

# Introduction to Algorithms

Time complexity, Asymptotic notations, Techniques- Divide and Conquer, Dynamic Programming, Greedy algorithms, Backtracking, Branch and Bound

# Algorithm

- An algorithm is any well defined computational procedure that takes some values or set of values as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into output
- An algorithm is a finite set of instructions/steps that if followed accomplishes a particular task
- An algorithm must have the following:
  - Zero or more input
  - At least one output
  - Definiteness – each instruction is clear and unambiguous
  - Finiteness – algorithm must terminate after finite number of steps
  - Effectiveness – every instruction must be basic

# Analysis of Algorithm

- The purpose of analysis of algorithms is two fold:
  - To obtain estimates of bounds on the storage or run-time which the algorithms needs to successfully process the input
  - To find an efficient algorithm for solving a particular problem
- There are two different ways of analyzing an algorithm:
  - Priori analysis
  - Posteriori analysis
- Space Complexity: amount of memory that the algorithm needs to run to completion
- Time Complexity: amount of computer time that the algorithm needs to run to completion

# Time Complexity Analysis

- The time complexity  $T(p)$  of an algorithm  $p$  can be defined in terms of the number of steps the algorithm takes to run to completion
- The number of steps taken by an algorithm is proportional to the problem size
- For a given problem size, the time complexity can be defined in terms of:
  - number of operations performed
  - time of each operation
  - frequency of each operation
- It is assumed that each operation takes more or less same time to execute. Let this time be  $t$  then, an algorithm with  $n$  steps will take  $tn$  time to run
- The goal is to find the upper and lower bound on this run time

# Asymptotic Notations

- asymptotic notations describe the bounds on the run time of an algorithm in terms of some function of the problems size
- Thus, asymptotic notations essentially find out a functional relationship between the problems size and time required to solve the problem using a particular algorithm
- these notations try to give an upper or lower bound (or both) on the run time of an algorithm
- We will mainly focus on three different types of notations  $O, \theta, \omega, \Omega$
- Among these Big-O notation is most commonly used

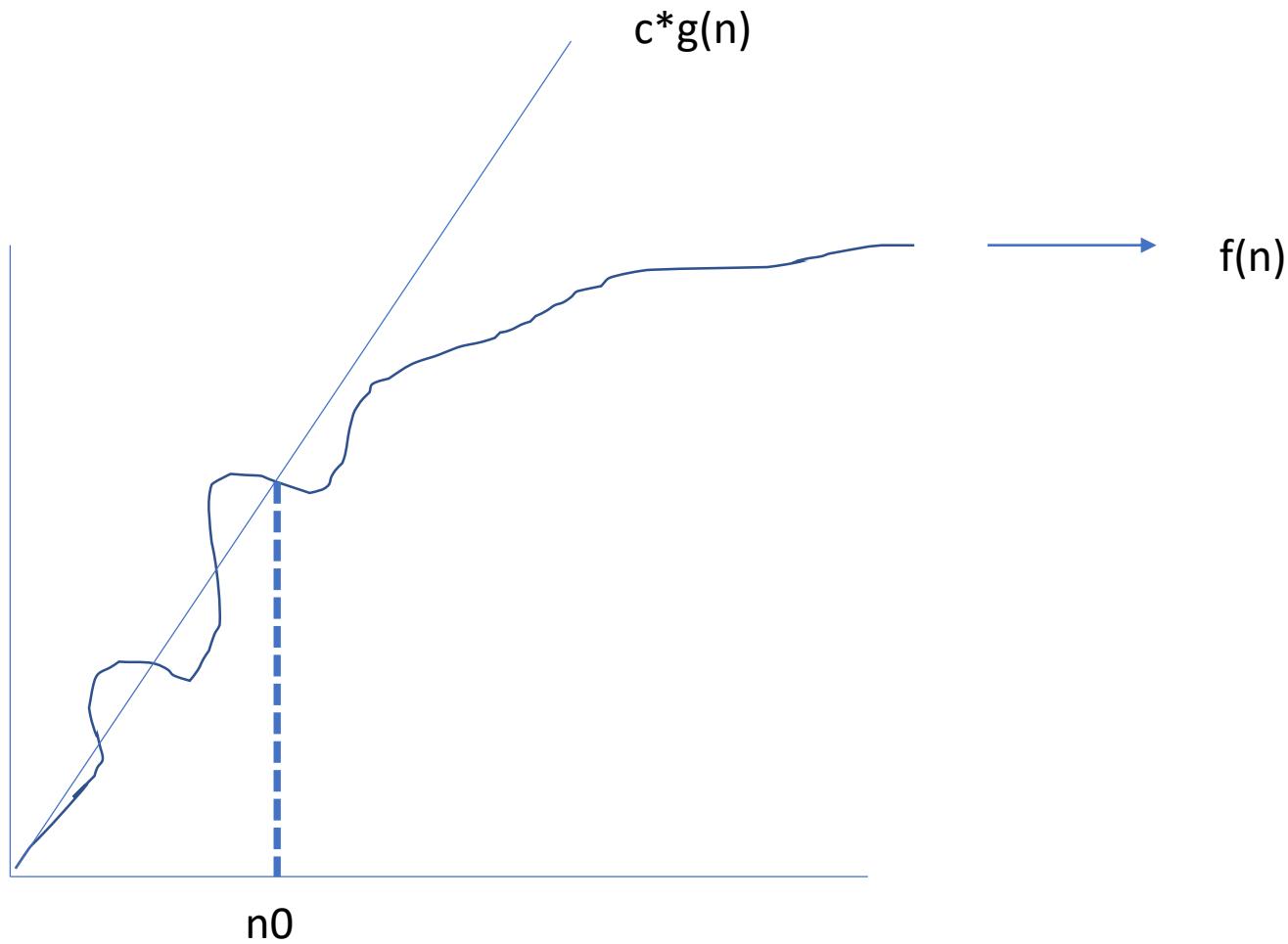
# Big-O Notation

- For a given function  $f(n)$  we define order of  $g(n)$  to be the set of functions such that

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

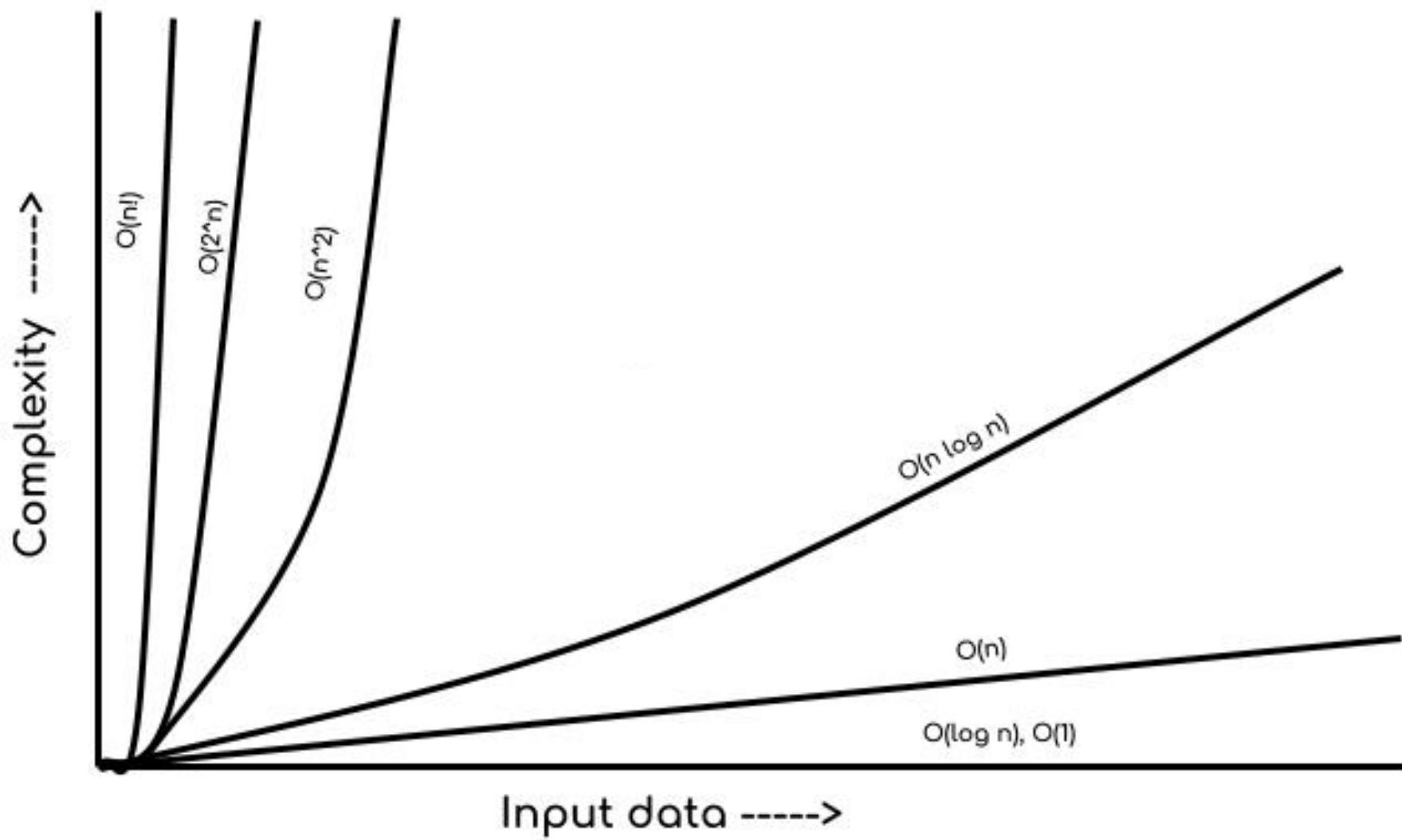
$$f(n) \leq c * g(n) \quad \forall n >= n_0$$

- $f(n)$  essentially provides an upper bound on  $g(n)$  i.e. value of  $g(n)$  will always be less than  $f(n)$
- For lower values of  $n$ , this might not be true but for sufficiently large  $n$ , the value is always lower than  $f(n)$
- Establishes lowest upper bound



# Examples: Big-O

- $3n+3 = O(n)$  { $c = 4: 4n \ n_0 \geq 3$ }
- $10n^2 + 4n + 2 = O(n^2)$  { $\leq 11n^2 \ \forall n \geq 5$ }
- $6 * 2^n + n^2 = O(2^n)$  { $\leq 7 * 2^n \ \forall n \geq 4$ }
- Here,  $n$  is the problem size
- It indicates the number of inputs to the algorithm
- $O(1)$  or  $O(p)$  where  $p$  is a constant, indicates that the algorithm takes constant amount of time no matter what the problem size is
- That is, the algorithm is independent of the problem size



- The statement  $f(n) \leq g(n)$  only says that  $f(n)$  is an upper bound on  $g(n)$  for all values of  $n \geq n_0$ , but it is silent about how good this upper bound is.
- But, for the statement  $f(n) \leq g(n)$  to be informative, the function  $g(n)$  should be as small a function of  $n$  as one can come up with

# (Big)- $\Omega$ Notation

- For a given function  $f(n)$  we define  $\Omega(g(n))$  to be the set of functions such that

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

$$f(n) \geq c * g(n) \quad \forall n \geq n_0$$

- $f(n)$  essentially provides an lower bound on  $g(n)$  i.e. value of  $g(n)$  will always be greater than  $f(n)$
- For lower values of  $n$ , this might not be true but for sufficiently large  $n$ , the value is always greater than  $f(n)$
- This establishes highest lower bound

# Examples – Big Ω

- $3n+3 = \Omega(n)$                           as  $3n+3 \geq 3n \forall n$
- $10n^2 + 4n + 2 = \Omega(n^2)$                   as  $10n^2 + 4n + 2 \geq 10n^2 \forall n$
- $6 * 2^n + n^2 = \Omega(2^n)$                   as  $6 * 2^n + n^2 \geq 6 * 2^n \forall n$
- bound is always chosen as close as possible
- The function  $f(n)$  is only a lower bound on  $g(n)$  but for the statement  $f(n) = \Omega(g(n))$  to be informative  $g(n)$  should be as large a function of  $n$  as possible

# $\theta$ – Notation

$$f(n) = \theta(g(n)) \quad \text{iff}$$

$\exists$  positive constant  $c_1, c_2$  and  $n_0$  such that

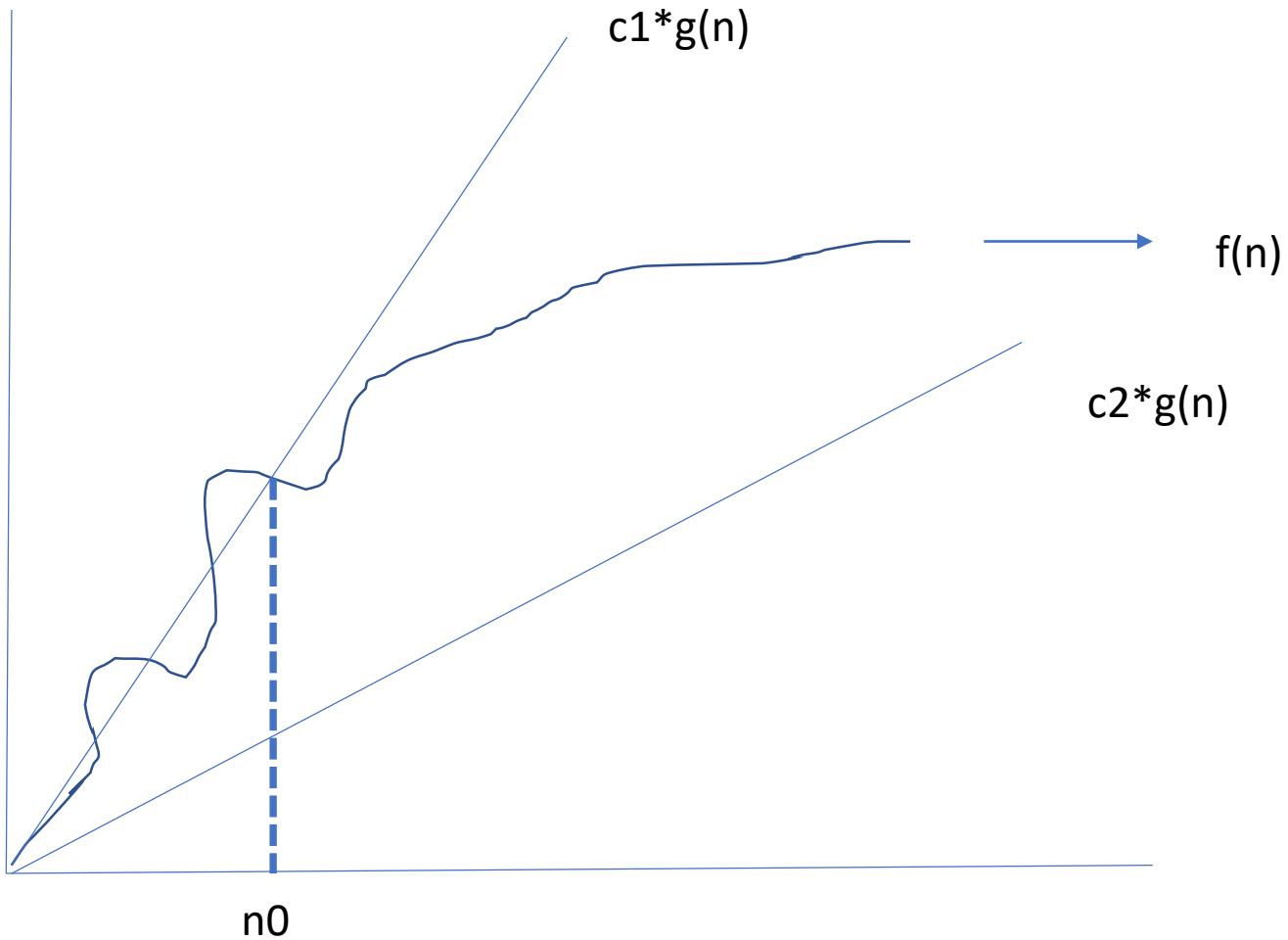
$$c_1 * g(n) \leq f(n) \leq c_2 * g(n) \quad \forall n \geq n_0$$

- establishes both upper and lower bound

➤  $3n \leq 3n + 2 \leq 4n \quad \forall n_0 \geq 2$

➤  $3n + 2 = \theta(n)$

➤ This method also establishes tight bound



# $\text{o}$ -Notation

$$f(n) = o(g(n)) \text{ iff } 0 \leq f(n) < c * g(n) \quad \forall n \geq n_0$$

or

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

Establishes loose upper bound

# Examples: o-Notation

- $3n + 2 = o(n \log n)$   
 $= o(n^2)$   
 $= o(2^n)$   
 $\neq o(n)$

We always try to find lose bound not tight

# $\omega$ – Notation

$$f(n) = \omega(g(n)) \quad \text{iff} \quad 0 < c^*g(n) < f(n) \quad \forall n \geq n_0$$

or

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

Establishes loose lower bound

# Examples - $\omega$ Notation

- $10n^3 + 6 = \omega(n)$   
=  $\omega(n^2)$   
 $\neq \omega(n^3)$

We always try to find lose lower bound

# Comparing Functions

Transitivity:

$$f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \text{ imply } f(n) = \Theta(h(n)) ,$$

$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \text{ imply } f(n) = O(h(n)) ,$$

$$f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \text{ imply } f(n) = \Omega(h(n)) ,$$

$$f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) \text{ imply } f(n) = o(h(n)) ,$$

$$f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \text{ imply } f(n) = \omega(h(n)) .$$

**Reflexivity:**

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

$$f(n) = O(f(n)) ,$$

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

**Symmetry:**

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) .$$

**Transpose symmetry:**

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)) ,$$

$$f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n)) .$$

# Trichotomy

- For any two real numbers  $a$  and  $b$ , exactly one of the following holds:
  - $a=b$
  - $a < b$
  - $a > b$
- However, this is not true for asymptotic notation
- It is possible that for a function neither  $f(n) = O(g(n))$  nor  $f(n) = \Omega(g(n))$  holds
- E.g. the two functions  $n$  and  $n^{1+\sin(n)}$  cannot be compared using asymptotic notation

# Time Complexity Analysis

- While carrying out time complexity analysis for an algorithm, we normally work out time for several values of  $n$ . This is done because:
  - Asymptotic analysis tells us the behavior only for sufficiently large values of  $n$ . For smaller values of  $n$ , the run-time may not follow the asymptotic curve. To determine the break-even point beyond which the asymptotic curve is followed, we need to examine the times for several values of  $n$
  - Even in the region where asymptotic behavior is exhibited, the times may not lie exactly on the predicted curve because of the effect of low order terms that are ignored in the asymptotic analysis

# Best, Worst and Average case analysis

- When an algorithm is run in minimum time (i.e. minimum number of steps), we call it the Best-Case Execution
  - When an algorithm is run in maximum time (i.e. maximum number of steps), we call it Worst-Case Execution
  - The average case lies somewhere in between best and worst case
- Normally, we report only the worst-case running time for an algorithm due to several reasons:

- The worst case running time of an algorithm is an upper bound on the running time or execution time for any input instance. Knowing this gives us a guarantee that the algorithm will never take time longer than this
- For some algorithms, the worst case fairly often occurs. E.g. Searching for a particular data item in the list when the list doesn't contain it, the worst case fairly often occurs
- The average case is roughly as bad as the worst case. Moreover, it may not be apparent what constitutes an average case input for a particular algorithm
- The scope of average-case analysis is limited, because it may not be apparent what constitutes an “average” input for a particular problem. Often we assume that all inputs of a given size are equally likely

# Polynomial and Non-Polynomial Time Algorithm

- For most of the problems, we often come across the best algorithm for their solutions have computing times that cluster into two groups:
- The first group consists of problems whose solution times are bound by polynomials of small degree e.g. ordered searching ( $O(\log n)$ ), polynomial evaluation ( $O(n)$ ), sorting( $O(n \log n)$ ) and storing editing ( $O(mn)$ )
- The second group comprises of problems whose best known algorithms are bounded by non-polynomial time. E.g. TSP( $O(n^2 * 2^n)$ ) and Knap-sack problem ( $O(2^n)$ )
- We are interested in finding a polynomial time algorithm for any given problem if possible

# Decision Problem and Optimization Problem

- Any problem that has only a two-class solution either 0(no) or 1(yes) is called a decision problem
- An algorithm for a decision problem is termed as decision algorithm
- Any problem that involves identification of an optimal value (either maximum or minimum) of a given cost function is known as an optimization problem
- Many optimization problems can be recast into decision problems with the property that the decision problem can be solved in polynomial time if and only if the corresponding optimization problem is solvable in polynomial time

# Greedy Algorithms

- It is hard to define exactly what is meant by greedy algorithms
- An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion
- Often more than one greedy algorithms may be designed for a particular problem
- If a greedy algorithm is able to solve a non-trivial problem, it indicates that there exists a local decision rule that can be used to construct optimal solutions for the problem
- In some cases, greedy algorithms can find close to optimal solutions
- In general, it is easy to implement greedy algorithm for almost any problem however, finding cases in which they work well and proving that they work well is a challenge
- Algorithms for finding shortest path and minimum spanning tree are based on greedy technique

# Divide and Conquer

- Divide and conquer refers to a class of algorithmic techniques where we break the input into several parts, solve the problem in each part recursively, and then combine the solutions to these sub-parts into an overall solution
- For many problems, this technique can prove to be a simple and yet powerful method
- Analyzing the running time here generally involves solving a recurrence relation
- In many cases, divide and conquer is combined with other algorithm design techniques to give efficient algorithms
- Example: Merge sort algorithm that you will see later is based on divide and conquer technique

# Dynamic Programming

- Greedy approach for problem solving is the most natural and intuitive way
- Divide and conquer technique proves helpful in cases where no natural greedy algorithm can be identified
- However, in more complex cases even the divide and conquer technique may not give reasonable improvement over the brute force search
- In such cases, a more powerful and subtle design technique known as dynamic programming is applied
- This technique draws its intuition from divide and conquer and is essentially opposite of greedy technique

- Here, we implicitly explore the space of all possible solutions, by carefully decomposing things into a series of subproblems and then build up correct solutions to larger and larger subproblems
- Thus, we follow a bottom-up approach in solving the problem
- As we try to design correct algorithm by implicitly exploring the solution space, there is always danger of reaching close to a brute-force search, application of this technique therefore requires carefully designing the algorithm
- It should be noted that although dynamic programming works by examining exponentially large set of solutions, it never examines them all explicitly, this careful balancing of limited examination and building solution by combining the subproblems that makes this technique tricky to get used to
- Floyd's algorithm for finding all pair shortest path is based on dynamic programming

# Backtracking

- It is a general technique to find all possible solutions to a given computational problem
- It is mainly used for constraint satisfaction problems
- It builds the solution in an incremental manner and discards a candidate as soon as it realizes that the candidate might possibly not lead to a solution
- In practice, this technique also solves the problem in a recursive manner by satisfying some constraint, a candidate is rejected as soon as it is unable to satisfy the constraint

# Branch and Bound

- This technique is used in optimization problem, i.e. for finding the optimal solution to a given problem
- This is the main difference between backtracking and branch and bound
- Here, the candidate solutions are thought to be arranged in the form of a rooted tree and the solution is explored by traversing through the branches of the tree by solving the subsets of problems
- Each branch is first tested for the upper and lower bounds expected on the solution and the branch is discarded if it doesn't satisfy the bounds
- Branch and bound is mainly used for discrete and combinatoric optimization problems

# Divide and Conquer Approach

Merge sort, Integer multiplication, Solving recurrence relation

# General Idea

- In this approach, the problem is solved recursively, i.e., a procedure calls itself several times in order to solve the problem
- Idea is to divide the original problem into several sub-problems that are either same or closely related to the original problem but are smaller in size
- The results of sub problems is combined to get the solution to original problem
- The combination procedure may be slightly unrelated to the original problem and might be treated as an overhead

# Steps:

- The divide and conquer paradigm has three steps:
  1. **Divide**: the problem into a number of subproblems that are similar instances of the original problem
  2. **Conquer**: the subproblems by solving them recursively, if the subproblems are small however, solve them in a straightforward manner
  3. **Combine**: the solutions to the subproblems into the solution of the original problem

# Merge Sort

- The merge sort procedure closely follows the divide and conquer approach of problem solving.
- Steps of merge sort:
  1. Divide: the  $n$ -element array into two arrays of size  $n/2$  each
  2. Conquer: sort the two sub-arrays recursively
  3. Combine: Merge the two sorted sub-arrays to get the final sorted array
- The procedure is said to “bottom out” when there are no more elements left to divide
- At this point, the merge procedure is invoked

# Merge Process

*Time complexity:  $\theta(n)$*

**MERGE( $A, p, q, r$ )**

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

# Analyzing the Merge Process

- Steps 1 to 3 take constant time (say  $c_0$ )
- Steps 4 to 5 take –  $n_1*c_1$  time
- Steps 6 to 7 take –  $n_2*c_1$  time
- Steps 8 to 11 take constant time  $c_3$
- Steps 12 to 17 take  $(r-p+1)*c_4$  time
- Total time –  $c_0 + n_1*c_1 + n_2*c_2 + c_3 + (r-p+1)*c_4$
- $\Rightarrow c_0 + (r-p+1)c_1 \Rightarrow n*c_1$
- $\Rightarrow \theta(n)$  where  $n = r-p+1$

# Merge Sort Procedure

$$T(n) = \theta(1) \quad \text{for } n = 1$$

**MERGE-SORT( $A, p, r$ )**

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

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

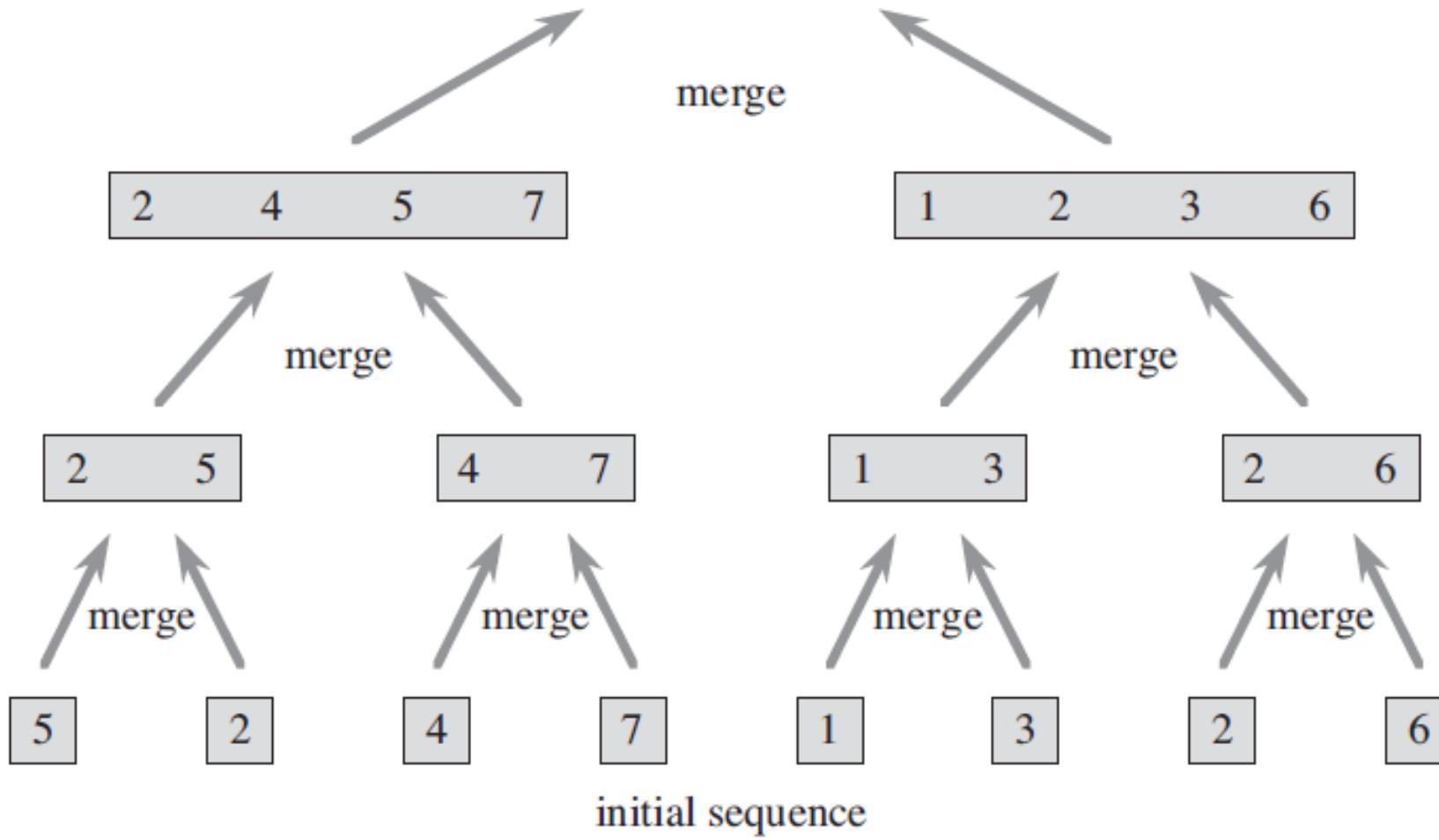
# Analysis of Divide and Conquer Algorithms

- The running time can be described using recurrence equation or simply recurrence
- It describes the overall running time of the algorithm on a problem size of ‘n’ in terms of running time on smaller inputs
- Mathematical tools can be used to solve the recurrence and obtain the bounds on running time

sorted sequence

1	2	2	3	4	5	6	7
---	---	---	---	---	---	---	---

$$\frac{n}{2^i} = 1$$

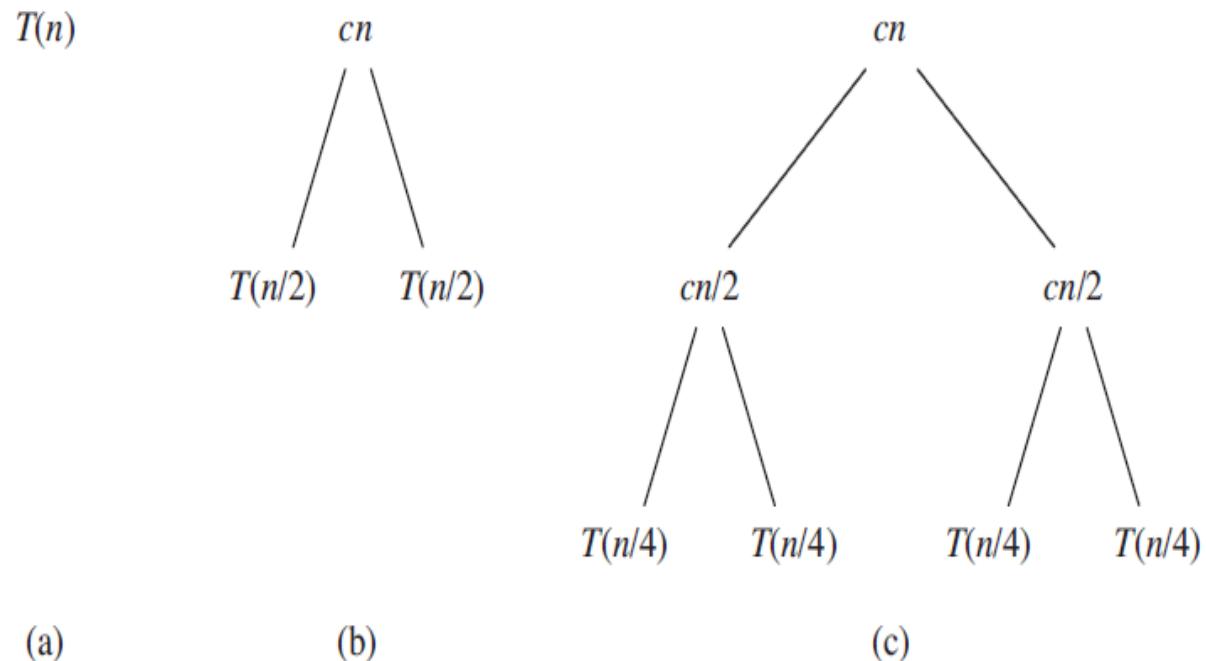


# Recurrence relation for Merge Sort

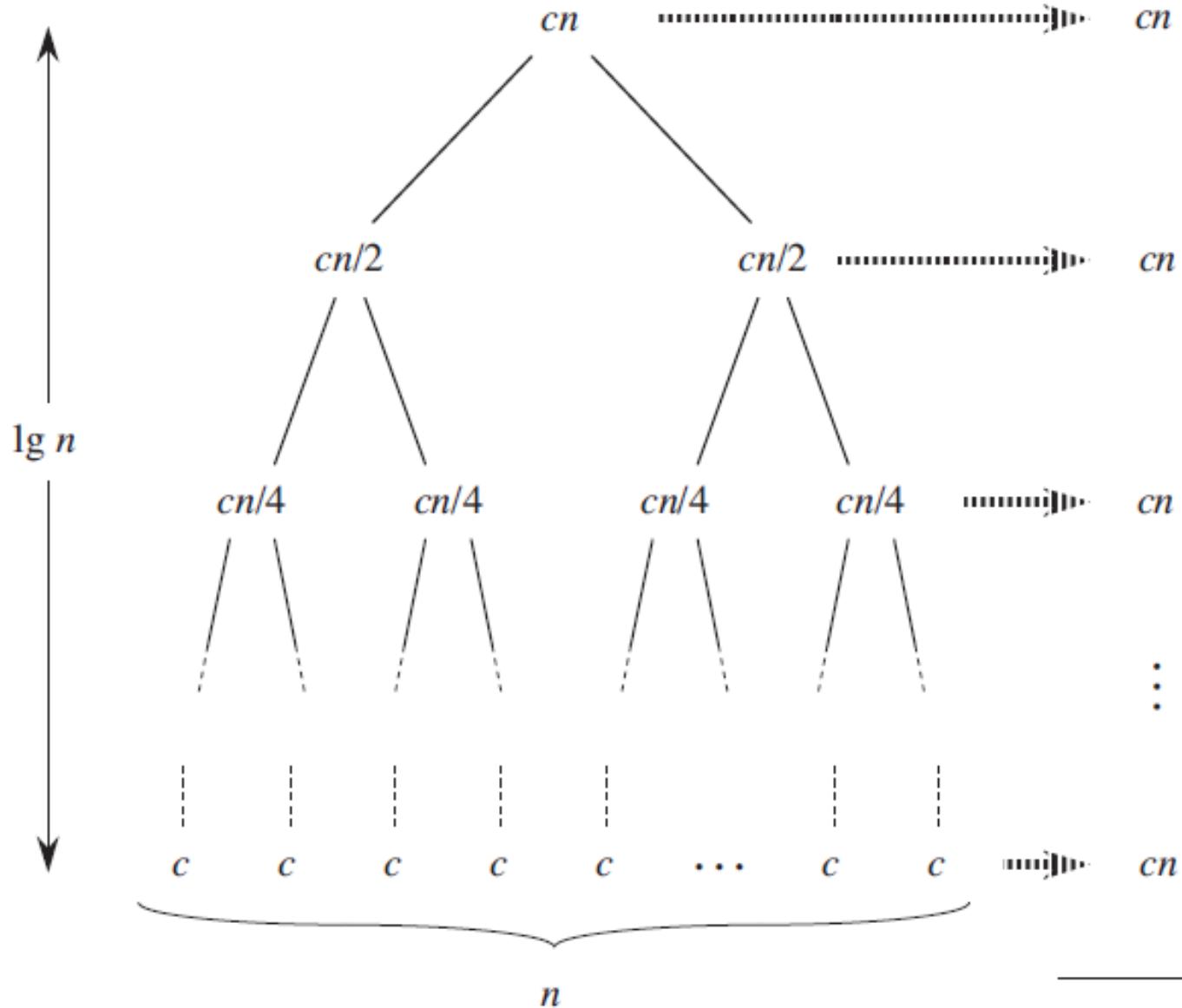
- Assuming that the list can always be divided into two equal halves
- Division takes constant time
- Merger takes constant time
- Each sub-problem takes half the time of original problem
- When the subproblem has just one element it takes constant time for sorting (no sorting required)
- Suppose,  $T(n)$  be the time for solving the original problem of size ‘n’ then, the recurrence relation can be written as:

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

# Solving Recurrence using Recursion Tree



- Recursion Tree can be used to solve simple recurrence equations
- Solving the above recurrence using recursion tree yields  $\theta(n \log n)$  time complexity for merge sort



(d)

Total:  $cn \lg n + cn$

# Integer Multiplication

- Brute force method:
  - Multiply each digit of one number by each of the other and add the result
  - Drawback: time complexity is high, takes  $O(n^2)$  time
- Divide and Conquer method:
  - Improvement over brute force method
  - Time complexity:  $O(n^{1.3})$

# Gauss's method for multiplication

- Improvement in calculating product of two complex numbers attributed to C. F. Gauss
- $(a+bi)(c+di) = ac - bd + i(bc + ad)$
- Looks like 4 multiplications but actually only two multiplications are required as shown below
- $bc + ad = (a+b)(c+d) - ac - bd$

# Representation of Binary Numbers

- An n-digit binary number can be divided into two equal parts of  $n/2$  size each (assuming that n is even)
- The original number can be represented in terms of the two divisions as below:

$$x = 2^{n/2} * \boxed{x_L} + \boxed{x_R}$$

$$y = 2^{n/2} * \boxed{y_L} + \boxed{y_R}$$

- For example, if  $x = 10110110_2$  we have,  $x_L = 1011_2$  and  $x_R = 0110_2$   
$$x = 1011_2 * 2^4 + 0110_2$$

- The product  $xy$  can then be written as:

$$\begin{aligned} xy &= (2^{\frac{n}{2}} * x_L + x_R)(2^{\frac{n}{2}} * y_L + y_R) \\ &= 2^n * x_L y_L + 2^{\frac{n}{2}}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

- The addition takes  $O(n)$  time for size of  $n$
- Four multiplications to be calculated, the recurrence relation can be expressed as:

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

- The above equation works out to  $O(n^2)$

- Using Gauss's trick we can re-write the multiplications as:

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$$

Thus, the expression for evaluation becomes:

$$2^n * x_L y_L + 2^{\frac{n}{2}} \{ (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R \} + x_R y_R$$

- Only three multiplication operations are there, addition can still be done in  $O(n)$  time
- The recurrence relation changes to:

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

- The branching factor now changes to 3 instead of 4
- Length of tree is still  $\log_2^n$
- This creates a quite lower time bound of  $O(n^{1.59})$
- The above change in order of growth creates a lot of difference in running time
- It is not necessary to go up to bit-1 in order to solve this problem
- For most processors, 16-bit or 32-bit multiplication can be performed as a single operation, built-in procedure can be used as soon as the problem size reduces to this level
- The Fast-Fourier Transform, another important divide-and-conquer method can be used for even faster performance

# Integer Multiplication Algorithm

```
function multiply(x, y)
```

Input: Positive integers  $x$  and  $y$ , in binary

Output: Their product

```
 $n = \max(\text{size of } x, \text{ size of } y)$ 
```

```
 $\text{if } n = 1: \text{return } xy$ 
```

$x_L, x_R = \text{leftmost } \lceil n/2 \rceil, \text{ rightmost } \lfloor n/2 \rfloor \text{ bits of } x$

$y_L, y_R = \text{leftmost } \lceil n/2 \rceil, \text{ rightmost } \lfloor n/2 \rfloor \text{ bits of } y$

```
 $P_1 = \text{multiply}(x_L, y_L)$ 
```

```
 $P_2 = \text{multiply}(x_R, y_R)$ 
```

```
 $P_3 = \text{multiply}(x_L + x_R, y_L + y_R)$ 
```

```
 $\text{return } P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{\lfloor n/2 \rfloor} + P_2$ 
```

# Implementation Details

- Implement functions for addition and multiplication separately
- Function for addition should be defined for n-digit number
- Output array should be of size  $n+1$  to handle carry
- Function for multiplication should have two cases for  $n=1$ , in order to handle 1-bit multiplication
- Value of  $n$  for each call can be calculated implicitly using array size

# Matrix Multiplication

SQUARE-MATRIX-MULTIPLY( $A, B$ )

```
1   $n = A.\text{rows}$   $\theta(n^3)$ 
2  let  $C$  be a new  $n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4      for  $j = 1$  to  $n$ 
5           $c_{ij} = 0$ 
6          for  $k = 1$  to  $n$ 
7               $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ 
8  return  $C$ 
```

# Divide and Conquer Approach

- Simple approach is to divide each of the  $n \times n$  matrices into two  $n/2 \times n/2$  matrices
- We can then re-write the multiplication in the form of these smaller matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

- This can be solved by using the equations below and can be used to derive a recursive procedure for calculating the product

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21},$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22},$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21},$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}.$$

### SQUARE-MATRIX-MULTIPLY-RECURSIVE( $A, B$ )

```
1   $n = A.\text{rows}$ 
2  let  $C$  be a new  $n \times n$  matrix
3  if  $n == 1$ 
4       $c_{11} = a_{11} \cdot b_{11}$ 
5  else partition  $A$ ,  $B$ , and  $C$  as in equations (4.9)
6       $C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21})$ 
7       $C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22})$ 
8       $C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21})$ 
9       $C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22})$ 
10 return  $C$ 
```

# Time Complexity

$$T(n) = \theta(1) + 8T\left(\frac{n}{2}\right) + \theta(n^2)$$

$$= 8T\left(\frac{n}{2}\right) + \theta(n^2)$$

$$\Rightarrow T(n) = \theta(n^3)$$

# Strassen's Method

- Key idea is to reduce the branching factor
- Branching factor is reduced from 8 to 7
- The reduction of one matrix multiplication is compensated with several matrix additions
- Constant number of additions at each recursive step only adds a constant amount of time
- The time complexity is reduced to  $n^{lg 7}$

# Solving Recursion

Substitution method, recursion tree, master method

# Substitution Method

- Two step process:
  - Guess the form of the solution
  - Use mathematical induction to find the constants and show that the solution works
- The guessed solution is substituted when applying the inductive hypothesis to smaller values
- Challenge is to be able to guess the form of solution (recursion tree can be helpful)
- Can be used for establishing both upper and lower bounds

# Example

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

- Let's guess that the solution is  $O(nlgn)$
- Then, we need to prove that  $T(n) \leq cnlgn$  for appropriate choice of  $n$
- Suppose, the bound holds for some  $m < n$ , in particular  $m = \lfloor n/2 \rfloor$
- Therefore,

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

- Last step holds until  $c \geq 1$

# Handling Boundary Condition

- We need to prove that boundary conditions are suitable as base cases for inductive proof
- We must show that we can choose the constant  $c$  large enough such that the bounds hold for boundary conditions as well
- Might not be obvious typically
- Consider the case,  $T(n) \leq cn\lg n$ ,  $n \geq n_0$
- Suppose, we choose the sole boundary condition as  $T(1) = 1$
- This creates problem with the bounds on the runtime

$$T(1) = c1 \lg 1 = 0$$

- Which is contradictory to  $T(1) = 1$
- The above condition is avoided by absorbing the boundary condition in  $n_0$
- Inductive proof is applied for all other cases except the boundary conditions
- Suppose the bounds hold for  $n > 3$ , then recurrence does not directly depend on  $T(1)$
- Boundary condition for  $T(3)$  now depends on  $T(1)$
- We can derive that  $T(2) = 4$  and  $T(3) = 5$
- Now,  $c$  should be chosen large enough so that

$$T(2) \leq c2 \lg 2 \text{ and } T(3) \leq c3 \lg 3 \text{ holds} \Rightarrow c \geq 2$$

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

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

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$$

- While making the guess we can use the fact that addition of a constant term will make little difference as the problem size increases
- Therefore, the guess would be  $T(n) = O(n \lg n)$

- In some cases, the inductive proof does not work even though the guess is correct
- Such cases can be handled by subtracting a lower order term
- Tighter bound is easier to prove

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

Suppose, we guess the solution as  $T(n) = O(n)$  then, we need to prove that  $T(n) \leq cn$  for appropriate choice of constant c

- Suppose, it is true for  $m = \lfloor n/2 \rfloor$  then,

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

$$\Rightarrow T(n) \leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1$$

$$\Rightarrow T(n) \leq cn + 1$$

- Above will not equate to  $T(n) \leq cn$  for any positive constant  $c$
- Intuitively however, the bound on the solution is correct that is we need not increase our bound
- Such situations can be handled by making a revised guess by subtracting a lower order term

- Let's make a new guess as  $T(n) \leq cn - d$  for some constant  $d \geq 0$
- Therefore,

$$T(n) \leq (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1$$

$$\Rightarrow T(n) \leq cn - 2d + 1$$

$$\Rightarrow T(n) \leq cn - d$$

- Inductive hypothesis is to be proved in its exact form

# Changing Variables

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

- Above equation can be solved by changing variables
- We convert the equation using the following

$$m = \lg n \Rightarrow n = 2^m$$

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

- This equation can be solved by again replacing  $T(2^m)$  by  $S(m)$

$$S(m) = 2S(m/2) + m$$

Which is a familiar form of equation and has solution  $O(m \lg m) = O(\lg n \lg \lg n)$

# Exercise

- $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$
- Find the bounds of the above equation
- We will prove that,  $T(n) \leq cnlgn - d$
- Suppose, it is true for some  $m = \lfloor n/2 \rfloor + 17$   
*we choose  $n_1$  s.t.  $n > n_1$  implies  $\frac{n}{2} + 17 \leq 3n/4$*

Therefore,

$$T(\lfloor n/2 \rfloor + 17) \leq c \left( \frac{n}{2} + 17 - d \right) \lg \left( \frac{n}{2} + 17 - d \right)$$

$$\Rightarrow T(n) \leq 2 \left\{ c \left( \frac{n}{2} + 17 - d \right) \lg \left( \frac{n}{2} + 17 - d \right) \right\} + n$$

$$\Rightarrow T(n) \leq cn \log \left( \frac{n}{2} + 17 \right) + 17c \log \left( \frac{n}{2} + 17 \right) - 2d + n$$

$$\Rightarrow T(n) \leq cn \log \left( \frac{3n}{4} \right) + 17c \log \left( \frac{3n}{4} \right) - 2d + n$$

$$\Rightarrow T(n) \leq cn \log n - d + cn \log \left( \frac{3}{4} \right) + 17c \log \left( \frac{3n}{4} \right) - d + n$$

$$\Rightarrow T(n) \leq cn \lg n - d + cn \log \left( \frac{3}{4} \right) + 17c \log n + 17c \log \left( \frac{3}{4} \right) - d + n$$

- Last step holds if  $c \geq -\frac{2}{\log(\frac{3}{4})}$  and  $d = 34$

$$T(n) \leq cn\log n - d + 17c\log(n) - n$$

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

$\Rightarrow$  we can find  $n_2$  such that

$$n \geq n_2 \text{ implies } n \geq 17c\log(n)$$

$n_0$  is taken as maximum of  $n_1$  and  $n_2$  then for all  $n \geq n_0$

$$T(n) \leq cn\log n - d$$

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

# Solving Recurrence using Recursion Tree

# Concepts

- Serves as a straight forward to making a good guess for many problems
- Each node represents the cost of a single sub-problem in the set of recursive function invocations
- Cost of each sub-problem is summed to get cost-per level
- Cost of each level is summed to get the total cost of recursion
- While using recursion tree to generate a good guess minor variations in the boundary conditions are tolerable

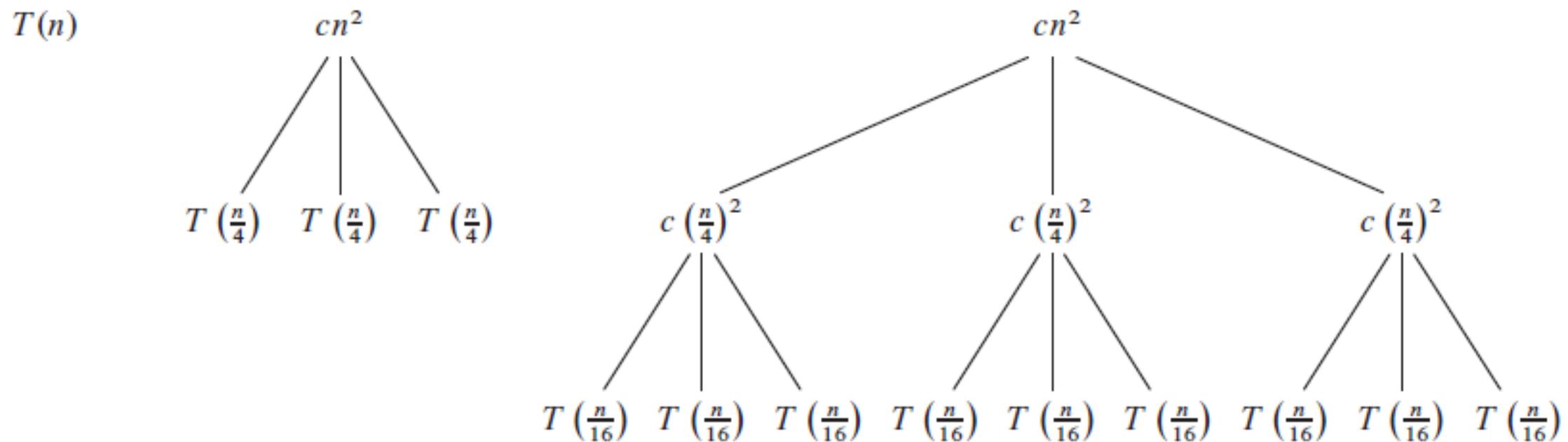
# Example

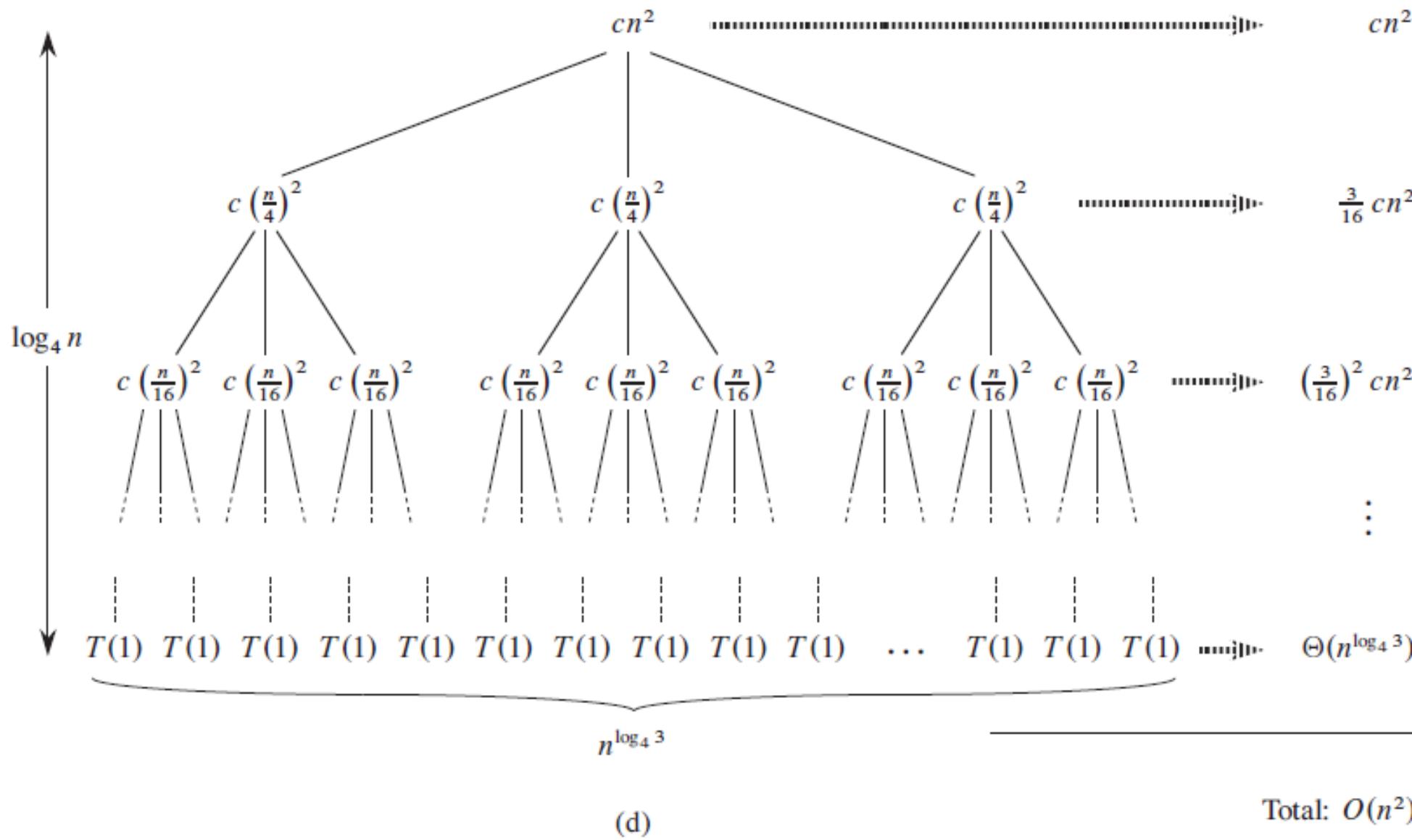
$$T(n) = 3T(\lfloor n/4 \rfloor) + \theta(n^2)$$

- Indicates division of problem into three smaller problems at each step
- Each sub-problem is  $1/4^{\text{th}}$  the size of original problem
- Merger step takes  $n^2$  time at each step

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

- The recursion tree for this problem can be designed as follows





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

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

# Verification using Substitution Method

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

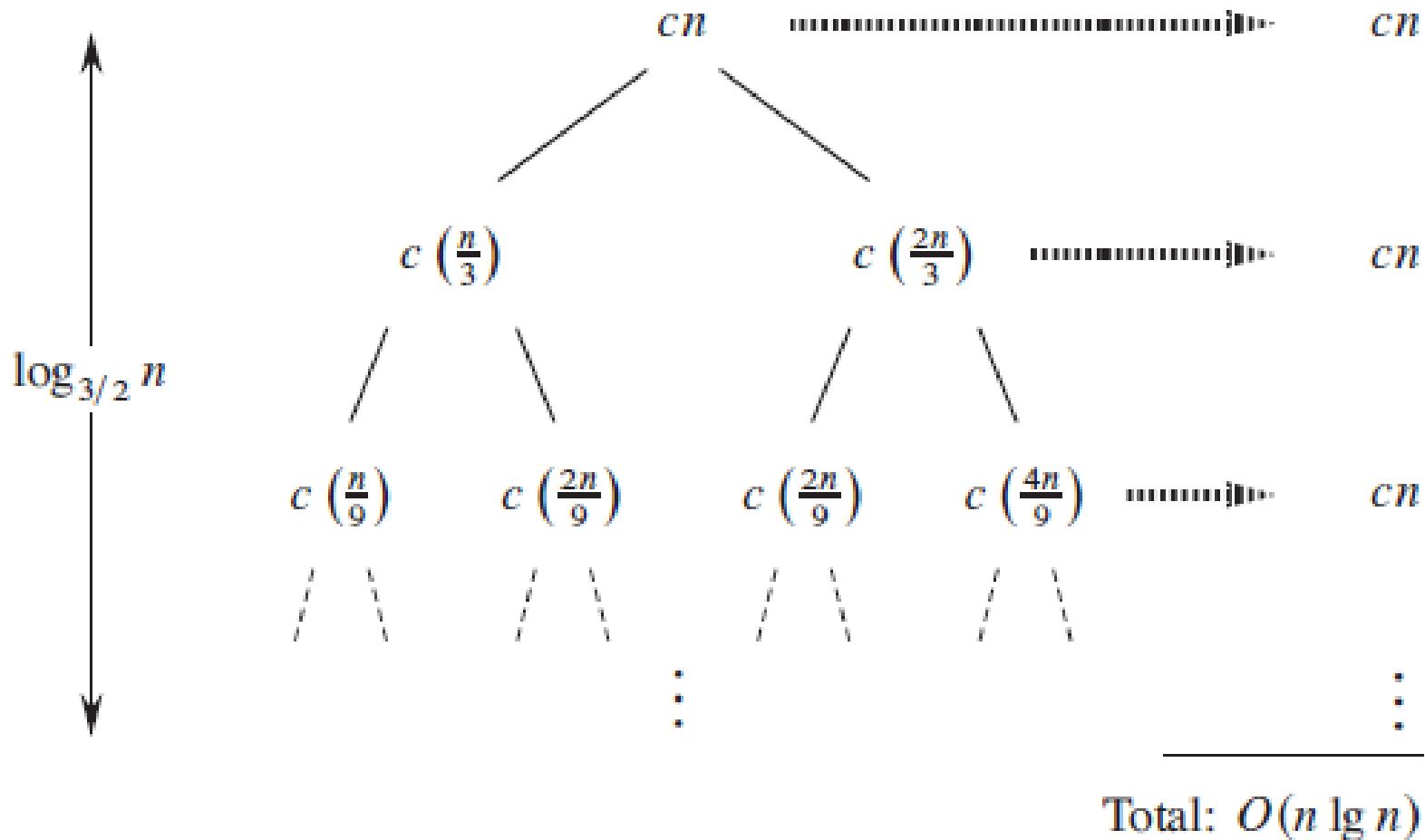
where the last step holds as long as  $d \geq (16/13)c$ .

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n)$$

- Each time the problem is divided asymmetrically into sub-problems of sizes  $n/3$  and  $2n/3$
- Merger steps takes  $O(n)$  time
- The equation can be re-written as:

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn$$

- We use this to draw the recursion tree



- Height of the tree: longest sequence from root

$$\Rightarrow (2/3)^k n = 1 \Rightarrow k = \log_{3/2} n$$

- Root contributes  $cn$  to cost
- If each level contributes equally, total cost is bounded by  $O\left(cn \log_{\frac{3}{2}} n\right) \Rightarrow O(n \lg n)$
- Each level however does not contribute equally
- Contribution of leaves is different
- No. of leaves =  $2^{\log_{3/2} n} \Rightarrow n^{\log_{3/2} 2}$
- If each leaf contributes constant time then the bound is  $\theta(n^{\log_{3/2} 2})$  for leaves
- $\log_{3/2} 2$  is strictly greater than 1, therefore bound is  $\omega(n \lg n)$

# Verification using Substitution Method

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

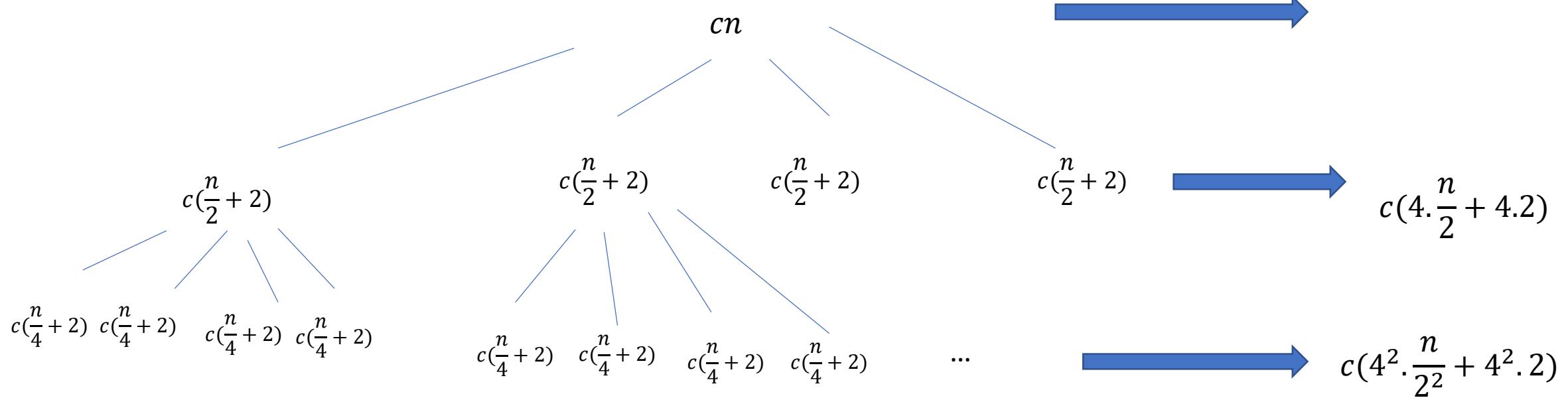
# Exercise

- Use a recursion tree to determine a good asymptotic upper bound on the recurrence

$$T(n) = 4T\left(\frac{n}{2} + 2\right) + n$$

- Use the substitution method to verify your answer.

$\lg n$



# Analysis

- Height of tree  $\Rightarrow \frac{n}{2^k} = 1 \Rightarrow k = \log_2 n$
- Total cost at ith level  $\Rightarrow c(4^i \times \frac{n}{2^i} + 4^i \times 2)$
- Cost of last level  $\Rightarrow 4^k \times T(1) \Rightarrow 4^{\log_2 n} \Rightarrow \theta(n^2)$

# Calculating total cost

$$T(n) = 4T\left(\frac{n}{2} + 2\right) + n$$

$$T(n) = cn + \sum_{i=0}^{\lg n - 1} c \left( 4^i \times \frac{n}{2^i} + 4^i \times 2 \right) - 2 + \theta(n^2)$$

$$= cn + cn(2^{\lg n - 1} - 1) + \frac{2}{3}c(4^{\lg n - 1}) + \theta(n^2) - 2$$

$$= cn\left(\frac{n}{2} - 1\right) + \frac{2}{3}c\left(\frac{n^2}{4}\right) + \theta(n^2) + cn - 2$$

$$= cn^2 - cn + \frac{1}{6}c(n^2) + \theta(n^2) - 2$$

$$\Rightarrow O(n^2)$$

# Verification using Substitution Method

- For this and all such problems, we will use the method of subtracting a lower order term from the equation for proving the bound
- Suppose, we guess the solution to be  $O(n^2)$
- Then, we have to prove that  $T(n) \leq c \cdot n^2 - 6n$
- Let us assume that the bound holds for  $m = n/2+2$
- Then,

$$T\left(\frac{n}{2} + 2\right) \leq c \left(\frac{n^2}{4} + 4 + 2n\right) - 6 \times \left(\frac{n}{2} + 2\right)$$

$$\Rightarrow T(n) \leq 4 \cdot c \left( \frac{n^2}{4} + 4 + 2n - 3n - 12 \right) + n$$

$$= cn^2 - 4cn - 32c + n$$

$$= cn^2 + (1 - 4c)n - 32c$$

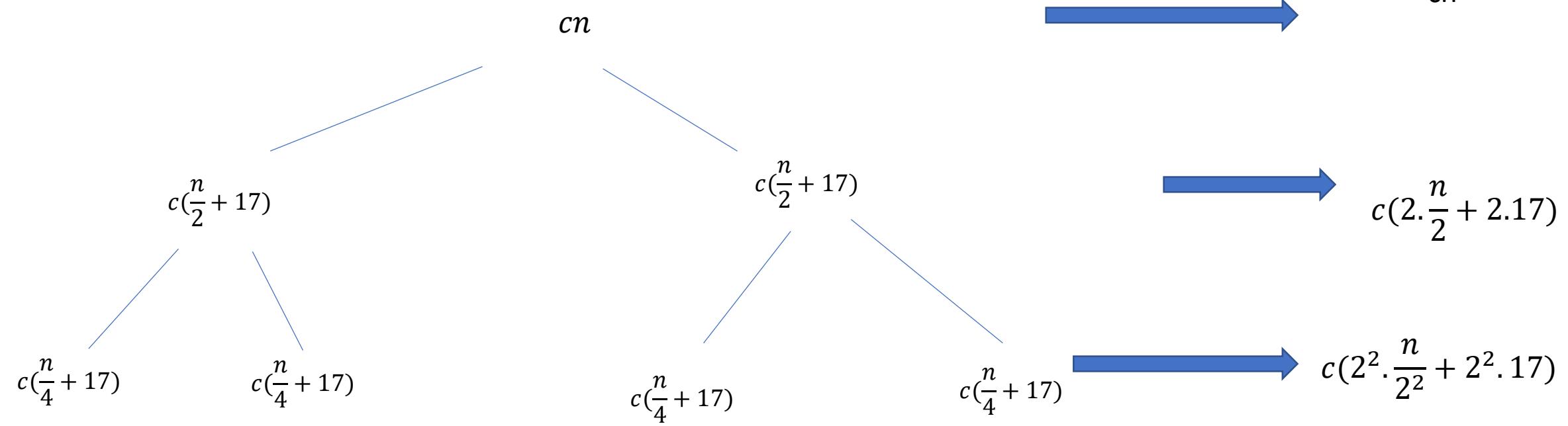
$$1 - 4c \leq -6 \Rightarrow c \geq \frac{7}{4}$$

## Example 2

$$T(n) = 2T\left(\frac{n}{2} + 17\right) + n$$

- Height of the tree  $\Rightarrow k = \log_2 n$
- Total Cost at the  $i$ th level  $\Rightarrow 2^i \cdot c \left( \frac{n}{2^i} + 17 \right)$
- Cost at the last level  $\Rightarrow 2^i \times T(1) = \theta(n)$

↑



$\lg n$

$cn$

$$c\left(\frac{n}{2} + 17\right)$$

$$c\left(\frac{n}{4} + 17\right)$$

$$c\left(\frac{n}{4} + 17\right)$$

$$c\left(2 \cdot \frac{n}{2} + 2 \cdot 17\right)$$

$$c\left(\frac{n}{4} + 17\right)$$

$$c\left(\frac{n}{4} + 17\right)$$

$$T(n) = cn + \sum_{i=0}^{\log_2 n - 1} (cn + 2^i \times 17) - 17 + \theta(n)$$

$$T(n) = cn + \frac{c}{2}n \log n + 17\left(\frac{n}{2} - 1\right) - 17 + \theta(n)$$

$$T(n) \leq cn \lg n$$

# Master Method

For solving recurrences

# Introduction

- “Cookbook” for solving recurrences
- No guessing, tree construction or calculation required
- Has three cases, that are used to decide the form of solution
- Given a recurrence equation, you need to decide which category it falls among the three in order to find the bounds on the running time

# General Recurrence relation

- For a recurrence described below:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- Here each time, the original problem is divided into ‘a’ sub-problems
- Each sub-problem is solved in  $T(n/b)$  time
- Both ‘a’ and ‘b’ are positive constants
- $f(n)$  is the time of combining the results of the a sub-problems generated at a stage

# Master Method

- We have three cases for the previously described recurrence:
1. If  $f(n) = O(n^{\log_b a - \epsilon})$  for some constant  $\epsilon > 0$ , then  $T(n) = \theta(n^{\log_b a})$
  2. If  $f(n) = \theta(n^{\log_b a})$ , then  $T(n) = \theta(n^{\log_b a} \lg n)$
  3. If  $f(n) = \Omega(n^{\log_b a + \epsilon})$  for some constant  $\epsilon > 0$ , and if  $a f\left(\frac{n}{b}\right) \leq c f(n)$  for some constant  $c < 1$  and all sufficiently large  $n$ , then  $T(n) = \theta(f(n))$

# Example 1

$$T(n) = 9T\left(\frac{n}{3}\right) + n$$

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

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

Therefore, case 1 applies and we have:

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

## Example 2

$$T(n) = 2T\left(\frac{n}{3}\right) + 1$$

$$n^{\log_b a} = n^{\log_3 2} = n^{0.631}$$

$$f(n) = O(1)$$

Therefore, case 1 applies and we get

$$T(n) = \theta(n^{\log_3 2})$$

## Example 3

$$T(n) = T\left(\frac{2n}{3}\right) + 1$$

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

$$f(n) = \theta(1)$$

Therefore, case 2 applies and we have

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

## Example 4

$$T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$$

$$n^{\log_b a} = n^{\log_4 3} = n^{0.792}$$

$$n^{\log_b a+\epsilon} = n^{\log_4 3+\epsilon} = n \text{ for } \epsilon = 1$$

$$\Rightarrow f(n) = \Omega(n^{\log_4 3+\epsilon})$$

Therefore, case 4 will apply if regularity condition holds

- $af\left(\frac{n}{b}\right) \leq cf(n)$  for some constant  $c < 1$  and all sufficiently large  $n$

$$af\left(\frac{n}{b}\right) \Rightarrow 3\left(\frac{n}{4}\right)\lg\left(\frac{n}{4}\right)$$

$$= \left(\frac{3}{4}\right)n\lg n - \left(\frac{3}{4}\right)n\lg 4$$

$$= \left(\frac{3}{4}\right)f(n) - 1.5n$$

Thus, the condition holds for  $c = 3/4$

Therefore, case 3 applies and we have

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

## Example 5

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

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

$$\log_b a + \epsilon = \log_2 2 + \epsilon \Rightarrow n \cdot n^{\epsilon_1}$$

$$\frac{f(n)}{\log_b a + \epsilon} = \frac{n \log n}{n \cdot n^\epsilon} = \frac{\lg n}{n^\epsilon}$$

- Above is not an asymptotic lower bound on  $n \lg n$  as  $\lg n$  is not asymptotically larger than  $n^\epsilon$  for any positive  $\epsilon$
- This might not be obvious for  $0 < \epsilon < 1$  but holds true for sufficiently large  $n$  in this case also
- Therefore, case 3 does not hold
- We can find the asymptotic bound using recursion tree and prove it using substitution method

# Proof using L'Hospital's Rule

Suppose for any two functions  $f(x)$  and  $g(x)$  and some real number 'a' or  $\pm\infty$ , we want to find

$$\lim_{x \rightarrow a} \frac{f(x)}{g(x)}$$

And one of the following cases occurs:

$$\lim_{x \rightarrow a} \frac{f(x)}{g(x)} = \frac{0}{0} \quad \text{or}$$

$$\lim_{x \rightarrow a} \frac{f(x)}{g(x)} = \frac{\pm\infty}{\pm\infty}$$

# L'Hospital's Rule

$$\text{if } \lim_{x \rightarrow a} \frac{f(x)}{g(x)} = \frac{0}{0} \quad \text{or} \quad \lim_{x \rightarrow a} \frac{f(x)}{g(x)} = \frac{\pm\infty}{\pm\infty}$$

*Then,*

$$\lim_{x \rightarrow a} \frac{f(x)}{g(x)} = \lim_{x \rightarrow a} \frac{f'(x)}{g'(x)}$$

In the previous example we have,

$$\lim_{n \rightarrow \infty} \frac{\lg n}{n^\epsilon} = \frac{\infty}{\infty}$$

Thus L'Hospital's rule applies and we get,

$$\lim_{n \rightarrow \infty} \frac{\lg n}{n^\epsilon} = \lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{\epsilon n^{\epsilon-1}} = \frac{0}{\infty} = 0$$

$$\lim_{n \rightarrow \infty} \frac{\lg n}{n^\epsilon} = 0 \Rightarrow \text{that } n^\epsilon \text{ is an upper bound on } \lg n$$

## Example 6

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

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

$$\Rightarrow f(n) = \theta(n^{\log_b a})$$

Therefore, case 2 applies and we have

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

## Example 7

$$T(n) = 8T\left(\frac{n}{2}\right) + \theta(n^2)$$

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

Above is polynomially larger than  $\theta(n^2)$  i.e.  $f(n) = O(n^{3-\epsilon})$

Therefore, case 1 applies and we have

$$T(n) = \theta(n^3)$$

## Example 8

$$T(n) = 7T\left(\frac{n}{2}\right) + \theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 7} = n^{2.81}$$

$$\Rightarrow f(n) = O(n^{2.81-\epsilon}) \text{ for } \epsilon = 0.8$$

Therefore, case 1 applies and we have

$$T(n) = \theta(n^{\lg 7})$$

## Example 9

Professor Caesar wishes to develop a matrix-multiplication algorithm that is asymptotically faster than Strassen's algorithm. His algorithm will use the divide and-conquer method, dividing each matrix into pieces of size  $n/4 \times n/4$ , and the divide and combine steps together will take  $\theta(n^2)$  time. He needs to determine how many subproblems his algorithm has to create in order to beat Strassen's algorithm. If his algorithm creates 'a' subproblems, then the recurrence for the running time  $T(n)$  becomes  $T(n) = aT\left(\frac{n}{4}\right) + \theta(n^2)$ . What is the largest integer value of  $a$  for which Professor Caesar's algorithm would be asymptotically faster than Strassen's algorithm?

$$n^{\log_b a} = n^{\log_4 a}$$

Time complexity of Strassen's Method =  $O(n^{\lg 7}) = O(n^{2.81})$

Using the Master method, the largest value of 'a' is possible when case 1 applies

For the time complexity of Professor's algorithm to be less than  $O(n^{2.81})$  we need to have

$$2 \leq \log_4 a < 2.81$$

$$\log_4 a < \log_2 7$$

$$\frac{\log_2 a}{2} < \log_2 7$$

$$\log_2 a < 2\log_2 7$$

$$\log_2 a < \log_2 7^2$$

$$a < 49$$

Therefore, the maximum possible value of 'a' is 48.

# Sorting

Selection Sort, Insertion Sort, Merge sort, Quicksort

# Selection Sort

- This is an in place sorting technique
- Here, in each pass the smallest value is chosen from the data and is placed at the correct position
- It requires mainly two operations – comparison and swapping
- For example, in the first pass, the smallest value is found and is swapped with  $a[0]$
- In the second pass, the next smallest value is chosen and swapped with  $a[1]$
- This is the reason why it is called as selection sort, as in each pass we ‘select’ the correct value for a particular place in array

57	48	79	65	15	33	52
0	1	2	3	4	5	6

15	48	79	65	57	33	52
0	1	2	3	4	5	6

15	33	79	65	57	48	52
0	1	2	3	4	5	6

15	33	48	65	57	79	52
0	1	2	3	4	5	6

<b>15</b>	<b>33</b>	<b>48</b>	<b>52</b>	<b>57</b>	<b>79</b>	<b>65</b>
0	1	2	3	4	5	6

<b>15</b>	<b>33</b>	<b>48</b>	<b>52</b>	<b>57</b>	<b>79</b>	<b>65</b>
0	1	2	3	4	5	6

<b>15</b>	<b>33</b>	<b>48</b>	<b>52</b>	<b>57</b>	<b>65</b>	<b>79</b>
0	1	2	3	4	5	6

# Analyzing Selection Sort

- In the first pass, we perform  $n-1$  comparisons
- In the second pass, we perform  $n-2$  comparisons and so on
- Total time taken:
  - $(n-1) + (n-2) + (n-3) + \dots + 1 = n(n-1)/2 \rightarrow O(n^2)$
- Since, insertion sort performs in-place sorting, it does not require any extra memory for sorting
- Another observation that can be made is, the method does no better if the data is ordered

# Insertion Sort

- Insertion sort works in the manner we sort a pile of cards
- We start with one element and put it in our other hand
- Then, we pick another card and place it in such manner that the cards in our other hand are sorted
- Then, we insert the third card again in the same manner
- Since, we insert each element at an appropriate position in an already sorted array – we call this method insertion sort

<b>57</b>	<b>48</b>	<b>79</b>	<b>65</b>	<b>15</b>	<b>33</b>	<b>52</b>
0	1	2	3	4	5	6

<b>57</b>
0

<b>48</b>	<b>57</b>
0	1

<b>48</b>	<b>57</b>	<b>79</b>
0	1	2

<b>48</b>	<b>57</b>	<b>65</b>	<b>79</b>
0	1	2	3

<b>15</b>	<b>48</b>	<b>57</b>	<b>65</b>	<b>79</b>
0	1	2	3	4

<b>15</b>	<b>33</b>	<b>48</b>	<b>57</b>	<b>65</b>	<b>79</b>
0	1	2	3	4	5

<b>15</b>	<b>33</b>	<b>48</b>	<b>52</b>	<b>57</b>	<b>65</b>	<b>79</b>
0	1	2	3	4	5	6

# Insertion Sort: Implementation

```
for(int i =1; i<n; i++)
{  int key = array[i];
   int k = i-1;
   while(k>=0 && key < array[k])
   {  array[k+1] = array[k];
      --k;
   }
   array[k+1] = key;
}
```

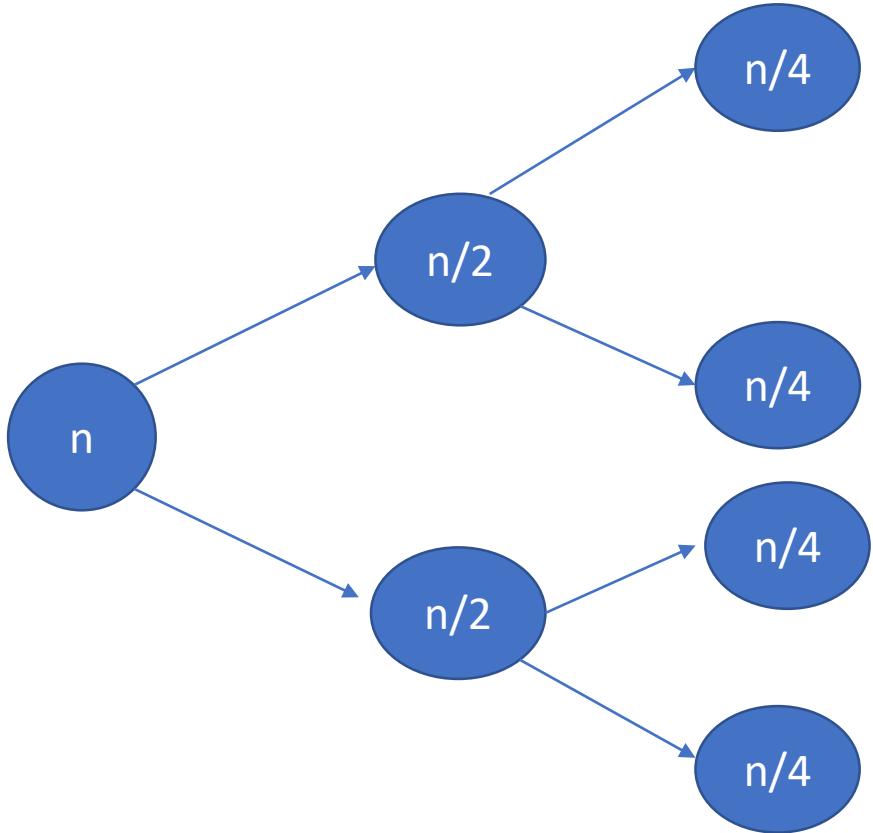
# Analysis of Insertion Sort

- Each time we pick an element  $k$  for insertion we might make only one comparison (if  $k$  is the largest element among sorted entities) or  $k-1$  comparisons (if  $k$  is the smallest among all sorted entities)
- For random data, we expect  $(k-1)/2$  comparisons on average
- Thus, for sorting an array of size  $n$ , we will expectedly make around  $(k-1)/2$  comparisons for  $k$  varying between 2 to  $n$  i.e.,

$$\sum_{k=2}^n \frac{(k-1)}{2} = \frac{1}{2} \{1 + 2 + \dots + n - 1\} = \frac{n(n-1)}{4} \xrightarrow{\text{O}(n^2)}$$

# Quicksort

- The main idea behind quicksort is to partition the list with respect to one of the values called *pivot*
- The elements of the list are then arranged so that all the values less than pivot lie left to it while all the values greater than pivot lie right to it
- After this step, the pivot element is at its correct position in the array
- The point at which the pivot is located is called the *division point (dp)*
- If we are able to sort the two sub-arrays obtained after this procedure, our problem will be solved
- However, we can use the same procedure to sort the two sub-arrays
- Thus, quicksort is used in recursive manner to sort a given array



- Total number of steps would be  $\log_2 n$
- In each step we are doing at most  $n$  comparisons
- Thus, time complexity in the best case would be  $O(n \log_2 n)$
- We will next see what do we mean by the best case for this technique

# Analysis of Quicksort

- This is an algorithm whose performance can range from very fast to very slow
- Typically takes  $O(n\log n)$  time for random data, the number of comparisons lie somewhere between  $n\log n$  and  $3n\log n$
- At the heart of the algorithm lies the partitioning step
- Ideally, the partition should be such that each time we partition the array we get around the same number of elements in the two sub-arrays
- In the worst case, we might generate completely unbalanced partition such that one part contains only one element and the other contains rest of the elements
- In this case, quicksort will be no better than insertion sort and will take  $O(n^2)$  time

- Such a case might arise when we are trying to sort a sorted array and choose the first element as pivot
- Such cases may be avoided by choosing the pivot element randomly from the data instead of always choosing the first element
- In case of random selection, there is little chance that the pivot element will be chosen wrongly and the algorithm achieves the worst case time complexity of  $O(n^2)$
- Modify the partition function to choose different elements as pivot and see the effect on sorting time required for different inputs

# Merge Sort

- Merge sort is based on a similar idea as quicksort
- Here, we divide the array each time into equal sub-arrays and sort each array separately
- The complete sorted array is obtained by merging the two sub-arrays
- For sorting the sub-arrays we use the same procedure
- This is continued until we reach an array containing just one element
- The main difference between quicksort and merge sort is that in merge sort the two sub-arrays are merged instead of being partitioned

- While partitioning, quicksort might partition the arrays into two unequal partitions, which can lead to poor performance
- In merge sort however, at each pass the array is partitioned into two almost equal partitions
- Similar to the partition function in quicksort, at the heart of merge sort is the merge function that performs the merging of two sorted lists

```

void merge(int [], int , int, int)
void mergeSort(int A[], int min, int max)
{ if(min <max)
  { int mid = (min + max)/2;
    mergeSort(A, min, mid);
    mergeSort(A, mid+1, max);
    merge(A, min, mid, max);
  }
}

```

53	12	98	63	18	32	80	46	72	21
----	----	----	----	----	----	----	----	----	----

Here, we will sort the first 5 elements and last 5 elements separately

12	18	53	63	98	21	32	46	72	80
----	----	----	----	----	----	----	----	----	----

12	18	21	32	46	53	63	72	80	98
----	----	----	----	----	----	----	----	----	----

# Analysis of Merge Sort

- Since merge sort divides the array into two (almost) equal parts each time and for merging we require at most size of the array number of comparisons
- The running time for merge sort is  $O(n \log n)$
- This is the best possible time complexity we have achieved among all the algorithms seen yet
- However, merge sort requires an extra array for sorting (external sorting method)
- Therefore, space complexity of merge sort is higher than other methods

# Heap Data Structure

Heapsort, Priority Queue

# Definition

- A heap can be viewed as a nearly complete binary tree.
- Each node in the heap satisfies the *heap property*
- A heap could be of two types: *min-heap and max-heap*
- For a max-heap, the heap property states that the value at each node is at most as large as its parent
- For min-heap, the heap property is satisfied if the value at each node is at least as large as its parent
- Heaps are used for sorting and for implementing priority queue

# Types of Binary Trees

- Full Binary tree: each node is either of degree 0 or 2. There is no deg-1 node
- Complete Binary tree: each node has degree either 0 or 2 and all the leaf nodes are at the same level
- Nearly Complete Binary tree: A complete binary tree accept possibly at the last level where the nodes might have deg-1 such that the positional information is preserved
- i.e. if the nodes are numbered from left to right then the numbering does not change in a complete binary and a nearly complete binary tree

# Array elements as Heap

- Array elements can be represented in the form of a heap
- The array elements can be viewed as forming a nearly complete binary tree.
- For a given array element indexed at  $i$  we can compute the index of its parents and children as:
  - $\text{Parent}(i) = \lfloor i/2 \rfloor$
  - $\text{Left\_child}(i) = 2i$
  - $\text{Right\_child}(i) = 2i + 1$
- For an array  $A$  of size  $A.\text{length}$ , the size of the heap is defined as  $A.\text{heapsize}$  and we have  $1 \leq A.\text{heapsize} \leq A.\text{length}$

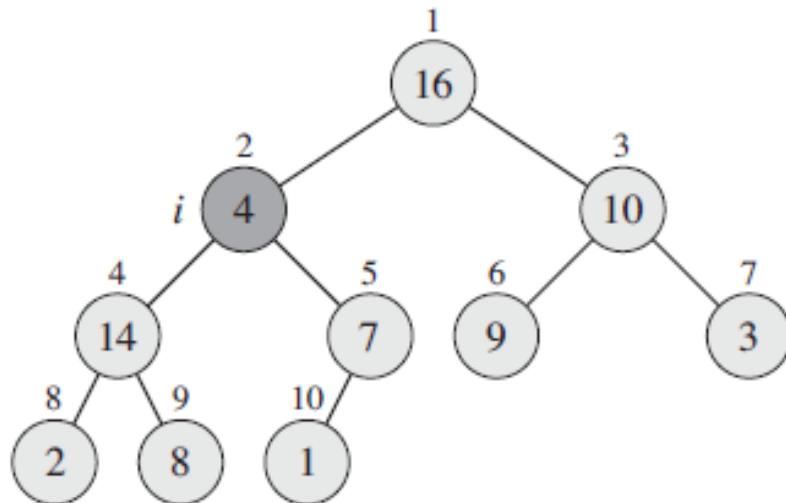
# Heap Procedures

- *Max-Heapify*: runs in  $O(\lg n)$  time, used to maintain the heap property
- *Build-Max-Heap*: runs in  $O(n)$  time, produces a max-heap from an unordered array
- *Heapsort*: runs in  $O(n \lg n)$  time, sorts an array *in place*
- *Max-Heap-Insert*, *Heap-Extract-Max*, *Heap-Increase-Key*, *Heap-Maximum* procedures, run in  $O(\lg n)$  time, allow priority queue to be implemented using heap data structure

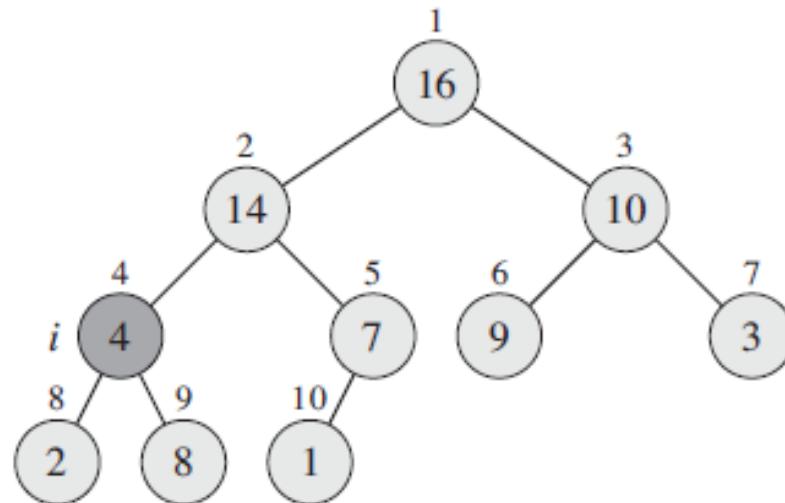
# Maintaining the Heap property

**MAX-HEAPIFY( $A, i$ )**

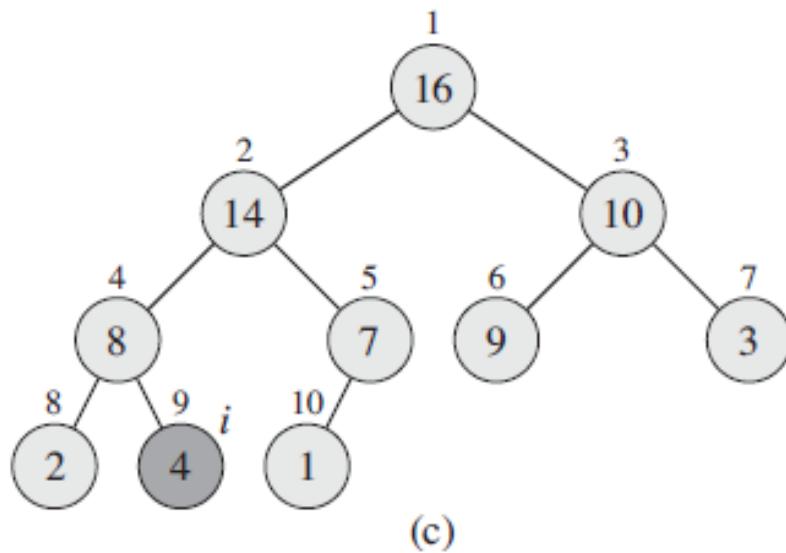
- 1  $l = \text{LEFT}(i)$
- 2  $r = \text{RIGHT}(i)$
- 3 **if**  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
  - 4      $\text{largest} = l$
  - 5 **else**  $\text{largest} = i$
  - 6 **if**  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
    - 7      $\text{largest} = r$
  - 8 **if**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



(a)



(b)



(c)

# Complexity Analysis

- Steps 1 to 9 take constant time
- Step 10 is a recursive call to Max-Heapify
- In order to write the recurrence, we need to find the maximum size of a sub-tree
- Suppose the heap has  $n$  nodes in total. Since, heap is a complete binary tree, the left and right-subtrees can have at most a difference of 1 in their heights

- Thus, we get the following expression:

$$1 + 2^{h+1} - 1 + 2^{h+2} - 1 = n$$

$$\Rightarrow 2^{h+1} (2 + 1) - 1 = n$$

$$\Rightarrow 2^{h+1} = \frac{n + 1}{3} \approx \frac{n}{3}$$

$$\Rightarrow 2^{h+2} \approx \frac{2n}{3}$$

- Thus, the maximum height of the subtree could be  $2n/3$

- We can write the recurrence as below:

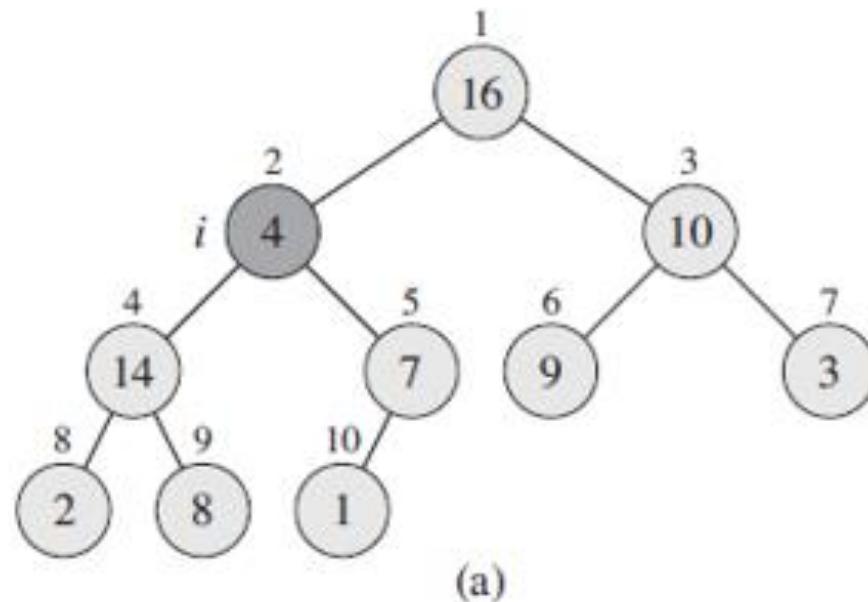
$$T(n) = T\left(\frac{2n}{3}\right) + \theta(1)$$

Using the case 2 of Master method, the solution to the above recurrence is  $T(n) = O(lgn)$

Thus, the Heapify method runs in  $O(lgn)$  time

# Exercise

- Show that, with the array representation for storing an  $n$ -element heap, the leaves are the nodes indexed by  $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots n$



# Exercise

- What is the effect of calling MAX-HEAPIFY ( $A, i$ ) for  $i > A.\text{heap-size}/2$ ?

# Exercise

- Show that the worst-case running time of MAX-HEAPIFY on a heap of size  $n$  is  $\Omega(\lg n)$
- Worst case occurs when the Max-Heapify is called along the longest path from the root to the leaf
- In this case the running time will be bounded by the height of the heap i.e.  $\Omega(\lg n)$

# Building a Heap

- Elements of an array can be built-up into a max-heap
- We use the procedure Build-max-heap for this
- All elements of the array from  $\lfloor n/2 \rfloor + 1$  upto  $n$  will comprise the leaf nodes and therefore each is a heap in itself
- In our algorithm, we will add higher level node one by one and use the heapify procedure to maintain the heap property as we add elements

# Build-max-Heap Procedure

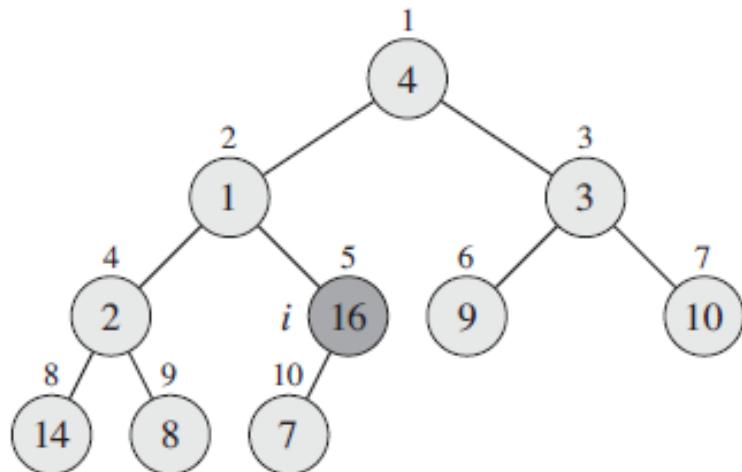
BUILD-MAX-HEAP( $A$ )

- 1     $A.\text{heap-size} = A.\text{length}$
- 2    **for**  $i = \lfloor A.\text{length}/2 \rfloor$  **downto** 1
- 3        MAX-HEAPIFY( $A, i$ )

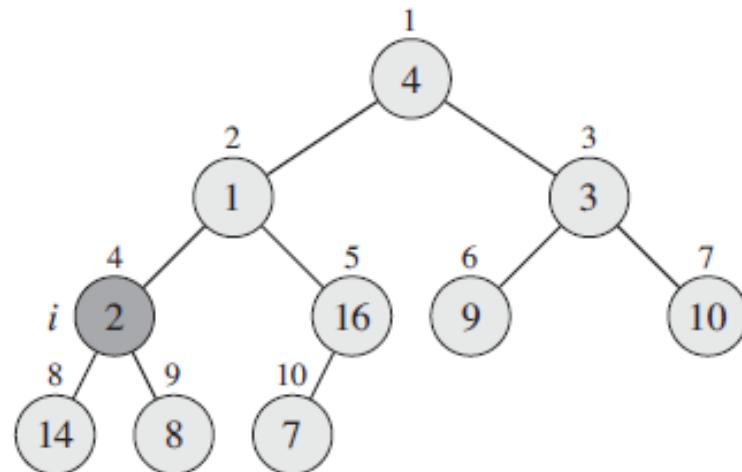
# Proof of Correctness

- Find the loop invariant
- Show that it is true prior to execution of the loop
- Show that it holds after execution of the loop
- Show that the loop invariant provides a useful property to show the correctness of the algorithm when loop terminates
- Loop invariant => all the elements from the initial value down to the highest value of i form a max-heap

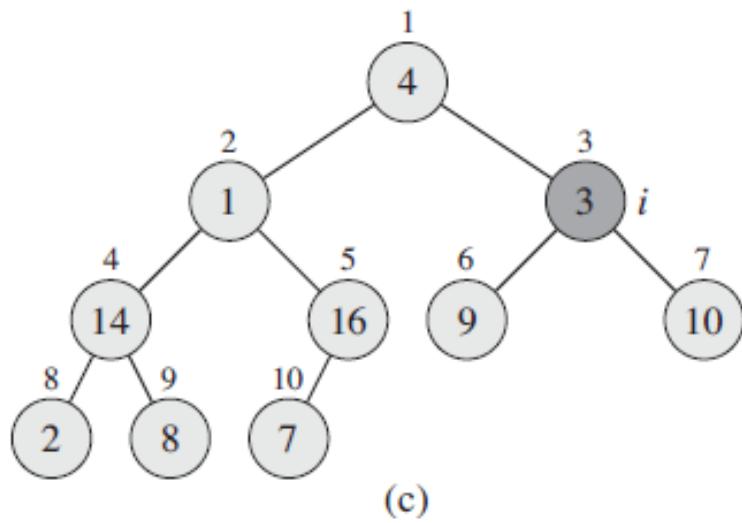
$A$	4	1	3	2	16	9	10	14	8	7
-----	---	---	---	---	----	---	----	----	---	---



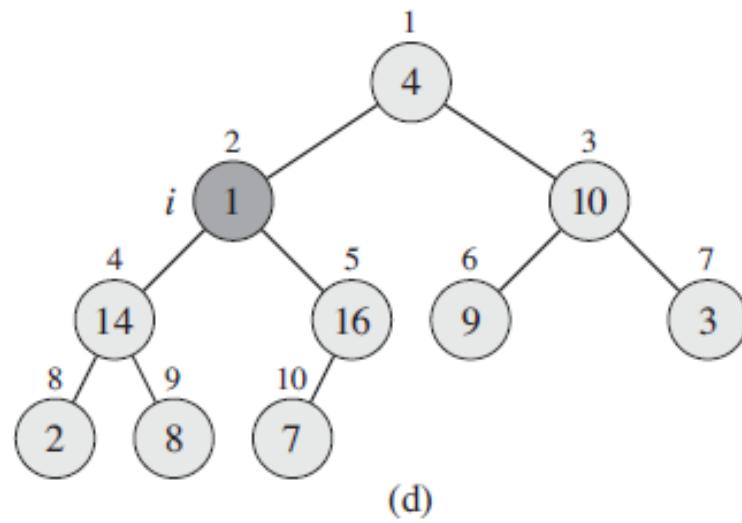
(a)



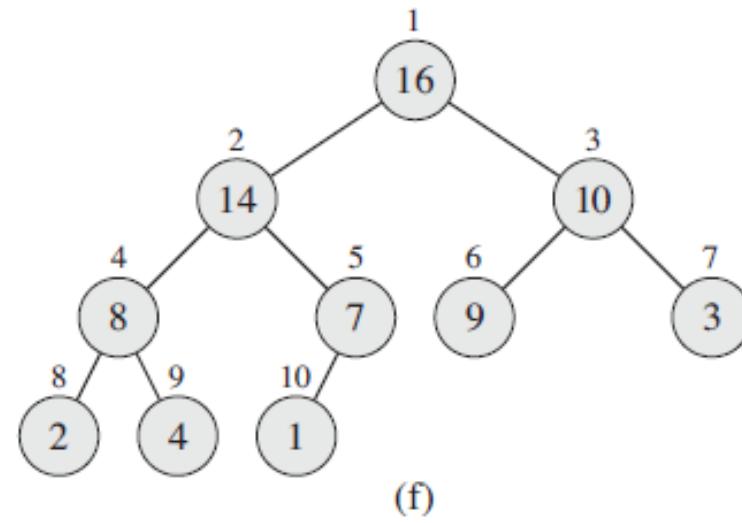
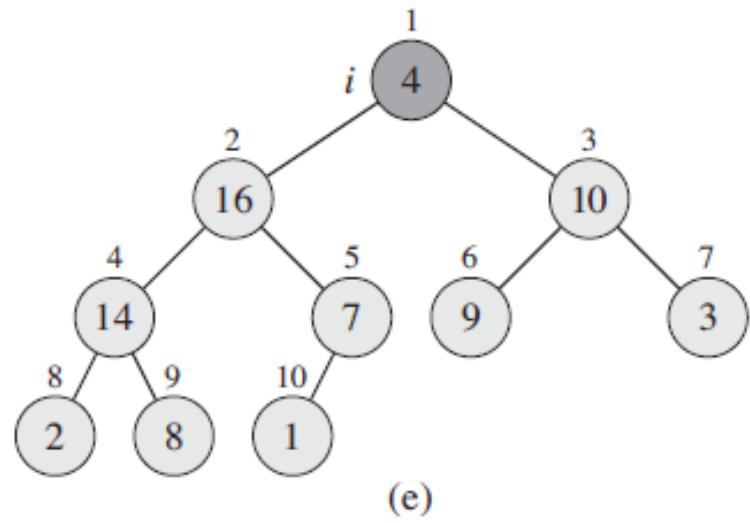
(b)



(c)



(d)



# Complexity Analysis

- Each call to heapify takes at most  $O(\lg n)$  time
- The procedure heapify is called for at most the total number of nodes in the heap i.e.  $n$
- Therefore the upper bound on the running time of the algorithm is  $O(n \lg n)$
- This is not however asymptotically tight
- A tighter bound can be obtained by noting that an  $n$ -element heap has:
  - Maximum height =  $\lfloor \lg n \rfloor$
  - Maximum number of nodes with height  $h$  =  $\left\lceil \frac{n}{2^{h+1}} \right\rceil$

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

We evaluate the last summation by substituting  $x = 1/2$  in the formula (A.8), yielding

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1 - 1/2)^2} \\ &= 2. \end{aligned}$$

Thus, we can bound the running time of BUILD-MAX-HEAP as

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

# Proof: Maximum nodes with height h

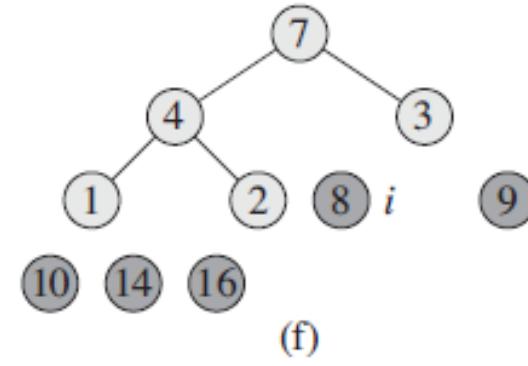
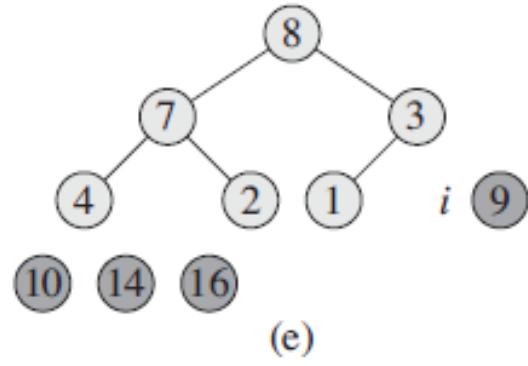
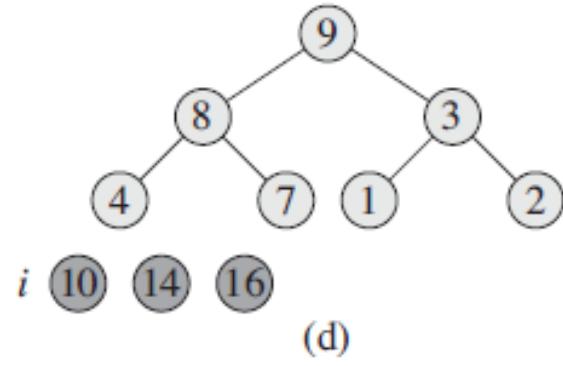
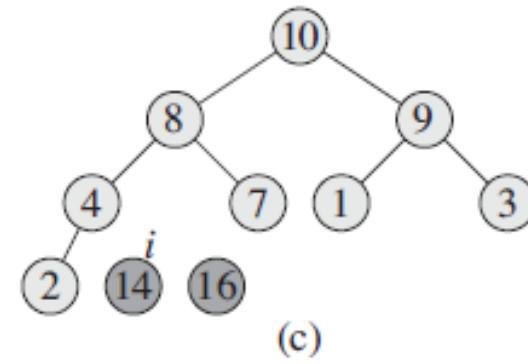
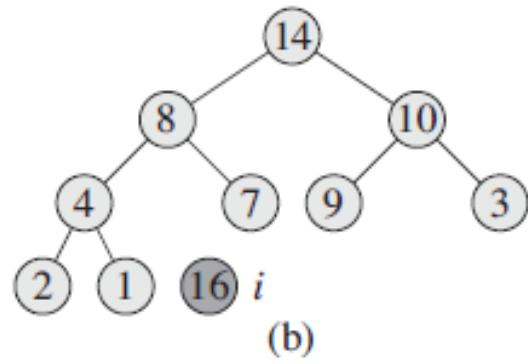
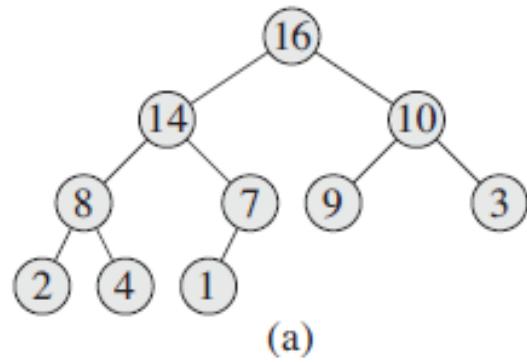
- All the nodes from  $\lceil n/2 \rceil + 1, \dots, n$  are leaf nodes
- Therefore, no. of nodes with height = 0  $\Rightarrow \lceil n/2 \rceil$
- No. of nodes with height = 1 will be half of this value  $\Rightarrow \lceil n/2 \rceil \times \frac{1}{2}$
- No. of nodes with height = 2 will be half of this value  $\Rightarrow \lceil n/2 \rceil \times \frac{1}{2^2}$
- Maximum no. of nodes with height h  $= \left\lceil \frac{n}{2^{h+1}} \right\rceil$

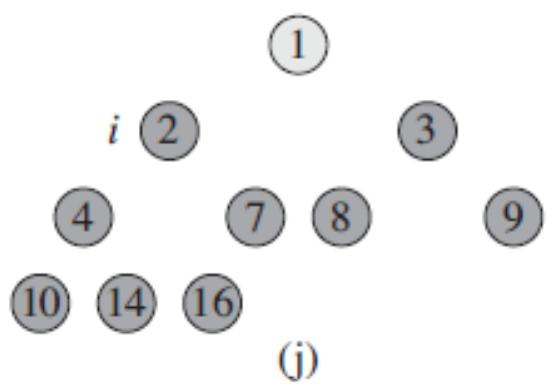
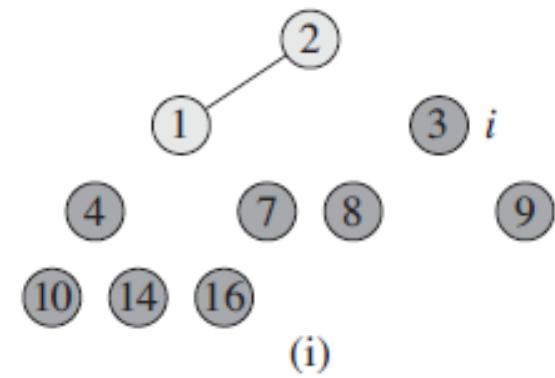
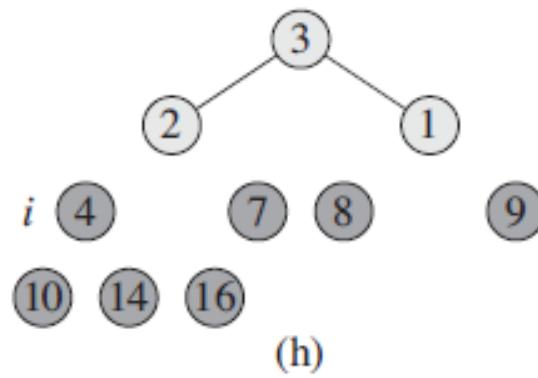
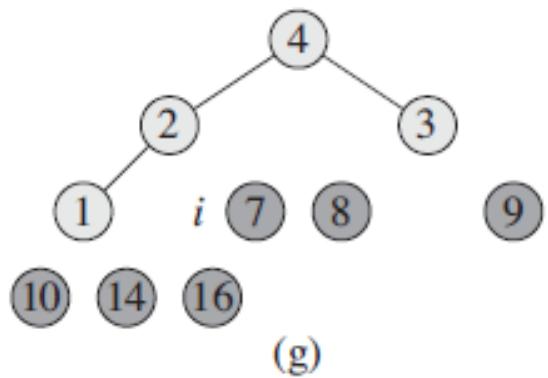
# Heapsort

HEAPSORT( $A$ )

- 1 BUILD-MAX-HEAP( $A$ )
- 2 **for**  $i = A.length$  **downto** 2
- 3     exchange  $A[1]$  with  $A[i]$
- 4      $A.heap-size = A.heap-size - 1$
- 5     MAX-HEAPIFY( $A, 1$ )

- Largest element is at the top and therefore can be placed at the end of the array
- This is done by exchanging the value of 1<sup>st</sup> element with the last element of the heap
- Since one element of the array is at its correct sorted position, we remove it from the heap by reducing the heap-size by 1
- After exchange, the root of the heap might not follow the heap property so we call Heapify with the root (1)





<i>A</i>	1	2	3	4	7	8	9	10	14	16
----------	---	---	---	---	---	---	---	----	----	----

(k)

# Complexity Analysis

- Build-heap takes  $O(n)$  time
- Each call to Heapify takes  $O(\lg n)$  time
- The total running time is thus  $O(n \lg n)$

# Priority Queue

- A data structure where the elements are accessed in the order of their priority
- For a set of elements  $S$ , each element is associated with a *key* value
- Priority queue is of two types: max-priority queue and min-priority queue
- Operations (Max-priority queue):
  - Insert( $S, x$ )
  - Maximum( $S$ )
  - Extract-Max( $S$ )
  - Increase-Key( $S, x, k$ )

# Implementation of Priority Queue

- The priority queue designed for a particular application stores only the key values
- Each element of the priority queue is associated with an object of the application for which priority queue is designed
- Key values are nothing but the priorities of the associated objects
- All the manipulation is done using the key values only
- A handle is maintained that associates the key value with the corresponding object
- The program object stores the index value of the key and the key stores the handle of the program object (could be a pointer to the object)
- Each time a key is moved from its position, the handles are updated so as to maintain the correct index value

# Heap-Maximum

HEAP-MAXIMUM( $A$ )

1   **return**  $A[1]$

# Heap-Extract-Max( $A$ )

HEAP-EXTRACT-MAX( $A$ )

```
1  if  $A.\text{heap-size} < 1$ 
2      error “heap underflow”
3   $max = A[1]$ 
4   $A[1] = A[A.\text{heap-size}]$ 
5   $A.\text{heap-size} = A.\text{heap-size} - 1$ 
6  MAX-HEAPIFY( $A, 1$ )
7  return  $max$ 
```

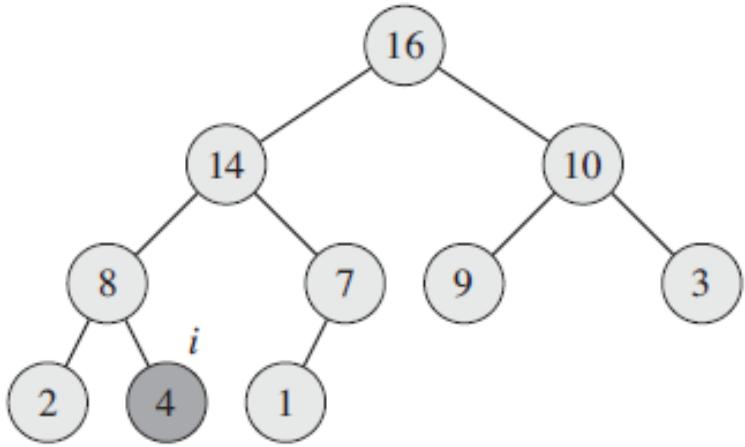
- Performs only constant amount of operations other than the call to Max-Heapify
- Running time is same as that of Max-Heapify i.e.  $O(\lg n)$

# Heap-Increase-Key( $A$ , $i$ , $key$ )

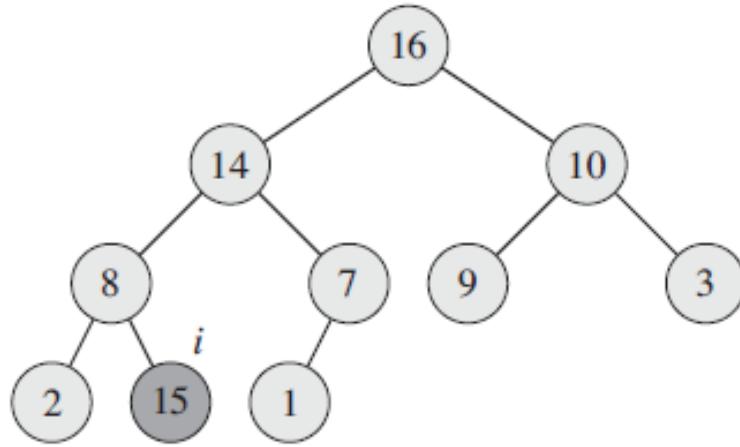
HEAP-INCREASE-KEY( $A$ ,  $i$ ,  $key$ )

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

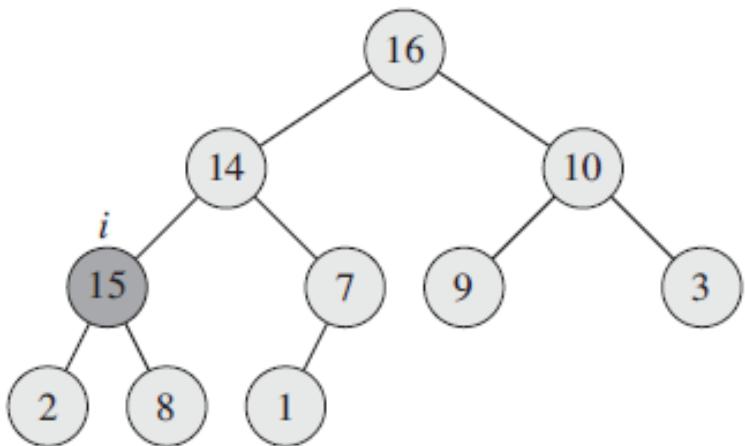
- The while loop runs at most up to the complete height of the heap
- Running time of the algorithm is  $O(\lg n)$



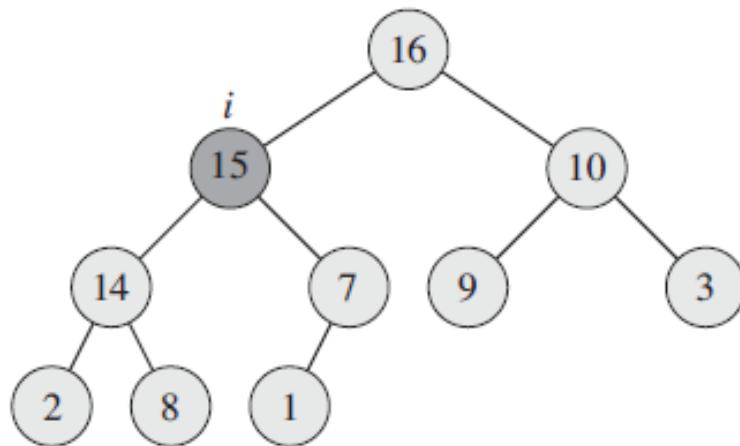
(a)



(b)



(c)



(d)

# Max-Heap-Insert( $A$ , $key$ )

**MAX-HEAP-INSERT**( $A, key$ )

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

- Steps 1 and 2 add constant time
- Step 3 runs in  $O(\lg n)$  time
- Therefore, runtime is bounded by  $O(\lg n)$

# Proof of Correctness: Heap-Increase-Key

- Loop invariant => At the start of each iteration of the **while** loop of lines 4–6, the subarray  $A[1,..A.\text{heap-size}]$  satisfies the max-heap property, except that there may be one violation:  $A[i]$  may be larger than  $A[\text{Parent}(i)]$
- **Initialization:** initially, the loop invariant is satisfied
- **Maintenance:** After each iteration the element at index ‘i’ is set to its correct value and i is set to the parent of node which might now violate the max-heap property and therefore, loop invariant is satisfied after every iteration
- **Termination:** loop terminates when either  $A[i] < \text{Parent}[i]$  or  $i = 1$ . If  $A[i] < \text{Parent}[i]$  then, all the elements  $A[1,..A.\text{heap-size}]$  satisfy the max-heap property
- If  $i=1$ , then we are at the root, which has no parent and again, all the elements  $A[1,..A.\text{heap-size}]$  satisfy the max-heap property.
- Thus, every element satisfies the max-heap property and therefore, the algorithm is correct.

# Exercise

- Give an  $O(nlgk)$ -time algorithm to merge  $k$  sorted lists into 1 sorted list, where  $n$  is the total number of elements in all the input lists.

# Quicksort

# Description

- Divide and conquer based algorithm, the steps can be summarized as:
- *Divide*: given an array  $A[p, \dots, r]$ , partition it into two (possibly empty) sub-arrays  $A[p, \dots, q-1]$  and  $A[q+1, \dots, r]$  such that  $A[p\dots q-1]$  contains all the elements less than or equal to  $A[q]$  and  $A[q+1, \dots, r]$  contains all the elements greater than or equal to  $A[q]$
- *Conquer*: Sort the two sub-arrays  $A[p, \dots, q-1]$  and  $A[q+1, \dots, r]$  by recursive calls to the quicksort procedure
- *Combine*: as the resulting sub-arrays are already sorted, nothing is to be done in the combine step

# Algorithm

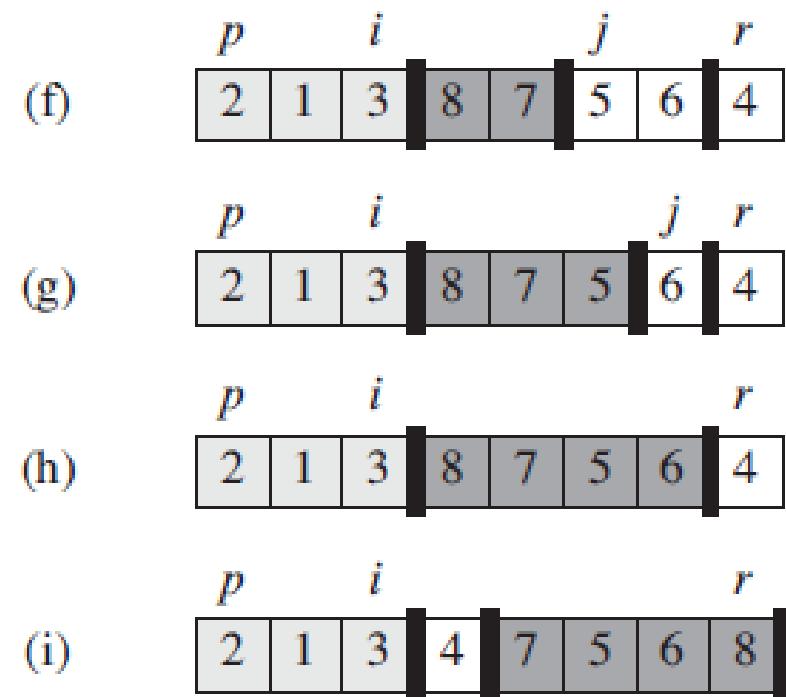
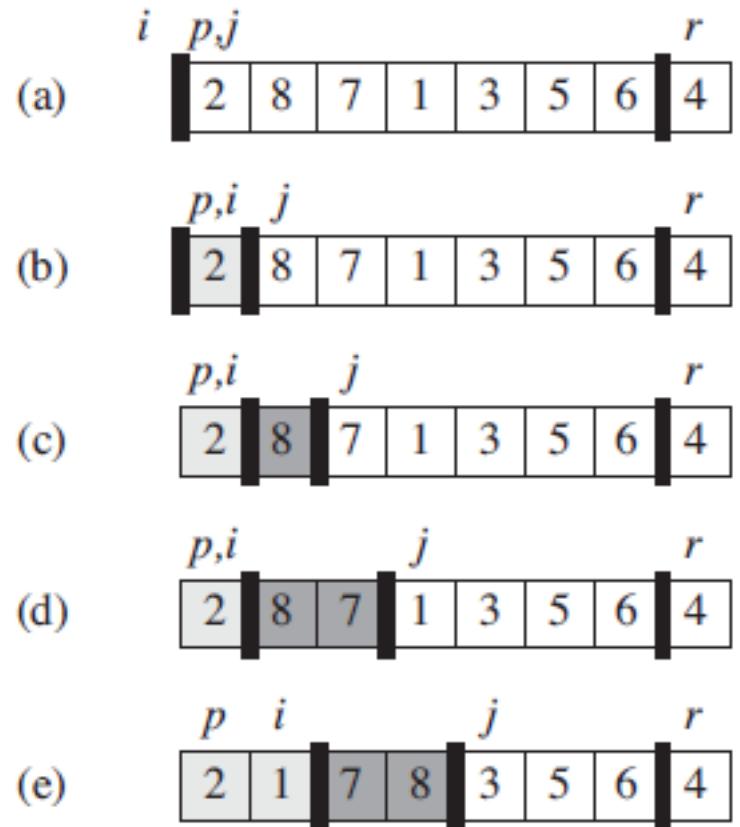
QUICKSORT( $A, p, r$ )

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

# Partition Function

PARTITION( $A, p, r$ )

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



$i = p-1$   
 $j = p$   
 $x = 4$



Compare  $A[j]$  and  $x$

$i = p$

$j = p$

Exchange  $A[i]$  with  $A[j]$  has no effect

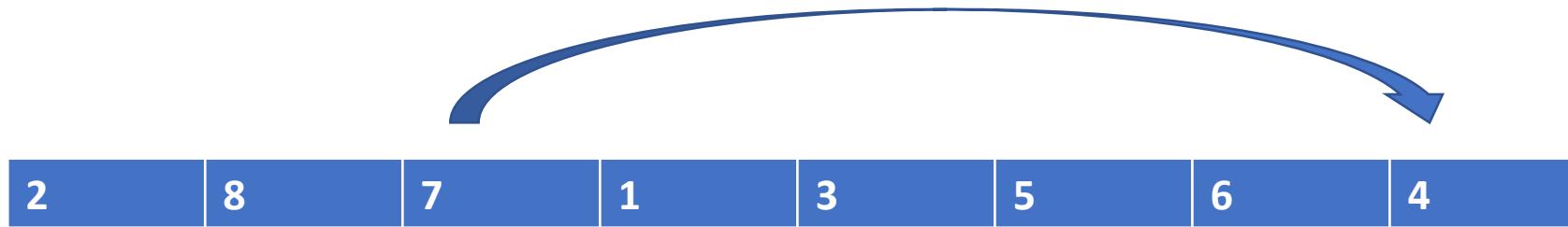
$i = p$   
 $j = p+1$



Compare  $A[j]$  and  $x$

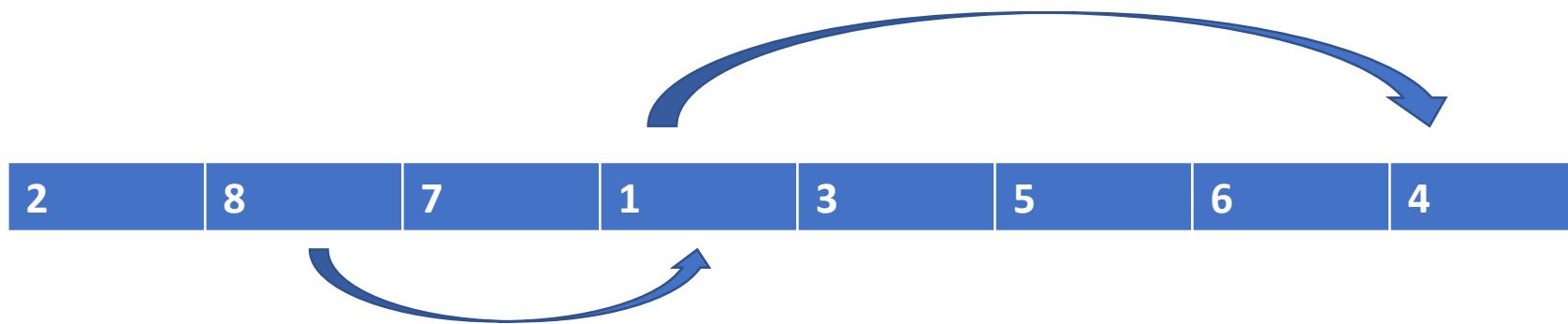
No action

$i = p$   
 $j = p+2$



Compare  $A[j]$  and  $x$   
No action

$i = p$   
 $j = p+3$



Compare  $A[j]$  and  $x$   
 $i = p+1$   
 $j=p+3$   
Exchange  $A[i]$  and  $A[j]$

$i = p+1$   
 $j=p+4$



Compare  $A[j]$  and  $x$

$i = p+2$

$j=p+4$

Exchange  $A[i]$  and  $A[j]$

$i = p+2$   
 $j=p+5$



Compare  $A[j]$  and  $x$

No action

$i = p+2$   
 $j=p+6$



Compare  $A[j]$  and  $x$   
No action

$i = p+2$   
 $j=r \rightarrow$  loop stops  
here



Exchange  $A[i+1]$  with  $A[r]$



# Performance Analysis

- Worst-case Partitioning: Will happen when each time the partition function returns an element  $A[q]$  such that all the other elements are either less than or equal to  $A[q]$  or greater than or equal to  $A[q]$  thus always resulting into  $n-1$  elements in one sub-array and 0 in the other
- The recurrence relation for the worst-case partitioning can be written as:

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

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

# Complexity Analysis of worst case

- Solution to the above recurrence is  $T(n) = \theta(n^2)$
- Thus, quicksort take  $\theta(n^2)$  time in the worst-case
- This worst case occurs only when the input array is already sorted
- It should be noted that insertion sort runs in  $\theta(n)$  time in this situation

# Best-case partitioning

- The best case occurs when the pivot element  $A[q]$  partitions the array  $A$  into two (almost) equal-size subarrays each time
- The performance of quicksort improves considerably in this case and it runs as fast as the merge sort
- The recurrence for the best case can be given as:

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

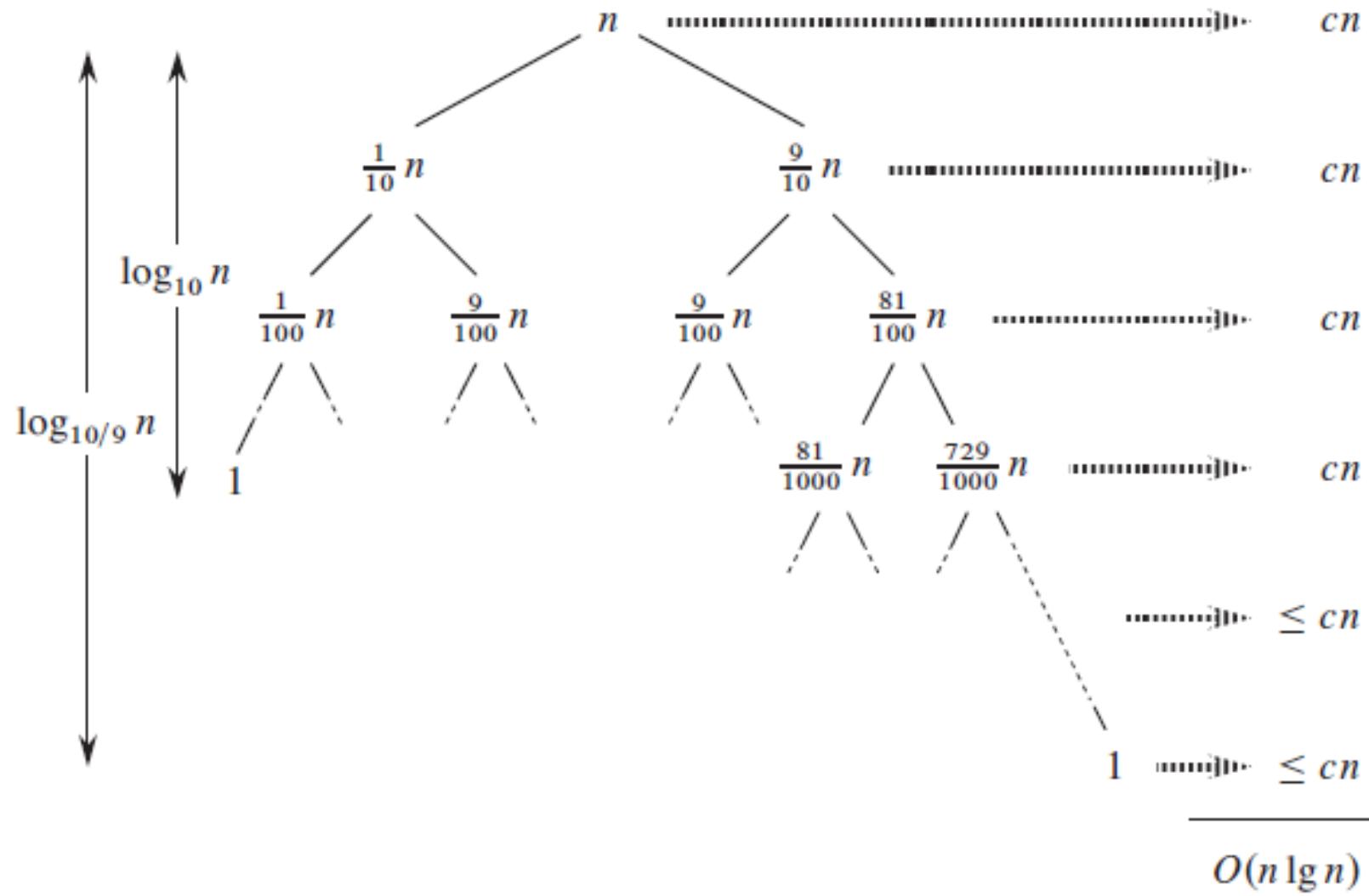
- Solution to this recurrence,  $T(n) = \theta(nlgn)$
- Thus, we get asymptotically faster algorithm if the partitioning is balanced

# Balanced Partitioning

- Even for very unbalanced partitioning like 9-to-1 at each level, we get good performance from quicksort i.e.  $O(n \log n)$

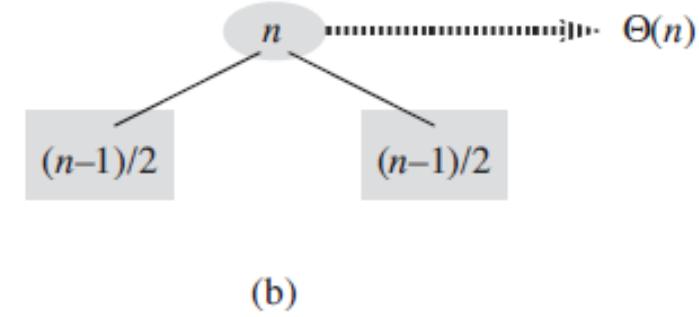
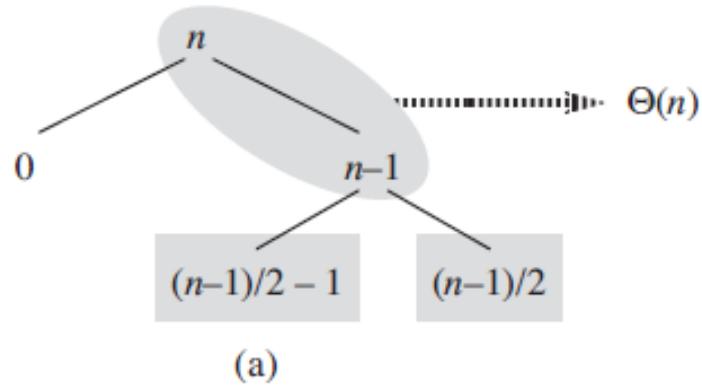
$$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn$$

- For even bad split like 99-to-1 as well, we get similar performance
- $O(n \log n)$  time complexity is always achieved if the partition has constant proportion



# Average Case

- For a random input, partitioning in constant proportion at each level is highly unlikely
- Some partitions could be unbalanced, some could be reasonably balanced and some fairly balanced
- The performance depends on the relative ordering of input numbers and not their actual value
- We consider all combinations of the input sequence equally likely
- As in next example, combination of alternative good and bad splits also results into  $O(nlgn)$  time complexity



Partitioning: 0,  $(n-1)/2 - 1$ ,  $(n-1)/2$

$$\text{Time} = \theta(n) + \theta(n - 1) = \theta(n)$$

The extra cost can be absorbed in the constant and thus the time complexity is  $O(n \lg n)$

# Analysis of Quicksort

- We will now show that the worst case running of quicksort takes  $O(n^2)$  time using substitution method
- Let  $T(n)$  be the worst case running time on input size  $n$ , we get the recurrence as:

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n - q - 1)) + \Theta(n)$$

- The sub-problems produced by the partition are of size  $n-1$  therefore,
- We guess that the solution is  $T(n) \leq cn^2$  for some constant  $c$ .

$$T(n) = \max_{0 \leq q \leq n-1} (cq^2 + c(n - q - 1)^2) + \Theta(n)$$

$$T(n) = c \cdot \max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) + \Theta(n)$$

- Above expression reaches its maximum at either of the end points as the second derivative is positive giving a concave curve
- Both the end points result into  $(n - 1)^2$
- Thus,

$$T(n) = c \cdot \max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) + \Theta(n) \leq (n - 1)^2$$

$$T(n) = c \cdot \max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) + \Theta(n) \leq n^2 - 2n + 1$$

$$T(n) \leq c(n^2 - 2n + 1) + \Theta(n)$$

$$\leq cn^2$$

- The value of  $c$  is chosen large enough to dominate  $\Theta(n)$  term

$$\Rightarrow T(n) = O(n^2)$$

- Thus,  $T(n) = \Theta(n^2)$

# Randomized Quicksort

RANDOMIZED-PARTITION( $A, p, r$ )

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

RANDOMIZED-QUICKSORT( $A, p, r$ )

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

# Expected Running Time

Lemma: Let  $X$  be the total number of operations performed during partitioning over the entire execution of Quicksort on an  $n$ -element array. Then, its expected running time is  $O(n+X)$ .

Proof:

- The partition step adds to the complexity of algorithm.
- We will consider all the steps in partitioning accept the condition check at line 4
- All the steps except line 4 and the two calling of quicksort takes  $O(n)$  time in total
- Thus, the complete running time of the algorithm is  $O(n+X)$
- $X$  remains to be calculated in the above discussion

- We consider a particular ordering of input sequence as  $Z_{ij} = \{z_i, z_{i+1}, \dots, z_j\}$  such that,  $z_i$  is the  $i$ th smallest element in the input sequence and  $Z_{ij}$  defines the set of elements between  $z_i$  and  $z_j$  inclusively
- In the above arrangement any pair  $z_i$  and  $z_j$  is compared **at most once**
- We define an indicator random variable  $X_{ij}$  as:

$$X_{ij} = I\{z_i \text{ is compared to } z_j\}$$

Here, we consider the comparison that takes place anytime during the entire execution of the algorithm

- Since each pair is compared at most once, the total number of comparisons can be characterized as:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}$$

Taking expectation both sides we get,

$$E[X] = E\left[ \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij} \right]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr\{z_i \text{ is compared to } z_j\}$$

$$\begin{aligned}\Pr\{z_i \text{ is compared to } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is chosen as first pivot from } Z_{ij}\} \\ &= \Pr\{z_i \text{ is chosen as first pivot from } Z_{ij}\} \\ &\quad + \Pr\{z_j \text{ is chosen as first pivot from } Z_{ij}\}\end{aligned}$$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1}.$$

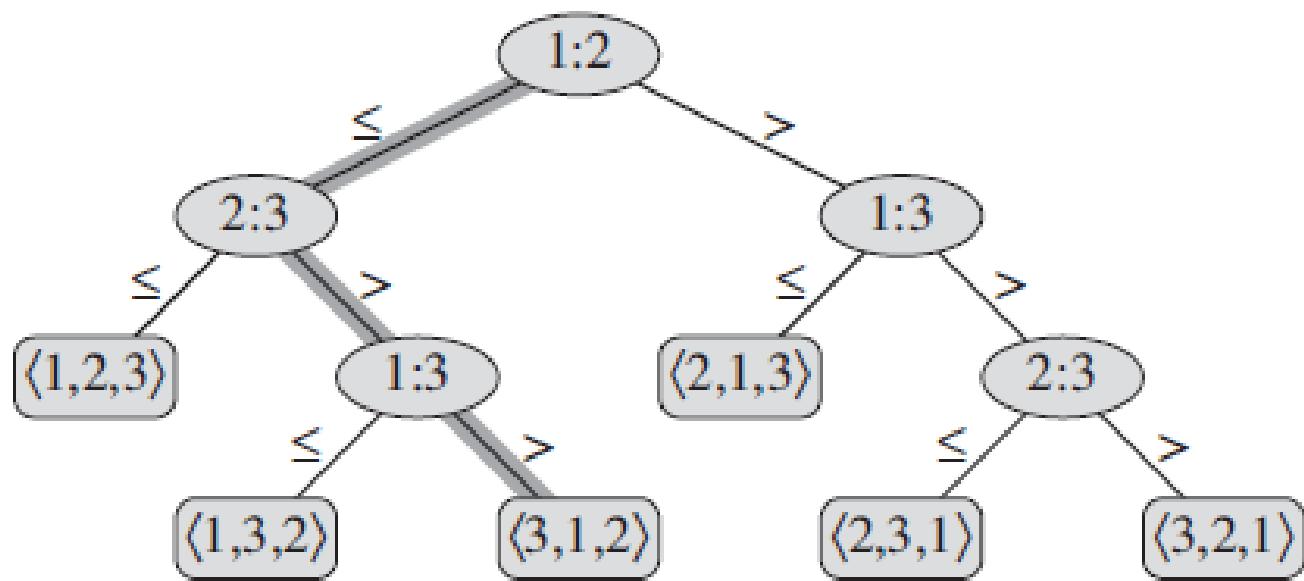
$$\begin{aligned}\mathbb{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\ &< \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k} \\ &= \sum_{i=1}^{n-1} O(\lg n) \\ &= O(n \lg n).\end{aligned}$$

$$\sum_{i=1}^n 1/i = \int_1^n \frac{dx}{x} \Rightarrow O(\lg n)$$

# Sorting in Linear Time

Counting sort, Radix sort, Bucket sort

# Decision tree model for comparison sort



# Decision tree model

- Each node represents a comparison between elements  $A[i]$  and  $A[j]$
- The result of comparison directs the search path
- If  $A[i] < A[j]$ , the search takes left path and right otherwise
- Each leaf node represents one of the  $n!$  possible permutations of the input sequence
- Thus, the leaf node that is obtained in the end corresponds to the correct sorted sequence
- For any comparison based sorting algorithm to be correct, there must be  $n!$  leaf nodes in it and every leaf node must be reachable from the root

# Lower Bound for Comparison sort

- For any algorithm, we can represent its working in the form of the decision tree, the height of the decision tree represents a bound on the worst case
- Suppose, the decision tree has height ‘ $h$ ’ and has ‘ $l$ ’ reachable nodes
- A binary tree with height  $h$  can have at most  $2^h$  leaf nodes  
$$\Rightarrow l \leq 2^h$$
- Further, since all possible permutations of the input sequence must appear as reachable leaf nodes for the algorithm to be correct we have,  
$$\begin{aligned} n! &\leq l \\ \Rightarrow n! &\leq 2^h \\ \Rightarrow h &\geq \lg(n!) \end{aligned}$$
- Since  $n\lg n$  is a lower bound on  $\lg(n!)$  we have  
$$h = \Omega(n\lg n)$$

# Counting Sort

COUNTING-SORT( $A, B, k$ )

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

```
i=1
for j = 0 to k
  while(C[j]!=0)
    B[i] = j;
    C[j]--;
    i++;
```

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

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

(a)

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

(b)

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

	2	2	4	6	7	8
--	---	---	---	---	---	---

(c)

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

	1	2	4	6	7	8
--	---	---	---	---	---	---

(d)

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

	1	2	4	5	7	8
--	---	---	---	---	---	---

(e)

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

(f)

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

// Execution of Step 4

	1	2	3	4	5	6
C:	0	0	0	0	0	0

j=1

A[j] = 2

C[A[j]] = C[2]

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

j=2

A[j] = 5

C[A[j]] = C[5]

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

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

j=3

A[j] = 3

C[A[j]] = C[3]

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

j=4

A[j] = 0

C[A[j]] = C[0]

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

j=5

A[j] = 2

C[A[j]] = C[2]

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

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

j=6

A[j] = 3

C[A[j]] = C[3]

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

j=7

A[j] = 0

C[A[j]] = C[0]

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

j=8

A[j] = 3

C[A[j]] = C[3]

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

// Execution of Step 7

C:	1	2	3	4	5	6
	2	0	2	3	0	1

k=1

$$C[1] = C[1]+C[0]$$

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

k=2

$$C[2] = C[2]+C[1]$$

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

k=3

$$C[3] = C[3]+C[2]$$

	0	1	2	3	4	5
	2	2	4	7	0	1

k=4

$$C[4] = C[4]+C[3]$$

	0	1	2	3	4	5
	2	2	4	7	7	1

k=5

$$C[5] = C[5]+C[4]$$

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

// Execution of Step 10

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

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

j=8, A[j] = 3  
B[C[A[j]]] = A[j]  
B[C[3]] => B[7] = 3  
C[3]=C[3]-1 = 6

	0	1	2	3	4	5		
C:	2	2	4	6	7	8		
	1	2	3	4	5	6	7	8
B:	-	-	-	-	-	-	3	-

j=7, A[j]=0  
B[C[0]] => B[2] = 0  
C[0]=C[0]-1 = 1

	0	1	2	3	4	5	6	7	8
C:	1	2	4	6	7	8			
B:	-	0	-	-	-	-	3	-	

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

j=6, A[j]=3  
 $B[C[3]] \Rightarrow B[6] = 3$   
 $C[3]=C[3]-1 = 5$

C:	0	1	2	3	4	5	6	7	8
B:	-	0	-	-	-	3	3	-	

j=5, A[j]=2  
 $B[C[2]] \Rightarrow B[4] = 2$   
 $C[2]=C[2]-1 = 3$

C:	0	1	2	3	4	5		
B:	-	0	-	2	-	3	3	-

j=4, A[j]=0  
 $B[C[0]] \Rightarrow B[1] = 0$   
 $C[0]=C[0]-1 = 0$

C:	0	1	2	3	4	5	6	7	8
B:	0	0	-	2	-	3	3	-	

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

j=3, A[j]=3

B[C[3]] => B[5] = 3

C[3]=C[3]-1 = 4

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

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

j=2, A[j]=5

B[C[5]] => B[8] = 5

C[5]=C[5]-1 = 7

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

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

j=1, A[j]=2

B[C[2]] => B[3] = 2

C[2]=C[2]-1 = 2

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

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

# Analysis

- Runs in  $\theta(n + k)$  time
- Often used when  $k = O(n)$  when it runs in  $\theta(n)$  time
- Beats the lower bound of comparison sort i.e.  $\Omega(n \lg n)$
- It is a *Stable* sorting algorithm: numbers in the output array appear in the same order as in the input array
- Useful in Radix sort

# Radix Sort

**RADIX-SORT( $A, d$ )**

- 1 **for**  $i = 1$  **to**  $d$
- 2       use a stable sort to sort array  $A$  on digit  $i$

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

# Analysis

- For a sequence of  $n$  numbers in the range 0 to  $k$ , each having  $d$ -digits, the algorithm runs in  $\theta(d(n + k))$  time
- Runs in linear time when  $d$  is constant and  $k = O(n)$
- Usually less preferred over quicksort
- Quicksort can use memory caches etc. in better manner
- Counting sort which is used as an intermediate sorting method, does external sorting which can be avoided in quicksort

# Proof that counting sort works

- Induction method is used to prove that it works
- For a 1-digit number, it sorts the array correctly
- For 2-digit number sequence, the sorting on the most significant digits maintains the lower order sorting order for tie and therefore, the sorting is correct for 2-digit
- Assume true for n-digit and prove the same for n+1th digit
- This proves that the counting sort works correctly

# Bucket Sort

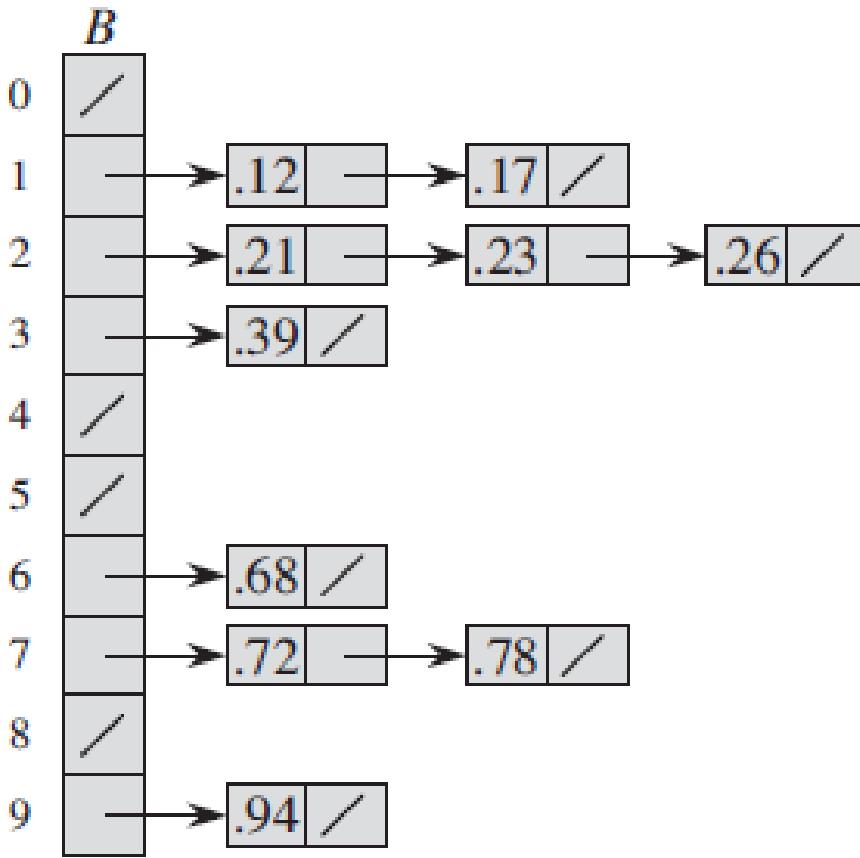
- Assumes that the input is drawn from a uniform distribution
- Average running time is  $O(n)$
- Input is generated using a random process and uniformly and independently distributed over  $[0, 1]$
- For sorting the interval is divided into  $n$  equal-size sub-intervals called *buckets*
- Numbers are distributed among the buckets
- We expect each bucket to have only small number of elements
- Each bucket is then sorted and elements are listed in order

## BUCKET-SORT( $A$ )

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

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

(a)



(b)

# Correctness of Bucket Sort

- For any two elements  $A[i]$  and  $A[j]$
- Suppose  $A[i] \leq A[j] \Rightarrow \lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$
- Therefore, either  $A[i]$  will go to same bucket as  $A[j]$  or it will go to a bucket lower than that of  $A[j]$
- In either case, it will be placed before  $A[i]$  in the final sorted array
- Thus, the algorithm is correct

# Complexity Analysis

- Each step takes  $O(n)$  time except lines 7 and 8 that perform sorting of each bucket using insertion sort
- The running time can be described as:

$$T(n) = \theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

- Since, the input is a uniform distribution we can calculate the expected running time by calculating the expectation of the above equation

$$E[T(n)] = E \left[ \theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \right]$$

$$= \theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)]$$

$$= \theta(n) + \sum_{i=0}^{n-1} O[E(n_i^2)]$$

- $E[n_i^2]$  value will be same for all the buckets as the elements are distributed uniformly
- It can be shown that  $E[n_i^2] = 2 - 1/n$
- Thus, the average case running time of bucket sort comes out to be

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

- Thus, bucket sort runs in linear time if the input is drawn from uniform distribution
- The linear running time can also be obtained if the input is not drawn from a uniform distribution as long as the sum of squares of bucket sizes is linear in total number of elements

# Dynamic Programming

Matrix Chain Multiplication

# Introduction

- Greedy problem solving technique, dynamic programming and divide and conquer are all similar in that they try to solve the problem by dividing it into smaller sub-problems and solving the sub-problems in order to get the solution of the original problem
- Dynamic programming and greedy techniques are used in solving **optimization problems**
- Unlike divide and conquer, the dynamic programming technique is useful when the subproblems overlap
- The idea in dynamic programming is to solve the overlapping subproblems only once and save the result which can be *looked up* in future rather than solving the same problem again
- When subproblems do not overlap, dynamic programming may not be a good approach

# When to apply Dynamic Programming

- Optimization Problem – the problem has several solutions including potentially more than one optimal solution and the task is to find **an** optimal solution
- Optimal substructure – the solution to the optimization problem involves solving the generated subproblems optimally (and the subproblems can be optimized independently).
- Overlapping Subproblems – the space of the generated subproblems is **small** in the sense that many subproblems overlap
- It is an example of a technique where we can increase space complexity to reduce time complexity i.e. there exists **space-time trade-off**

# Steps involved in Dynamic Programming

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution.
- Construct an optimal solution from computed information.

# Matrix-Chain Multiplication

- We are given a sequence (chain)  $\langle A_1, A_2, \dots, A_n \rangle$  of  $n$  matrices to be multiplied, and we wish to compute the product

$$A_1 A_2 \dots A_n$$

- The number of computations required for calculating the above product might change dramatically with the order of computation of products
- The computation problem in Matrix-Chain multiplication is to find an optimal order of multiplication for the  $n$  matrices so that the number of computations involved is minimum

# Example

- Suppose, A1, A2 and A3 are three matrices having dimensions (10x100), (100x5) and (5x50) respectively
- $((A_1, A_2)A_3) - (10 \times 100 \times 5) + (10 \times 5 \times 50) = 7500$  scalar multiplications
- $(A_1(A_2, A_3)) - (10 \times 100 \times 50) + (100 \times 5 \times 50) = 50000 + 25000 = 75000$  scalar multiplications
- The problem in matrix chain multiplication is not actually calculating the matrix product but finding an optimal sequence of matrices for calculating the product so that the number of scalar multiplications is minimum

# Brute Force Approach

- This would involve enumerating all possible sequences of multiplication for  $n$  matrices and calculating the cost for each
- A minimum cost sequence will comprise the solution to the problem
- Multiplication of a sequence of  $n$  matrices can be obtained by multiplying two subsequences of size  $k$  and  $n-k$  each where we put a parenthesis between  $k$ th and  $(k+1)$ th matrices
- The two sequences can in turn be evaluated by again putting the parenthesis between any two matrices in the sequence
- Thus, we get a recurrence relation for solving the problem

$$P(n) = \begin{cases} 1, & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n - k), & \text{if } n \geq 2 \end{cases}$$

- Using an inductive proof assuming  $P(n) \geq 2^n - 1 \Rightarrow \Omega(2^n)$
- We can use substitution method to show that  $P(n) = \Omega(2^n)$
- Thus, the brute-force approach yields an algorithm that grows exponentially with the problem size  $n$

# Applying Dynamic Programming

- We note that once we have generated two parenthesized sequences of  $k$  and  $n-k$  lengths
- We must solve each of them optimally in order to get the optimal solution to the problem
- If it were not so then there would be some other sequence within them which could give even smaller number of multiplications and therefore, a solution better than what we have got which contradicts our assumption of finding the optimal sequence
- Therefore, we must solve both the sub problems optimally in order to get optimal solution to the original problem
- Further, both the subproblems are independent as the optimal sequence of the first  $k$  matrix products has nothing in common with optimal sequence of remaining  $n-k$  products

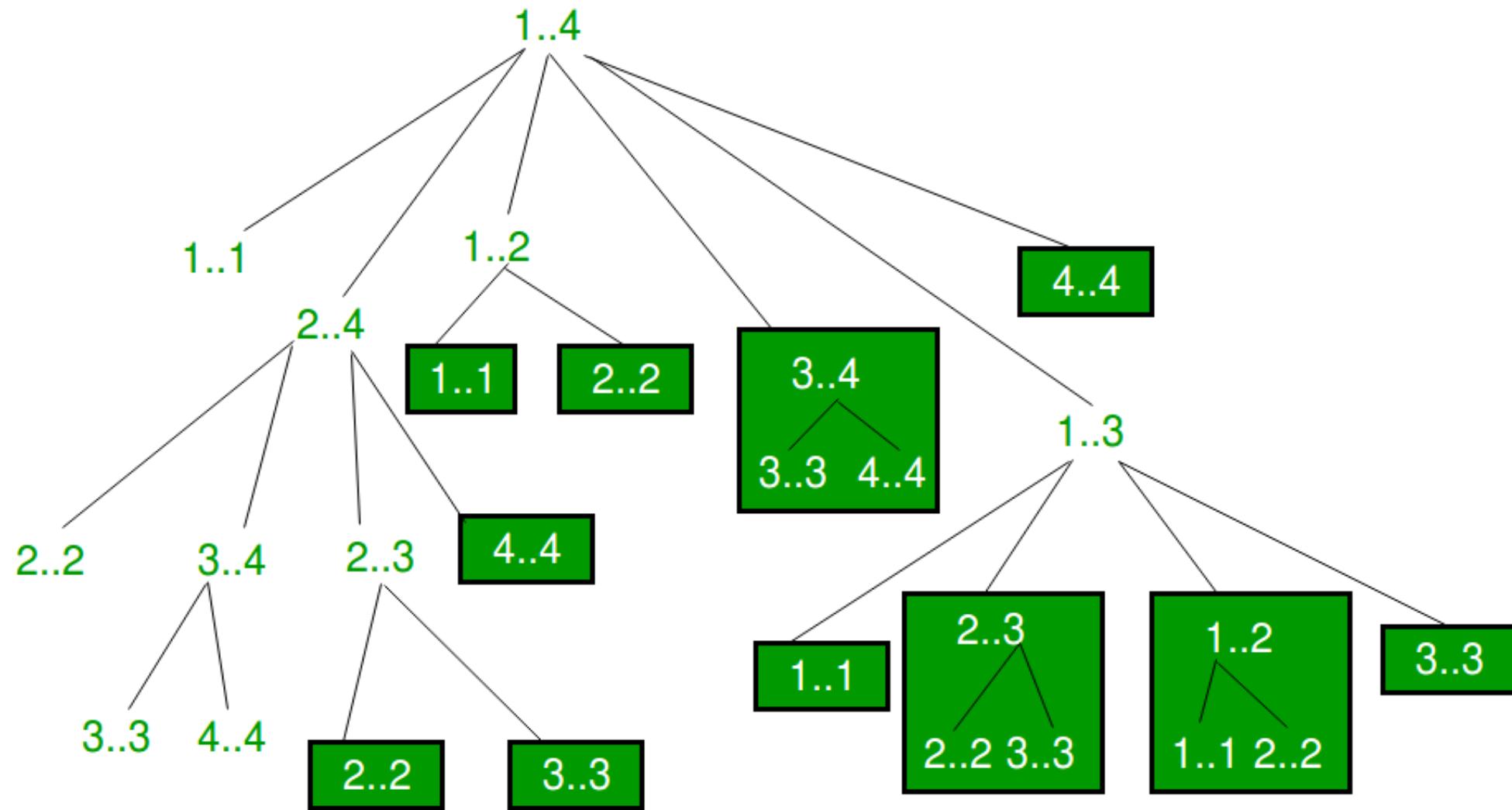
- We further note that for each possible value of the initial division performed at  $k$ , we get two subsequences, each of which has many subsequences that are common for different values of  $k$
- Thus, while enumerating the possible sequences for multiplication recursively we are generating subproblems that overlap considerably
- Thus, the space of subproblems is not very large and we are actually generating same subproblems over and over again
- Therefore, the **optimal substructure and overlapping subproblems**, two important guiding properties for a dynamic programming based solution, both hold true for this problem
- Therefore, we can look for a dynamic programming based solution where we only solve a subproblem once and use its result every time the same subproblem is generated

$\langle(A1A2A3A4A5A6)(A7A8A9A10)\rangle$

$\langle(A1A2A3A4A5)(A6A7A8A9A10)\rangle$

$\langle((A1A2A3)(A4A5A6))((A7A8A9)A10)\rangle$

$\langle((A1A2A3)(A4A5))(A6(A7A8A9)A10)\rangle$



# DP based solution

- We solve the problem step-by-step as given in slide 4
- In the first step we assume that the optimal division takes place at the kth matrix (characterizing the optimal solution)
- The two subsequences generated after this step must also give optimal solution for generating the optimal solution to original problem (recursively defining optimal solution)
- We will then compute the value of the optimal solution
- The fourth step might not always be required, if required some extra information may need to be stored for constructing the solution

- So, now we can characterize the optimal solution for a sequence from  $i$  to  $j$  as follows:

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$$

Here,  $m[i, j]$  denotes the total number of scalar multiplications required for a sequence of length  $j-i+1$  and each matrix  $A_i$  has dimension  $p_{i-1} p_i$

Above is a recursive equation and we need to solve the subproblems to get the optimal solution

The above process is repeated for all possible values of  $k$

Thus the recursive equation becomes:

$$m[i, j] = \begin{cases} 0, & \text{if } i = j \\ \min_{i \leq k \leq j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\}, & \text{if } i < j \end{cases}$$

- The above equation will be solved in a bottom-up manner to get the optimal sequence
- Each matrix  $A_i$  has dimension  $p_{i-1} p_i$
- The method takes as input a sequence  $\langle p_0, p_1, \dots, p_n \rangle$  of length  $n+1$  and uses the matrix  $m[i, j]$  to store the optimal cost for the sequence  $i$  to  $j$
- It also uses an auxiliary matrix  $s[i, j]$  to store the index of the optimal split

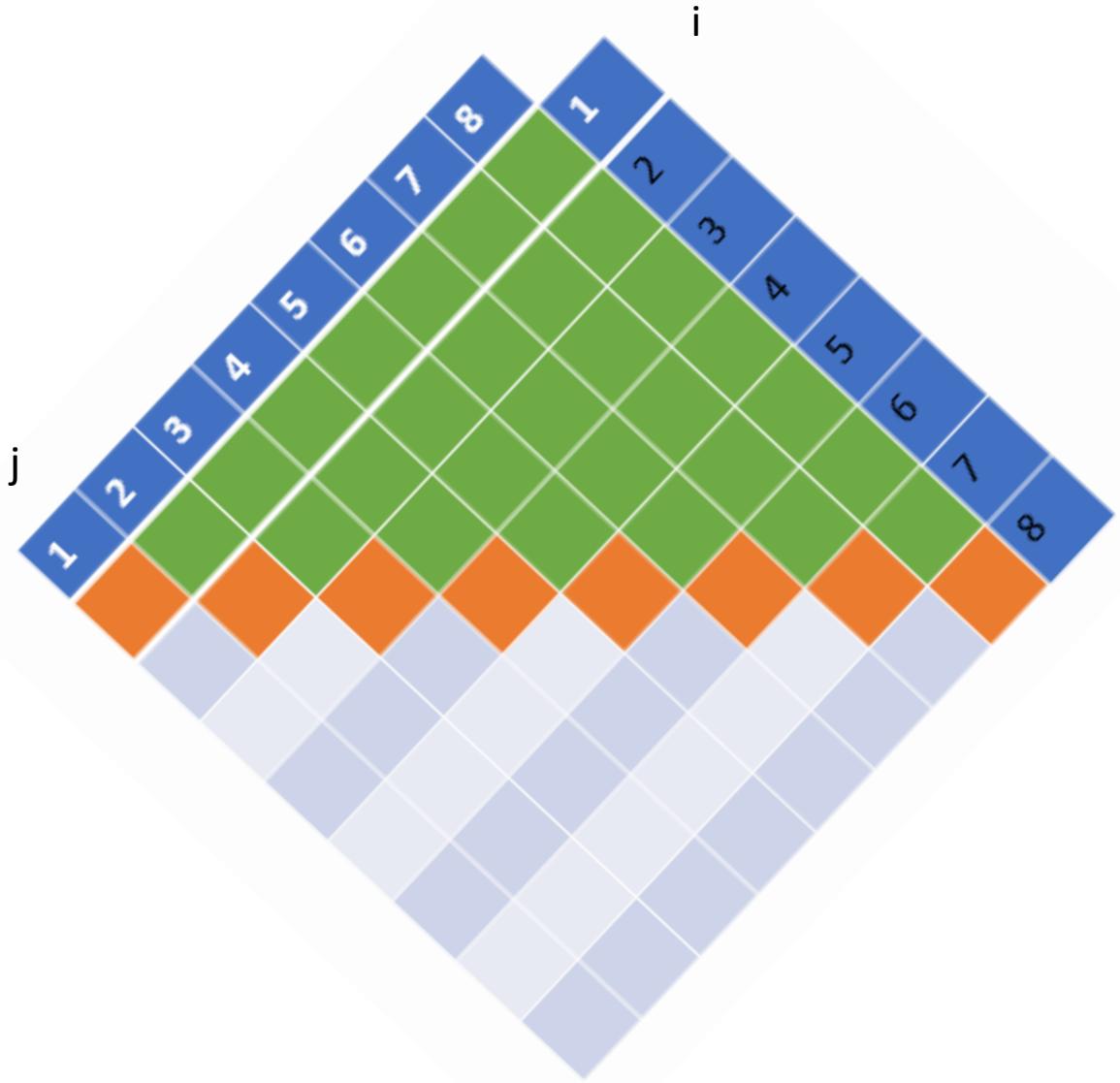
## MATRIX-CHAIN-ORDER( $p$ )

```
1   $n = p.length - 1$ 
2  let  $m[1..n, 1..n]$  and  $s[1..n - 1, 2..n]$  be new tables
3  for  $i = 1$  to  $n$ 
4       $m[i, i] = 0$ 
5  for  $l = 2$  to  $n$           //  $l$  is the chain length
6      for  $i = 1$  to  $n - l + 1$ 
7           $j = i + l - 1$ 
8           $m[i, j] = \infty$ 
9          for  $k = i$  to  $j - 1$ 
10              $q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$ 
11             if  $q < m[i, j]$ 
12                  $m[i, j] = q$ 
13                  $s[i, j] = k$ 
14  return  $m$  and  $s$ 
```

j

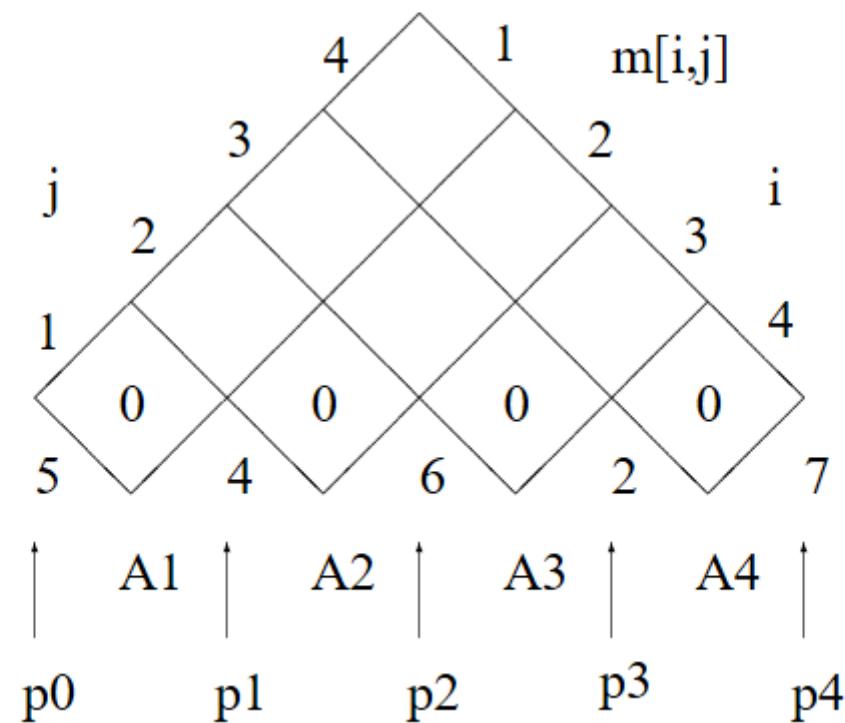
	1	2	3	4	5	6	7	8
1	orange	green	green	green	green	green	green	blue
2	light blue	orange	green	green	green	green	green	blue
3	white	orange	green	green	green	green	green	blue
4	light blue	orange	green	green	green	green	green	blue
5	white	orange	green	green	green	green	green	blue
6	light blue	orange	green	green	green	green	green	blue
7	white	orange	green	green	green	green	green	blue
8	light blue	orange	green	green	green	green	green	blue

i



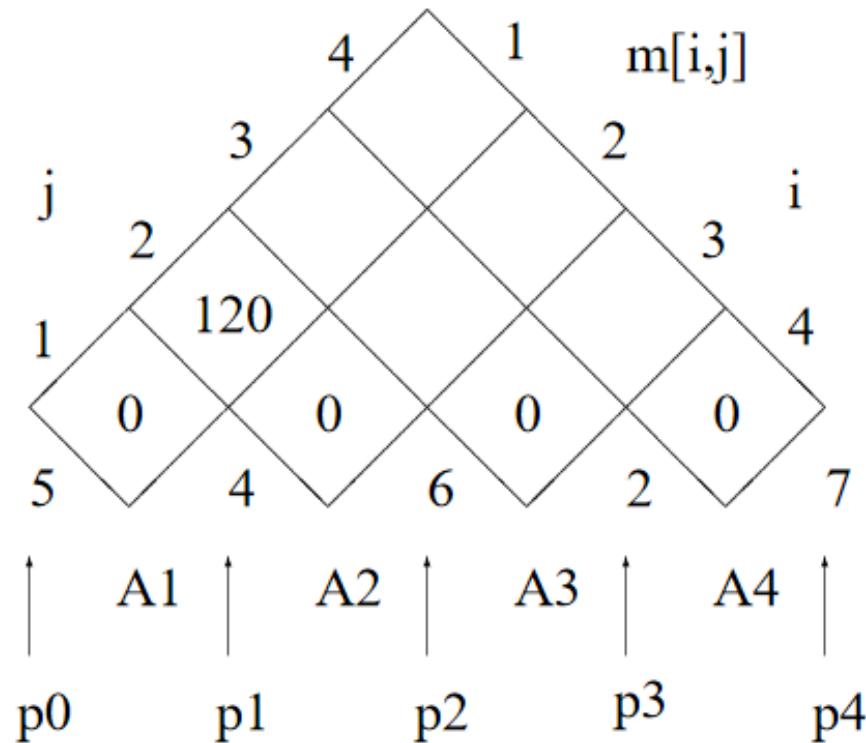
**Example:** Given a chain of four matrices  $A_1, A_2, A_3$  and  $A_4$ , with  $p_0 = 5, p_1 = 4, p_2 = 6, p_3 = 2$  and  $p_4 = 7$ . Find  $m[1, 4]$ .

### S0: Initialization



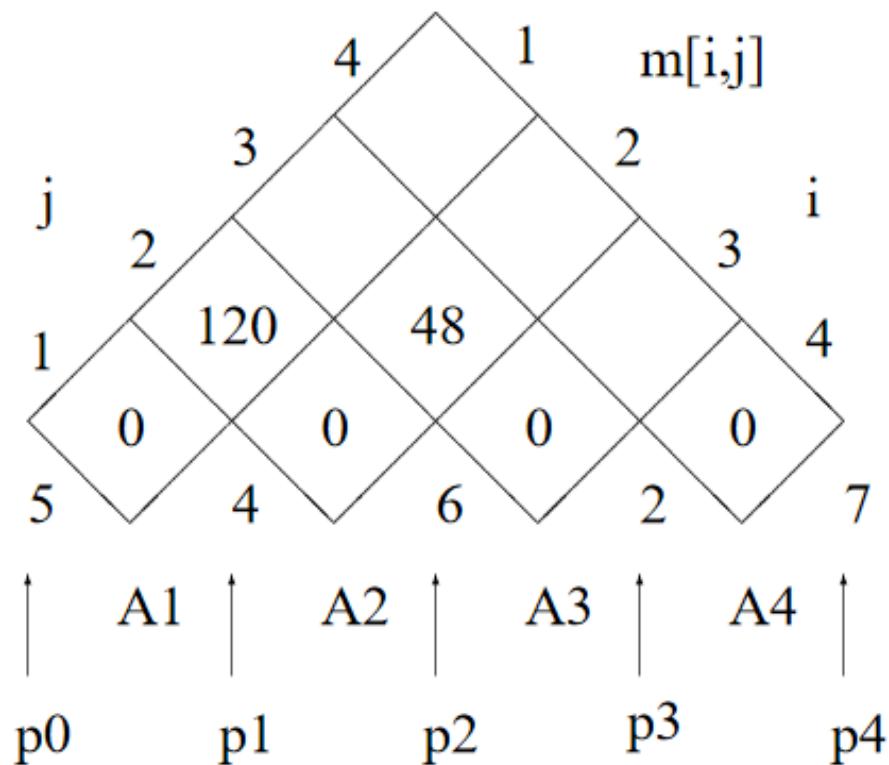
**Step 1: Computing  $m[1, 2]$**  By definition

$$\begin{aligned}m[1, 2] &= \min_{1 \leq k < 2} (m[1, k] + m[k + 1, 2] + p_0 p_k p_2) \\&= m[1, 1] + m[2, 2] + p_0 p_1 p_2 = 120.\end{aligned}$$



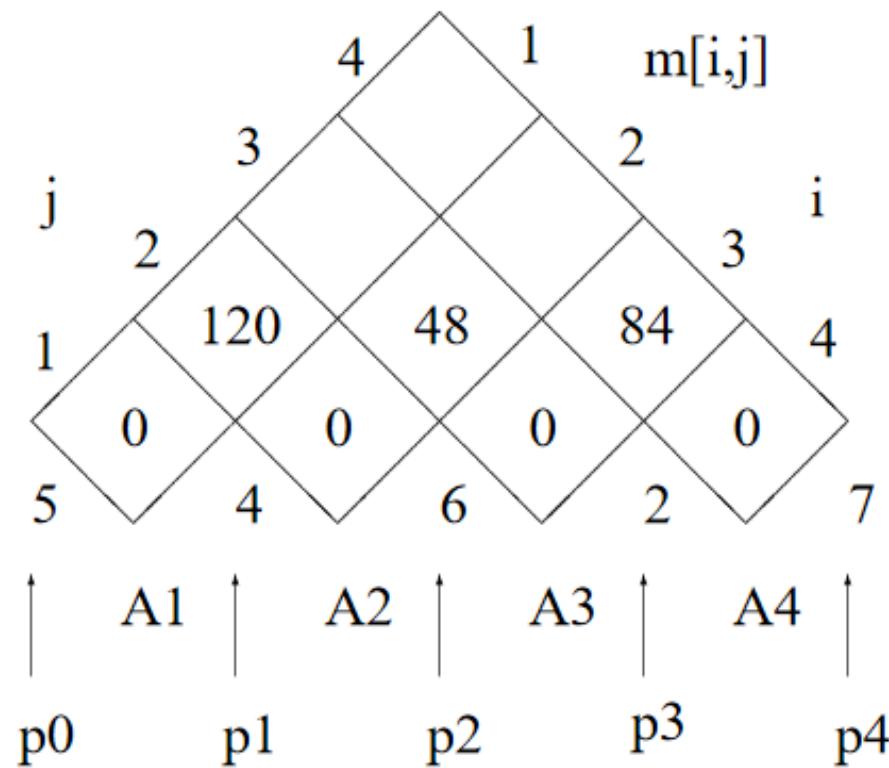
**Step 2: Computing  $m[2, 3]$**  By definition

$$\begin{aligned}m[2, 3] &= \min_{2 \leq k < 3} (m[2, k] + m[k + 1, 3] + p_1 p_k p_3) \\&= m[2, 2] + m[3, 3] + p_1 p_2 p_3 = 48.\end{aligned}$$



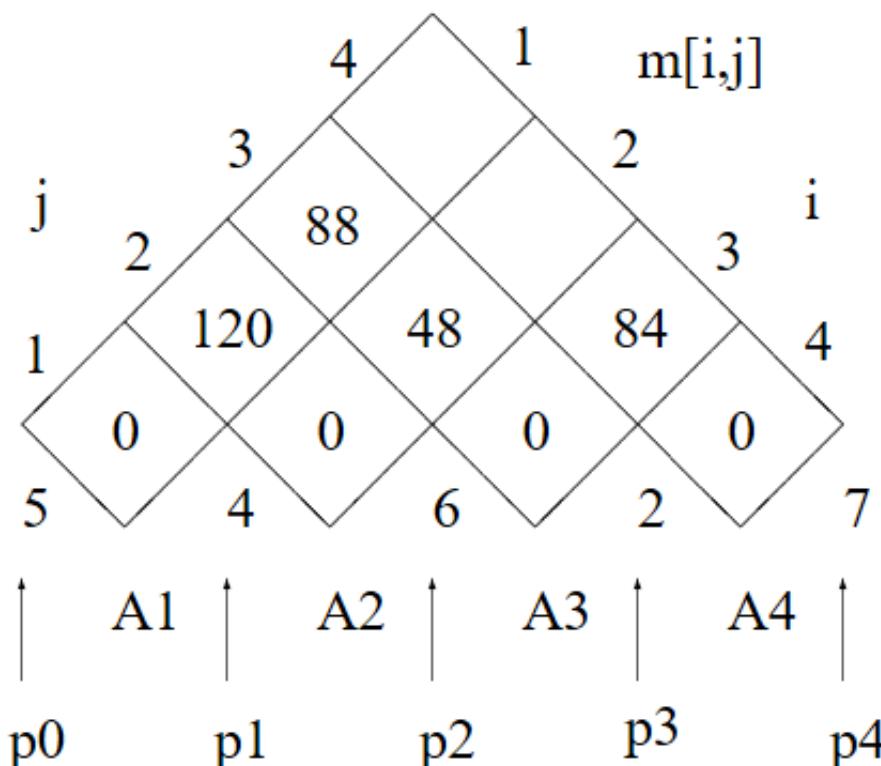
**Step 3: Computing  $m[3, 4]$**  By definition

$$\begin{aligned}m[3, 4] &= \min_{3 \leq k < 4} (m[3, k] + m[k + 1, 4] + p_2 p_k p_4) \\&= m[3, 3] + m[4, 4] + p_2 p_3 p_4 = 84.\end{aligned}$$



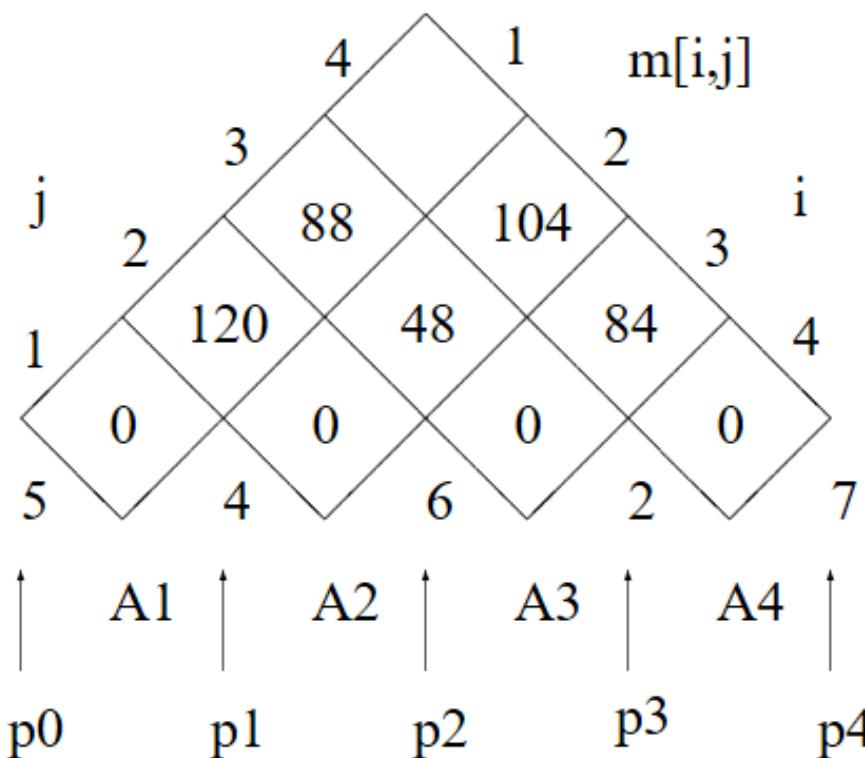
#### Step4: Computing $m[1, 3]$ By definition

$$\begin{aligned}m[1, 3] &= \min_{1 \leq k < 3} (m[1, k] + m[k + 1, 3] + p_0 p_k p_3) \\&= \min \left\{ \begin{array}{l} m[1, 1] + m[2, 3] + p_0 p_1 p_3 \\ m[1, 2] + m[3, 3] + p_0 p_2 p_3 \end{array} \right\} \\&= 88.\end{aligned}$$



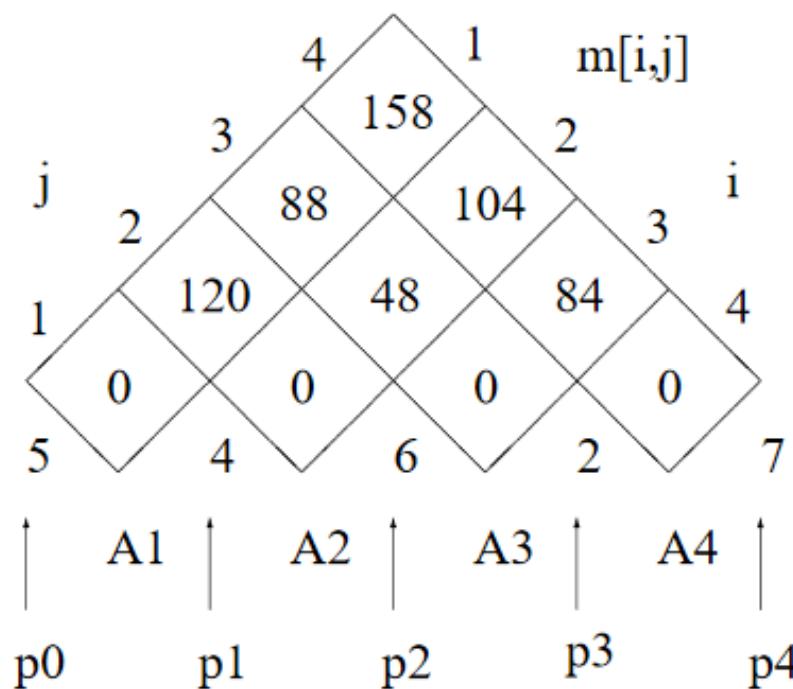
### Step 5: Computing $m[2, 4]$ By definition

$$\begin{aligned} m[2, 4] &= \min_{2 \leq k < 4} (m[2, k] + m[k + 1, 4] + p_1 p_k p_4) \\ &= \min \left\{ \begin{array}{l} m[2, 2] + m[3, 4] + p_1 p_2 p_4 \\ m[2, 3] + m[4, 4] + p_1 p_3 p_4 \end{array} \right\} \\ &= 104. \end{aligned}$$



**St6: Computing  $m[1, 4]$**  By definition

$$\begin{aligned}
 m[1, 4] &= \min_{1 \leq k < 4} (m[1, k] + m[k + 1, 4] + p_0 p_k p_4) \\
 &= \min \left\{ \begin{array}{l} m[1, 1] + m[2, 4] + p_0 p_1 p_4 \\ m[1, 2] + m[3, 4] + p_0 p_2 p_4 \\ m[1, 3] + m[4, 4] + p_0 p_3 p_4 \end{array} \right\} \\
 &= 158.
 \end{aligned}$$



# Top-Down v/s Bottom Up Approach

- For solving such problems, instead of using a recursive algorithm directly, we apply a tabular bottom-up approach
- A recursive algorithm applies a top-down approach solving a subproblem just when it is encountered, we do not want to do it here because sub problems overlap
- In the bottom-up approach as we shall see, we start with the smallest instance, solve it and save the result and then move further
- In this method, whenever a new problem is encountered all its subproblems are already solved and therefore, we only have to look at the optimal solution of the subproblem already stored somewhere

- Since, the smallest instance is solved first, the bottom-up approach does not use a recursive algorithm for problem solving
- For solving such problems, top down approach may also be applied with a technique called **memoization**
- Memoization refers to memorization of the results of sub-problems generated while solving the problem
- The algorithm before making a recursive call checks if the result is already available or not
- If yes, then the saved result is used rather than solving the subproblem again
- Among the two approaches bottom-up algorithm is useful if all the subproblems are generated at least once, this is because it won't involve the overhead of making recursive calls
- Top-down approach on the other hand, is useful when not all the subproblems need be generated, it saves time by only solving the subproblems that are generated in the course of solving a particular problem instance

**RECURSIVE-MATRIX-CHAIN**( $p, i, j$ )

```
1  if  $i == j$ 
2      return 0
3   $m[i, j] = \infty$ 
4  for  $k = i$  to  $j - 1$ 
5       $q = \text{RECURSIVE-MATRIX-CHAIN}(p, i, k)$ 
         +  $\text{RECURSIVE-MATRIX-CHAIN}(p, k + 1, j)$ 
         +  $p_{i-1} p_k p_j$ 
6      if  $q < m[i, j]$ 
7           $m[i, j] = q$ 
8  return  $m[i, j]$ 
```

**MEMOIZED-MATRIX-CHAIN**( $p$ )

```
1   $n = p.length - 1$ 
2  let  $m[1..n, 1..n]$  be a new table
3  for  $i = 1$  to  $n$ 
4      for  $j = i$  to  $n$ 
5           $m[i, j] = \infty$ 
6  return LOOKUP-CHAIN( $m, p, 1, n$ )
```

**LOOKUP-CHAIN**( $m, p, i, j$ )

```
1  if  $m[i, j] < \infty$ 
2      return  $m[i, j]$ 
3  if  $i == j$ 
4       $m[i, j] = 0$ 
5  else for  $k = i$  to  $j - 1$ 
6       $q = \text{LOOKUP-CHAIN}(m, p, i, k)$ 
           + LOOKUP-CHAIN( $m, p, k + 1, j$ ) +  $p_{i-1} p_k p_j$ 
7      if  $q < m[i, j]$ 
8           $m[i, j] = q$ 
9  return  $m[i, j]$ 
```

**PRINT-OPTIMAL-PARENS**( $s, i, j$ )

```
1  if  $i == j$ 
2      print “A” $_i$ 
3  else print “(”
4      PRINT-OPTIMAL-PARENS( $s, i, s[i, j]$ )
5      PRINT-OPTIMAL-PARENS( $s, s[i, j] + 1, j$ )
6      print “)”
```

# Dynamic Programming

Longest Common Subsequence

# Longest Common Subsequence

- A DNA strand consists of a finite sequence over four types of proteins Adenine, Guanine, Cytosine and Thymine i.e. {A,G,C,T}
- We might need to find the similarity between two DNA strands obtained from two different organisms
- One way to find this similarity is to find a common subsequence
- A subsequence that is common to both the strands contains a sequence of proteins that occurs in the same order in both the stands except that there could be some omits in the subsequence

# Problem of Finding LCS

- E.g.  $S_1 = ACCGGTCGAGTGCGCGGAAGGCCGGCCGAA$   
 $S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA$
- $S_3 = GTCGTCGGAAGGCCGGCCGAA$  is a longest common subsequence of  $S_1$  and  $S_2$
- Here, all the bases in  $S_3$  appear in both  $S_1$  and  $S_2$  in the same order but not necessarily consecutively
- There could be more than one LCSs and the task is to find any one LCS
- More formally, in this for given two sequences  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$  we wish to find out the longest common subsequence of  $X$  and  $Y$

# Characterizing a Subsequence

- The subsequence of X and Y can be characterized as another sequence  $Z = \langle z_1, z_2, \dots, z_k \rangle$  for which there exists a strictly increasing sequence  $\langle i_1, i_2, \dots, i_k \rangle$  of X such that for all  $j = 1, 2, \dots, k$ , we have  $x_{i_j} = z_j$
- E.g.  $Z = \langle B, C, D, B \rangle$  is a subsequence of  $X = \langle A, B, C, B, D, A, B \rangle$  with index sequence  $\langle 2, 3, 5, 7 \rangle$
- E.g.  $X = \langle A, B, C, B, D, A, B \rangle$  and  $Y = \langle B, D, C, A, B, A \rangle$
- Sequence  $\langle B, C, A \rangle$  is a subsequence of X and Y
- $\langle B, C, D, A \rangle$  and  $\langle B, D, A, B \rangle$  are two longest subsequences of X and Y

# Brute Force Approach

- In the simplest way of solving the problem of finding the optimal subsequence we would enumerate all possible subsequences of the two given sequences and choose the one which is longest
- This approach is not good especially when the two given sequences are very long
- We can apply dynamic programming to solve this problem
- We shall characterize the problem as one having the optimal substructure and overlapping sub-problems properties and use dynamic programming technique to solve it

# Optimal Substructure

- Suppose  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$  be two given sequences and let  $Z = \langle z_1, z_2, \dots, z_k \rangle$  be an LCS of  $X$  and  $Y$

Then we have,

- If  $x_m = y_n$ , then  $z_k = x_m = y_n$  and  $Z_{k-1}$  is an LCS of  $X_{m-1}$  and  $Y_{n-1}$
  - If  $x_m \neq y_n$ , then  $z_k \neq x_m$  implies that  $Z$  is an LCS of  $X_{m-1}$  and  $Y$
  - If  $x_m \neq y_n$ , then  $z_k \neq y_n$  implies that  $Z$  is an LCS of  $X$  and  $Y_{n-1}$
- 
- From the above, it is clear that the LCS problem has the optimal substructure property as the solution to the problem involves optimally solving the subproblems

# Recursive Solution

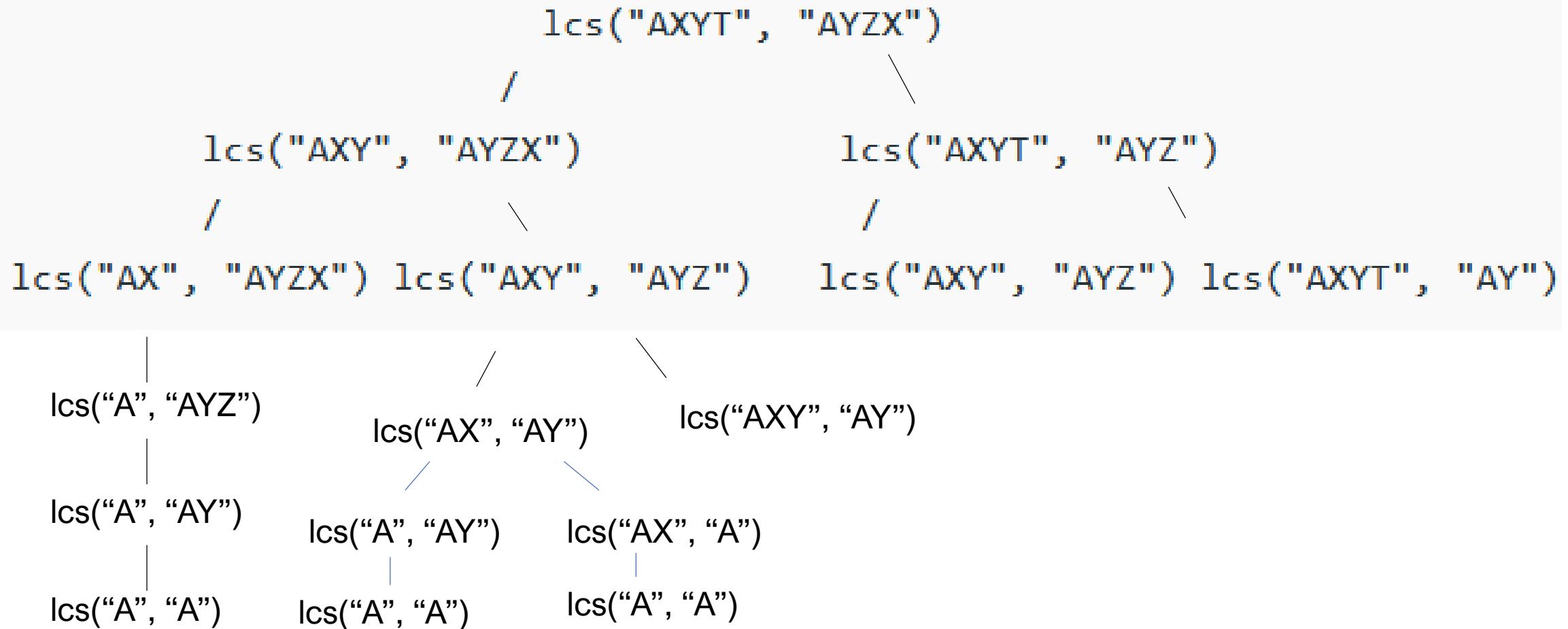
- A recursive solution to the above can be constructed by utilizing the property that an optimal solution to the problem involves finding an optimal subsequence for a sequence of smaller length
- Further, we can see that for evaluating each of the two subsequences optimally we need to enumerate all possible subsequences of smaller length within it
- Thus, many of the sequences will overlap within a subproblem
- We can apply DP in order to reduce the number of subproblems by evaluating each possible subsequence only once

# Recursive Equation

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

- In the above equation, we rule out some of the subproblems based on the problem conditions while in matrix chain multiplication, no subproblems were ruled out
- There are other dynamic programming problems in which some subproblems may be completely ruled out based on problem conditions
- Above equation results in at most  $\theta(mn)$  distinct subproblems and we can use dynamic programming to solve it

Strings: “AXYT” and “AYZX”



## LCS-LENGTH( $X, Y$ )

```
1   $m = X.length$ 
2   $n = Y.length$ 
3  let  $b[1..m, 1..n]$  and  $c[0..m, 0..n]$  be new tables
4  for  $i = 1$  to  $m$ 
5       $c[i, 0] = 0$ 
6  for  $j = 0$  to  $n$ 
7       $c[0, j] = 0$ 
8  for  $i = 1$  to  $m$ 
9      for  $j = 1$  to  $n$ 
10         if  $x_i == y_j$ 
11              $c[i, j] = c[i - 1, j - 1] + 1$ 
12              $b[i, j] = "\nwarrow"$ 
13         elseif  $c[i - 1, j] \geq c[i, j - 1]$ 
14              $c[i, j] = c[i - 1, j]$ 
15              $b[i, j] = "\uparrow"$ 
16         else  $c[i, j] = c[i, j - 1]$ 
17              $b[i, j] = "\leftarrow"$ 
18 return  $c$  and  $b$ 
```

**PRINT-LCS**( $b, X, i, j$ )

```
1  if  $i == 0$  or  $j == 0$ 
2      return
3  if  $b[i, j] == \nwarrow$ 
4      PRINT-LCS( $b, X, i - 1, j - 1$ )
5      print  $x_i$ 
6  elseif  $b[i, j] == \uparrow$ 
7      PRINT-LCS( $b, X, i - 1, j$ )
8  else PRINT-LCS( $b, X, i, j - 1$ )
```

$j$	0	1	2	3	4	5	6
$i$	$y_j$	B	D	C	A	B	A
0	$x_i$	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	-1	-1	1	-2
3	C	0	1	1	2	-2	2
4	B	0	1	1	2	2	3
5	D	0	1	2	2	3	3
6	A	0	1	2	2	3	4
7	B	0	1	2	2	3	4

X = <A, B, C, B, D, A, B>

Y= <B, D, C, A, B, A>

BCBA

BCAB

BDAB

# Complexity Analysis

- LCS-Length takes  $\theta(mn)$  time
- Print-LCS takes  $\theta(m + n)$  time as it decrements either i or j in every iteration
- The b table can be removed altogether and the construction of the optimal sequence can be done using c entries with additional comparison to decide which among  $c[i-1,j-1]$ ,  $c[i-1,j]$  and  $c[i,j-1]$  was used to calculate  $c[i,j]$  for each  $c[i,j]$ th entry

# Time taken in case of Brute Force Search

- The maximum length of LCS will be smaller among m and n
- We have the following possibilities for LCS:

$$\binom{n}{1} + \binom{n}{2} + \binom{n}{3} + \dots + \binom{n}{n} = 2^n - 1$$

- Total time for calculation =  $O(n \cdot 2^n)$  as we will take atmost n steps in verifying if the obtained subsequence is present in S2
- Thus, brute-force takes -  $O(n \cdot 2^n)$

# Dynamic Programming

Travelling Salesman Problem

# Travelling Salesperson Problem

- Given a directed graph  $G=(V,E)$  with edge costs  $c_{ij}$ . The Travelling salesman problem is to find a tour of minimum cost
- A tour is a directed simple cycle that includes every vertex in  $V$
- Cost of a tour is the sum of cost of all the edges in the tour
- The edge cost is defined as below:

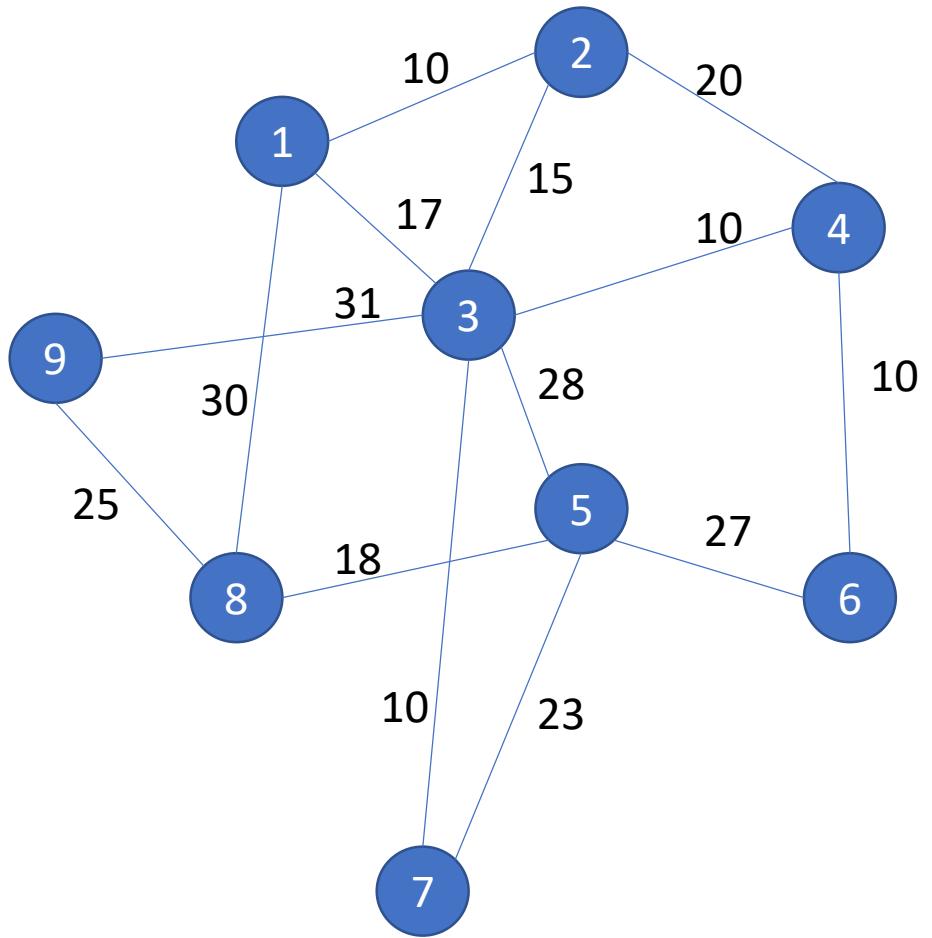
$$c_{ij} = \begin{cases} 0 & \text{if } i = j \\ > 0 & \text{if } \langle i, j \rangle \in E \\ \infty & \text{if } \langle i, j \rangle \notin E \end{cases}$$

# Example Applications

- Suppose a postal van is to be routed such that it picks mails from mail boxes located at  $n$  different sites and return to the head office. An  $n+1$  vertices graph can be used to represent the situation and optimal tour cost can be found
- A production environment where several commodities are manufactured on the same set of machines and the manufacturing proceeds in a cycle. Suppose in each cycle,  $n$  different commodities are produced and when the machines are changed from production of commodity  $i$  to commodity  $j$ , a change over cost  $c_{ij}$  is incurred. The task is to find an optimal order of manufacturing that minimizes the change over cost. It also involves cost of cycle change which is nothing but the change over cost from last commodity to the first commodity

# Brute Force Approach

- We notice that for any n-vertices graph we can have  $n!$  ways of moving from one vertex travelling through all other vertices and returning to the starting vertex
- Thus, generating all possible sequence of vertices will require  $O(n!)$  time
- However, we can recursively define an optimal cost solution and apply dynamic programming to reduce the time complexity



(1,1) (1,2) (1,3) (1,4)...(2,2) (2, 3)...

(2,4,3)/ (2,3,4)/(3,4,2)/(3,2,4).. Min cost

(2,3,4,5) => (2,3,4) + (4,5)

(2,4,3) + (3,5)

(3,4,2) + (2,5)

Min. cost

# Defining a Solution

- Without loss of generality, we assume that an optimal tour is a simple path that begins and ends at vertex 1
- Every tour will consist of an edge  $\langle 1, k \rangle$  for some  $k \in V - \{1\}$  and every vertex in  $V - \{1, k\}$  exactly once
- We want to find a minimum cost tour and therefore the selected sequence must have minimum cost
- Let us define a function  $g\{i, S\}$  to be the length of a shortest path starting at vertex  $i$ , going through all vertices in  $S$  and terminating at vertex 1
- Then, we can write the length of the optimal tour as  $g(1, V-\{1\})$

- We can thus write  $g(1, V - \{1\})$  as:

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{c_{1,k} + g(k, V - \{1, k\})\}$$

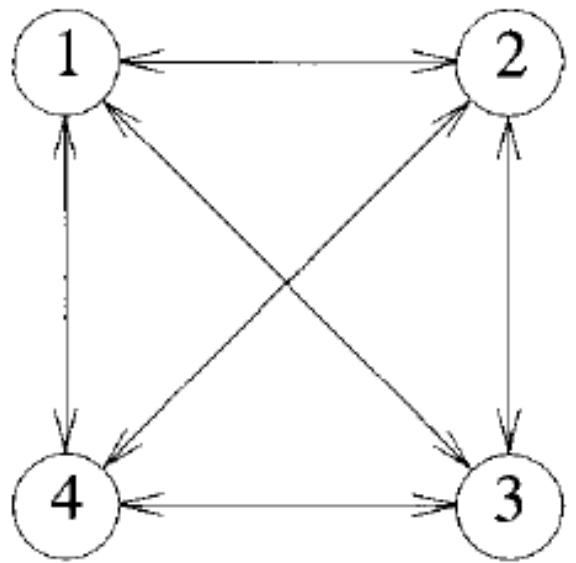
- We notice that in order to solve the above equation optimally, we must solve  $g(k, V - \{1, k\})$  optimally
- This is because if it were not true then we could find a minimum cost tour with same value of  $c_{1,k}$  resulting into an even lower cost tour for  $g(1, V - \{1\})$  hence, contradicting our assumption that it gives minimum cost tour
- Thus, this problem exhibits **optimal substructure**
- Further, each  $g(k, V - \{1, k\})$  has many subproblems in common for different values of  $k$  and this holds for any subproblem generated within. Thus, this problem has many **overlapping subproblems**

# Computing the Value of Optimal Solution

- We can calculate the value of an optimal solution by following a bottom-up approach
- We start by calculating the value of an optimal tour for  $S=0$  and move upward to compute the optimal tour by adding an additional vertex in each pass
- A top-down approach having recursive calling with memoization can also be used to solve the problem
- Here, we discuss the bottom up solution by generalizing the previous equation as below:

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\}$$

We can calculate the  $g(1, V - \{1\})$  if we know the value of  $g(k, V - \{1, k\})$  for all values of  $k$



(a)

0	10	15	20
5	0	9	10
6	13	0	12
8	8	9	0

(b)

# Solution

- For all values  $i$ , we have  $g(i, \phi) = c_{i1}, 1 \leq i \leq n$
- We can use the above values to calculate the  $g(i, S)$  value for  $S=1$  then  $S=2$  and so on

$$g(2, \phi) = c_{21} = 5, g(3, \phi) = c_{31} = 6, \text{ and } g(4, \phi) = c_{41} = 8.$$

$$\begin{array}{lll} g(2, \{3\}) & = & c_{23} + g(3, \phi) = 15 \\ g(3, \{2\}) & = & 18 \\ g(4, \{2\}) & = & 13 \end{array} \quad \begin{array}{lll} g(2, \{4\}) & = & 18 \\ g(3, \{4\}) & = & 20 \\ g(4, \{3\}) & = & 15 \end{array}$$

$$g(2, \{3, 4\}) = \min \{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\} = 25$$

$$g(3, \{2, 4\}) = \min \{c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})\} = 25$$

$$g(4, \{2, 3\}) = \min \{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\} = 23$$

$$\begin{aligned} g(1, \{2, 3, 4\}) &= \min \{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\} \\ &= \min \{35, 40, 43\} \\ &= 35 \end{aligned}$$

# Constructing an Optimal Tour

- For constructing the optimal cost tour, we need to store the value of  $j$  that minimizes the cost value for each  $g(i,S)$
- This  $j$  value can be traced in reverse order in order to find the optimal cost tour. Suppose  $J(,S)$  be this value then we can trace the solution for above problem as
- $J(1,\{2,3,4\}) = 2 \Rightarrow J(2,\{3,4\}) = 4 \Rightarrow J(4,\{3\}) = 3$
- Thus, optimal tour is 1,2,4,3,1

# Complexity Analysis

- Let  $N$  be the number of  $g(i, S)$  that have to be computed before  $g(1, V-\{1\})$  can be computed
- For each value of  $|S|$  there are  $n-1$  choices for  $i$
- The number of distinct sets  $S$  of size  $k$  not including 1 and  $i$   $\binom{n-2}{k}$

$$N = \sum_{k=0}^{n-2} (n-1) \binom{n-2}{k} = (n-1)2^{n-2}$$

- For each  $k$ ,  $(k-1)$  comparisons will also be required for finding the minimum cost
- Thus, the algorithm will require  $\theta(n^2 2^n)$  time for computation of optimal tour
- A serious drawback of the algorithm is the high space complexity  $O(n 2^n)$ . This is too large even for modest values of  $n$

# All Pair Shortest Path

# All Pair Shortest Path

- Given a graph (represented in the form of an adjacency matrix) we want to find the shortest path between every pair of vertices in the graph
- The single source shortest path algorithm can be applied to do this but it has a high time complexity
- Here, we shall see a dynamic programming based solution to solving the shortest path problem

# Structure of Shortest Path

- Let  $l_{ij}^m$  be the minimum weight of any path between i and j containing at most m vertices. Then, for m = 0 we have:

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j , \\ \infty & \text{if } i \neq j . \end{cases}$$

- For  $m \geq 1$ , the minimum weight path can be defined as:

$$\begin{aligned} l_{ij}^{(m)} &= \min \left( l_{ij}^{(m-1)}, \min_{1 \leq k \leq n} \{ l_{ik}^{(m-1)} + w_{kj} \} \right) \\ &= \min_{1 \leq k \leq n} \{ l_{ik}^{(m-1)} + w_{kj} \} . \quad \because w_{jj} = 0 \text{ for all } j \end{aligned}$$

## EXTEND-SHORTEST-PATHS( $L, W$ )

```
1   $n = L.\text{rows}$ 
2  let  $L' = (l'_{ij})$  be a new  $n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4      for  $j = 1$  to  $n$ 
5           $l'_{ij} = \infty$ 
6          for  $k = 1$  to  $n$ 
7               $l'_{ij} = \min(l'_{ij}, l_{ik} + w_{kj})$ 
8  return  $L'$ 
```

- This algorithm takes  $\theta(n^3)$  time
- This calculation needs to be repeated each time by extending the path length one at a time and considering all possible paths for every path length
- We can however, improve reduce the computation time of complete algorithm by utilizing the analogy of this algorithm with matrix multiplication

$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj} .$$

## SQUARE-MATRIX-MULTIPLY( $A, B$ )

```

1   $n = A.\text{rows}$ 
2  let  $C$  be a new  $n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4      for  $j = 1$  to  $n$ 
5           $c_{ij} = 0$ 
6          for  $k = 1$  to  $n$ 
7               $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ 
8  return  $C$ 
```

Observe that if we make the substitutions

$$\begin{aligned} l^{(m-1)} &\rightarrow a, \\ w &\rightarrow b, \\ l^{(m)} &\rightarrow c, \\ \min &\rightarrow +, \\ + &\rightarrow \cdot \end{aligned}$$

and replace  $l'_{ij} = \infty$  by 0

We can use matrix multiplication algorithm for calculation of shortest path which will run in  $\theta(n^4)$  time

## SLOW-ALL-PAIRS-SHORTEST-PATHS( $W$ )

```
1   $n = W.\text{rows}$ 
2   $L^{(1)} = W$ 
3  for  $m = 2$  to  $n - 1$ 
4      let  $L^{(m)}$  be a new  $n \times n$  matrix
5       $L^{(m)} = \text{EXTEND-SHORTEST-PATHS}(L^{(m-1)}, W)$ 
6  return  $L^{(n-1)}$ 
```

- One thing to be noted here is that, we do not need all of the  $L^m$  matrices for each value of  $m$
- We only need the final  $L^{n-1}$  matrix which gives the solution of the problem
- Therefore, we can avoid calculating all the intermediate matrices by using the following property

$$\begin{aligned}
L^{(1)} &= W, \\
L^{(2)} &= W^2 = W \cdot W, \\
L^{(4)} &= W^4 = W^2 \cdot W^2 \\
L^{(8)} &= W^8 = W^4 \cdot W^4, \\
&\vdots \\
L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil}-1} \cdot W^{2^{\lceil \lg(n-1) \rceil}-1}.
\end{aligned}$$

Since  $2^{\lceil \lg(n-1) \rceil} \geq n - 1$ , the final product  $L^{(2^{\lceil \lg(n-1) \rceil})}$  is equal to  $L^{(n-1)}$ .

## FASTER-ALL-PAIRS-SHORTEST-PATHS( $W$ )

```
1   $n = W.\text{rows}$ 
2   $L^{(1)} = W$ 
3   $m = 1$ 
4  while  $m < n - 1$ 
5      let  $L^{(2m)}$  be a new  $n \times n$  matrix
6       $L^{(2m)} = \text{EXTEND-SHORTEST-PATHS}(L^{(m)}, L^{(m)})$ 
7       $m = 2m$ 
8  return  $L^{(m)}$ 
```

- This algorithm runs in  $O(V^2 \log V)$  because each of the  $\lceil \log n - 1 \rceil$  products take  $\theta(n^3)$  time

# Greedy Technique

Activity Selection, Knapsack, Huffman Coding

# Introduction

- Applied on optimization problems
- Greedy algorithm always makes a choice that looks best at the moment
- That is, a locally optimal choice is made in the hope of achieving a globally optimal one
- Making local choice does not always yield an optimal solution but as we shall see in this section, for many problems it does
- Whenever applicable it drastically reduces the number of subproblems required to be solved at each level thus, reducing the computation time manifold
- Applied in many graph algorithms as well such as Dijkstra's algorithm for single source shortest path and calculation of minimum spanning tree

# Proving that Greedy strategy works

- Greedy problem solving requires two properties to be satisfied by the original problem
  - Optimal substructure
  - Greedy choice property
- For proving that a greedy algorithm works, we need to show that making a locally optimal choice yields a global optimal
- This is usually done by assuming a globally optimal choice and showing that it can be replaced by some greedy choice

# Activity-Selection Problem

- Suppose, we have a set  $S = \{a_1, a_2, \dots, a_n\}$  of  $n$  proposed activities that wish to use a resource that can serve only one activity at a time (for example a set of events to be organized in a single lecture hall)
- Each activity has a start time  $s_i$  and a finish time  $f_i$ , where  $0 \leq s_i < f_i < \infty$
- If an activity  $a_i$  is selected for using the resource, it takes place during the half-open interval  $[s_i, f_i)$
- Two activities  $a_i$  and  $a_j$  are said to be ***compatible*** if the intervals  $[s_i, f_i)$  and  $[s_j, f_j)$  do not overlap that is, if  $s_i \geq f_j$  or  $s_j \geq f_i$
- In **activity-selection problem**, a maximum-size subset of mutually compatible activities

# Example

$i$	1	2	3	4	5	6	7	8	9	10	11
$s_i$	1	3	0	5	3	5	6	8	8	2	12
$f_i$	4	5	6	7	9	9	10	11	12	14	16

- $\{a_3, a_9, a_{11}\}$  is a mutually compatible set of activities
- $\{a_1, a_4, a_8, a_{11}\}$
- $\{a_2, a_4, a_9, a_{11}\}$
- Above are two largest mutually compatible set of activities

# Solution

- We can use a dynamic programming approach for solving this problem
- Suppose, the initial optimal choice is at some point ‘k’ so that the activity  $a_k$  belongs to the optimal set of activities
- We will define the optimal solution recursively using this as follows:
- Let us define  $S_{ij}$  to be the set of activities that start after  $a_i$  finishes and finish before  $a_j$  starts
- Let us further define a set  $A_{ij}$  that consists of the largest number of mutually compatible activities in  $S_{ij}$

- Let activity  $a_k$  belongs to the maximal set  $A_{ij}$  of  $S_{ij}$
- We can describe the remaining selection of activities into two subproblems
- One is the finding mutually compatible activities in the set  $S_{ik}$  and the other is finding mutually compatible activities in the set  $S_{kj}$
- Now, the two subproblems must be solved optimally in order for the original problem to be solved optimally (optimal substructure holds)
- We can define  $A_{ik}$  as -  $A_{ik} = A_{ij} \cap S_{ik}$
- Similarly,  $A_{kj}$  can be defined as -  $A_{kj} = A_{ij} \cap S_{kj}$
- $A_{ij}$  can be defined as the union of three mutually exclusive sets:  

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$
- The optimal substructure thus yields the following recurrence:  

$$c[i, j] = c[i, k] + c[k, j] + 1$$

- The above steps will be repeated for all values of k
- Therefore, the optimal solution can be given as:

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \phi \\ \max_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \phi \end{cases}$$

- Either a top-down approach with memoization can be used for solving this or a bottom up approach can be used by filling the table entries

# Greedy Approach

- It turns out that the activity selection problem has the interesting property that a greedy choice can be made at each step thus reducing the number of subproblems to be solved to just 1
- The greedy choice made at each step is dependent on some criteria that gives optimal solution
- For the activity selection problem, greedy choice could be to select the activity that finishes earliest as it would leave maximum space to accommodate more activities
- We assume that the input is sorted in the order of finishing time and therefore the activity with smallest finishing time appears in the front, thus we will choose activity  $a_1$

- If we make the greedy choice at the first step, we don't need to look for all possible values of 'k', thus reducing the number of subproblems that need be solved
- We are left with only one subproblem to be solved  $S_k$  – making a greedy choice among activities that start after activity  $a_1$  finishes
- Since the problem has optimal substructure, making a greedy choice among the remaining activities yields an optimal global solution
- We are making a greedy choice at each step and are left with only one subproblem to solve
- Greedy algorithms are therefore, solved in top-down manner
- We can initially write a recursive algorithm to solve the problem and later re-write it as an iterative algorithm

# Proof that the Greedy choice works

- Let us consider a subproblem  $S_k$  and let  $a_m$  be any activity in  $S_k$  with the earliest finish time. Then, we will prove that  $a_m$  is included in some maximal subset of mutually compatible activities in  $S_k$
- Let  $A_k$  be an optimal subset of mutually compatible activities in  $S_k$  and let  $a_j$  be the first activity to finish in  $A_k$ . Then, we have the following:
  - Case 1:  $a_j = a_m$  then, we have already proved that  $a_m$  is included in  $A_k$

- Case 2:  $a_j \neq a_m$
- Let us define another set of activities  $A'_k = A_k - \{a_j\} \cup \{a_m\}$
- We claim that  $A'_k$  defined above is disjoint. This is because  $A_k$  is disjoint and  $a_m$  is the first activity to finish in  $S_k$ . Therefore,  $f_m \leq f_j$
- Further,  $|A'_k| = |A_k|$  therefore,  $A'_k$  is also a maximal subset of mutually compatible activities and hence a solution.
- Thus, we have proved that making a greedy choice at each step always yields an optimal solution for the activity selection problem

# Recursive Greedy Algorithm for Activity-Selection Problem

RECURSIVE-ACTIVITY-SELECTOR( $s, f, k, n$ )

- 1  $m = k + 1$
- 2 **while**  $m \leq n$  and  $s[m] < f[k]$  // find the first activity in  $S_k$  to finish
- 3      $m = m + 1$
- 4 **if**  $m \leq n$
- 5         **return**  $\{a_m\} \cup$  RECURSIVE-ACTIVITY-SELECTOR( $s, f, m, n$ )
- 6 **else return**  $\emptyset$

# Iterative Greedy Algorithm

**GREEDY-ACTIVITY-SELECTOR( $s, f$ )**

```
1   $n = s.length$ 
2   $A = \{a_1\}$ 
3   $k = 1$ 
4  for  $m = 2$  to  $n$ 
5      if  $s[m] \geq f[k]$ 
6           $A = A \cup \{a_m\}$ 
7           $k = m$ 
8  return  $A$ 
```

- Both the algorithms run in  $\theta(n)$  time

# Not all greedy choices work

- The greedy choice should be made carefully so as to obtain the global optimum by making greedy choices. For example, the following greedy choices for the activity selection problem do not yield an optimal solution
- Selection of activities with least duration
- Selection of compatible activities with earliest start time
- Selecting activities that overlap with fewest remaining activities

# When to apply Greedy Algorithm

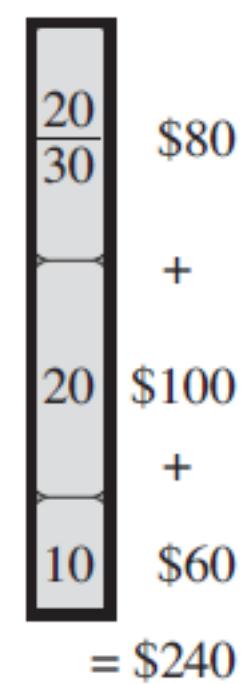
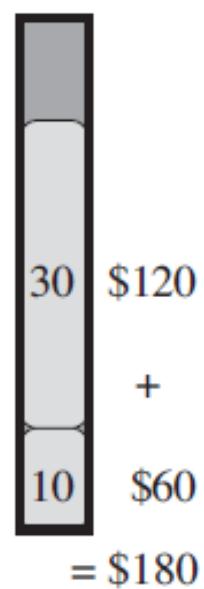
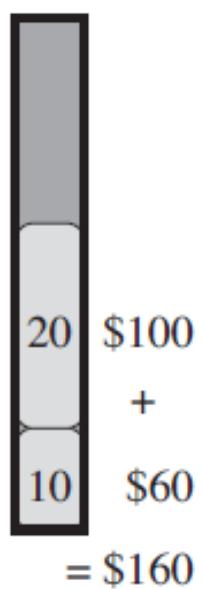
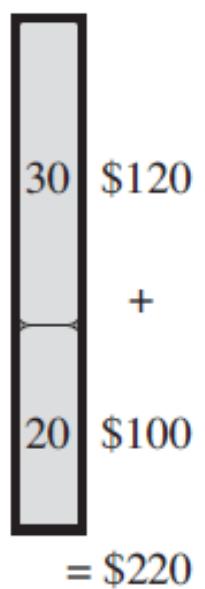
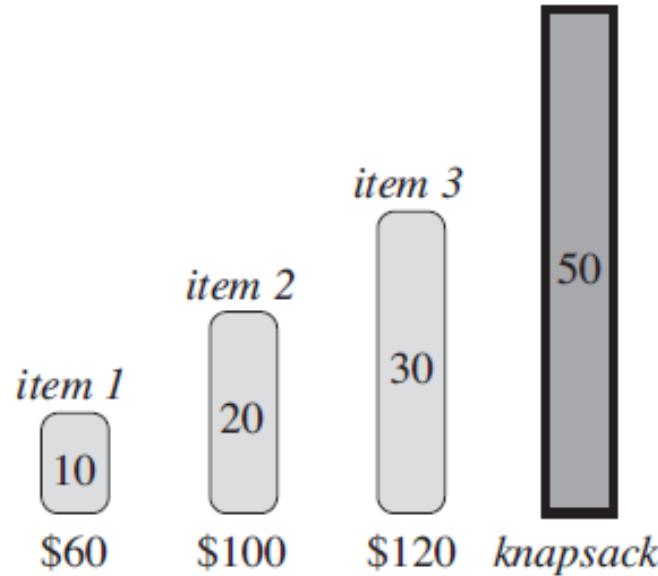
- Optimization Problem
- Greedy choice property – making a greedy choice at each step, yields a global optimum
- Optimal Sub-structure
- We usually prove point no. 2 in the same manner we proved it for activity-selection problem i.e. by assuming a global optimal solution and then proving that a random element of it can be replaced by a greedy choice made at some level and hence, making greedy choices every time will lead to an optimal solution
- Greedy choices can be made quickly if we process the data or use an appropriate data structure. E.g. sorting the activities in decreasing order of finish time causes making of greedy choice much easier

# Knapsack Problem

- A thief robbing a store finds  $n$  items. The  $i$ th item costs  $v_i$  dollars and weighs  $w_i$  pounds, where  $v_i$  and  $w_i$  are integers. The thief wants to take as valuable a load as possible, but he can carry at most  $W$  pounds in his knapsack for some integer  $W$ . Which items the thief should take?
- The above problem has two variants:
  - 0/1 Knapsack – each item can either be taken completely or not taken at all and the thief can take at most one item of one type
  - Fractional Knapsack – the thief can take fractions of an item rather than always making a binary (0-1) choice.

# Fractional Knapsack

- The fractional knapsack problem satisfies the greedy choice property and can be solved by making a greedy choice at each step.
- The greedy selection is made on the item that gives the largest cost/weight
- After such an item is totally exhausted, the thief can select the next item with largest cost/weight.
- The process is repeated until the total weight of selected items becomes equal to  $W$



# 0-1 Knapsack

- This problem does not obey the greedy choice property
- The 0-1 Knapsack problem is solved using dynamic programming
- The recursive equation is written by assuming that an item ‘k’ is an optimal choice for the knapsack then we are left with one subproblem in which select one item from remaining (n-1) items and add the cost of the item to the total cost of knapsack
- Since we do not know the value of k, this process is repeated for all values of k

$$C_n = \max_{k \in n} \{c_k + C_{n-1}\}$$

# Huffman Codes

- Used for data compression
- Causes around 20% to 90% saving
- For compression, a binary character code is used where each character is encoded with a sequence of unique binary numbers
- Either a fixed length or variable length code can be used
- Variable-length codes are much more efficient than fixed-length codes
- Huffman codes is also a variable length code
- The idea is to use smaller codes for the characters that occur more frequently in data while longer codes are used for less frequent characters

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Total number of bits required for encoding:

Case 1: Fixed-length codeword =  $1,00,000 \times 3 = 3,00,000$  bits

Case 2: Variable-length codeword =  $45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4 = 2,24,000$  bits

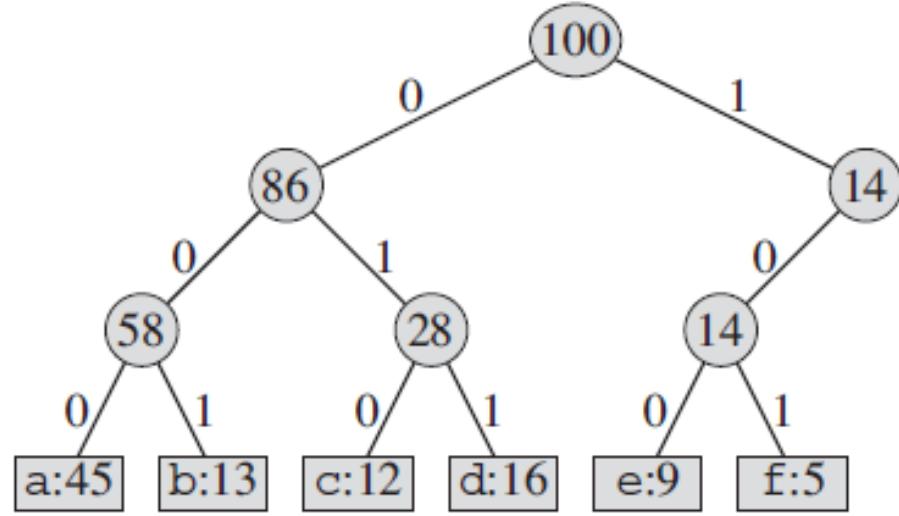
Total savings =  $(76,000 / 3,00,000) \times 100 \approx 25\%$

# Prefix Codes

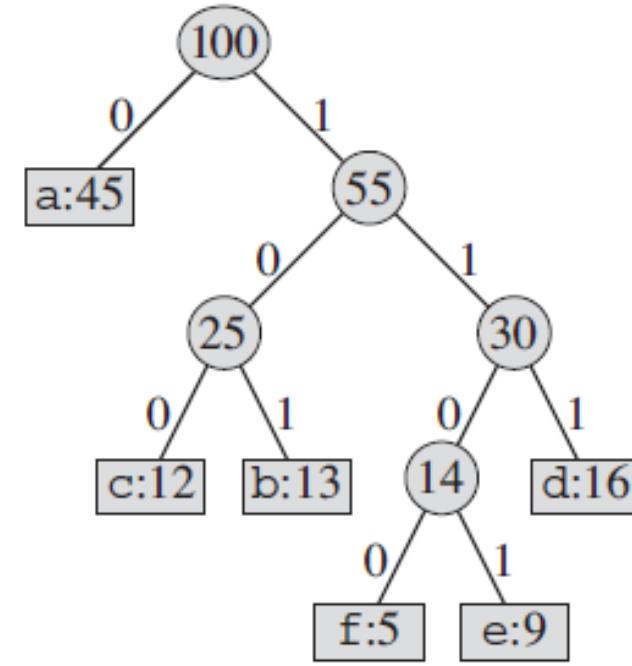
- Prefix codes are the ones in which no code is a prefix of other
- Prefix codes always yield optimal data compression
- The document is encoded by concatenating the prefix codes of the characters
- Prefix codes also simplify decoding, since no code is a prefix of other, the decoding is unambiguous
- For decoding, we identify the initial codeword, translate it back to the original character and continue with the next codeword

# Decoding Process

- To simplify the decoding process, it is convenient to represent the codes so that the initial codeword can be picked easily
- The codes are usually represented as the nodes of a binary tree
- The leaves of the binary tree represent the characters
- The code for a character is then defined as a simple path from the root to the leaf
- An optimal code for a file is always represented in the form of a full binary tree (i.e. every non-leaf node has exactly two children)
- While traversing the tree, a ‘0’ means “go to left” and ‘1’ means “go to right”



Fixed-length code => not optimal



Variable-length code => full binary tree => optimal

- For any file having  $C$  alphabets, let the frequency of each character  $c$  in the file be  $c.freq$  and  $d_T(c)$  the depth of  $c$ 's leaf in the tree (thus  $d_T(c)$  also denotes the number of bits required to code  $c$ )
- Then, the number of bits required to encode the complete file is:

$$B(T) = \sum_{c \in C} c.freq \times d_T(c)$$

Above is defined as the cost of the tree

# Huffman Code

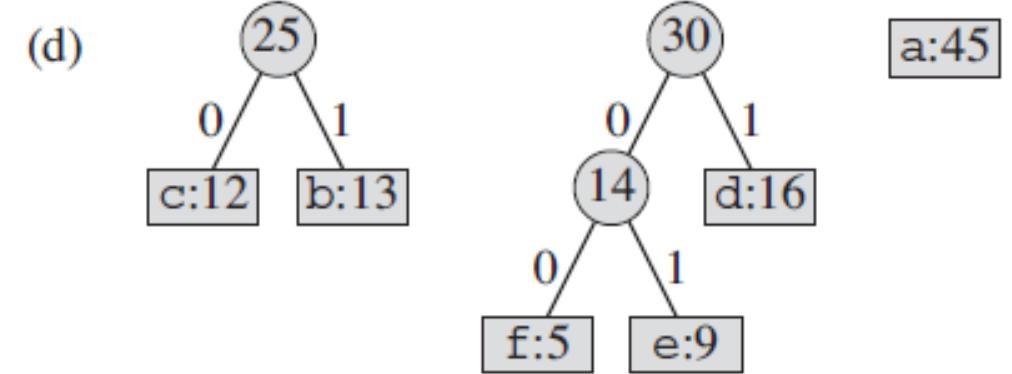
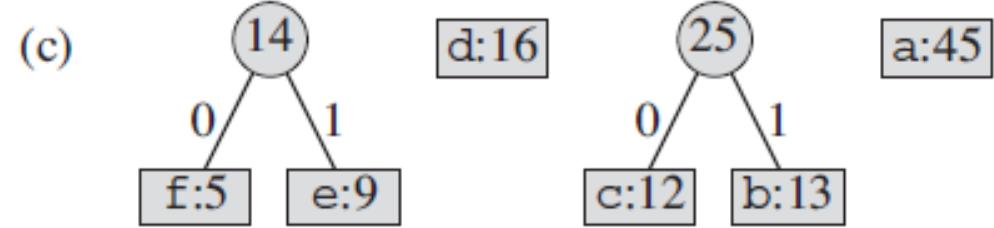
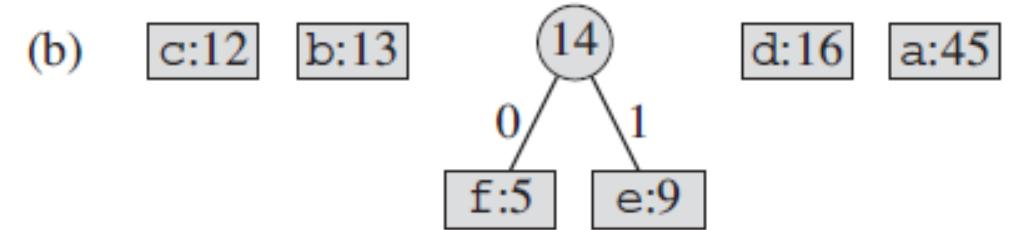
- Huffman invented a greedy algorithm for generating optimal prefix code also known as Huffman code
- For any optimal prefix code on alphabet C, the full binary tree representation has exactly  $|C|$  leaf nodes and exactly  $|C|-1$  internal nodes
- The Huffman coding is based on identifying the characters with smallest frequency and merging them to form an internal node of the tree
- The process is repeated until all the characters are merged into a single tree

# Huffman Code - Recursive Algorithm

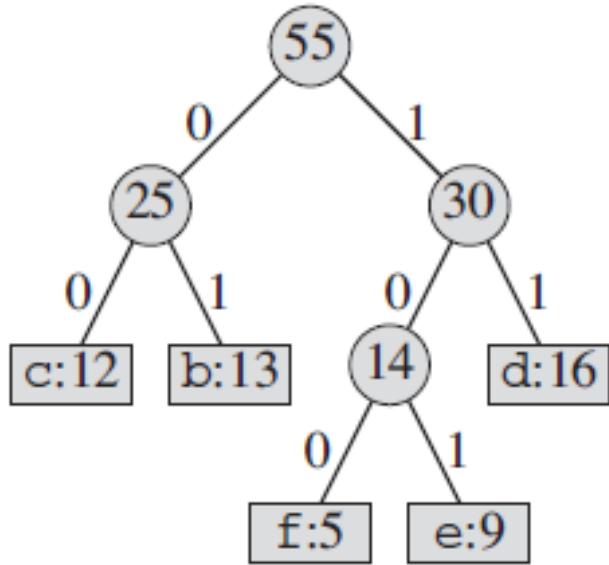
**HUFFMAN( $C$ )**

```
1   $n = |C|$ 
2   $Q = C$ 
3  for  $i = 1$  to  $n - 1$ 
4      allocate a new node  $z$ 
5       $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6       $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7       $z.freq = x.freq + y.freq$ 
8       $\text{INSERT}(Q, z)$ 
9  return  $\text{EXTRACT-MIN}(Q)$     // return the root of the tree
```

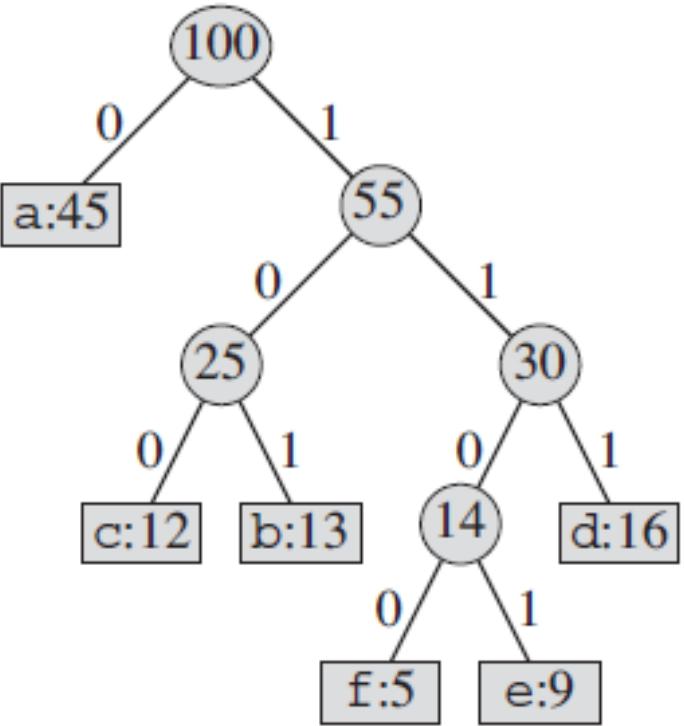
(a) f:5 e:9 c:12 b:13 d:16 a:45



(e) [a:45]



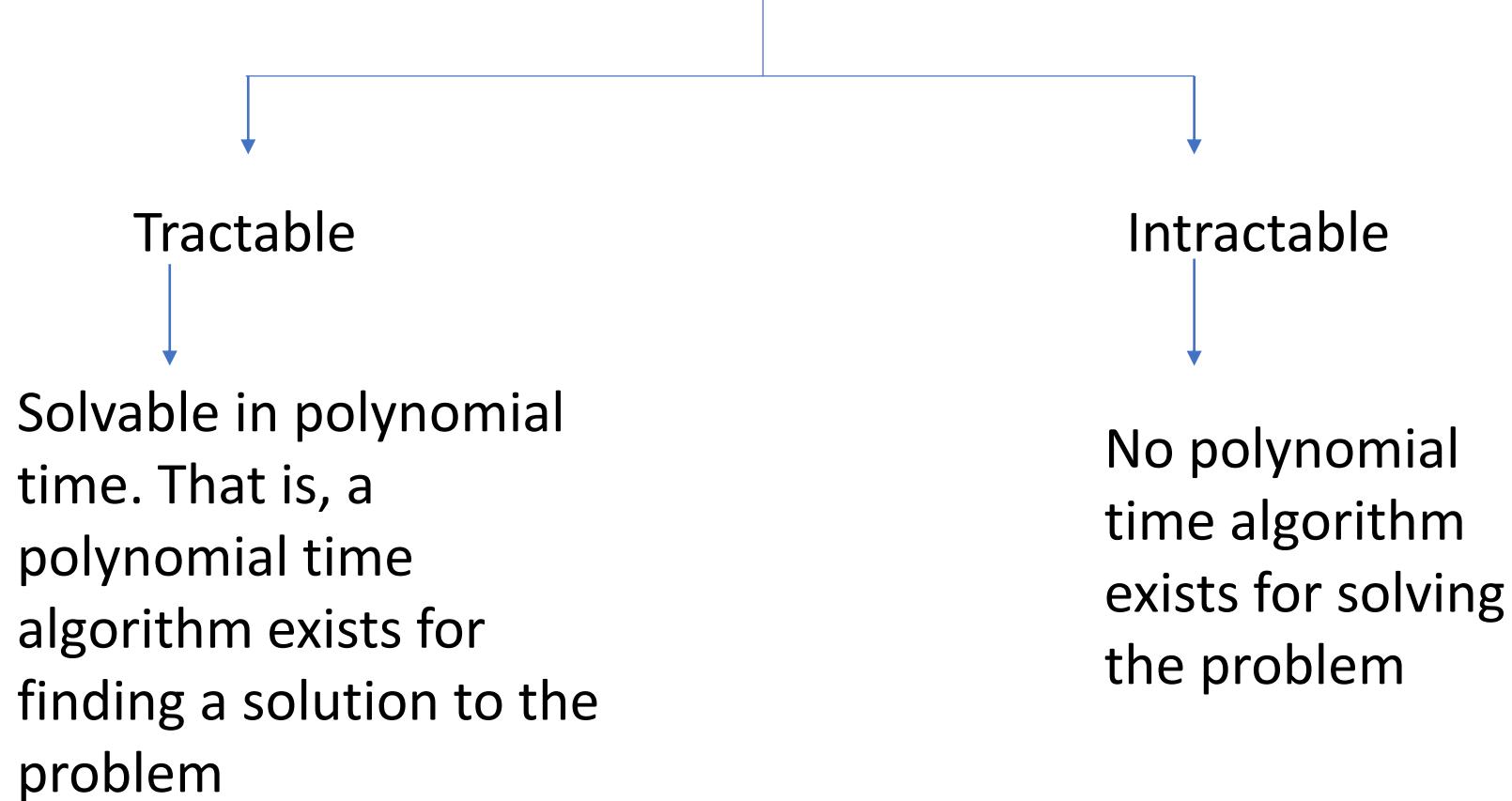
(f)



# Computational Complexity

Problem classes, P, NP, NP-Complete, NP-Hard, Reduction

## Problem classes



# Polynomial Time

- Problems having polynomial time algorithms are called tractable
- In practice, a polynomial of high order will be as bad as a non-polynomial-time algorithm but we keep with this idea for the following reasons:
  - For most practical problems that we encounter that have a polynomial time algorithm, the polynomial is found to be of lower order only
  - If a higher order polynomial algorithm is discovered for a problem, lower order ones follow. Therefore, if a problem exists for which we have a higher order polynomial solution it is highly likely that we will discover a lower order one
  - Polynomial class problems have the nice closure property i.e. if the output of one polynomial-time algorithm is fed to other the resulting algorithm will also be polynomial class

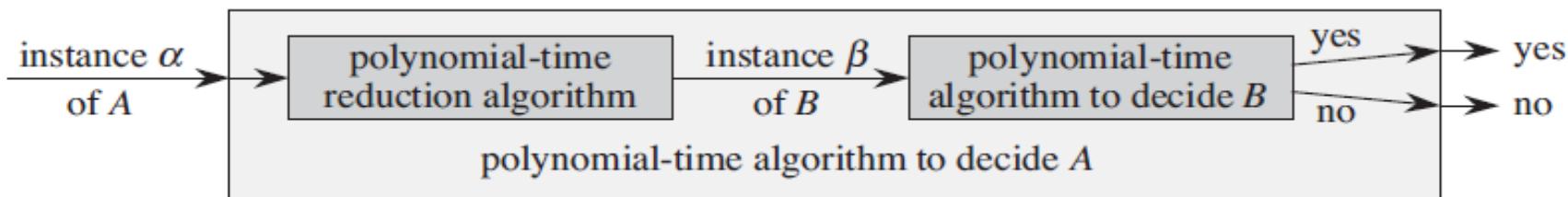
# Decision Problems v/s Optimization Problems

- An important class of problems that we will consider are NP-complete problems
- NP-complete problems are confined to the realm of Decision problems
- However, optimization problems can be recast into decision problem
- For example, the problem of finding shortest path between two given vertices in a graph can be recast into a decision problem by formulating a new problem as: given a graph  $G$ , two vertices  $u$  and  $v$  and an integer  $k$ , does there exist a path between  $u$  and  $v$  having at most  $k$  edges?
- The corresponding decision problem is no harder than the optimization problem so that if optimization problem can be solved in polynomial time, the decision problem is also solvable in polynomial time. Conversely, if we prove that the decision problem is hard then it gives sufficient evidence that the optimization problem is hard

# Reduction

- The recasting of problems can also be used for two different problems
- This technique is useful in showing that one problem is no easier than the other or it is at least as hard
- In reduction, we use a decision problem we know about. Suppose, we know how to solve a decision problem B in polynomial time. Given, a decision problem A, we device a transformation function that converts every instance  $\alpha$  of A into an instance  $\beta$  of B with the following characteristics:

- The transformation takes polynomial time
- If the solution for instance  $\alpha$  in A is yes then it is also yes for instance  $\beta$  in B and vice-versa
- We call such a procedure as *polynomial-time reduction algorithm*
- For solving A we do the following:
  - Use the polynomial time reduction algorithm to convert the given instance  $\alpha$  of A to instance  $\beta$  of B
  - Use the polynomial time algorithm of B to solve the instance  $\beta$
  - Use the answer of  $\beta$  as answer of  $\alpha$



# Proving NP-Completeness

- We can use the above procedure in opposite-way to show how hard a problem is
- Suppose, there is a problem A that we know that no polynomial time algorithm exists for and let there be a new problem B that can be reduced in polynomial time to problem A
- Then, we can simply prove by contradiction that no polynomial time algorithm exists for problem B because if it were not so then we could use the reduction algorithm to solve the problem A in polynomial time
- For proving NP-completeness, we need to have a “first” NP-complete problem that can be used to prove NP-completeness of other problems through reduction though for NP-complete problems we might not be able to prove that no polynomial time algorithm exists

# Abstract Problems

- An abstract problem  $Q$  is defined as a binary relation on a set  $I$  of problem *instances* and a set  $S$  of problem *solutions*
- For shortest path problem, an instance will be a triplet of a graph and two vertices and solution will be a sequence of zero or more vertices giving the shortest path between  $u$  and  $v$
- For NP-completeness, we need decision problems and therefore the solution set for such problems is  $\{0,1\}$
- For the decision problem corresponding to shortest path problem, instance is  $i = \langle G, u, v, k \rangle$  and solution set is  $\{0,1\}$
- Most optimization problems can be recast into a decision problem that is no harder to solve

# Encoding

- If a problem is to be solved using a computer, it needs to be converted in such a way that the computer can understand it
- The process of converting an abstract problem to a computer understandable representation is known as *Encoding*
- An encoding of a set  $S$  of abstract objects is a mapping  $e$  of the set  $S$  to a set of binary strings
- Like a number or a character that can be encoded as a binary string, complex objects such as graphs, polygons, functions etc. all can be encoded as binary strings
- Thus, a computer program solving a given problem actually takes as input a binary encoding of a problem instance

# Concrete Problem

- A problem whose instance set is the set of binary strings is known as *Concrete Problem*
- Suppose there is an instance  $i$  of a concrete problem
- We say that an algorithm solves the problem in  $O(T(n))$  if for the given instance  $i$  of length  $|i| = n$ , the algorithm is able to produce the solution in  $O(T(n))$
- Thus, a concrete problem is polynomial-time solvable if there exists an algorithm that solves the problem in  $O(n^k)$  for some constant  $k$

# Problem Conversion

- We can convert an abstract decision problem to a concrete problem using encoding
- Given an abstract decision problem  $Q$  that maps an instance set  $I$  to  $\{0,1\}$ , we can define an encoding  $e: I \rightarrow \{0,1\}^*$
- We denote this encoding by  $e(Q)$
- If the solution of the abstract decision problem on some instance  $i \in I$  is  $Q(i) \in \{0,1\}$  then the solution to the concrete-problem instance  $e(i) \in \{0,1\}^*$  is also  $Q(i)$
- Some binary strings in this set might represent no meaningful problem instance, we assume that all such strings arbitrarily map to 0

# Polynomial Time Solvability

- The definition of polynomial time solvability of a problem can be extended to abstract problems
- However, we would like the polynomial time solvability to be independent of the encoding i.e. whether a problem can be solved in polynomial time should not depend on the encoding used
- In practice, however it does depend heavily on the particular encoding used
- Nevertheless, we assume that we use optimal encodings so that whether an algorithm is polynomial-time solvable is independent of the encoding used

# Formal-Language Framework

- An alphabet  $\Sigma$  is a finite set of symbols
- A language  $L$  over  $\Sigma$  is any set of strings made up of symbols from  $\Sigma$
- E.g.  $\Sigma = \{0,1\}, L = \{10, 11, 101, 111, 1011, 1101, 10001, \dots\}$
- $L$  is the language of binary representation of prime numbers
- Empty string :  $\varepsilon$  and empty language:  $\emptyset$
- set of all strings over  $\Sigma$  is denoted by  $\Sigma^*$
- E.g.  $\Sigma = \{0,1\}, \Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$
- Every language  $L$  over  $\Sigma$  is a subset of  $\Sigma^*$

# Set Operations

- All set theoretic operations are defined on the set of languages
- Union, Intersection, Complement, Concatenation, Closure or Kleene star
- Complement of  $L$  is given by  $\bar{L} = \Sigma^* - L$
- Concatenation of two languages  $L_1$  and  $L_2$  is the language  $L$  given by:
- $L = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \dots$  where  $L^k$  is the language obtained by concatenating  $L$  to itself  $k$  times

# Formal languages and Complexity classes

- A decision problem  $Q$  can be viewed as simply the set  $\Sigma^*$  where  $\Sigma = \{0,1\}$
- Since,  $Q$  is entirely characterized by the set of strings in  $\Sigma^*$  that produce a 1 or (yes) as answer,  $Q$  can be defined as a language over  $\Sigma = \{0,1\}$ , where
- $L = \{x \in \Sigma^*: Q(x) = 1\}$
- This framework can be used to describe the relation between decision problems and algorithms used for solving them

# Acceptance

- An algorithm A is said to accept a string  $x \in \{0,1\}^*$  if, given input  $x$ , the algorithm's output  $A(x)$  is 1.
- Thus, the language accepted by an algorithm A is the set of strings  $L = \{x \in \Sigma^*: A(x) = 1\}$  that is, the set of strings that algorithm accepts
- An algorithm A rejects a string if  $A(x) = 0$
- An algorithm A that accepts a string  $x \in L$  not necessarily rejects a string  $x \notin L$  i.e. it might loop forever

# Decidability

- A language  $L$  is decided by an algorithm  $A$  if every binary string in  $L$  is accepted by  $A$  and every binary string not in  $L$  is rejected by  $A$
- A language  $L$  is ***accepted in polynomial time*** by an algorithm  $A$  if it is accepted by  $A$  and if in addition there exists a constant  $k$  such that for any length- $n$  string  $x \in L$ , algorithm  $A$  accepts  $x$  in time  $O(n^k)$
- A language  $L$  is ***decided in polynomial time*** by an algorithm  $A$  if there exists a constant  $k$  such that for any length- $n$  string  $x \in \{0,1\}^*$ , the algorithm correctly decides whether  $x \in L$  in time  $O(n^k)$
- For deciding a language  $L$  the algorithm must accept or reject every string in  $\Sigma^*$  while for accepting the language  $L$ , the algorithm only needs to produce an answer for strings in  $L$
- For many problems such as the decision problem corresponding to shortest path, an algorithm exists that can decide the path
- But for other problems, such as the Turing's Halting problem there exists an accepting algorithm but no decision algorithm exists

# Complexity Class

- A complexity class can be informally defined as a set of languages, membership in which is determined by a complexity measure, such as running time, of an algorithm that determines whether a given string  $x$  belongs to language  $L$
- We can define the complexity class P as:
- $P = \{L \subseteq \{0,1\}^*: \text{there exists an algorithm } A \text{ that decides } L \text{ in polynomial time}\}$
- Thus, P is also the class of problems that is accepted in polynomial time

# Polynomial Time Verification

- Suppose, we are given an instance  $\langle G, u, v, k \rangle$  of the decision problem corresponding to the shortest path problem.
- Suppose, we are further given an actual path  $p$  from  $u$  to  $v$
- Then, we can easily verify whether or not  $p$  belongs to the shortest path between  $u$  and  $v$  with at most  $k$  edges
- We can view  $p$  as a certificate and we are interested in finding the correctness of the certificate. This process is known as *Verification*
- Clearly, given a certificate, it is much easier to verify the certificate instead of solving the actual problem

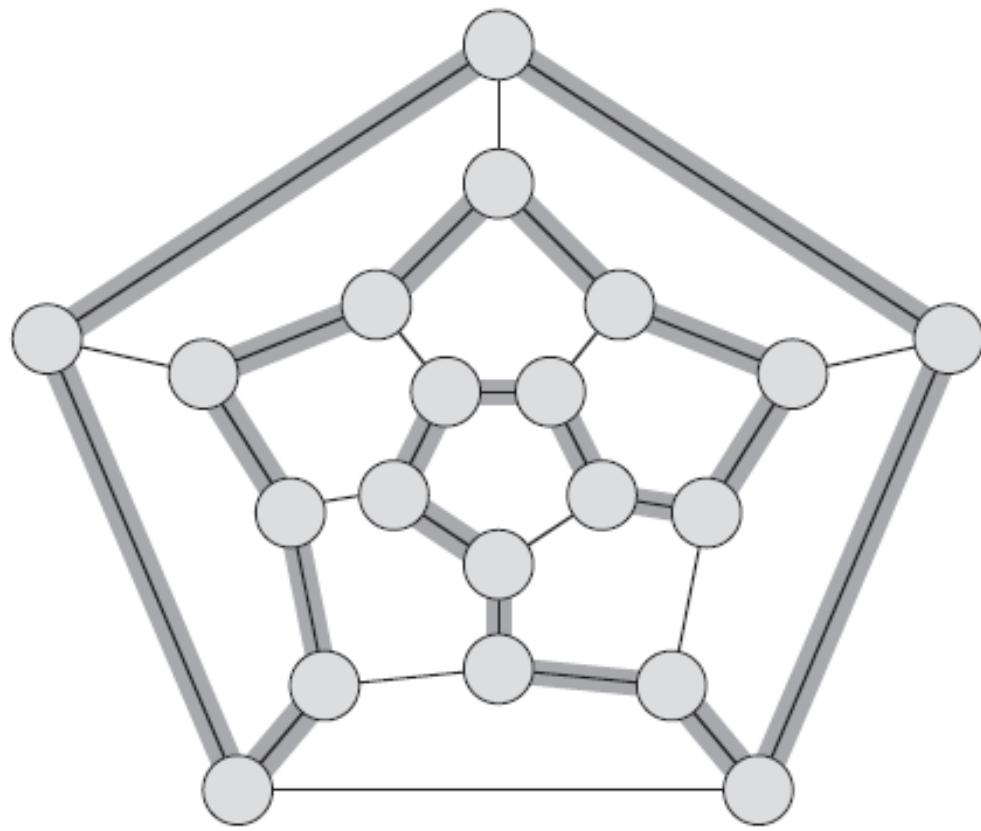
# Verification Algorithms

- It is defined as a two argument algorithm  $A$ , where one argument is an ordinary input string  $x$  and the other is a binary string  $y$  called a certificate.
- The algorithm verifies the input string such that  $A(x, y) = 1$
- The language verified by a verification algorithm  $A$  is:
- $L = \{x \in \{0,1\}^*: \text{there exists } y \in \{0,1\}^* \text{ such that } A(x, y) = 1\}$
- Thus,  $A$  verifies a language  $L$  if for any string  $x \in L$ , there exists a certificate  $y$  that  $A$  can use to prove that  $x \in L$ . Moreover, for any string  $x \notin L$ , there must be no certificate proving that  $x \in L$ .

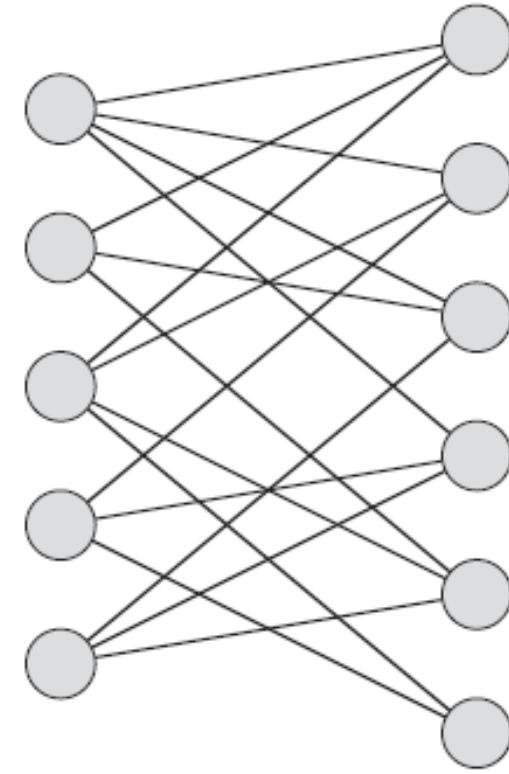
# Hamiltonian Circuit

- a ***hamiltonian cycle*** of an undirected graph  $G = (V, E)$  is a simple cycle that contains each vertex in  $V$ . A graph that contains a hamiltonian cycle is said to be ***hamiltonian***; otherwise, it is ***nonhamiltonian***.
- Hamiltonian Circuit problem: Given a graph, does it have a hamiltonian cycle?
- Suppose the graph  $G$  has  $m$  vertices and an encoding of length  $n = |<G>|$
- There are  $m!$  possible permutations of the vertices and therefore the lower bound on running time can be given by:
- $\Omega(m!) = \Omega(\sqrt{n!}) = \Omega(2^{\sqrt{n}})$  which is not  $O(n^k)$  for any constant  $k$

- For the corresponding verification problem, suppose you are told that a graph is hamiltonian and an arrangement of vertices that gives a hamiltonian cycle for the graph
- Then, it is easy to verify whether or not the arrangement actually gives a hamiltonian circuit, we will simply check for each consecutive set of vertices whether an edge exists and whether the complete arrangement actually creates a cycle in the graph
- This algorithm can be implemented in  $O(n^2)$  where  $n$  is the length of encoding
- Thus, the verification can be performed in polynomial time
- In the Hamiltonian cycle problem, the certificate  $y$  is the given cycle arrangement and it provides sufficient information to verify that the encoding  $x$  of the graph belongs to  $L$ . Conversely, if a graph is not Hamiltonian then, there is no possible arrangement of vertices that can fool the algorithm into believing that  $x \in L$ .



(a)

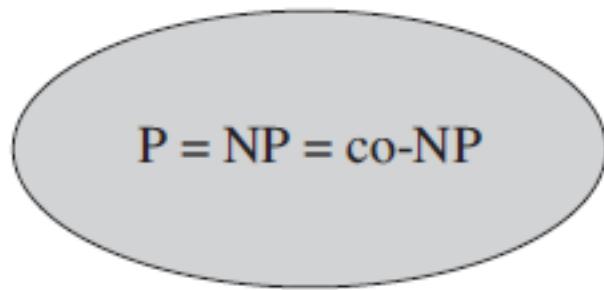


(b)

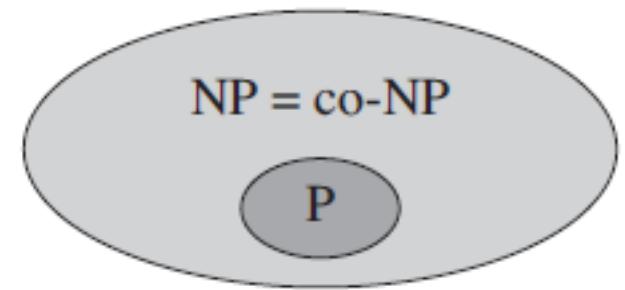
# The Complexity Class NP

- The complexity class NP is the class of languages that can be verified by a polynomial-time algorithm
- More precisely, a language L belongs to NP if and only if there exist a two-input polynomial-time algorithm A and a constant c such that
- $L = \{x \in \{0,1\}^*: \text{there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x, y) = 1\}$
- We say that A *verifies* language L *in polynomial time*

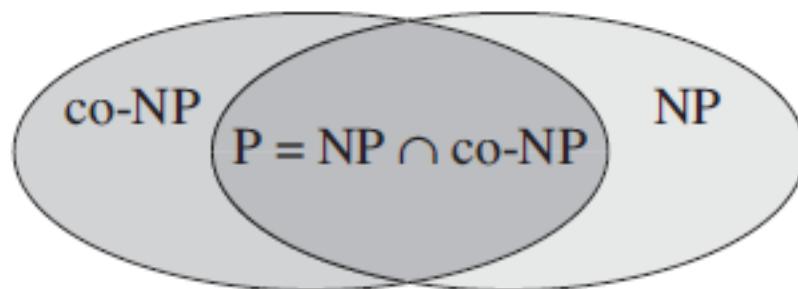
- Clearly, Hamiltonian cycle problem belongs to class NP
- Further, if a language  $L \in P$  then,  $L \in NP$
- This implies that  $P \subseteq NP$
- However, it is unknown whether  $P = NP$
- The most compelling reason to believe that  $P \neq NP$  is the existence of the class of languages that are NP-complete
- It is also not known whether the class NP is closed under complementation i.e. does  $L \in NP$  implies  $\bar{L} \notin NP$ . In other words whether  $NP = co - NP$
- Our understanding of the relationship between P and NP is very incomplete but we can at least prove that a particular problem is intractable, if we can prove that it is NP-complete and this gives valuable information about the problem



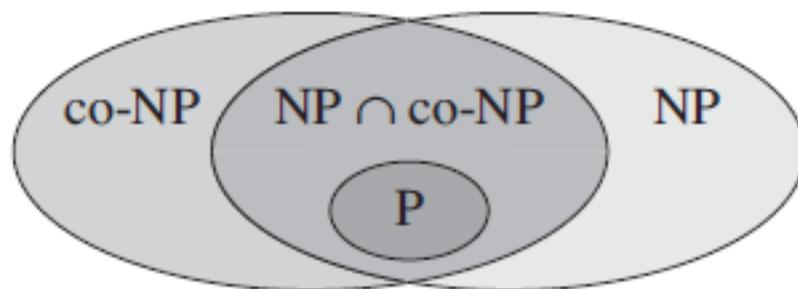
(a)



(b)



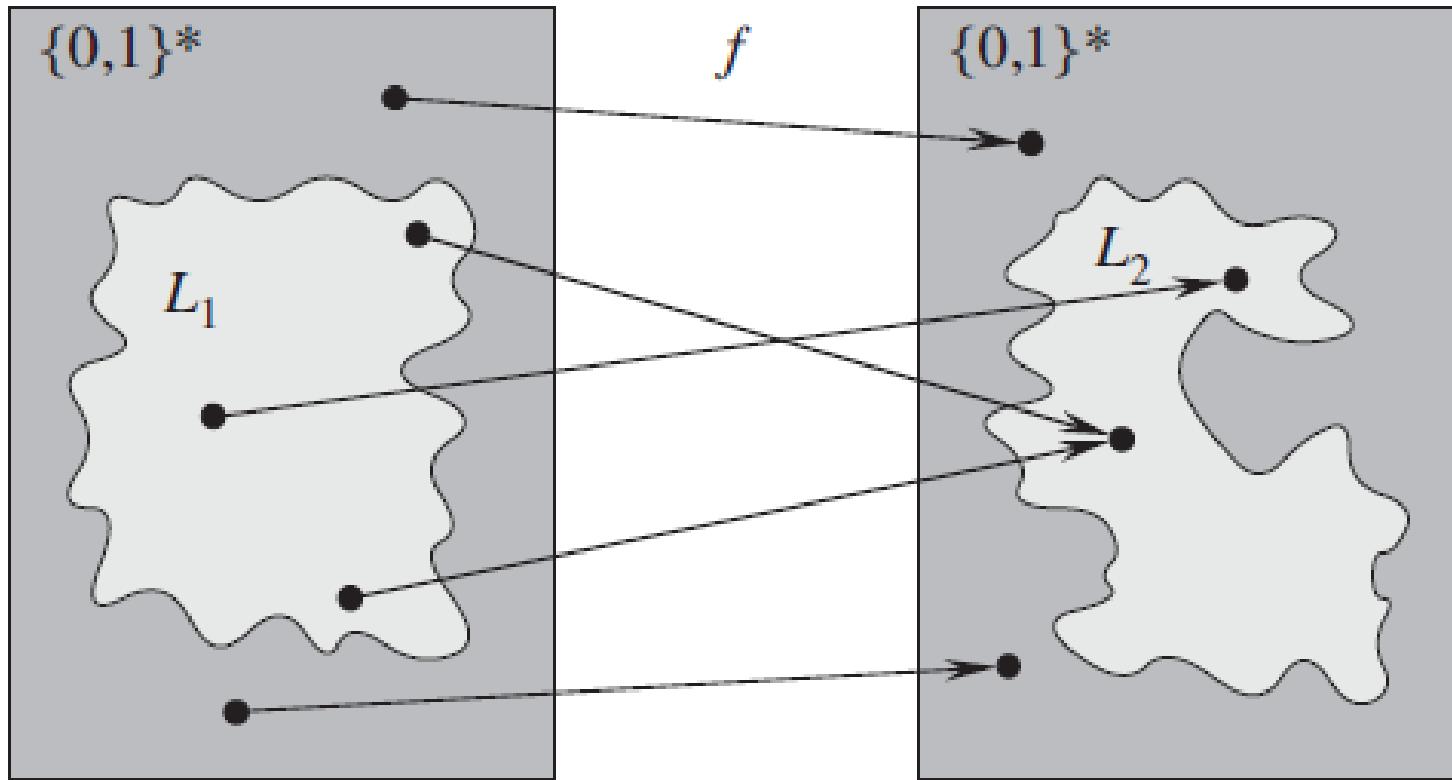
(c)



(d)

# NP-Completeness and Reducibility

- The existence of NP-complete problems is the most compelling reason for believing that  $P \neq NP$
- NP-complete problems have the special property that *if any* NP-complete problem can be solved in polynomial time then *every* NP-complete problem can be solved in polynomial time i.e.  $P = NP$
- The Hamiltonian cycle problem is NP-complete
- NP-complete languages are considered to be the hardest in NP class
- Circuit-satisfiability problem is also NP-complete



# Polynomial-time reducibility

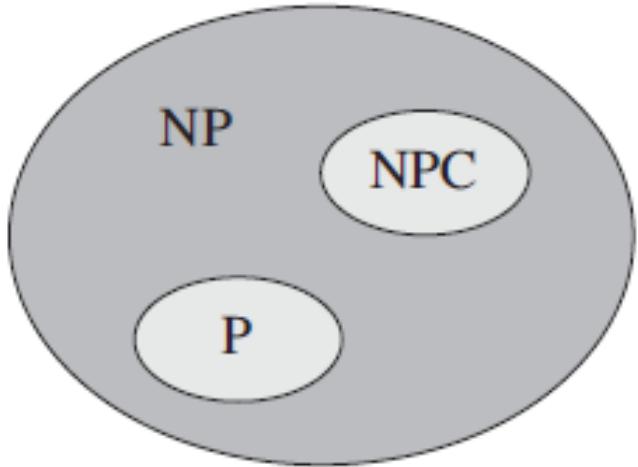
- a language  $L_1$  is ***polynomial-time reducible*** to a language  $L_2$ , written  $L_1 \leq_p L_2$ , if there exists a polynomial-time computable function  $f: \{0,1\}^* \rightarrow \{0,1\}^*$  such that for all  $x \in \{0,1\}^*$ ,

$$x \in L_1 \text{ if and only if } f(x) \in L_2$$

- The function  $f$  is called the reduction function and the polynomial-time algorithm  $F$  is called the reduction algorithm
- Lemma: If  $L_1, L_2 \subseteq \{0,1\}^*$  are two languages s.t.  $L_1 \leq_p L_2$ , then  $L_2 \in P$  implies  $L_1 \in P$

# NP-Completeness

- A language  $L \subseteq \{0,1\}^*$  is NP-complete if
  1.  $L \in NP$  and
  2.  $L' \leq_p L$  for every  $L' \in NP$
- If a language  $L$  satisfies property 2 but not necessarily property 1 then,  $L$  is called NP-hard
- Theorem: If any NP-complete problem can be solved in polynomial time then  $P = NP$ . Conversely, if any problem in NP is not polynomial time solvable, then no NP-complete problem is polynomial-time solvable



- Most theoretical computer scientists believe that  $P \neq NP$
- No polynomial time algorithm has ever been discovered for any NP-complete problem
- So, if we could prove that a problem is NP-complete, it in fact, gives excellent evidence that the problem is intractable

# NP-Completeness Example

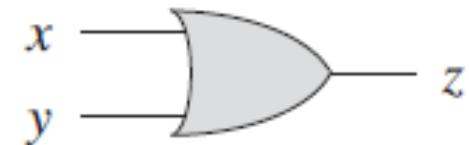
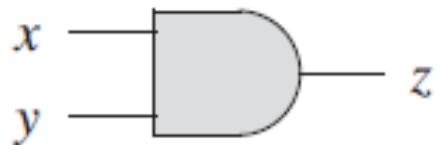
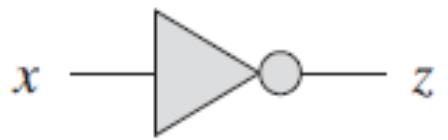
The Circuit-Satisfiability Problem

# Proving NP-completeness

- In order to prove that a problem is NP-complete we need to show that:
  - It is verifiable in polynomial time
  - An NP-complete problem is polynomial time reducible to this problem
- For proving the second criterion however, we need to have a first NP-complete problem for which we show that any problem in NP class is reducible to it in polynomial time
- After that, we can prove any problem to be NP-complete by only showing that my first NP-complete problem is reducible in polynomial time to this new problem
- We will choose circuit satisfiability problem as our first NP-complete problem

# Boolean Circuits

- We shall informally try to understand that circuit satisfiability problem is NP-complete
- A formal proof of NP-completeness is beyond the scope of the course
- A ***boolean combinational element*** is any circuit element that has a constant number of boolean inputs and outputs and that performs a well-defined function.
- A ***boolean combinational circuit*** consists of one or more boolean combinational elements interconnected by **wires**. A wire can connect the output of one element to the input of another, thereby providing the output value of the first element as an input value of the second.



$x$	$\neg x$
0	1
1	0

(a)

$x$	$y$	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

(b)

$x$	$y$	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

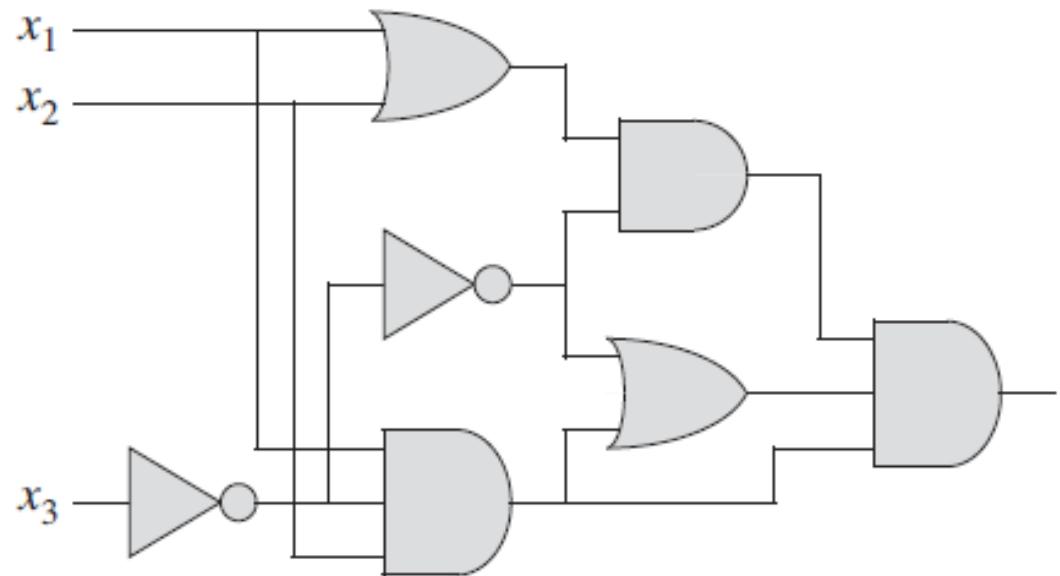
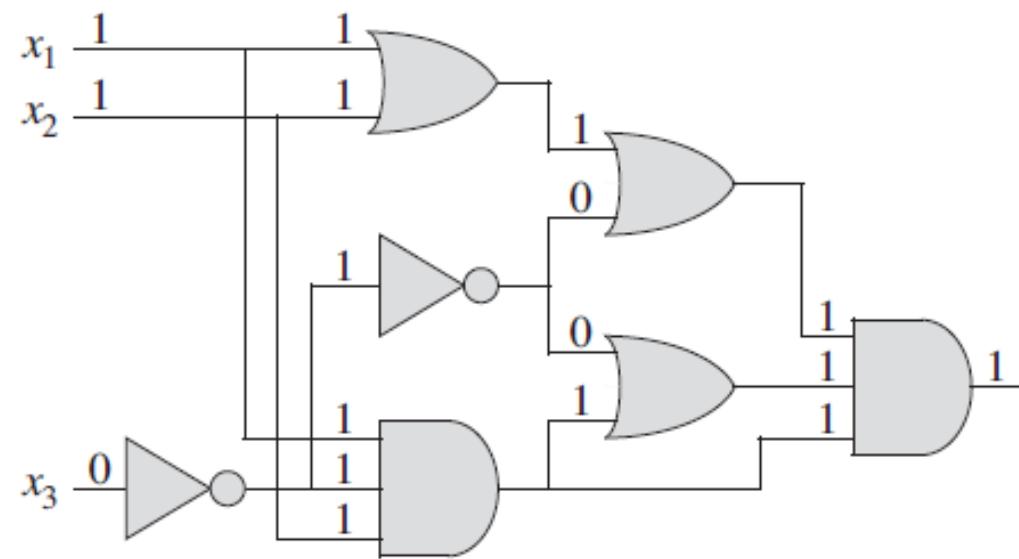
(c)

# Circuit input and output

- Although a single wire may have no more than one combinational-element output connected to it, it can feed several element inputs. The number of element inputs fed by a wire is called the *fan-out* of the wire. If no element output is connected to a wire, the wire is a *circuit input*, accepting input values from an external source. If no element input is connected to a wire, the wire is a *circuit output*, providing the results of the circuit's computation to the outside world.
- For the purpose of defining the circuit-satisfiability problem, we limit the number of circuit outputs to 1, though in actual hardware design, a boolean combinational circuit may have multiple outputs.
- Boolean combinational circuits **do not** contain any *cycles*.

# Circuit Satisfiability Problem

- A ***truth assignment*** for a boolean combinational circuit is a set of boolean input values. We say that a one-output boolean combinational circuit is ***satisfiable*** if it has a ***satisfying assignment***: a truth assignment that causes the output of the circuit to be 1.
- The ***circuit-satisfiability problem*** is, “Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?”
- In order to pose this question formally, however, we must agree on a standard encoding for circuits.



- The **size** of a boolean combinational circuit is the number of boolean combinational elements plus the number of wires in the circuit.
- We could devise a graph-like encoding that maps any given circuit  $C$  into a binary string  $\langle C \rangle$  whose length is polynomial in the size of the circuit itself.
- As a formal language, we can therefore define:
- $\text{CIRCUIT-SAT} = \{\langle C \rangle : C \text{ is a satisfiable boolean combinational circuit}\}$
- This problem is of practical importance in hardware optimizing. If a sub-circuit always produces a 0, then the designer can replace it with some other circuit that omits all logic gates and simply gives a 0 on every input
- If we apply brute-force to check satisfiability of a  $k$ -input circuit then we will have to test  $2^k$  possible circuit arrangements hence, making the procedure super-polynomial

# Proving NP-completeness

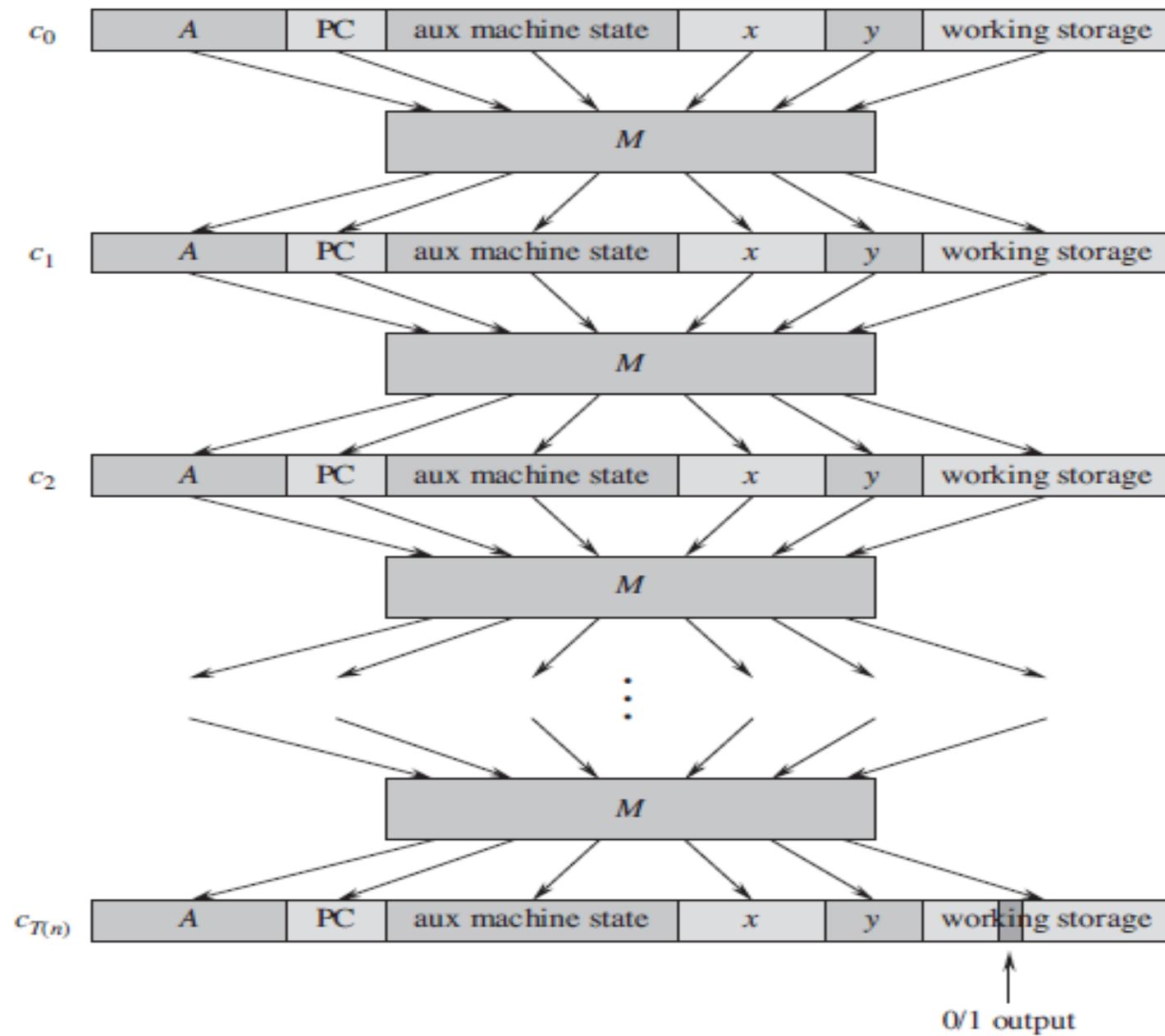
- In order to prove that CIRCUIT-SAT is NP-complete we will divide the proof into two parts:
  1. It belongs to NP class i.e. it is verifiable in polynomial time
  2. It is NP-hard i.e. any NP problem is reducible to it in polynomial time

# Polynomial time verification

- Suppose, we are given a possible assignment of the boolean values and we want to test whether the assignment satisfies the circuit or not
- Our algorithm will compute for each logic gate present in the circuit, whether the input to the wire correctly computes the value as claimed. With a good implementation it can be done in polynomial time
- Further, if the assignment does not yield a 1; there is no way our algorithm can be fooled into believing that the circuit is satisfied
- Thus, CIRCUIT-SAT can be verified in polynomial time

# NP-Hardness

- Let  $L$  be any language in NP. We shall describe a polynomial-time algorithm  $F$  computing a reduction function  $f$  that maps every binary string  $x$  to a circuit  $C = f(x)$  such that  $x \in L$  if and only if  $C \in \text{CIRCUIT-SAT}$ .
- The basic idea of the proof is that any algorithm  $A$  that accepts a language  $L$  can be implemented in the form of a circuit  $M$  in the computer
- The state of the computer changes from one state to other while solving the problem, each time the new state is input to  $M$
- When the algorithm halts, the state of the system stops changing



- We can imagine the reduction function  $F$  to consist of consecutive copies of  $M$  corresponding to the number of steps that  $A$  will take in reaching to the halting state
- Since, all the elements in the reduction function  $F$  – the problem description, the certificate, the memory required for saving the results, and the number of copies of  $M$  are all polynomial in ' $n$ ', we can say that  $F$  will run in no more than polynomial time
- Since,  $F$  will only output 1 when  $A(x,y)$  gives 1 and whenever  $A(x,y)$  gives output 1,  $F$  outputs 1 – the function  $F$  correctly reduces  $A$  to Circuit-satisfiability problem