex can be reached from any of the visited one

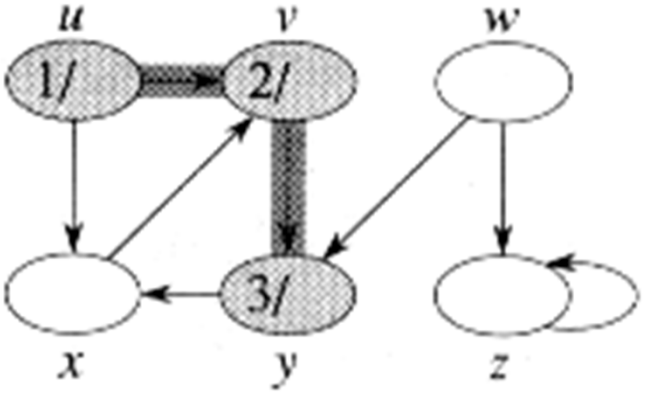
- **ELEMENTARY GRAPH ALGORITHMS**
  - **Depth First Search (Digragh)**



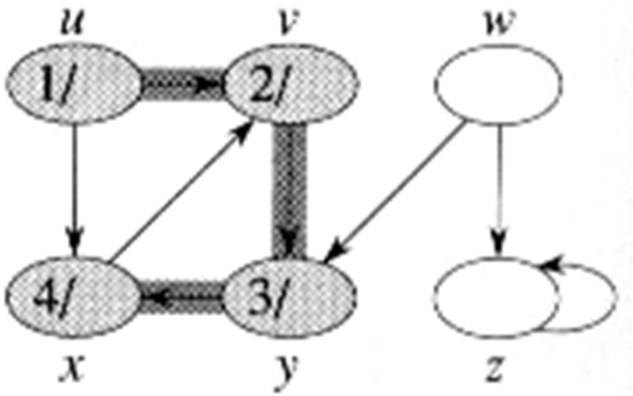
(a)



(b)



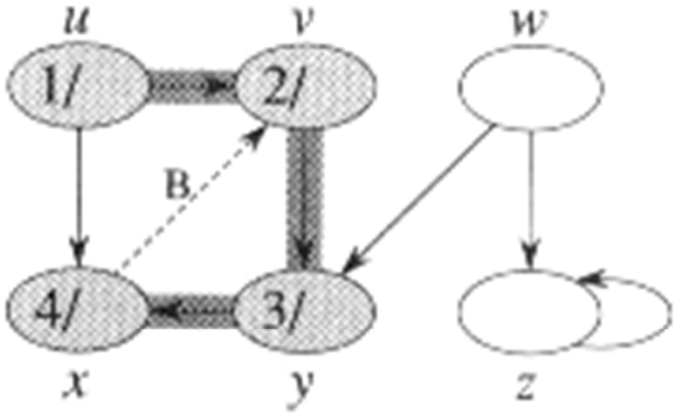
(c)



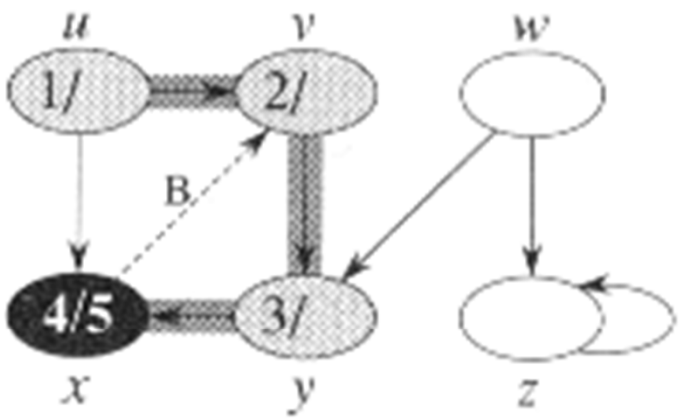
(d)

ELEMENTARY GRAPH ALGORITHMS

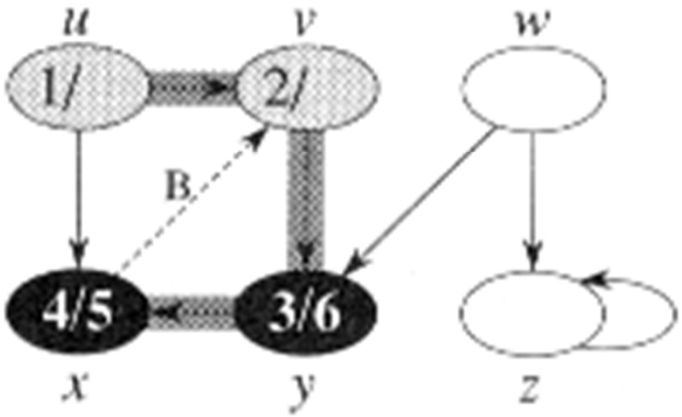
- **Depth First Search (Digragh)**



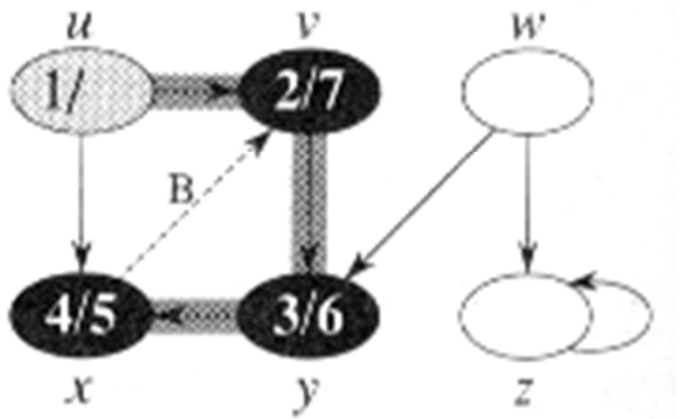
(e)



(f)



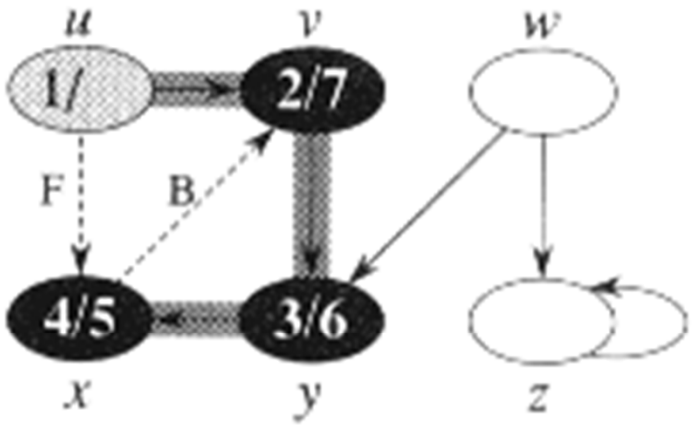
(g)



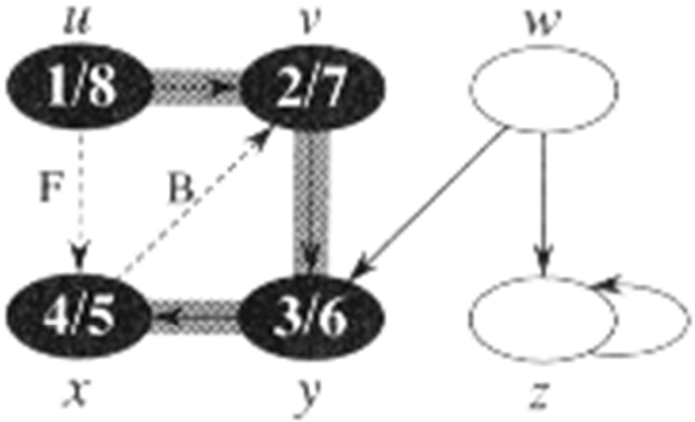
(h)

ELEMENTARY GRAPH ALGORITHMS

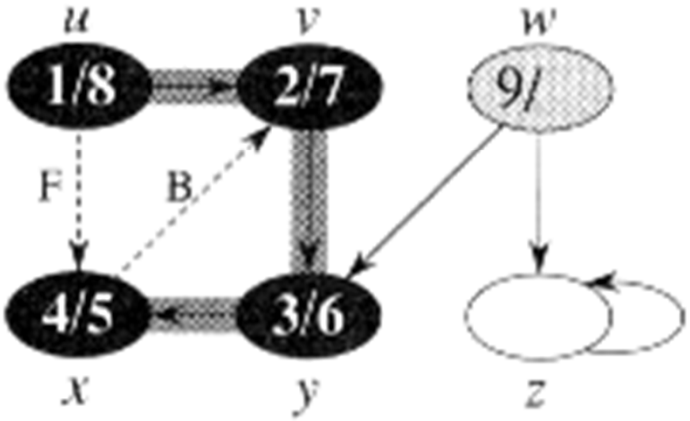
- **Depth First Search (Digragh)**



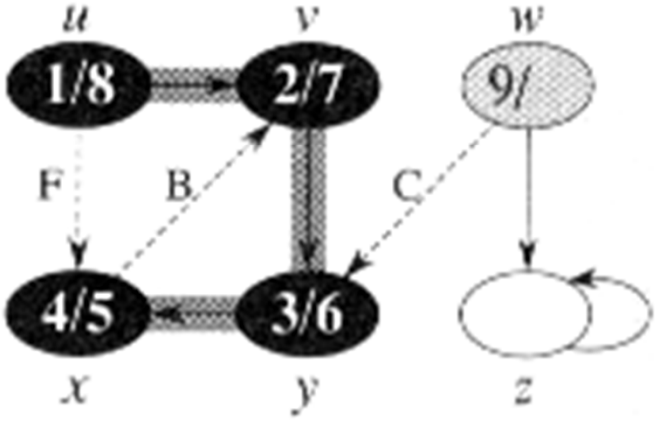
(i)



(j)



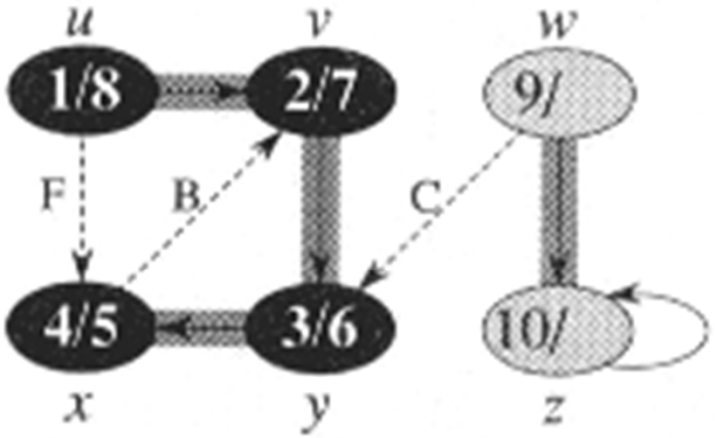
(k)



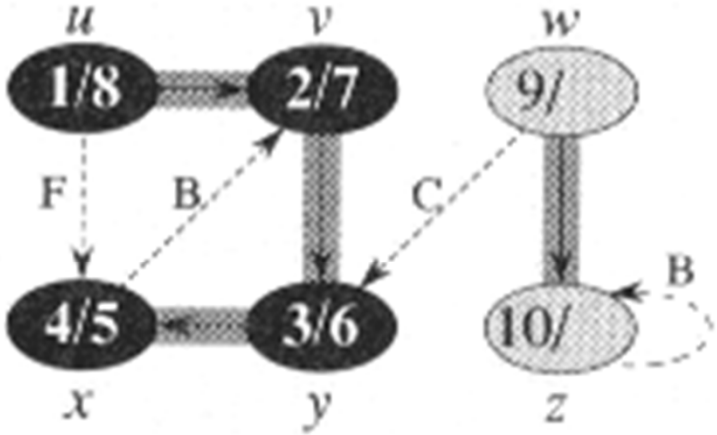
(l)

ELEMENTARY GRAPH ALGORITHMS

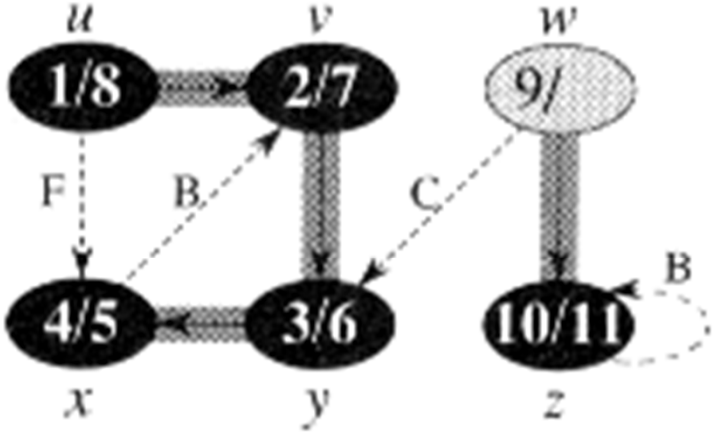
- **Depth First Search (Digragh)**



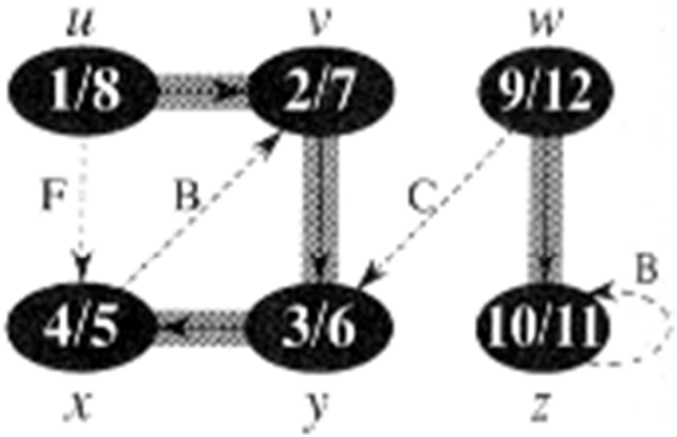
(m)



(n)



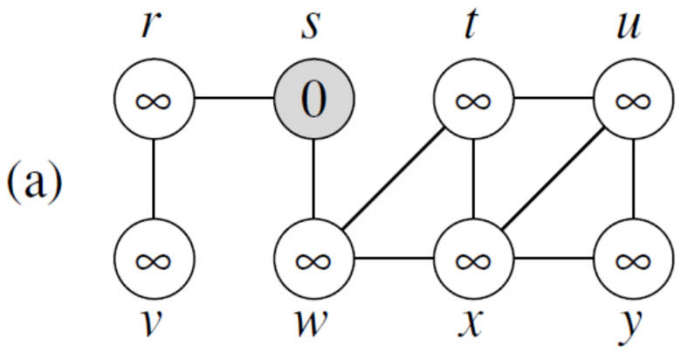
(o)

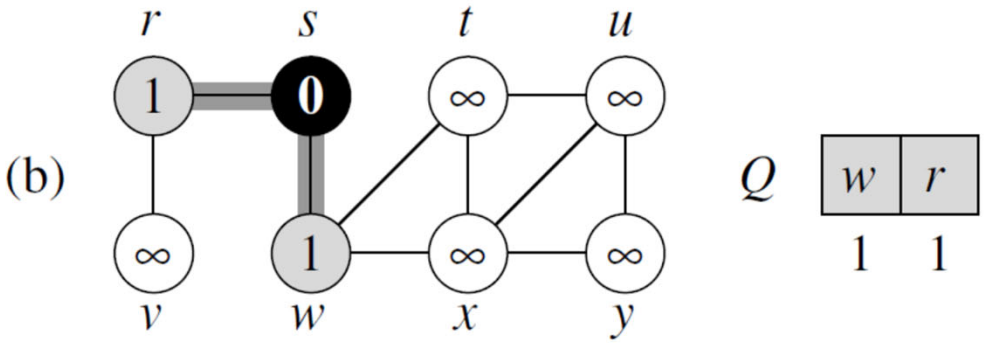


(p)

ELEMENTARY GRAPH ALGORITHMS

- **Breadth First Search**
- Starting at vertex *v* and marking it as visited, breadth first search differs from depth first search in that all unvisited vertices adjacent to *v* are visited next
- Then unvisited vertices adjacent to these vertices are visited and so on

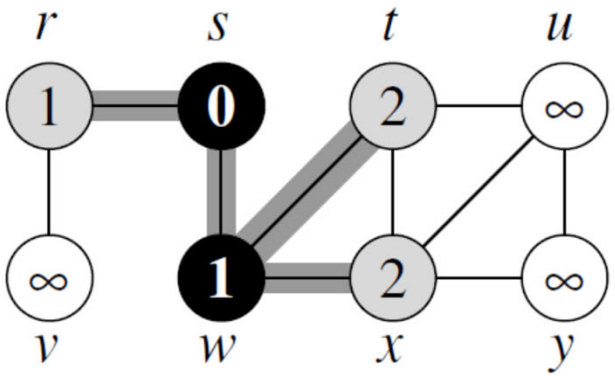




ELEMENTARY GRAPH ALGORITHMS

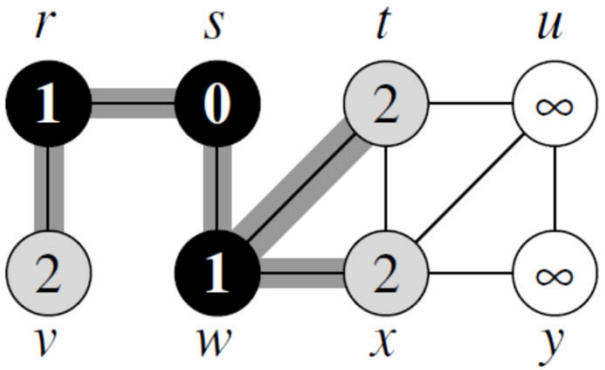
- **Breadth First Search**

(c)



Q | r | t | x |
1 2 2

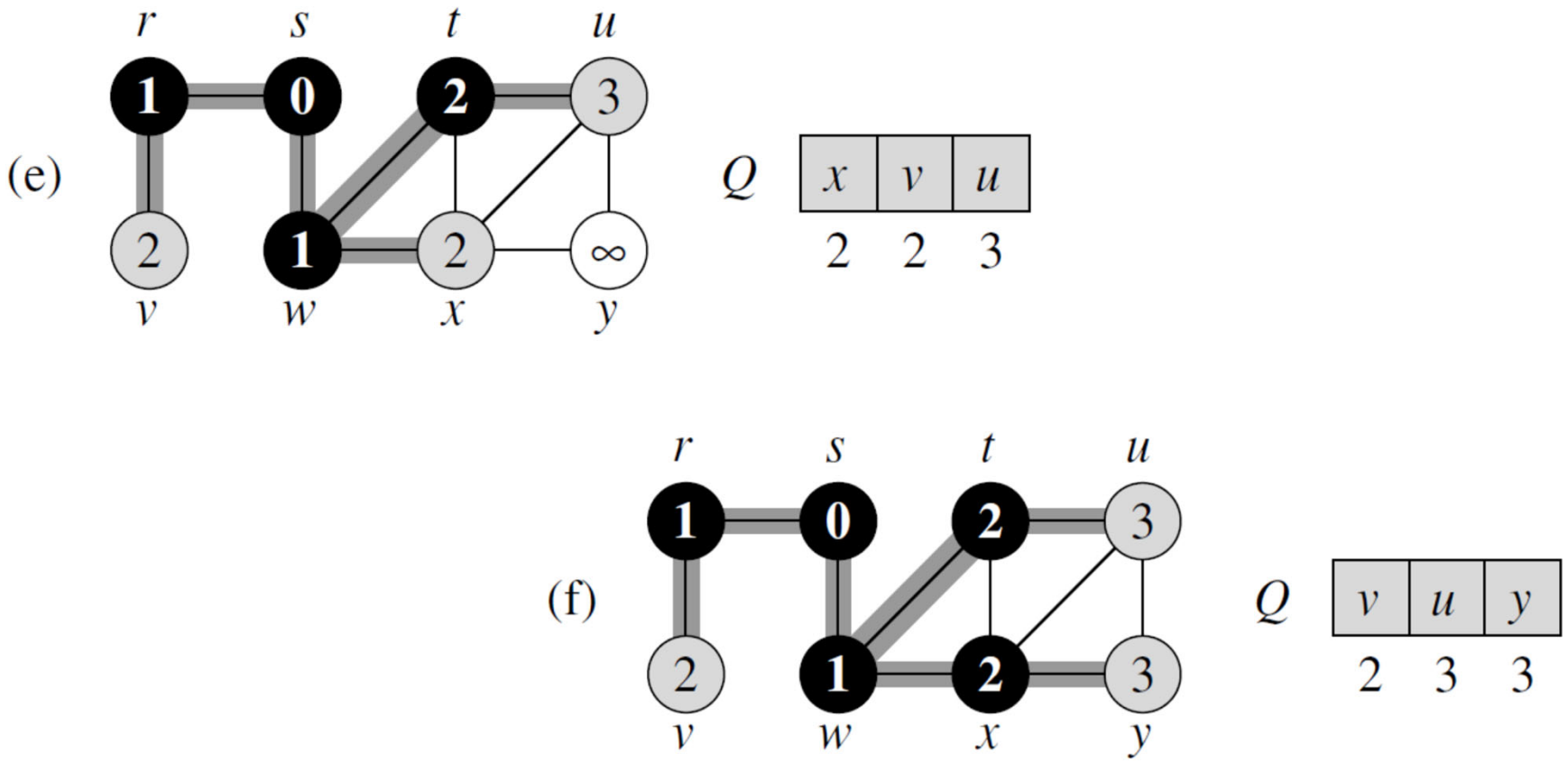
(d)



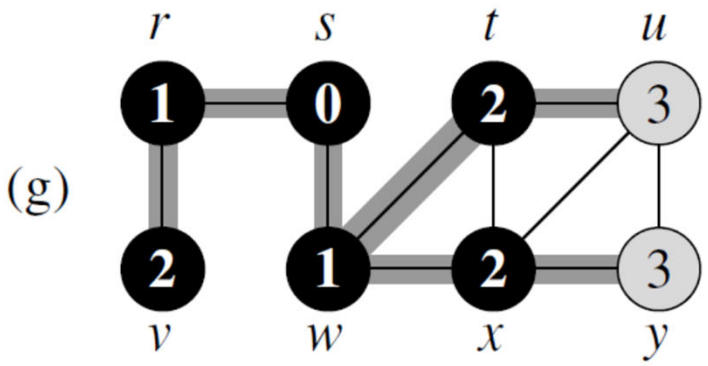
Q | t | x | v |
2 2 2

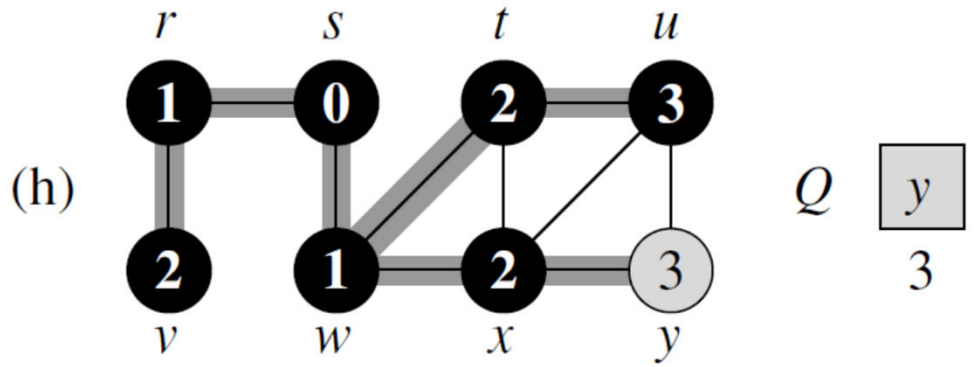
ELEMENTARY GRAPH ALGORITHMS

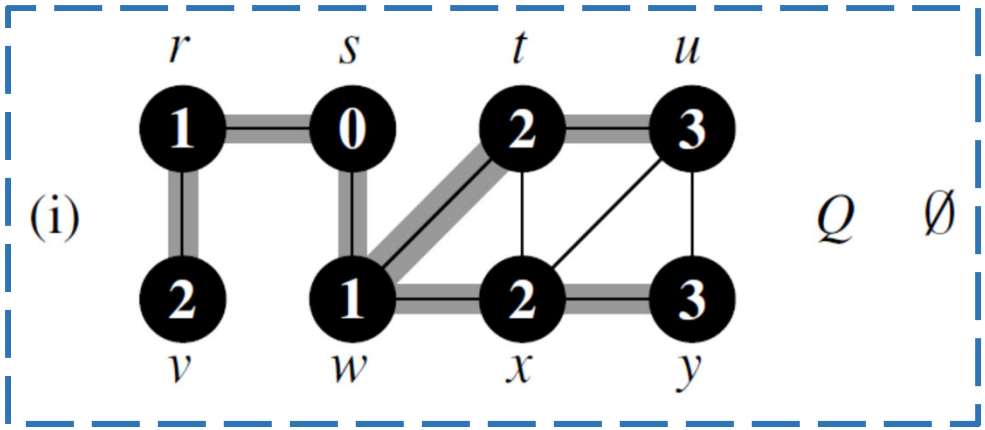
- **Breadth First Search**



Breadth First Search

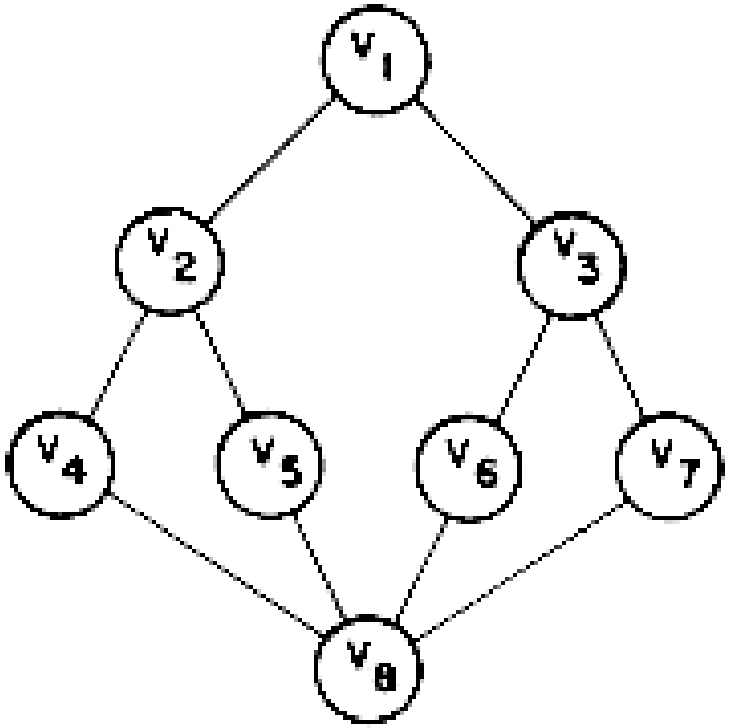






Assignment

Use DFS and BFS on following undirected graph from vertex v1

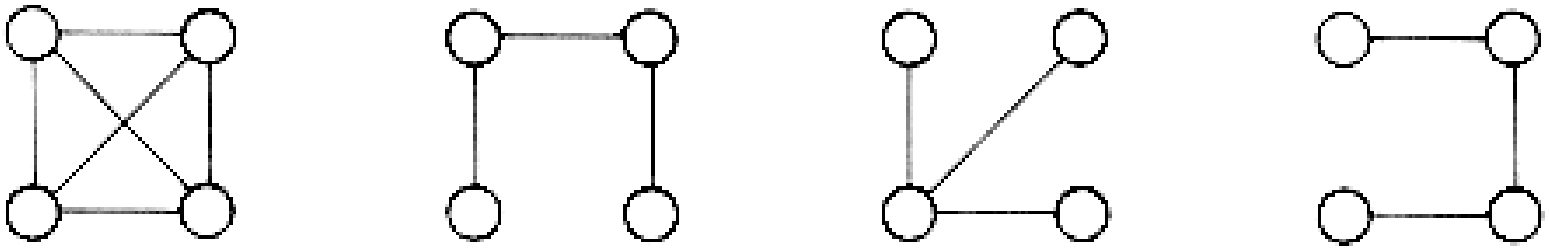


ELEMENTARY GRAPH ALGORITHMS

•Minimum Spanning Trees

• Spanning tree is a minimal subgraph G' of G such that $V(G') = V(G)$ and G' is connected (by a minimal subgraph, we mean one with the fewest number of edges).

• When the graph G is connected, a depth first or breadth first search starting at any vertex, visits all the vertices in G

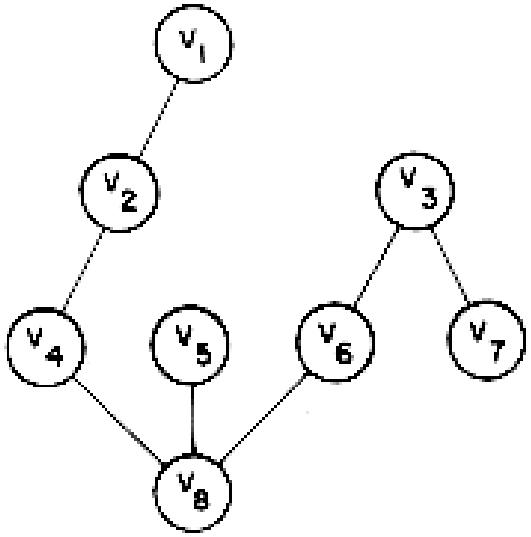


• The spanning tree resulting from a call to DFS is known as a *depth first spanning tree*.

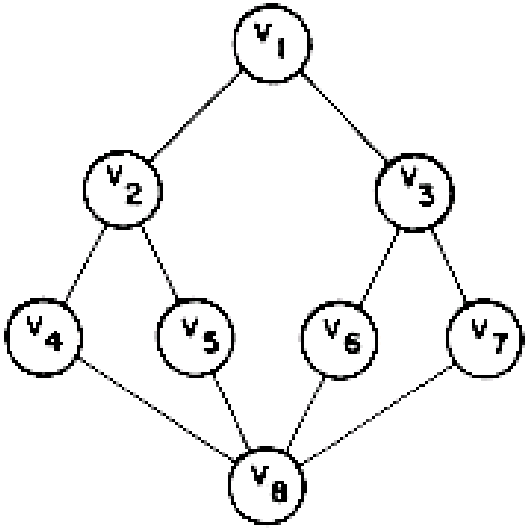
• When BFS is used, the resulting spanning tree is called a *breadth first spanning tree*

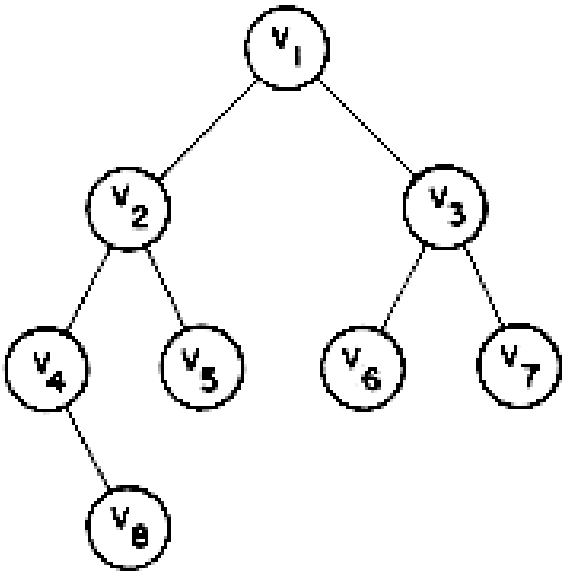
ELEMENTARY GRAPH ALGORITHMS

- **Trees and Minimum Cost Spanning Trees**



Depth first spanning tree





Breadth first spanning tree

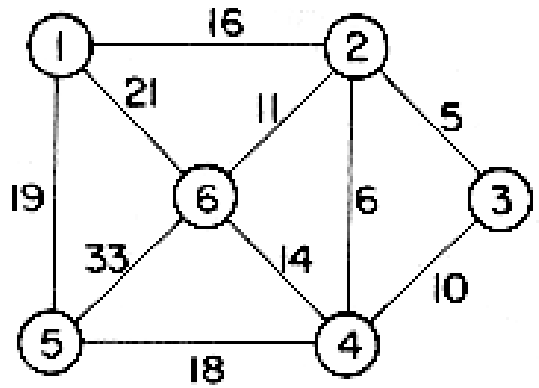
Minimum Cost Spanning Trees

- To interconnect a set of n pins, we can use an arrangement of n - 1 wires, each connecting two pins.
- Of all such arrangements, the one that uses the least amount of connection cost is usually the most desirable
- We can model this connection problem with undirected graph $G = (V, E)$, where V is the set of pins, E is the set of possible interconnections between pairs of edges, and for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost to connect u and v.
- We then wish to find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight $w(u,v)$ is minimized

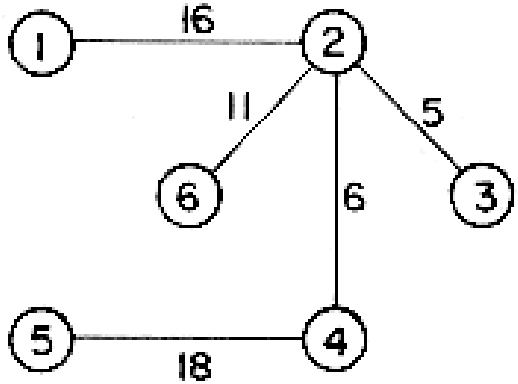
$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

Minimum Cost Spanning Trees

- One approach to determining a minimum cost spanning tree of a graph has been given by Kruskal.
- In this approach a minimum cost spanning tree, *T*, is built edge by edge.
- Edges are considered for inclusion in *T* in non-decreasing order of their costs.
- An edge is included in *T* if it does not form a cycle with the edges already in *T*



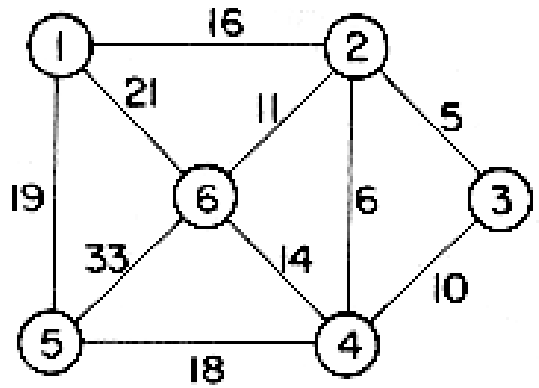
(a)



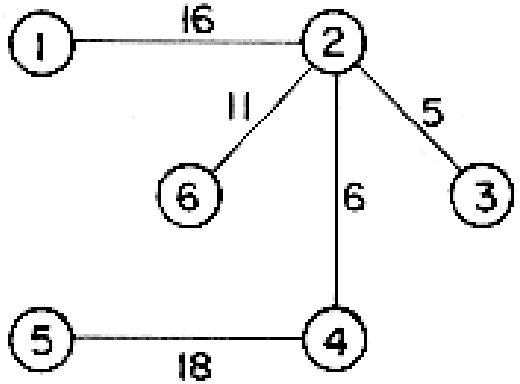
(b)

Minimum Cost Spanning Trees

```
MST-KRUSKAL(G, w)
1  A ← ∅
2  for each vertex v ∈ V[G]
3       do MAKE-SET(v)
4  sort the edges of E into nondecreasing order by weight w
5  for each edge (u, v) ∈ E, taken in nondecreasing order by weight
6       do if FIND-SET(u) ≠ FIND-SET(v)
7            then A ← A ∪ {(u, v)}
8                 UNION(u, v)
9  return A
```

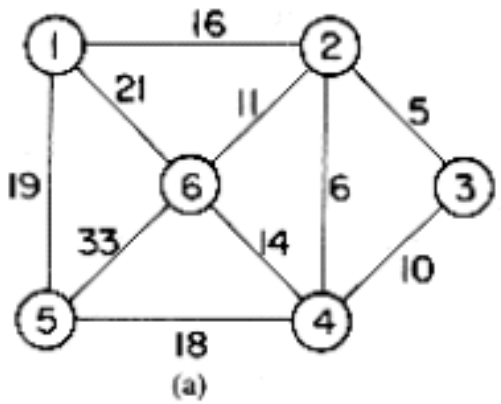


(a)

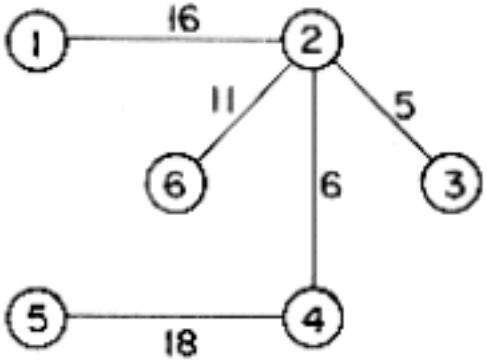


(b)

Minimum Cost Spanning Trees: Kruskal's algorithm



(a)

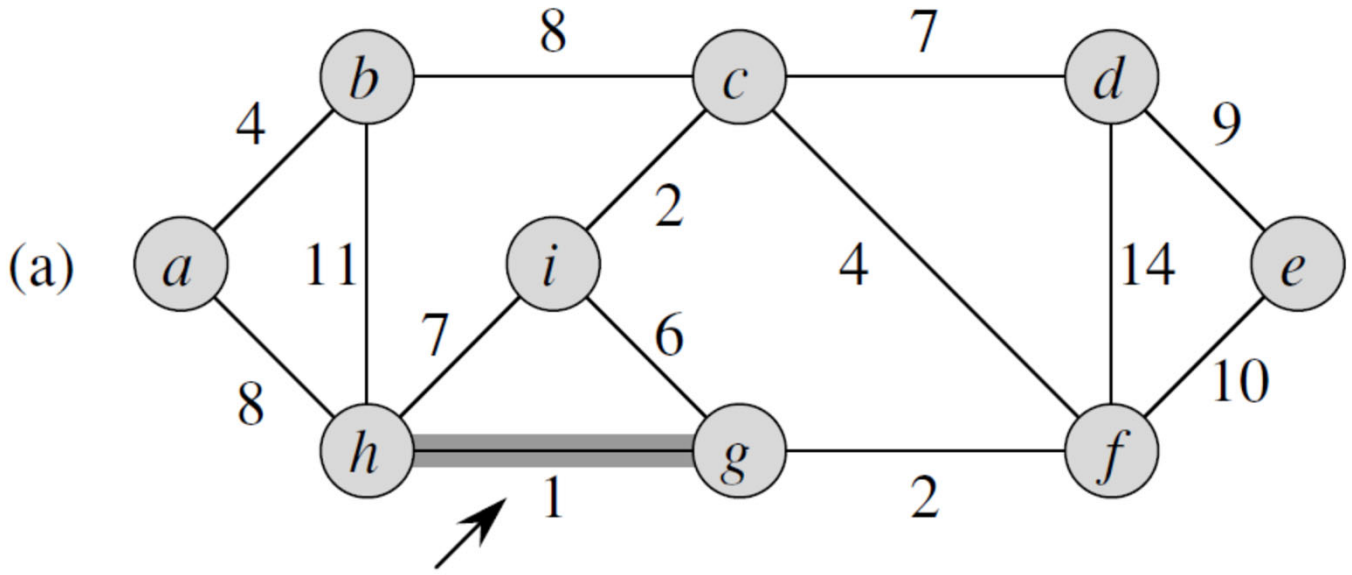


(b)

edge	cost	action	T
–	–	–	1 2 3 4 5 6
(2,3)	5	accept	1 2–3 4 5 6
(2,4)	6	accept	1 2–3, 2–4 5 6
(4,3)	10	reject	1 2–3, 2–4, 2–6 5
(2,6)	11	accept	
(4,6)	14	reject	1 2–3, 2–4, 2–6, 1–2 5
(1,2)	16	accept	
(4,5)	18	accept	1 2–3, 2–4, 2–6, 1–2, 4–5

Minimum Cost Spanning Trees

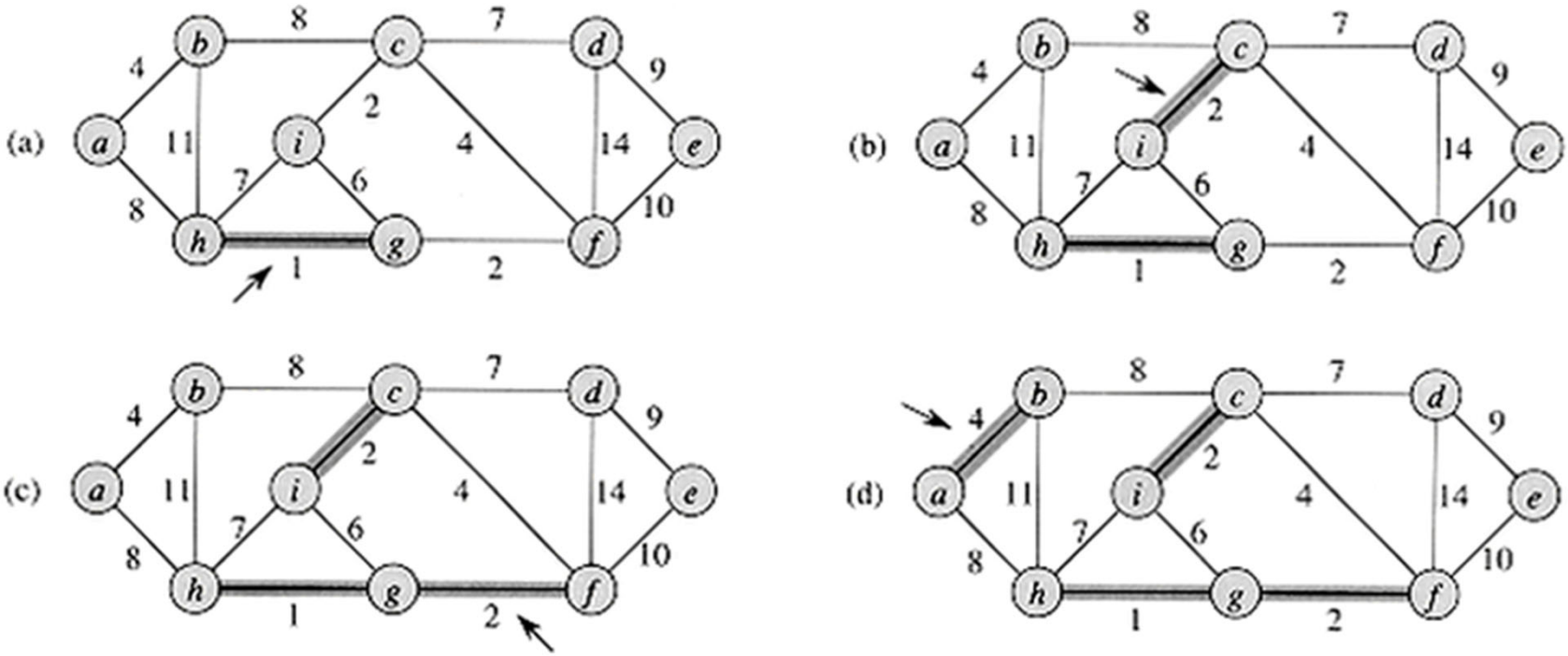
Assignment: Find MST using Kruskal's algorithm



Minimum Cost Spanning Trees

Assignment: Find MST using Kruskal's algorithm

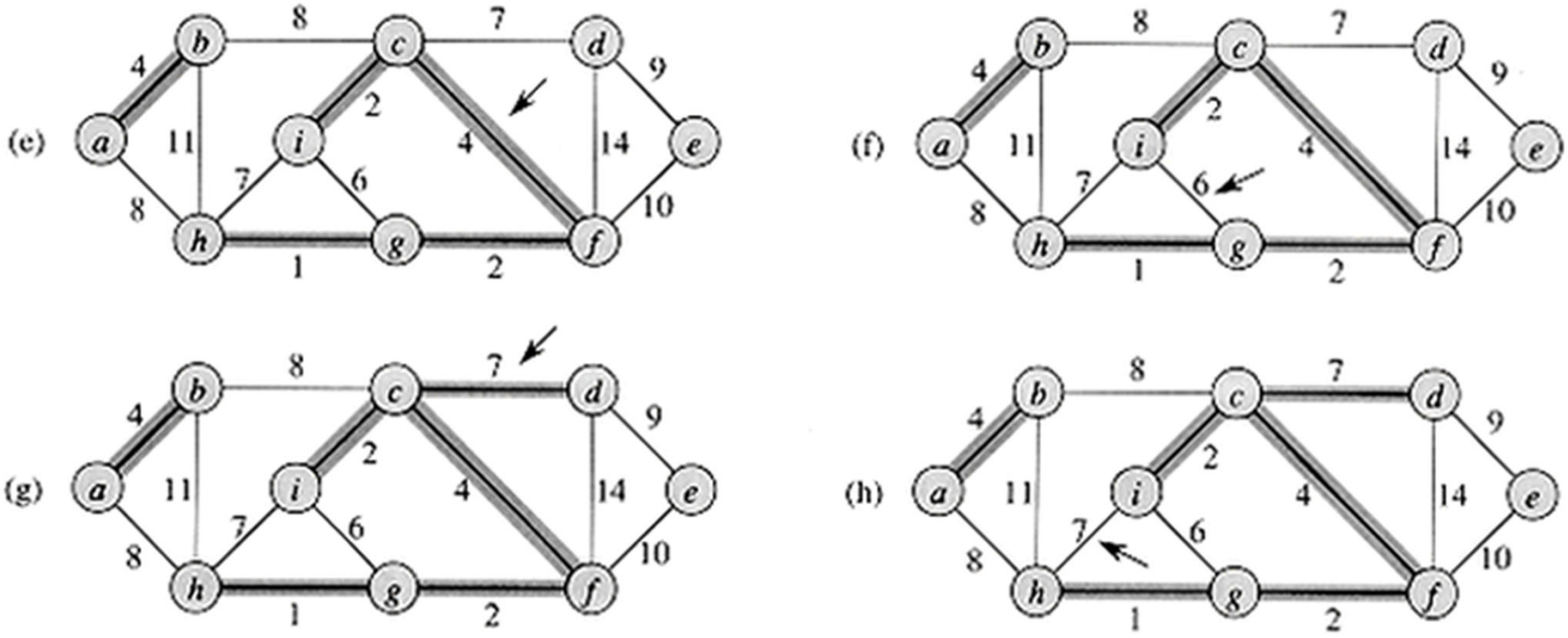
One possible **solution**



Minimum Cost Spanning Trees

Assignment: Find MST using Kruskal's algorithm

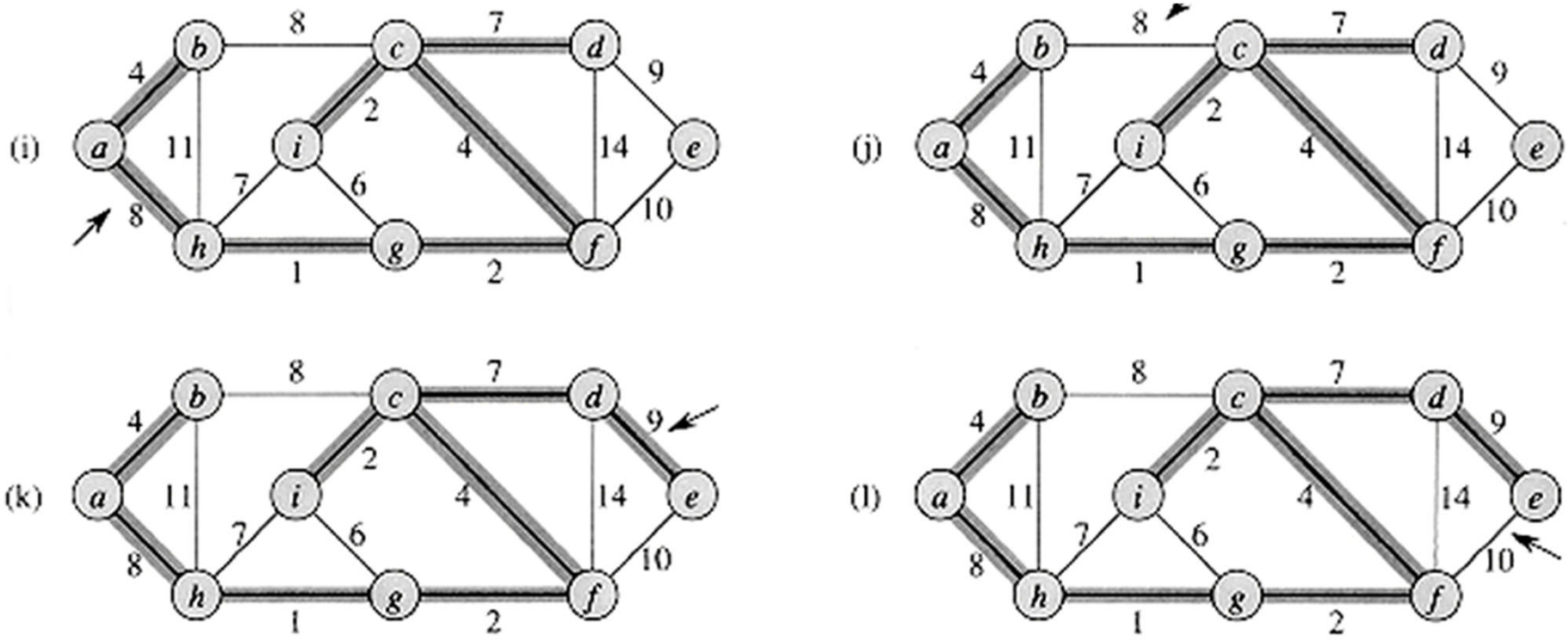
One possible **solution**



Minimum Cost Spanning Trees

Assignment: Find MST using Kruskal's algorithm

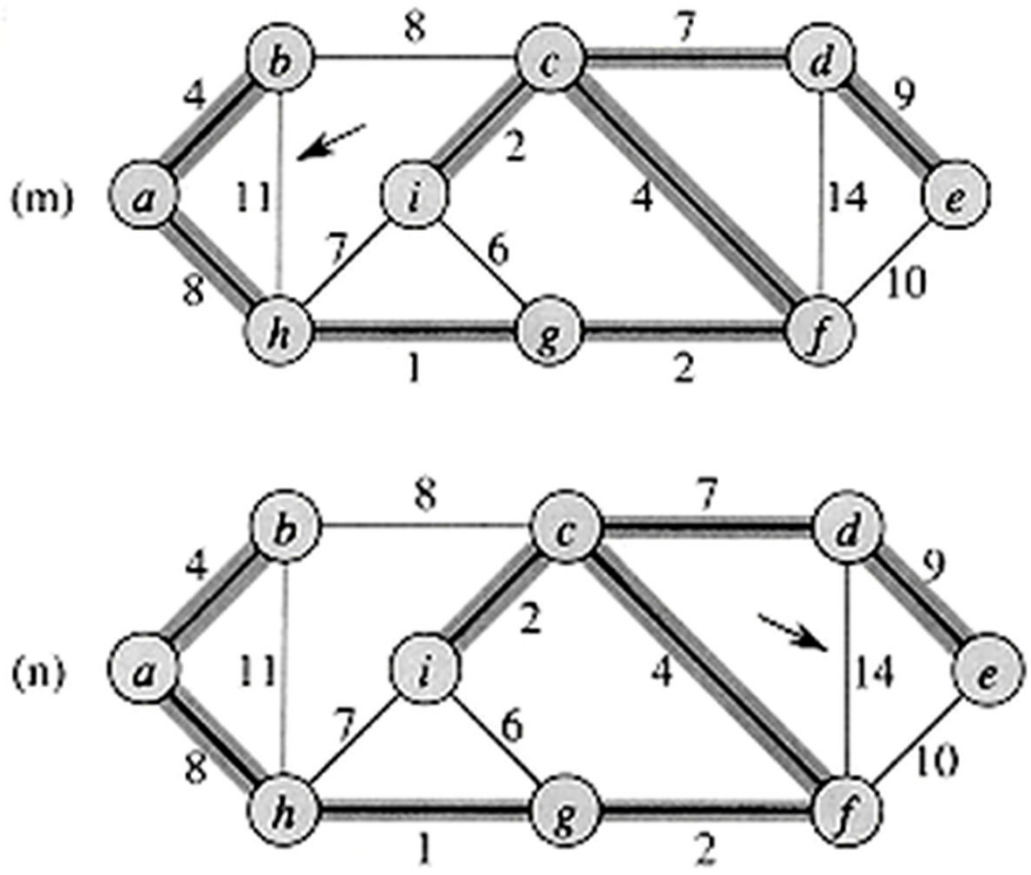
One possible **solution**



Minimum Cost Spanning Trees

Assignment: Find MST using Kruskal's algorithm

One possible **solution**



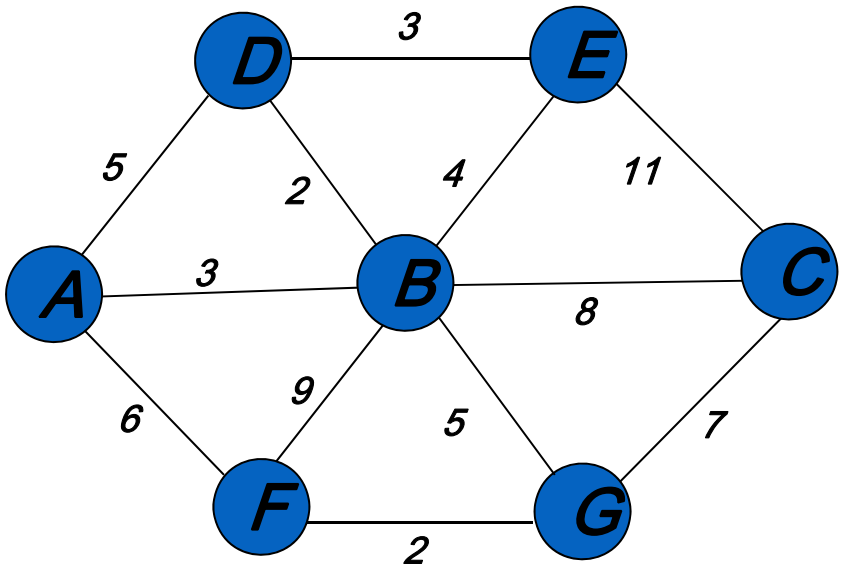
Minimum Cost Spanning Trees

Prim's Algorithm

- Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph, which we see in next lectures.
- Prim's algorithm has the property that the edges in the set A always form a single tree.
- The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V. At each step, a light edge is added to the tree A that connects A to an isolated vertex of $G_A = (V, A)$.

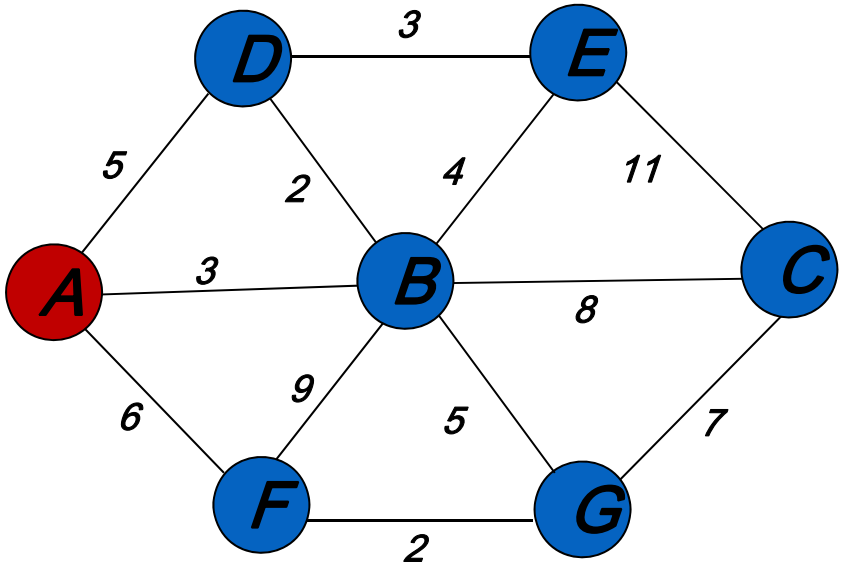
```
MST-PRIM(G, w, r)
 1  for each u ∈ V[G]
 2       do key[u] ← ∞
 3          π[u] ← NIL
 4  key[r] ← 0
 5  Q ← V[G]
 6  while Q ≠ ∅
 7       do u ← EXTRACT-MIN(Q)
 8          for each v ∈ Adj[u]
 9               do if v ∈ Q and w(u, v) < key[v]
10                     then π[v] ← u
11                          key[v] ← w(u, v)
```

Prim's Algorithm



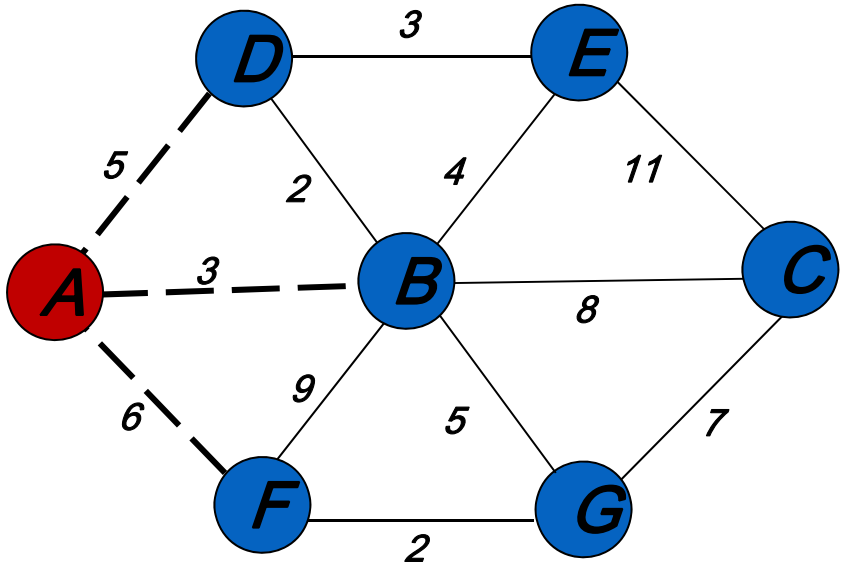
Prim's Algorithm

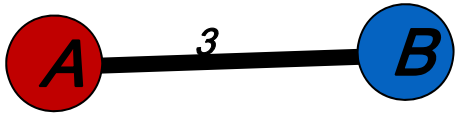
Select a random vertex for the start
Mark incident edges and select the one with min. weight



Prim's Algorithm

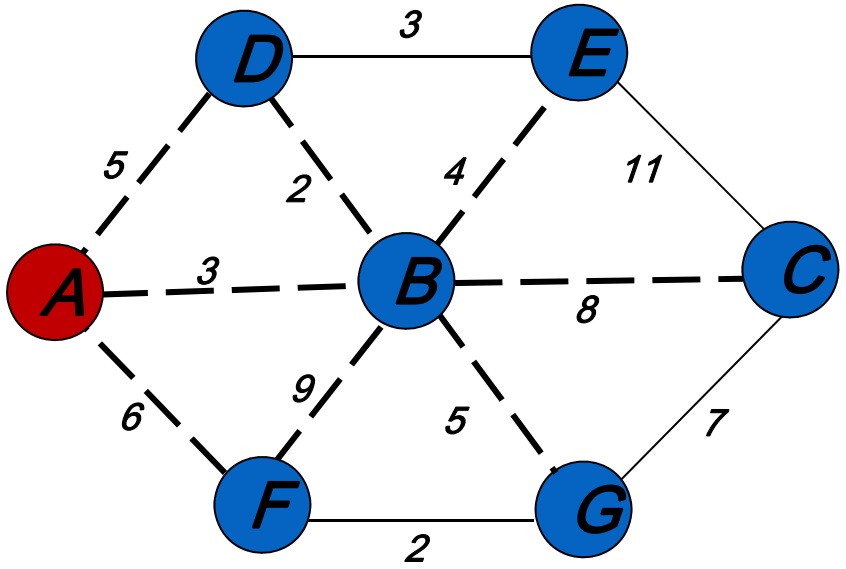
Select a random vertex for the start
Mark incident edges of A and select the one with min. weight

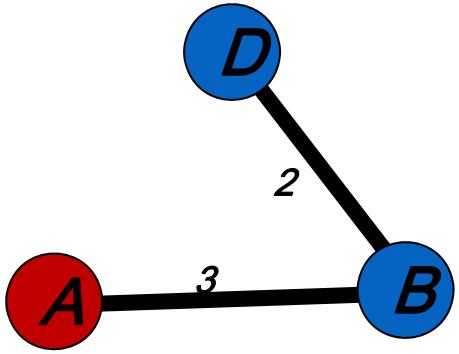




Prim's Algorithm

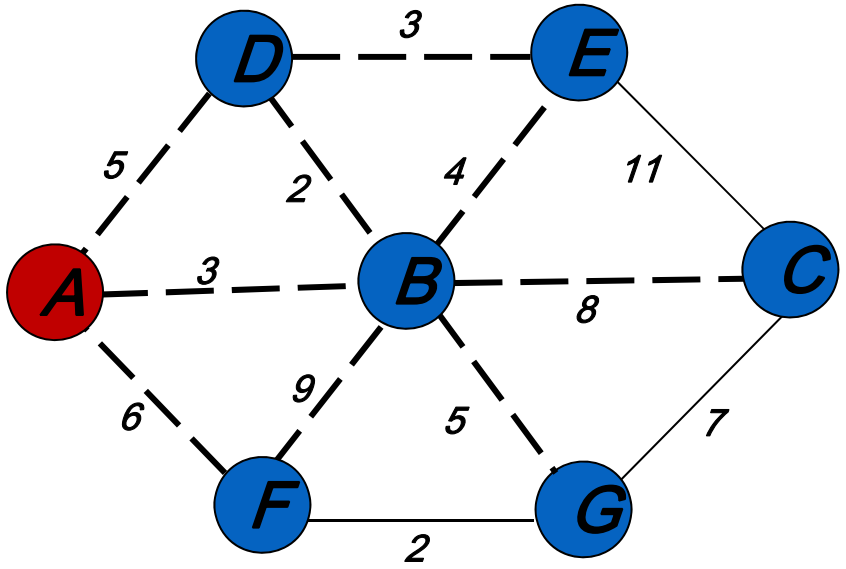
Mark incident edges of B and select from **all** marked edges the one with min. weight

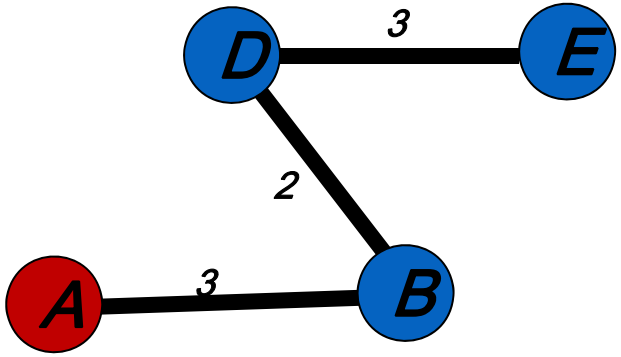




Prim's Algorithm

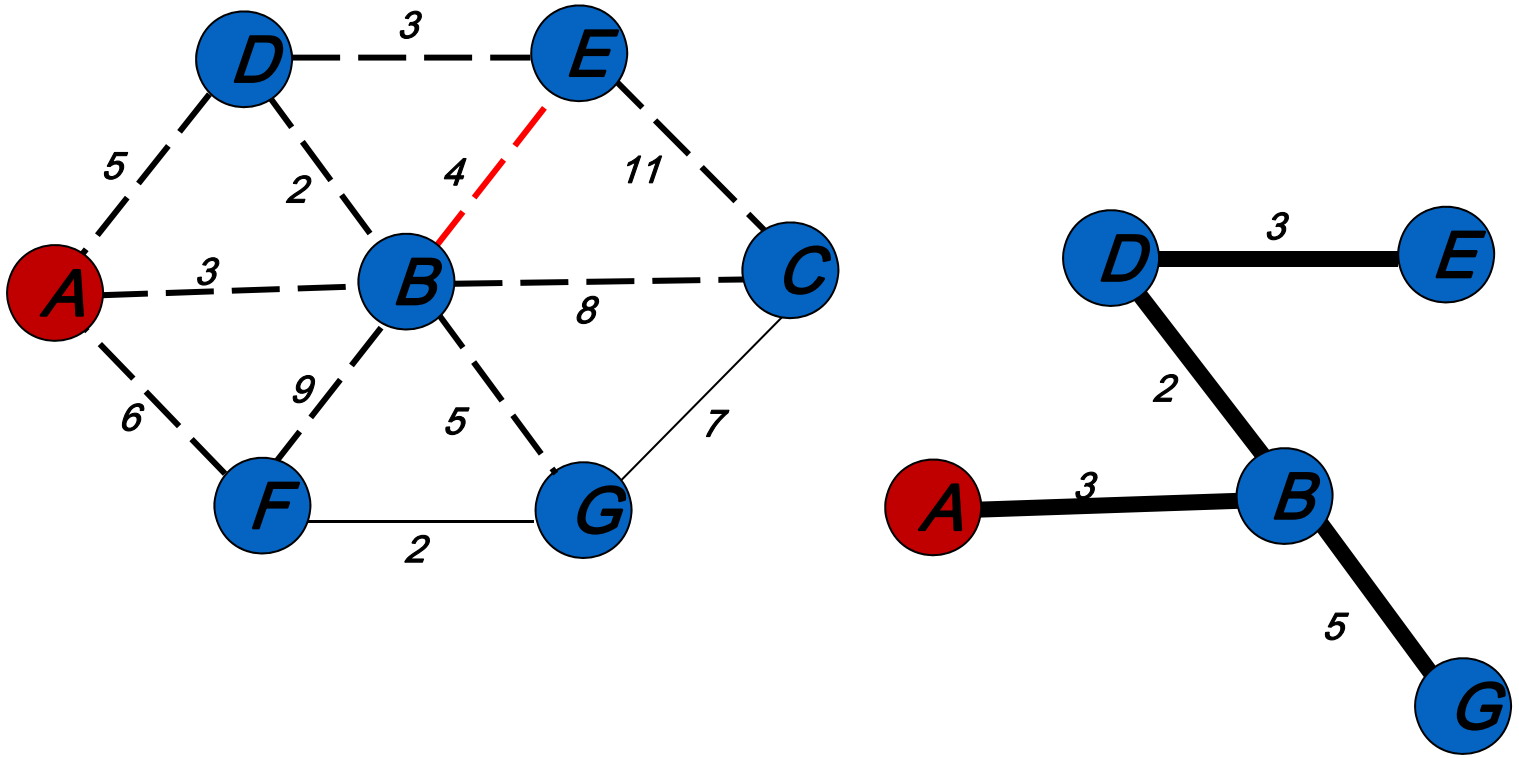
Mark incident edges of D and select from **all** marked edges the one with min. weight





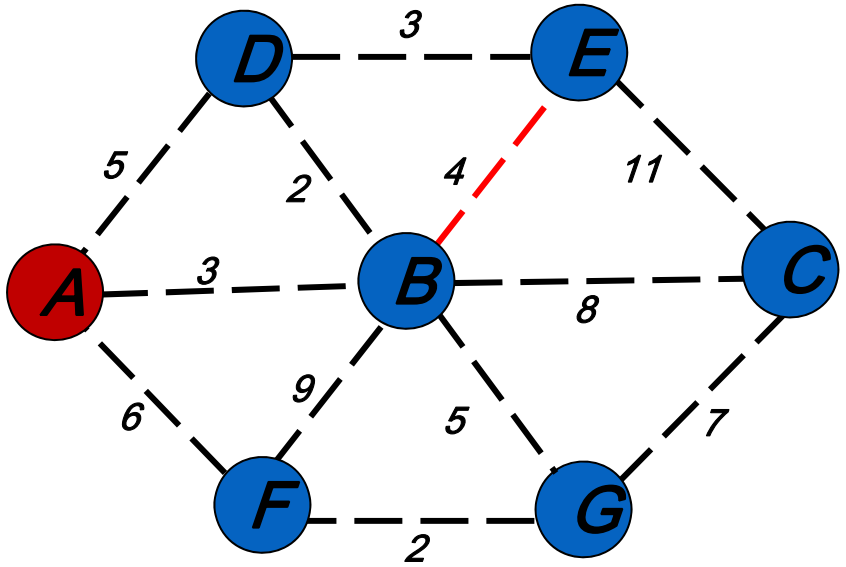
Prim's Algorithm

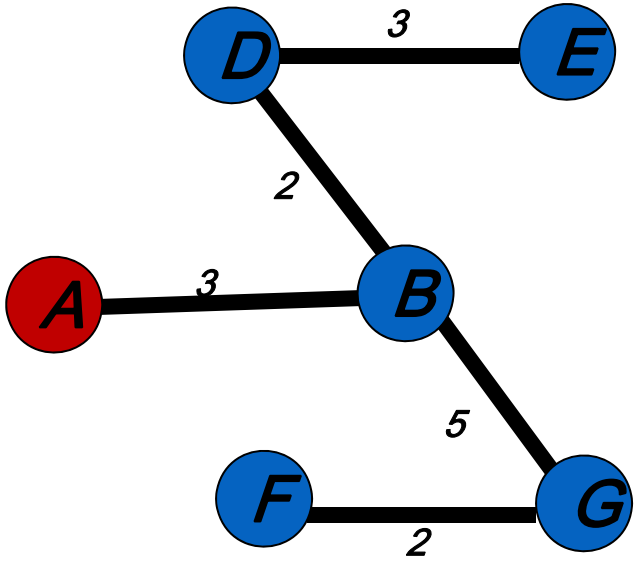
Mark incident edges of E and select from **all** marked edges the one with min. weight. It is 4 but it create a cycle, rejected. Select next edge with min. weight, 5.



Prim's Algorithm

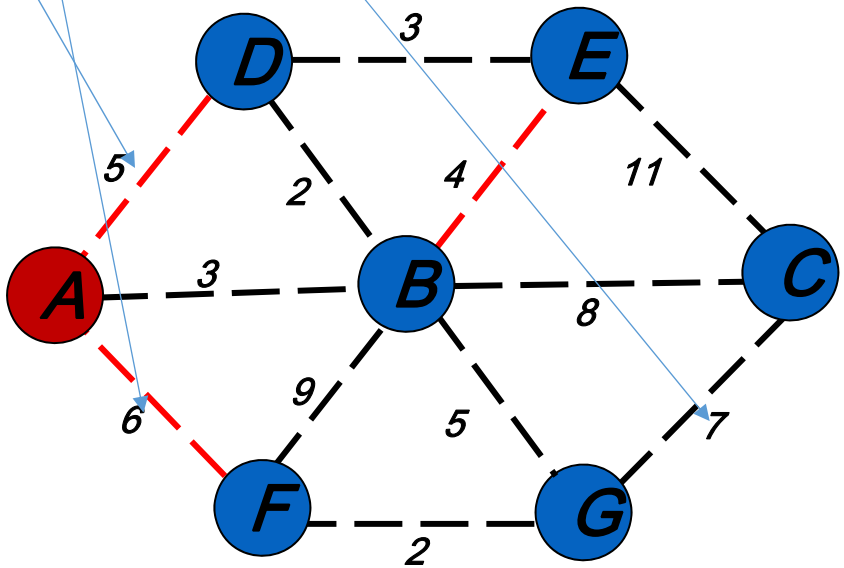
Mark incident edges of G and select from **all** marked edges the one with min. weight.

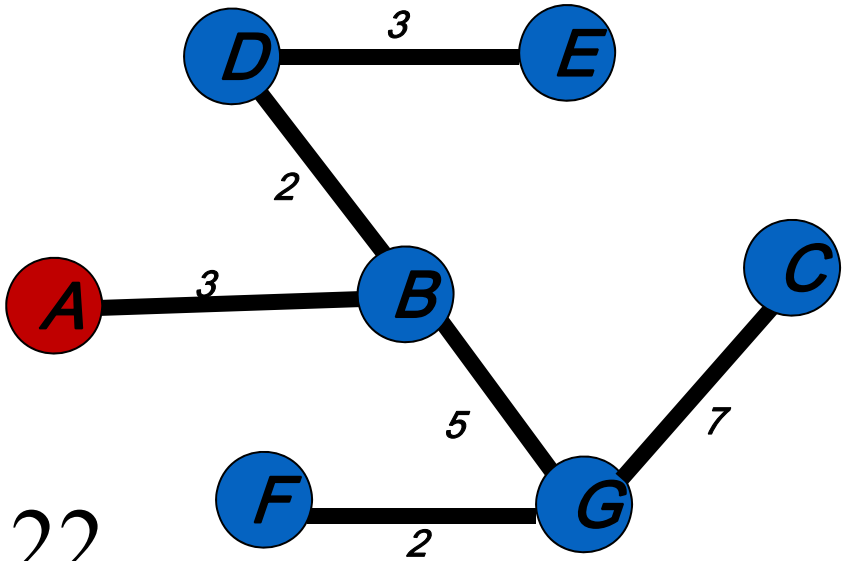




Prim's Algorithm

Vertex C has the incident edges with weights of 7, 8, 11. Before including C all edges with weights less than 7, 8, 11 has to be verified for spanning tree connection. If any edge creates a cycle, rejected, otherwise it will be included in the tree. Edges 5, 6 less than 7 are rejected and 7 is accepted.



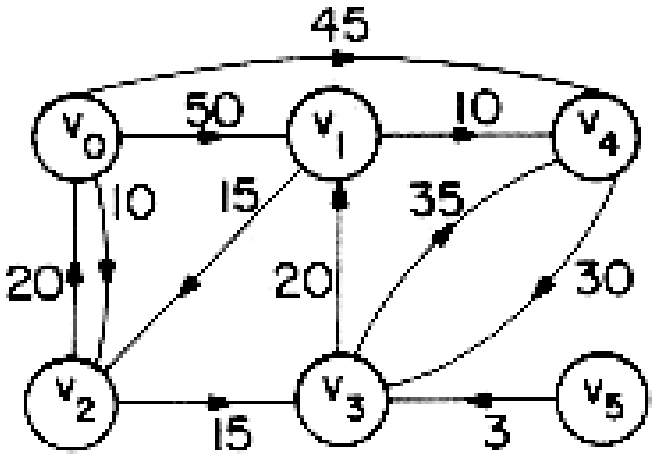


$$w(T) = \sum_{(u,v)\in T} w(u,v) = 22$$

Shortest Paths

- In a shortest-paths problem, we are given a weighted, directed graph $G = (V, E)$, with weight function $w : E \rightarrow R$ mapping edges to real-valued weights.
- The weight of path $p = (v_0, v_1, \ldots, v_k)$ is the sum of the weights of its constituent edges

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$



(a)

	Path	Length
1)	$v_0\ v_2$	10
2)	$v_0\ v_2\ v_3$	25
3)	$v_0\ v_2\ v_3\ v_1$	45
4)	$v_0\ v_4$	45

(b)

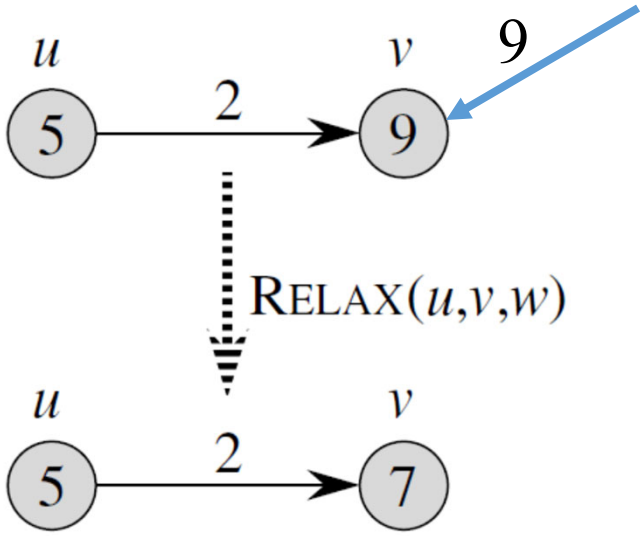
Relaxation

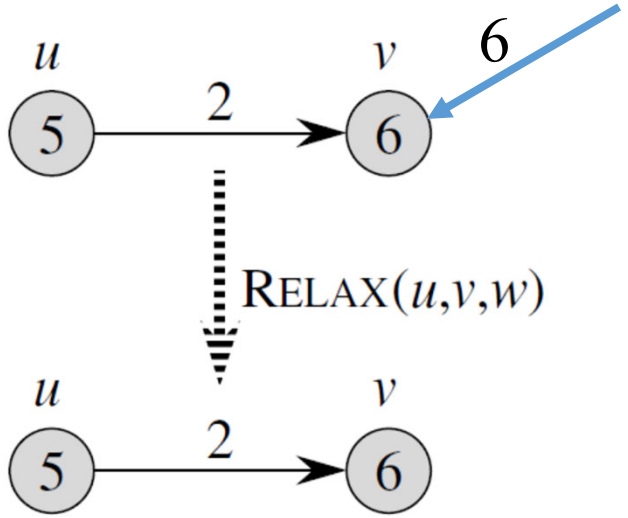
- The algorithms discussed here use the relaxation technique.
- For each vertex v $\in$ V, we maintain an attribute d[v]
- We call d[v] a shortest path
- The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating d[v]
- A relaxation step may decrease the value of the shortest path d[v]
- The outcome of a relaxation step can be viewed as a relaxation of the constraint $d[v] > d[u] + w(u, v)$
- That is, if $d[v] < d[u] + w(u, v)$, there is no "pressure" to satisfy this constraint, so the constraint is "relaxed."

Relaxation

- The following code performs a relaxation step on edge (u, v)

```
RELAX(u, v, w)
1   if d[v] > d[u] + w(u, v)
2       then d[v] ← d[u] + w(u, v)
3            π[v] ← u
```



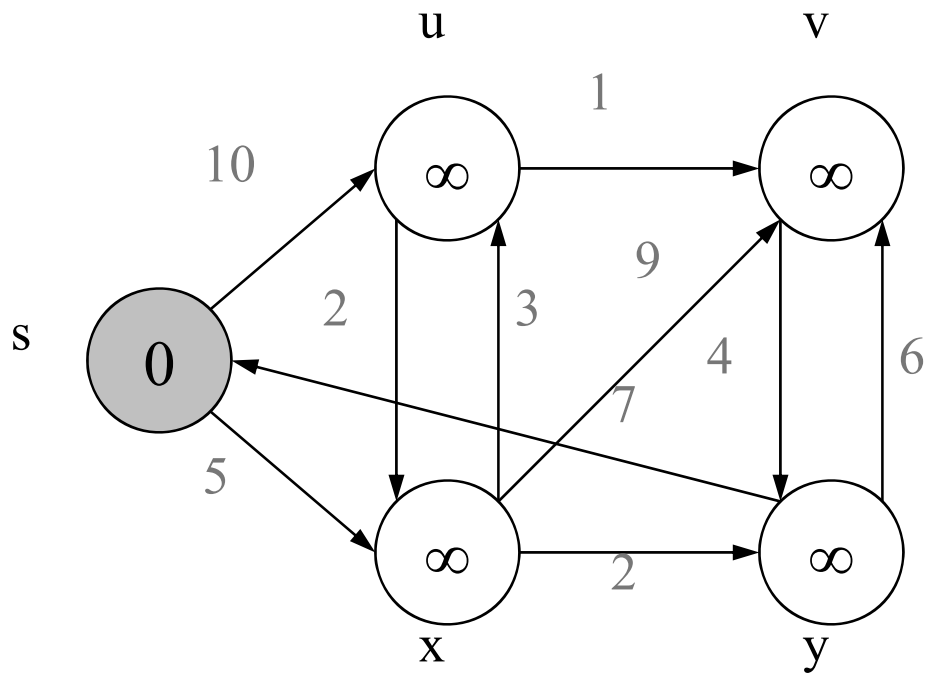


Shortest Paths (Dijkstra's algorithm)

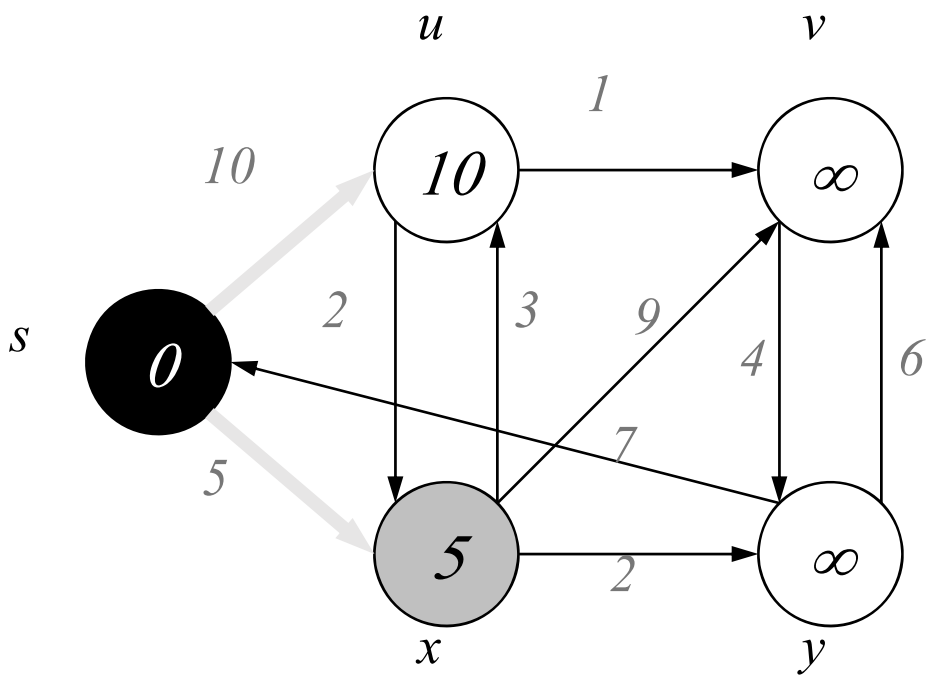
- This algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$
- **Definition**: for all vertices $v \in S$, we have $d[v] = (s, v)$.
- The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, inserts u into S, and relaxes all edges leaving u.

```
DIJKSTRA(G,w,s)
1  INITIALIZE-SINGLE-SOURCE (G,s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5        do u ← EXTRACT-MIN(Q)
6           S ← S ∪ {u}
7           for each vertex v ∈ Adj[u]
8               do RELAX (u,v,w)
```

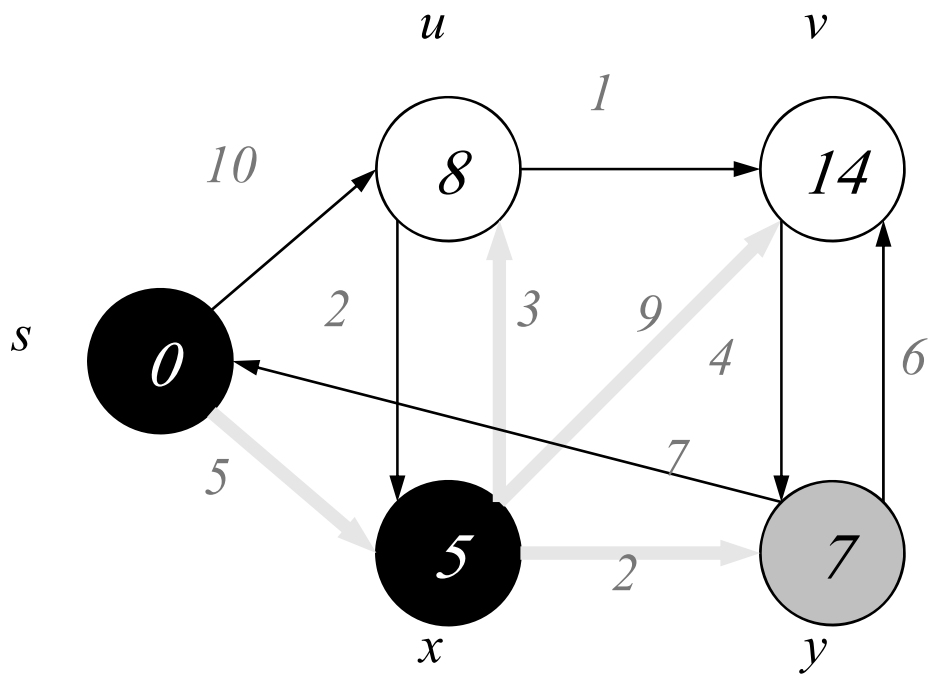
Shortest Path, Dijkstra's algorithm



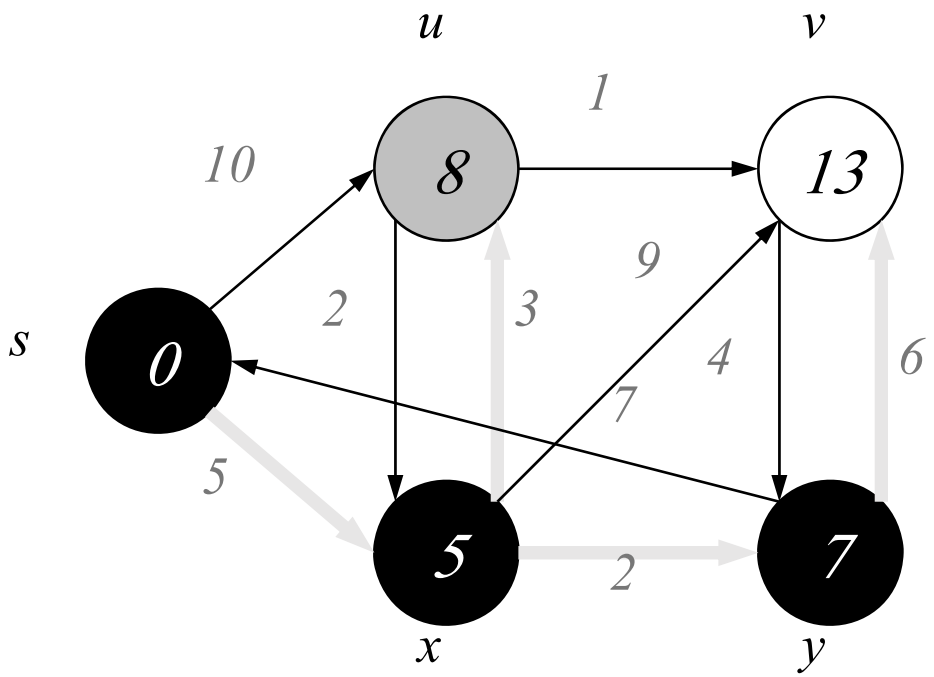
Shortest Path, Dijkstra's algorithm



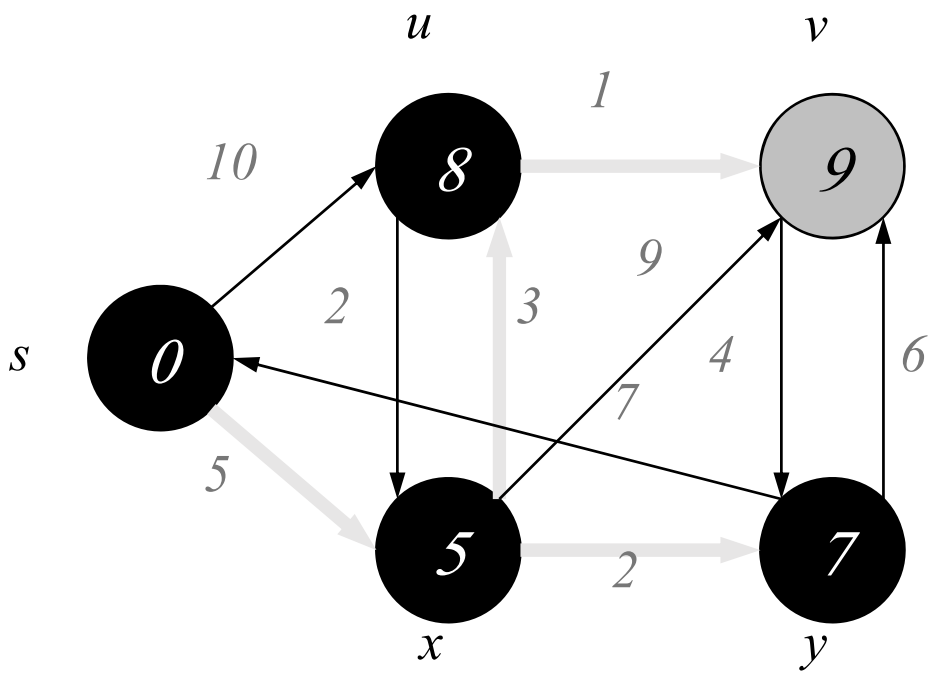
Shortest Path, Dijkstra's algorithm



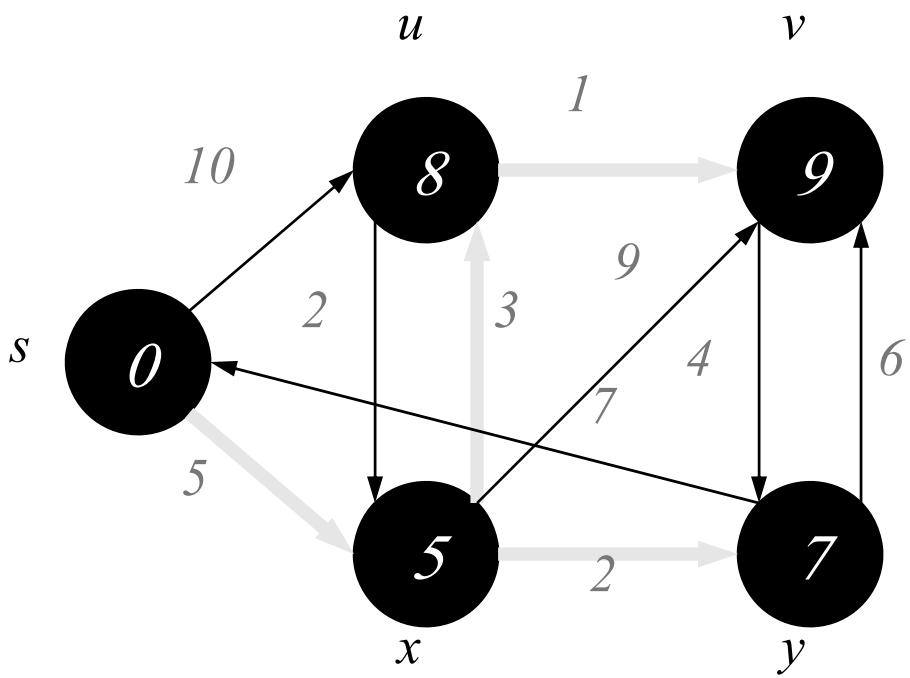
Shortest Path, Dijkstra's algorithm



Shortest Path, Dijkstra's algorithm

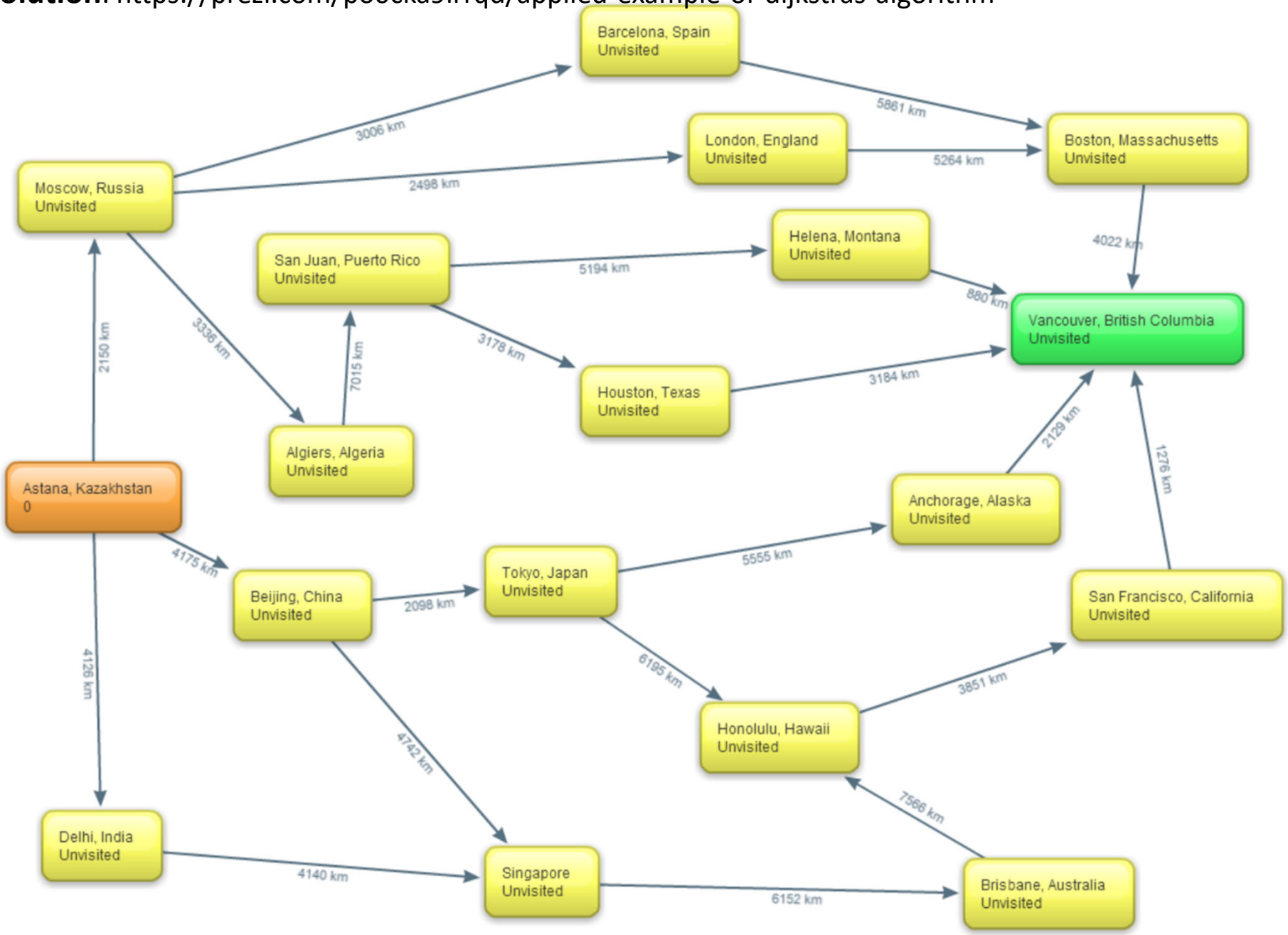


Shortest Path, Dijkstra's algorithm



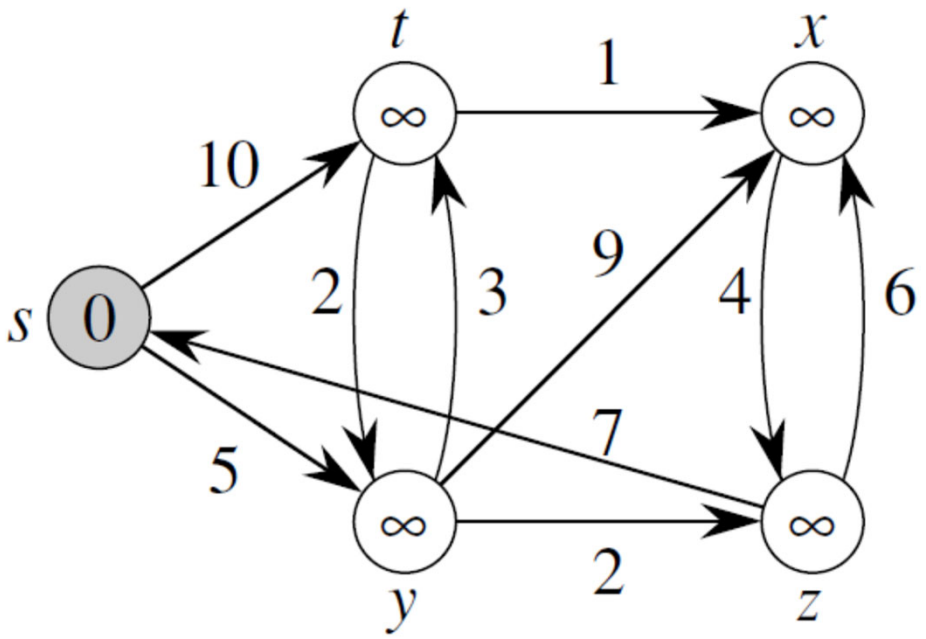
Assignment: Find the Shortest Paths using Dijkstra's algorithm

Solution: https://prezi.com/po0cka9lrrqd/applied-example-of-dijkstras-algorithm



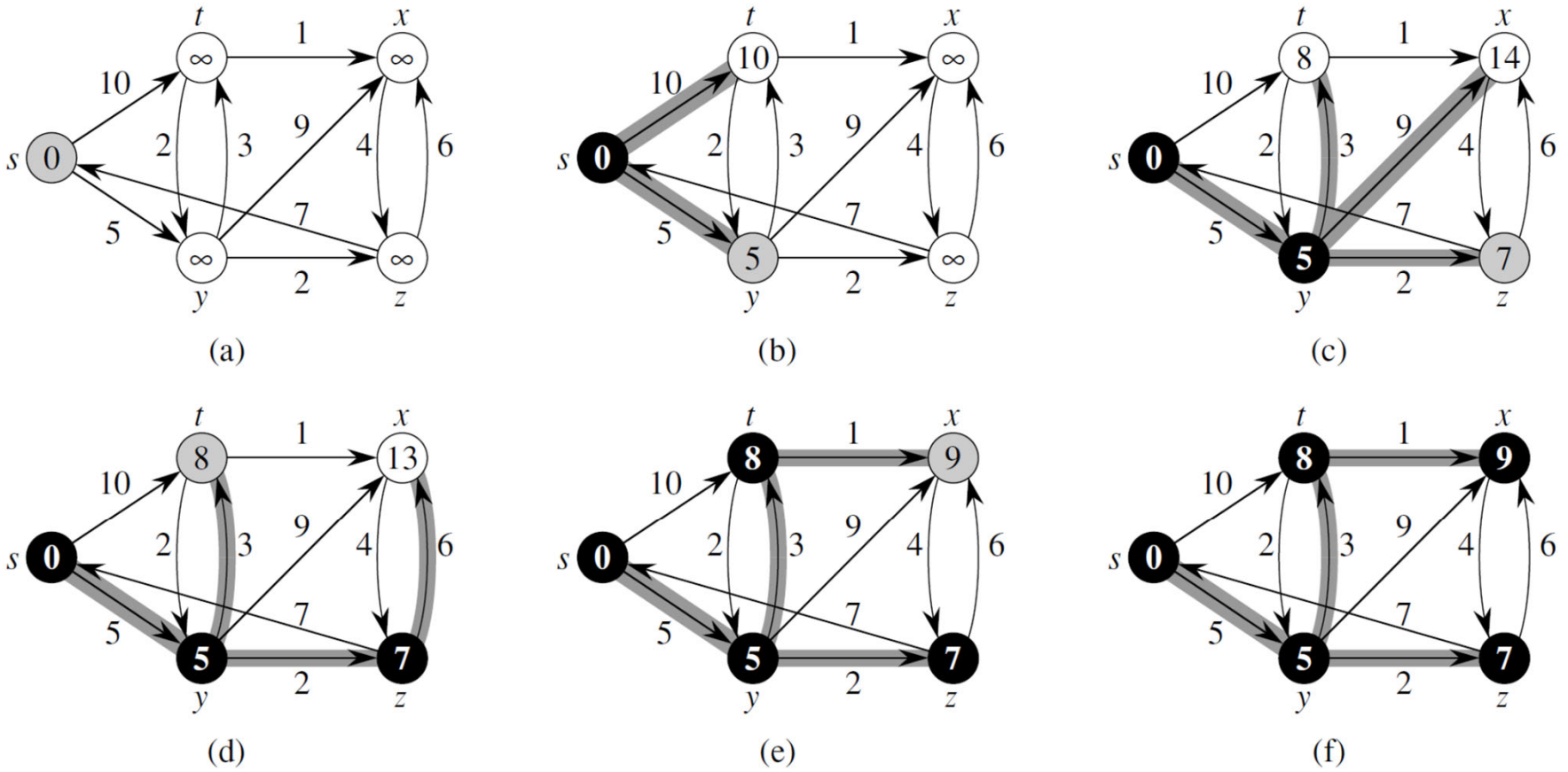
Assignment:

Find the Shortest Paths using Dijkstra's algorithm



Assignment: Solution

Find the Shortest Paths using Dijkstra's algorithm



Shortest Paths (The Bellman-Ford algorithm)

• Given is a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \rightarrow \mathrm{R}$

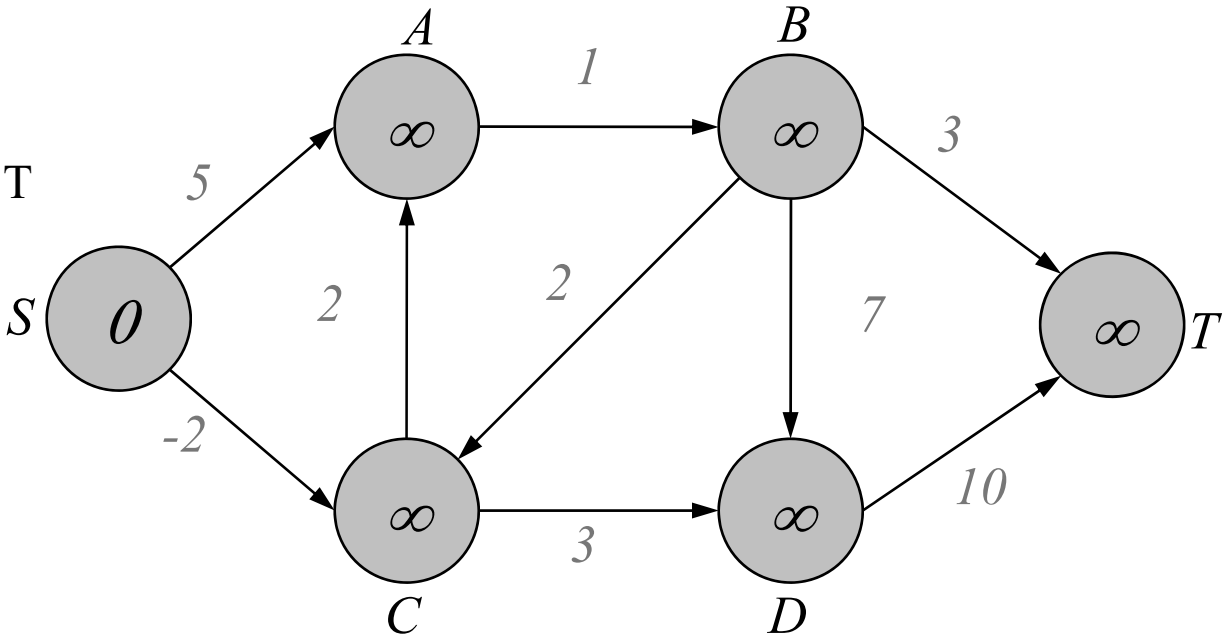
• Like Dijkstra's algorithm, the Bellman-Ford algorithm uses the technique of relaxation, progressively decreasing an estimate $d[v]$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest path weight (s, v).

•The algorithm returns `TRUE` if and only if the graph contains no negative-weight cycles that are reachable from the source

```
BELLMAN-FORD(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 for i ← 1 to |V[G]| - 1
3       do for each edge (u, v) ∈ E[G]
4             do RELAX(u, v, w)
5 for each edge (u, v) ∈ E[G]
6       do if d[v] > d[u] + w(u, v)
7             then return FALSE
8 return TRUE
```

Shortest Path, Bellman-Ford algorithm

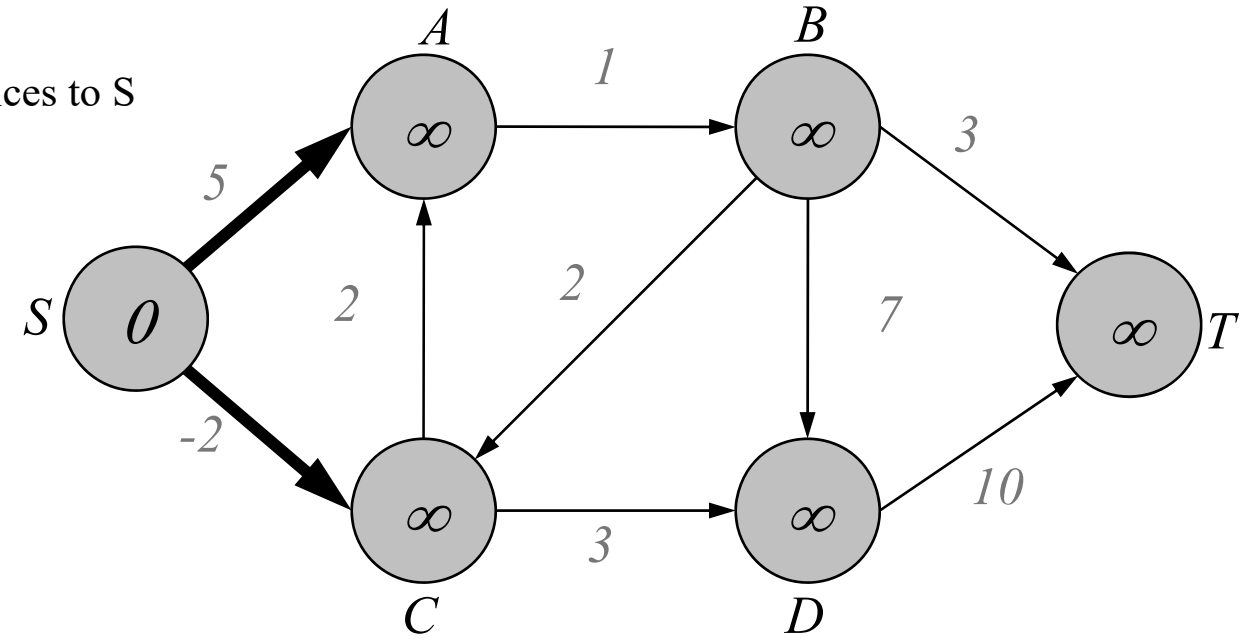
- Select Source S
- Select Destination T



	S	A	B	C	D	T
d[v]	0	∞	∞	∞	∞	∞
P[v]						

Shortest Path, Bellman-Ford algorithm

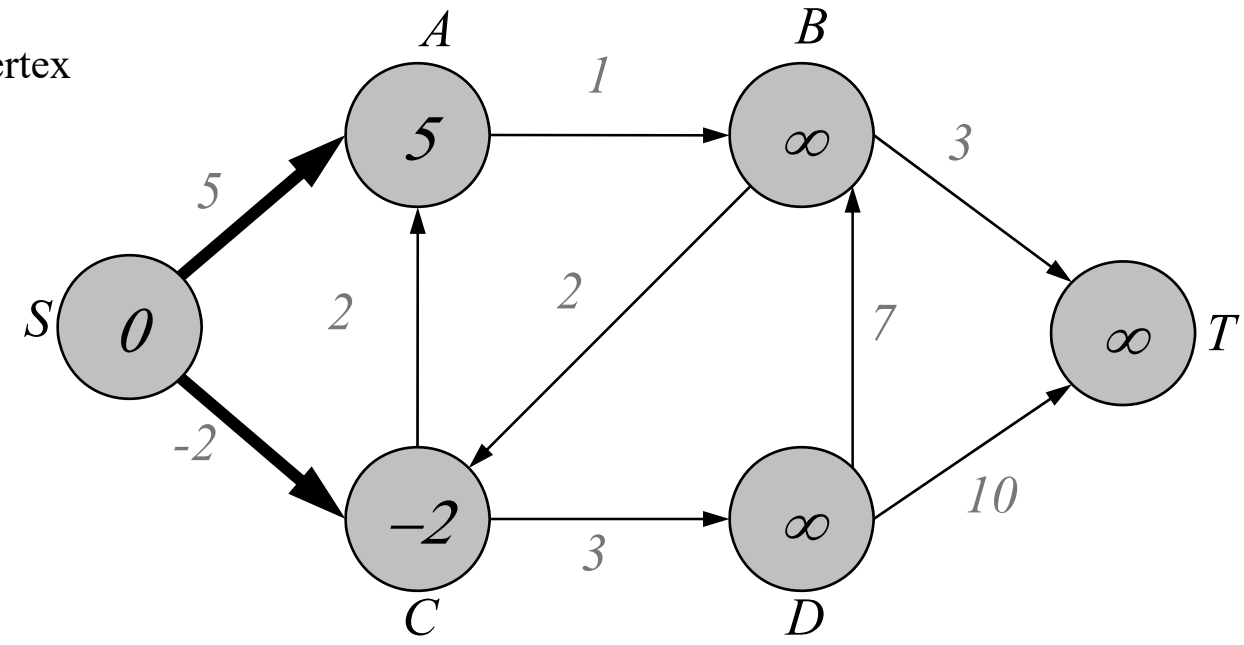
- Select adjacent vertices to S



	S	A	B	C	D	T
d[v]	0	5	∞	–2	∞	∞
P[v]		S		S		

Shortest Path, Bellman-Ford algorithm

- Continue with any relaxed vertex
- We do here alphabetical
- Select A

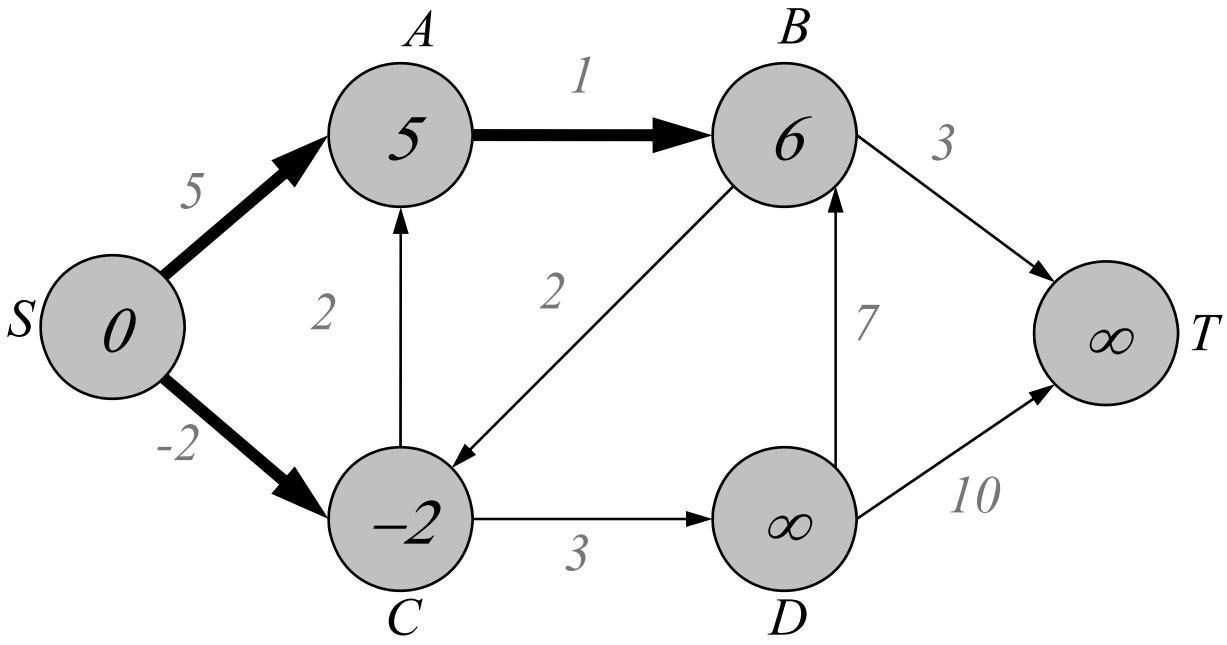


	S	A	B	C	D	T
d[v]	0	5	∞	–2	∞	∞
P[v]		S		S		

263

Shortest Path, Bellman-Ford algorithm

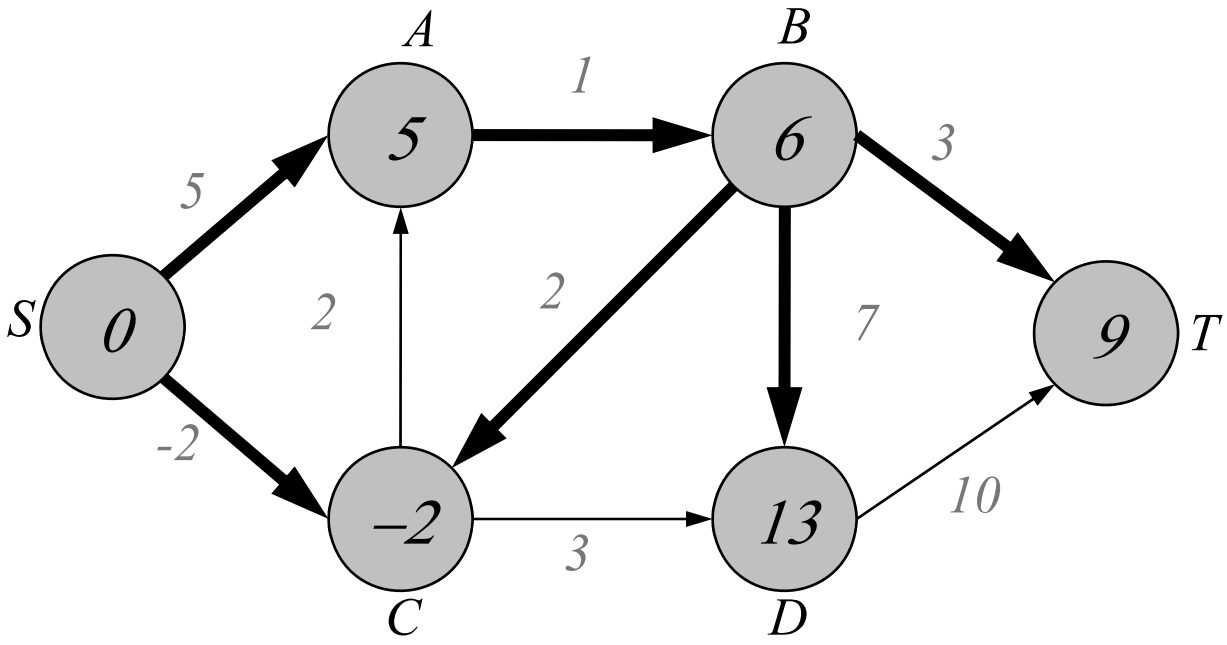
- Select A



	S	A	B	C	D	T
d[v]	0	5	6	–2	∞	∞
P[v]		S	A	S		

Shortest Path, Bellman-Ford algorithm

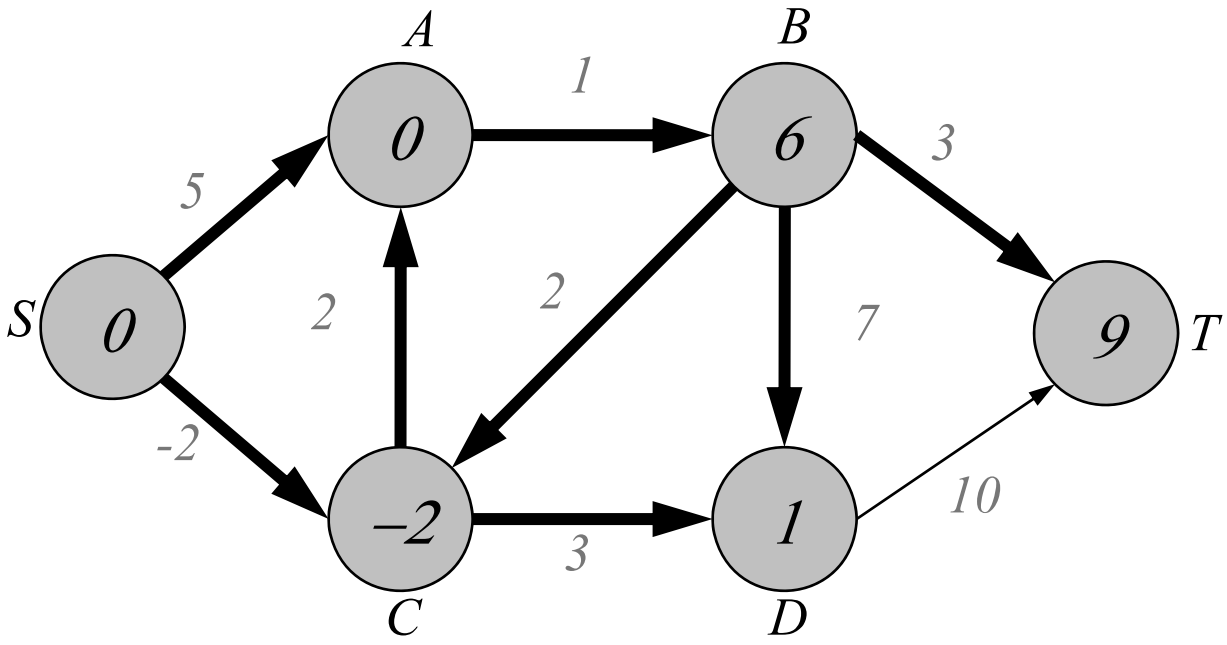
- Select any relaxed vertex
- Select B
- Relax adjacent vertices to B; C(already relaxed), D,T



	S	A	B	C	D	T
d[v]	0	5	6	–2	13	9
P[v]		S	A	S	B	B

Shortest Path, Bellman-Ford algorithm

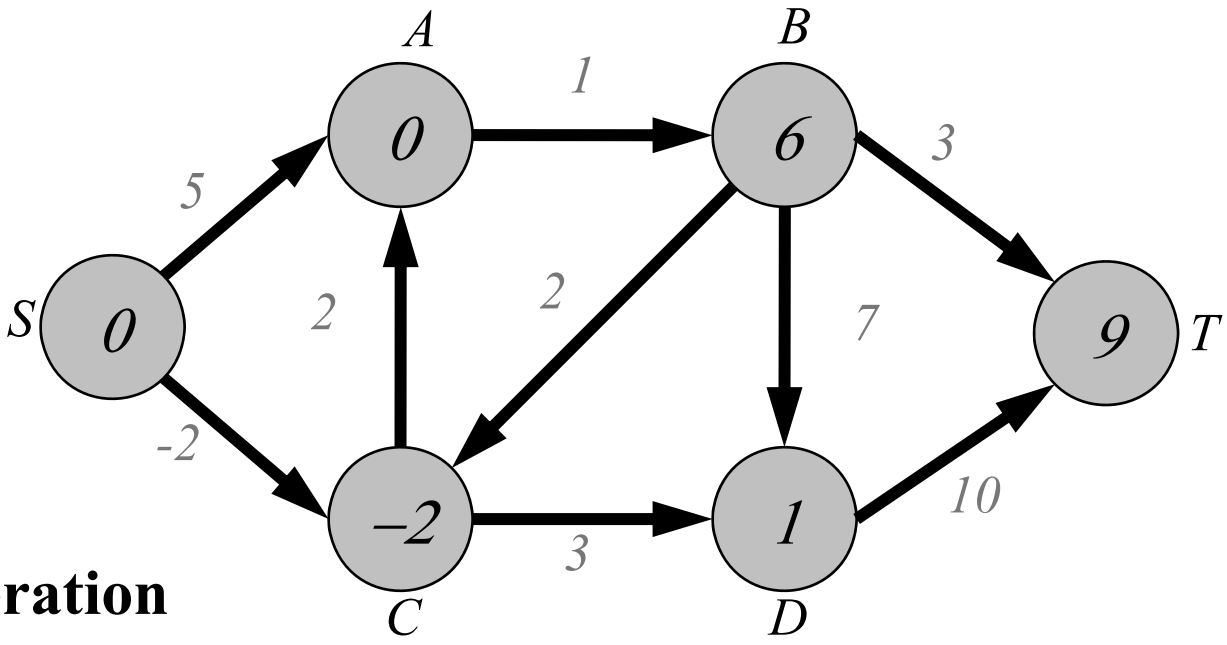
- Select any relaxed vertex
- Select C
- Relax adjacent vertices to C; A, D



	S	A	B	C	D	T
d[v]	0	~~5~~ 0	6	–2	~~13~~ 1	9
P[v]		~~S~~ C	A	S	~~B~~ C	B

Shortest Path, Bellman-Ford algorithm

- Select any relaxed vertex
- Select D
- Relax adjacent vertices to D; T(already relaxed)

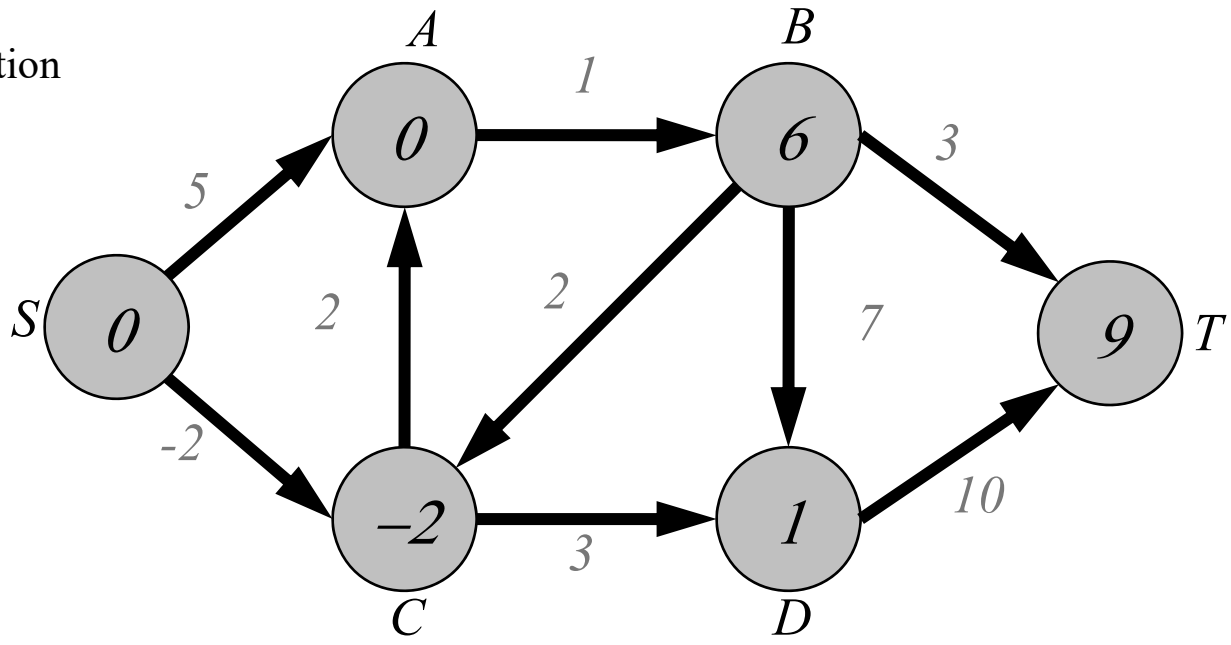


- **Done with first iteration**

	S	A	B	C	D	T
d[v]	0	~~5~~ 0	6	–2	~~13~~ 1	9
P[v]		~~S~~ C	A	S	~~B~~ C	B

Shortest Path, Bellman-Ford algorithm

- **Bellman-Ford** has V-1 iteration
- We have 6 vertices results to 6-1= 5 iterations
- Relax adjacent vertices to S; A(already relaxed), C(already relaxed)



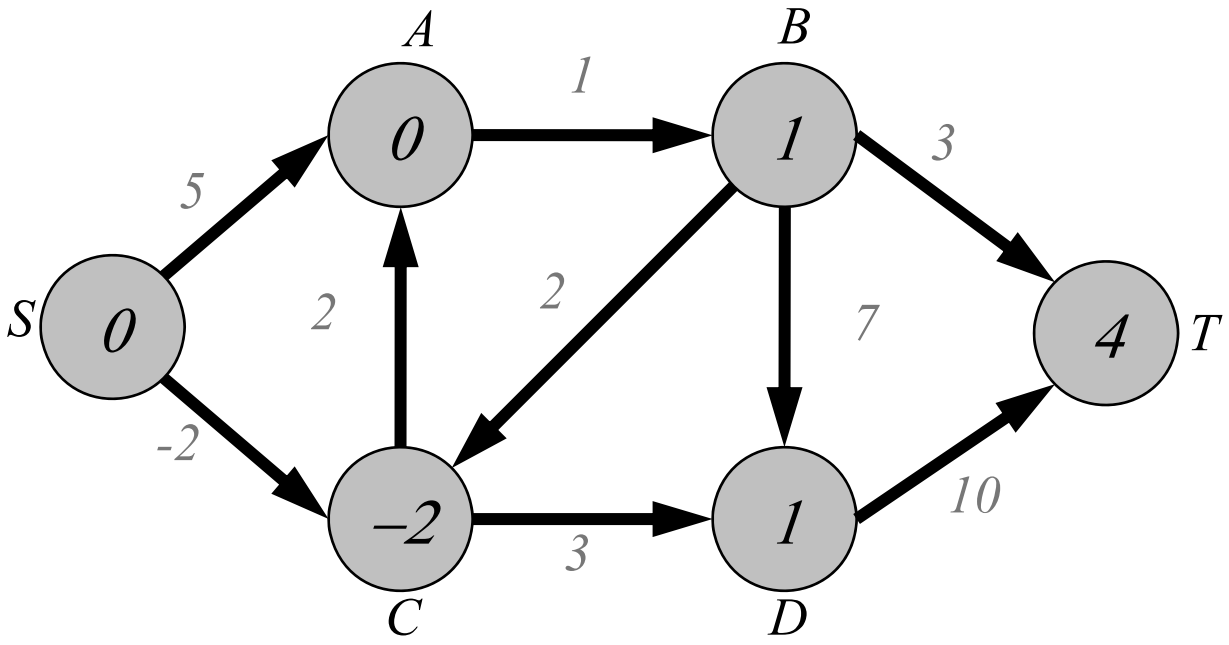
	S	A	B	C	D	T
d[v]	0	0	6	–2	~~13~~ 1	9
P[v]		C	A	S	~~B~~ C	B

Shortest Path, Bellman-Ford algorithm

- **Select A**
- Relax adjacent vertices
  B => 1

- **Select B**
- Relax adjacent vertices
  C(already relaxed) ,
  D(already relaxed),
  T => 4

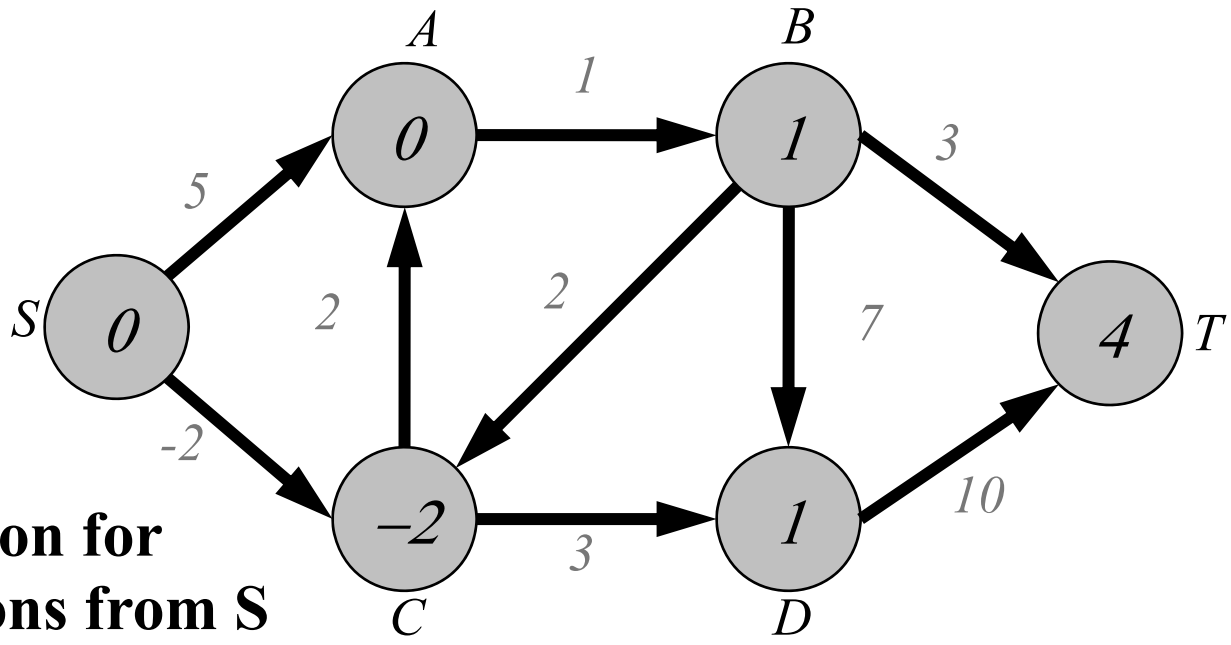
- **Select C**
- Relax adjacent vertices
  A(already relaxed)
  D(already relaxed)



	S	A	B	C	D	T
d[v]	0	0	~~6~~ 1	–2	1	~~9~~ 4
P[v]		C	A	S	C	B

Shortest Path, Bellman-Ford algorithm

- **Select D**
- Relax adjacent vertices T(already relaxed)
- **Select T**
- Relax adjacent vertices no adjacent vertices

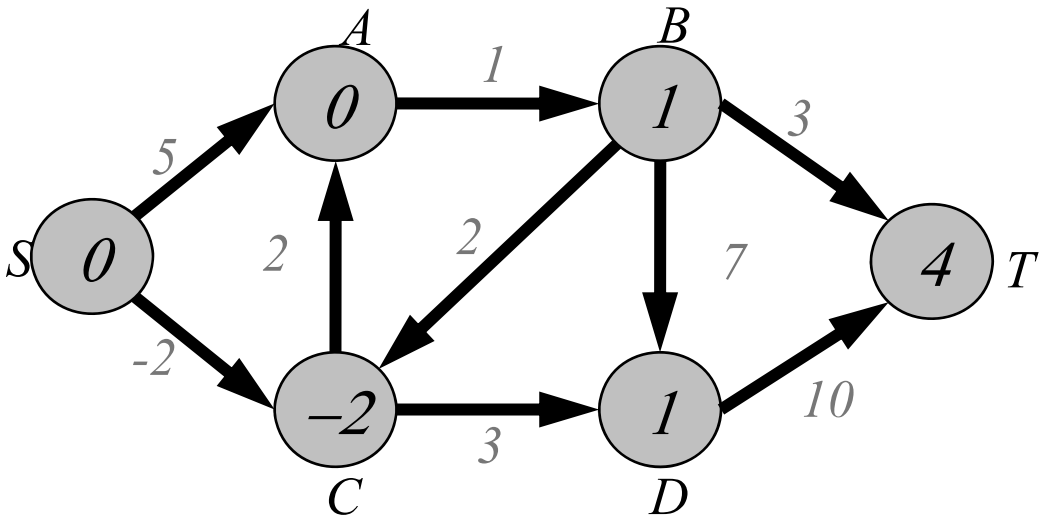


- **Repeat the relaxation for remaining 3 Iterations from S**

	S	A	B	C	D	T
d[v]	0	0	~~6~~ 1	−2	1	~~9~~ 4
P[v]		C	A	S	C	B

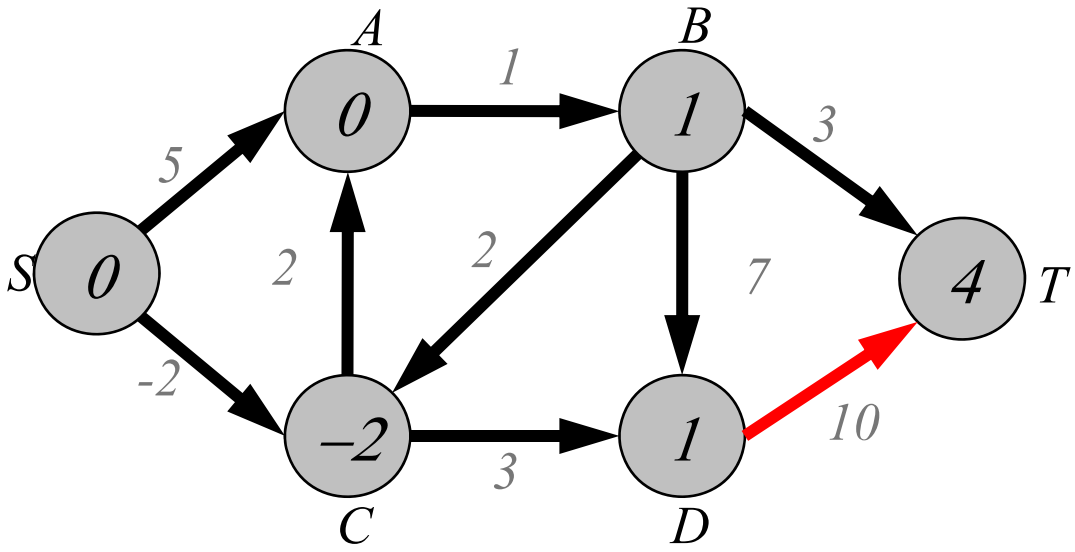
Shortest Path, Bellman-Ford algorithm

•**Repeat the relaxation for remaining 3 Iterations from S**



• If the d[v] still changes after n iterations (# of Vertices, here is 6) then the Bellman Ford can not find the shortest path. The graph has a negative cycle.

Shortest Path, Bellman-Ford algorithm



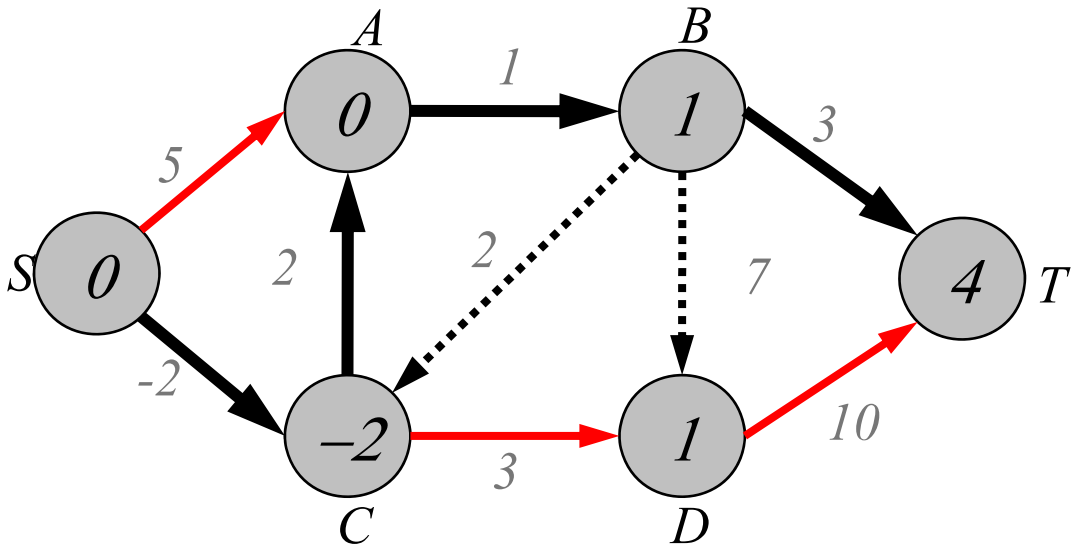
- **Last step is finding the shortest path**

- Return backwards from Destination T to Source S. If valid returns 0 in S

- T=4:

  to D returns 4-10=1 (invalid),
  to B returns 4-3=1 (Valid)

Shortest Path, Bellman-Ford algorithm



- B=1:

  to A returns 1-1=0 (Valid)

- A=0:

  to S returns 0-5=0 (Valid)
  to C returns 0-2=-2 (Valid)

- C=1:

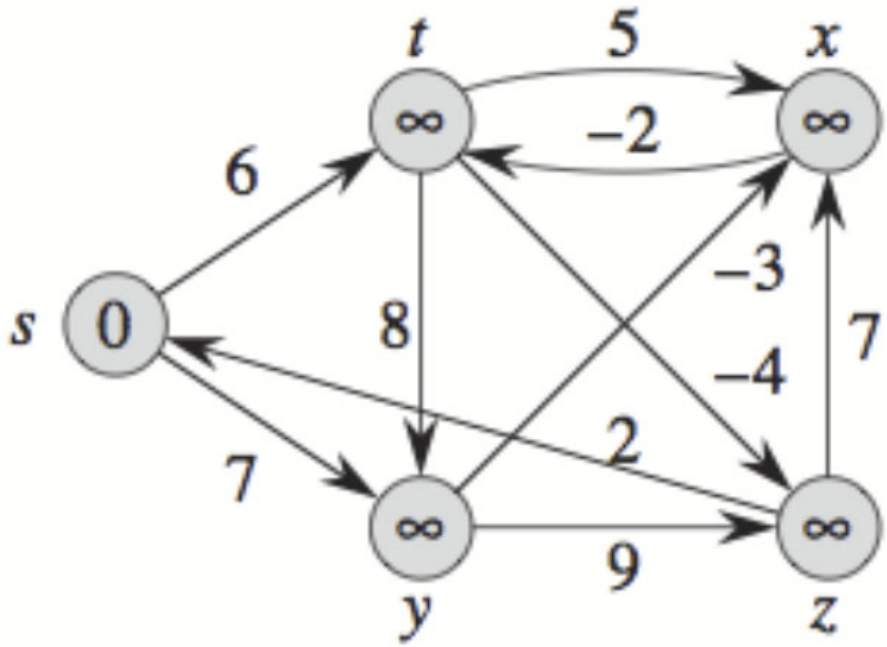
  to S returns -2-(-2)=0 (Valid)

Shortest path is

T ← B ← A ← C ← S

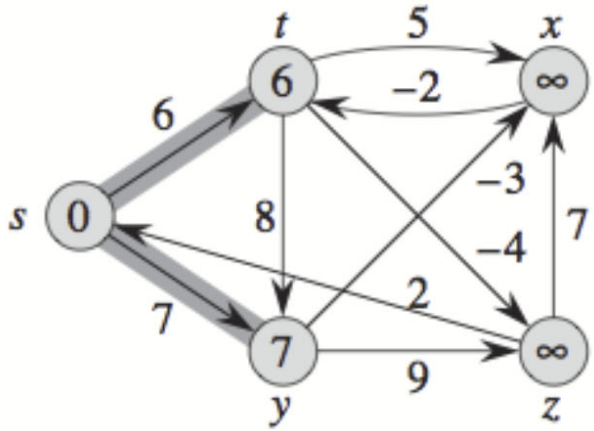
Shortest Path, Bellman-Ford algorithm

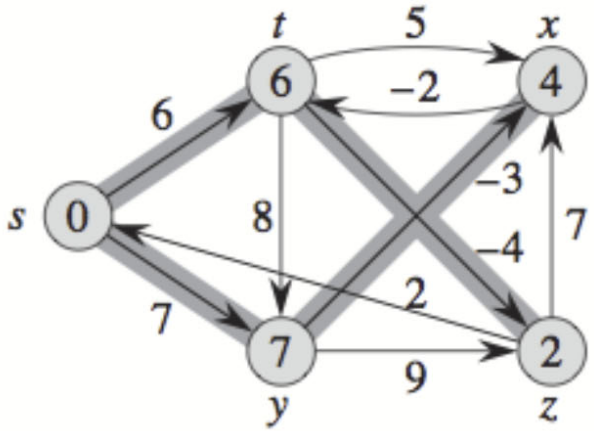
Assignment: *s* to *z*

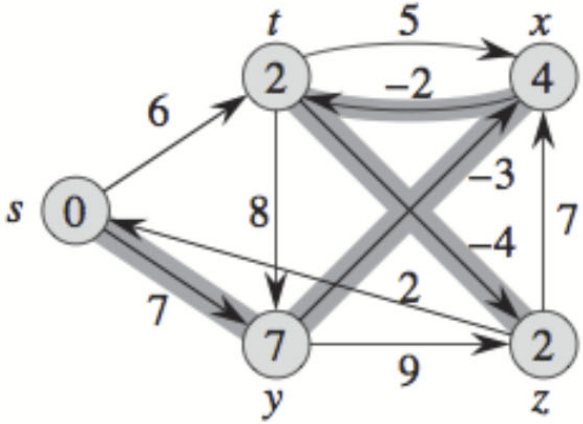


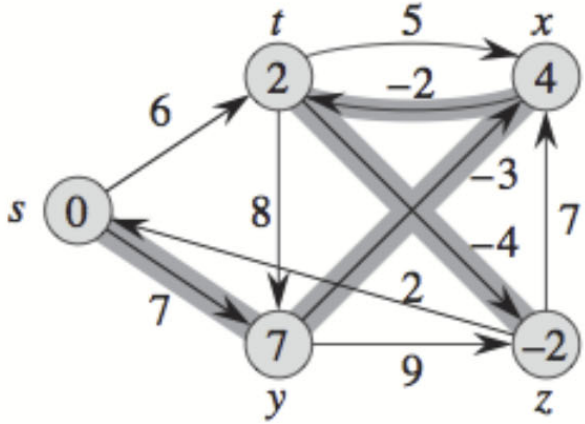
Shortest Path, Bellman-Ford algorithm

Solution

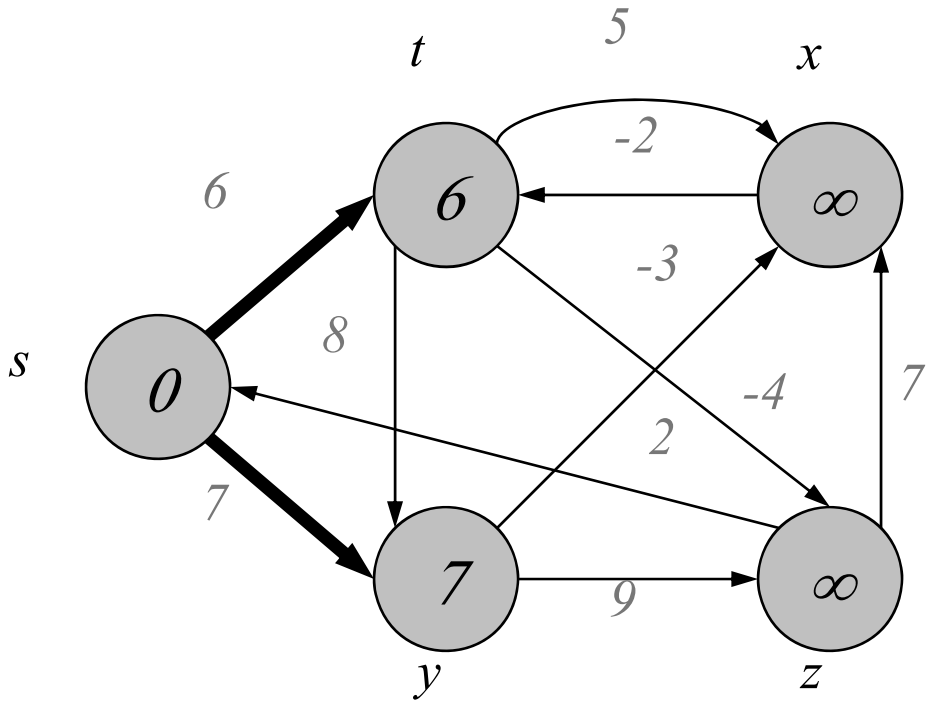




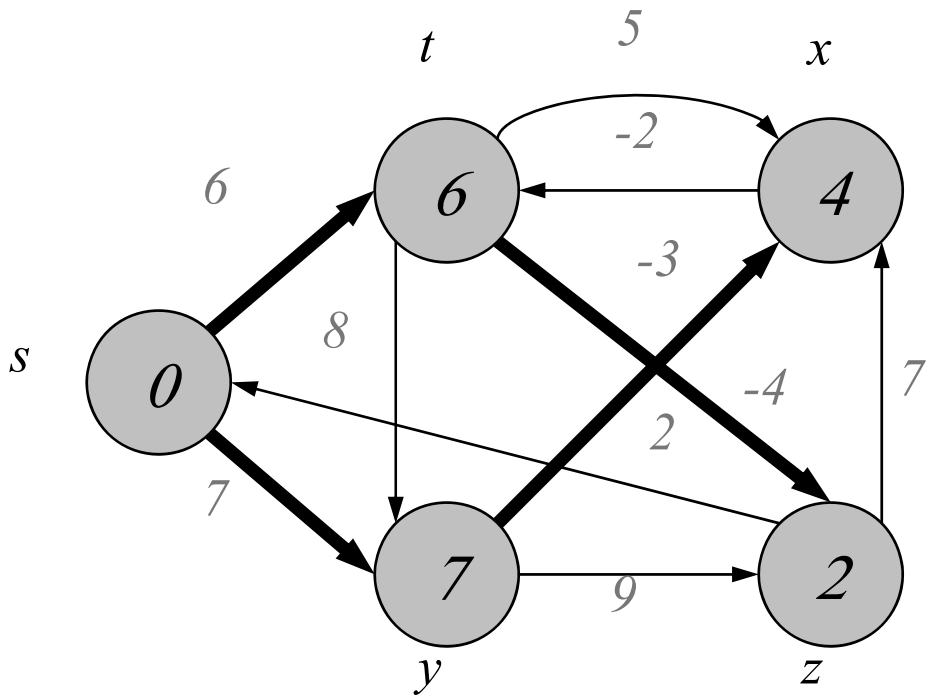




Shortest Path, Bellman-Ford algorithm



Shortest Path, Bellman-Ford algorithm



Shortest Path, Bellman-Ford algorithm