

Analýza a Zložitosť Algoritmov

AZA

Doc. RNDr. Silvester Czanner, PhD

silvester.czanner@stuba.sk (subject AZA)

Summary – session 1

- Lecturer introduction
- Refresher
 - Data structure taxonomy
 - Data structure landscape
 - C++ programming
 - Algorithms, pseudo-code, code
- Logistics
 - Semester and weekly schedule
 - Labs organization
- New stuff
 - **Complexity Analysis**

Who am I ? Overview of my academic journey

1989-00 Univerzita Komenského, Slovakia

2000-02 University of Aizu, Japan

2002-06 Carnegie Mellon University, USA

Pittsburgh University, USA

Harvard University, USA

2007-11 University of Warwick, UK

EPSRC, Robert's fund, IAS European Frontiers, Reinvention Centre Collaboration Fund,

RCUK academic fellowship

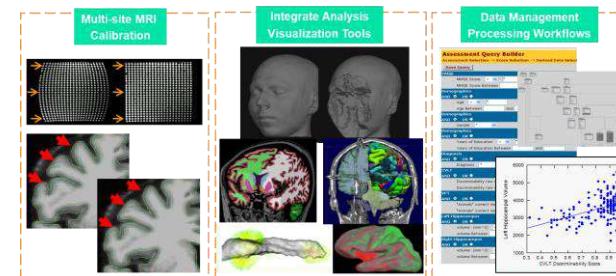
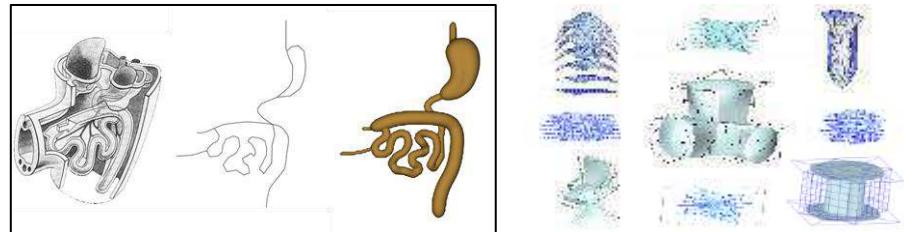
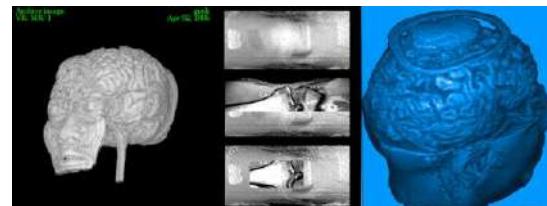
2011 - 2018 Manchester Met University, UK

KTP, H2020, MAN MET research fund, Erasmus teaching grant

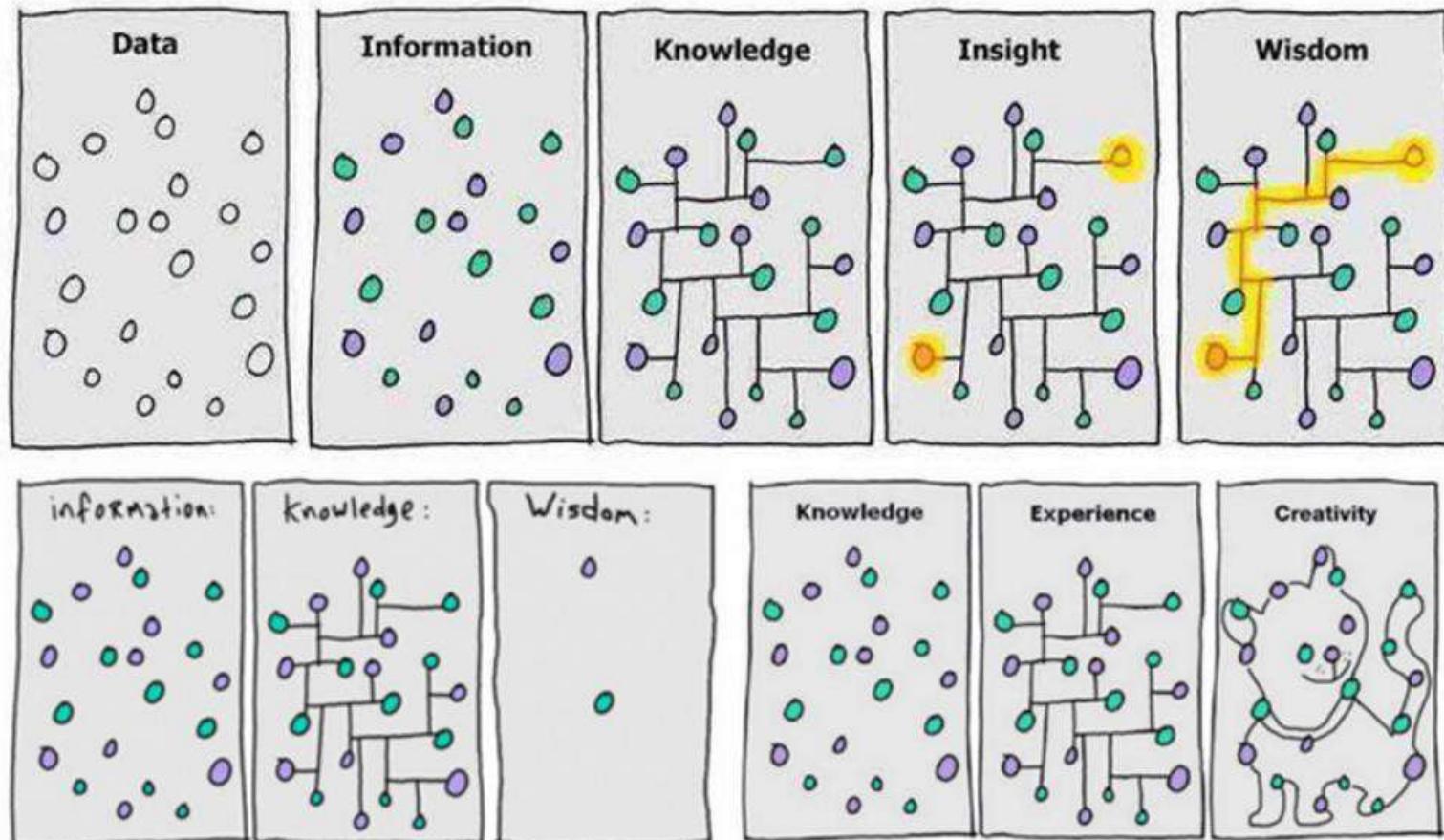
2019 - 2024 Liverpool John Moores Uni, UK

2024 – now Chester Uni, UK

2021 - now FIIT, STU



Refresher – Data Structure I.



Refresher – Data Structure II

Data

Data is information that has been translated into a form that is efficient for movement or processing.

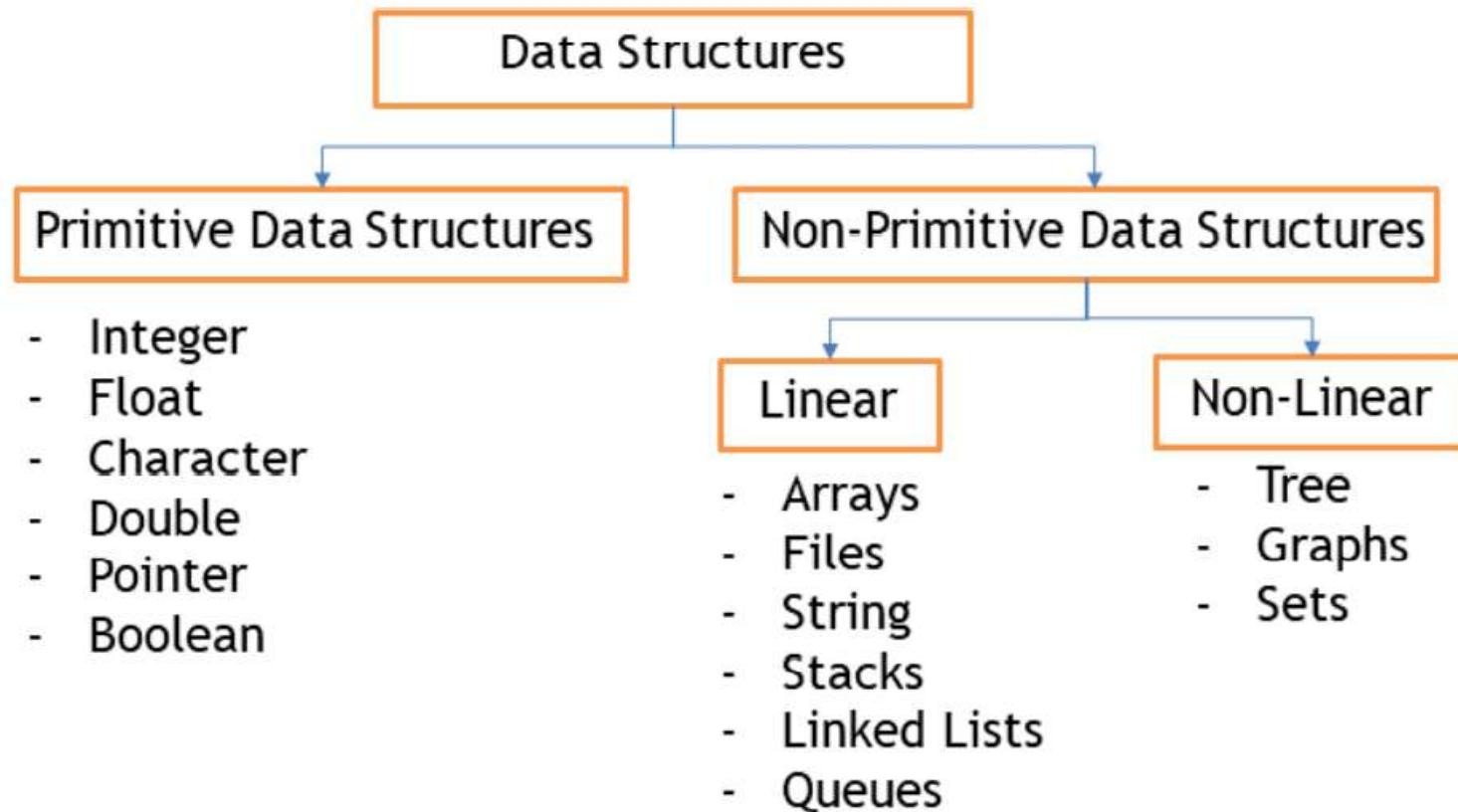
Data Type

A data type is a classification that specifies which type of value a variable has and what type of operations can be applied to it.

Data Structure

A data structure is a data organization, management, and storage format that enables efficient access and modification.

Refresher – Data Structure Taxonomy

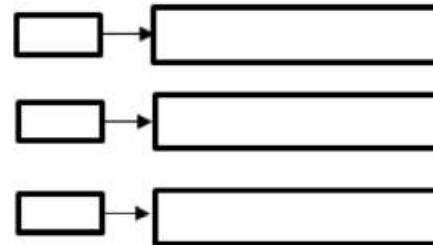


Refresher – Data Structure Landscape

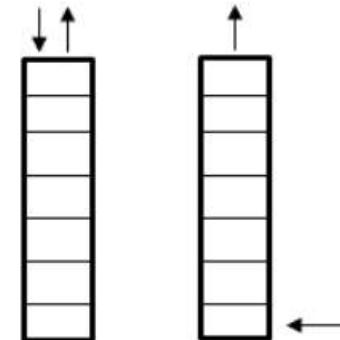
Arrays



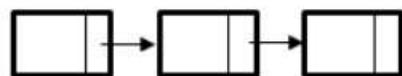
Hash Table



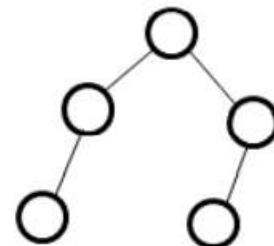
Stack and Queue



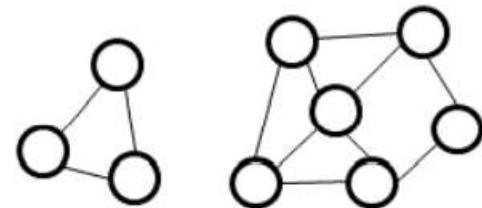
Linked List



Tree



Graph



Data Structure - Requirements

The five requirements of any data structure:

1. How to **Access** (one item / all items)
2. How to **Insert** (at end / at position)
3. How to **Delete** (form end / from position)
4. How to **Find** (if exists / what location)
5. How to **Sort** (sort in place / create sorted version)

C++ Programming

- Datatypes
- Structure and Union
- Functions
- Decision Structure (if-then-else, for, while)
- Class
- Template
- Memory Address
- Dynamic Memory management
- Static and Dynamic allocation
- Pointers

Algorithm, Pseudo-Code and Code

Algorithm

An algorithm expresses computational processes for solving a problem in terms of the actions. An algorithm is merely the sequence of steps taken to solve a problem.

Pseudo-Code

Pseudocode is human-friendly and informal but structured english for describing algorithms. It cannot be understood by a machine, but helps programmers develop algorithms.

Code

Implementation of a given algorithm in a specific programming language that can be interpreted or compiled by a machine.

Example of Pseudo-Code

Algorithm 1 Mystery Procedure

Input: Array A of n elements.

Output: Array sorted

```
1: procedure Mystery( $A, n$ )
2:   for  $j \leftarrow 1$  to  $n$  do
3:      $key \leftarrow A[j]$ 
4:      $i \leftarrow j - 1$ 
5:     while  $i > 0$  and  $A[i] > key$  do
6:       /* swap element */
7:        $A[i + 1] \leftarrow A[i]$ 
8:        $i \leftarrow i - 1$ 
9:     end while
10:     $A[i + 1] \leftarrow key$ 
11:  end for
12: end procedure
```

Example of Code

```
void Mystery(int A*, int n)
{
    int i, key, j;
    for (j = 1; j < n; j++)
    {
        key = A[j];
        i = j - 1;

        while (i >= 0 && A[i] > key)
        {
            A[i + 1] = A[i];
            i = i - 1;
        }
        A[i + 1] = key;
    }
}
```

Comparison – Pseudo-code vs CODE

Algorithm 1 Mystery Procedure

Input: Array A of n elements.

Output: Array sorted

```
1: procedure Mystery( $A, n$ )
2:   for  $j \leftarrow 1$  to  $n$  do
3:      $key \leftarrow A[j]$ 
4:      $i \leftarrow j - 1$ 
5:     while  $i > 0$  and  $A[i] > key$  do
6:       /* swap element */
7:        $A[i + 1] \leftarrow A[i]$ 
8:        $i \leftarrow i - 1$ 
9:     end while
10:     $A[i + 1] \leftarrow key$ 
11:  end for
12: end procedure
```

```
void Mystery(int A*, int n)
{
  int i, key, j;
  for (j = 1; j < n; j++)
  {
    key = A[j];
    i = j - 1;

    while (i >= 0 && A[i] > key)
    {
      A[i + 1] = A[i];
      i = i - 1;
    }
    A[i + 1] = key;
}
```

Semester Schedule

Lectures: 16/9, 30/9, 14/10, 21/10, 04/11, 11/11, 18/11

- Monday 8-10
- Monday 11-13

Labs: Martina Billichová, Dmytro Furman, Timotej Kralik, Michal Lüley a Martin Družbacký

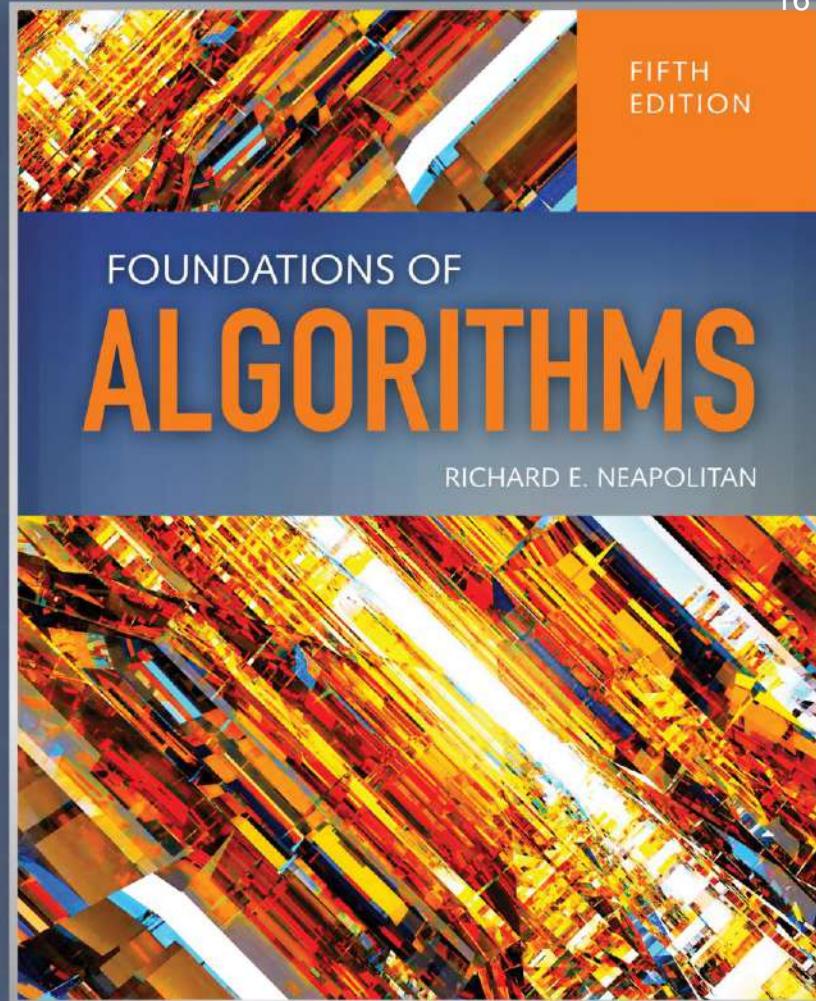
Assessment:

- 2 x assignments during the semester
 - Written test on theoretical analysis of an algorithm (20%) – **21/10/2024**
 - Practical implementation and analysis of an algorithm (30%) – deadline **6/12/2024**
- 1 x exam
 - Theoretical analysis and pseudocode of algorithms (50%) – **January 2025**

Semester Schedule II.

- 16/9
 - Session 1: Intro, Logistics, Algorithms: Efficiency, Analysis, and Order
 - Session 2: Divide-and-Conquer
- 30/09
 - Session 1: Dynamic programming,
 - Session 2: The Greedy approach
- 14/10
 - Session 1: Backtracking
 - Session 2: Branch-and-Bound
- **21/10**
 - Session 1: written test (1h) – material from first 5 labs
 - Session 2: written test (1h) – material from first 5 labs
- 04/11
 - Session 1: Computational Complexity: The Sorting Problem
 - Session 2: Computational Complexity: The Searching Problem
- 11/11
 - Session 1: An introduction to the theory of NP
 - Session 2: Limitations of Algorithm Power
- 18/11
 - Session 1: Wrap up, Q&A session
- Practical work submission deadline **06/12 5PM**

Algorithms: Efficiency, Analysis, and Order Chapter 1





Objectives

- Analyze techniques for solving problems
- Define an algorithm
- Define growth rate of an algorithm as a function of input size
- Define Worst case, average case, and best-case complexity analysis of algorithms
- Classify functions based on growth rate
- Define growth rates: Big O, Theta, and Omega



Methodology

- Approach to solving a problem
- Independent of Programming Language
- Independent of Style
- Sequential Search versus Binary Search
- Which technique results in the most efficient solution?



Problem?

- A question to which an answer is sought
- Parameters
 - Input to the problem
 - Instance: a specific assignment of values to the input parameters
- Algorithm:
 - Step-by-step procedure
 - Solves the Problem



Sequential Search vs Binary Search – Worst Case

- Input Array S size n
- $X \notin S$
- Sequential Search: n operations
- Binary Search: $\lg n + 1$ operations



Fibonacci: Iterative vs Recursive

- $\text{Fib}_0 = 0$
- $\text{Fib}_1 = 1$
- $\text{Fib}_n = \text{Fib}_{n-1} + \text{Fib}_{n-2}$
- Calculate the nth Fibonacci Term:
 - Recursive calculates $2^{n/2}$ terms
 - Iterative calculates $n+1$ terms

Fast computers vs efficient algorithms [CLRS 1]

Many recent innovations rely on

- fast computers
- efficient algorithms.

Which is more important?

The importance of efficient algorithms

The cost of an algorithm can be quantified by the number of steps $T(n)$ in which the algorithm solves a problem of size n .

Imagine that a certain problem can be solved by four different algorithms, with $T(n) = n, n^2, n^3$, and 2^n , respectively.

Question: what is the maximum problem size that the algorithm can solve in a given time?

Assume that a computer is capable of 10^{10} steps per second.

Cost $T(n)$ (Complexity)	Maximum problem size solvable in		
	1 second	1 hour	1 year
n	10^{10}	3.6×10^{13}	3×10^{17}
n^2	10^5	6×10^6	5×10^8
n^3	2154	33000	680000
2^n	33	45	58

Faster computers vs more efficient algorithms

Suppose a faster computer is capable of 10^{16} steps per second.

Cost $T(n)$	Max. size before	Max. size now
n	s_1	$10^6 \times s_1$
n^2	s_2	$1000 \times s_2$
n^3	s_3	$100 \times s_3$
2^n	s_4	$s_4 + 20$

A $10^6 \times$ increase in speed results in only a factor-of-100 improvement if cost is n^3 , and only an additive increase of 20 if cost is 2^n .

Conclusions As computer speeds increase ...

- ... it is algorithmic efficiency that really determines the increase in problem size that can be achieved.
- ... so does the size of problems we wish to solve.

Thus, designing efficient algorithms becomes even more important!

From Algorism to Algorithms

Invented in India around AD 600, the *decimal system* was a revolution in quantitative reasoning. Arabic mathematicians helped develop arithmetic methods using the Indian decimals.

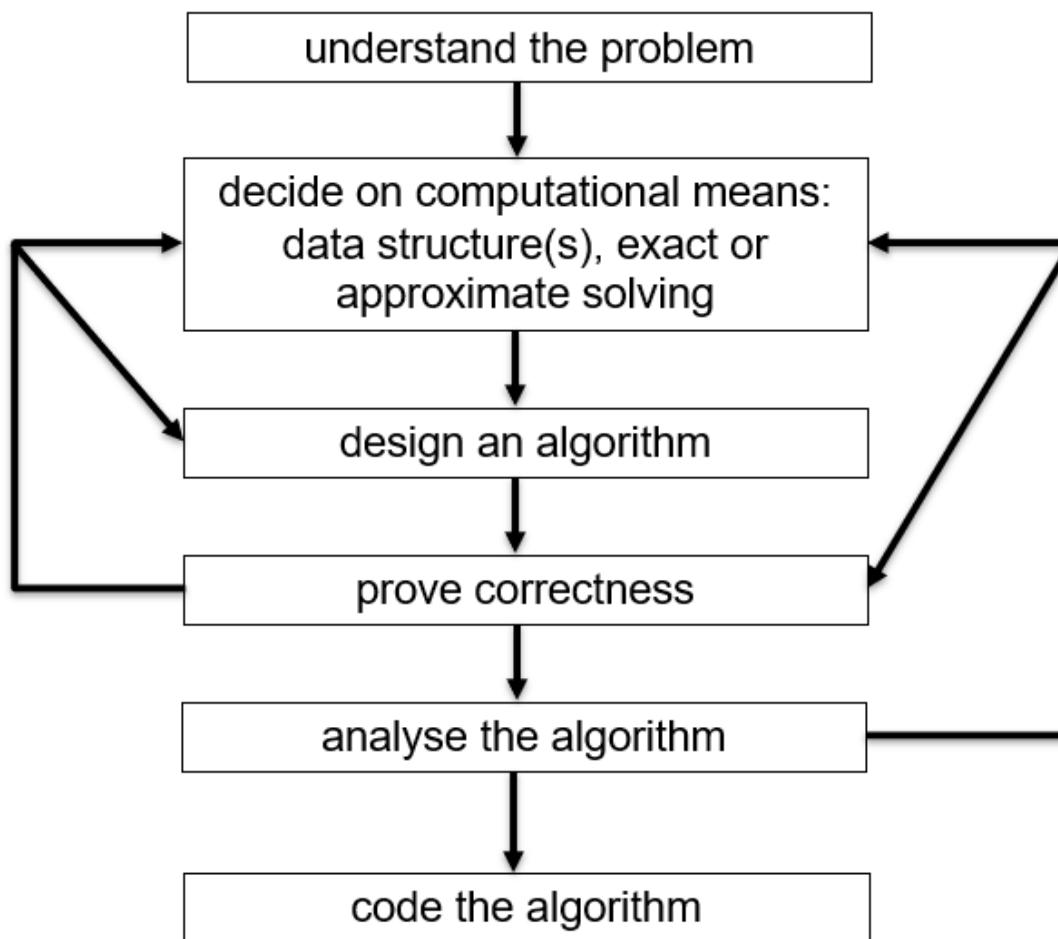
A 9th-century Arabic textbook by the Persian *Al Khwarizmi* was the key to the spread of the Indian-Arabic decimal arithmetic. He gave methods for basic arithmetic (adding, multiplying and dividing numbers), even the calculation of square roots and digits of π .

Derived from ‘*Al Khwarizmi*’, *algorism* means rules for performing arithmetic computations using the Indian-Arabic decimal system.

The word “*algorism*” devolved into *algorithm*, with a generalisation of the meaning to

Algorithm: a finite set of well-defined instructions for accomplishing some task.

Algorithm Design and Analysis



Evaluating algorithms

Two questions we ask about an algorithm

1. Is it correct?
2. Is it efficient?

Correctness is of utmost importance.

It is easy to design a highly efficient but incorrect algorithm.

Efficiency with respect to:

- Running time
- Space (amount of memory used)
- Network traffic
- Other features (e.g. number of times secondary storage is accessed)

Proving correctness and analysing the efficiency of programs are difficult problems, in general. Take for example the Collatz program: starting from a positive integer x repeat “if x is even then $x = x/2$, else $x = (3x + 1)/2$ ” until $x = 1$. It is an open problem whether it terminates for every positive x .

Computational Complexity

- Computational complexity **is the number of resources required to run an algorithm.**
- Focus is given to time and memory requirements.
- **Complexity of a problem** = complexity of the optimal algorithms solving the problem.
- The resource that is most commonly considered is **time**.
- Another important resource is the **size of memory** needed for running algorithms.
- Complexity in **time or space/memory**.
- Complexity expressed with respect to the **size of the input**.
- The **number of arithmetic operations** is another resource that is commonly used.
- Impossible to count the number of steps of an algorithm on all inputs.
- Therefore, several complexity functions are commonly used.

Space Complexity

- When memory was expensive, we focused on making programs as **space** efficient as possible and developed schemes to make memory appear larger than it really was

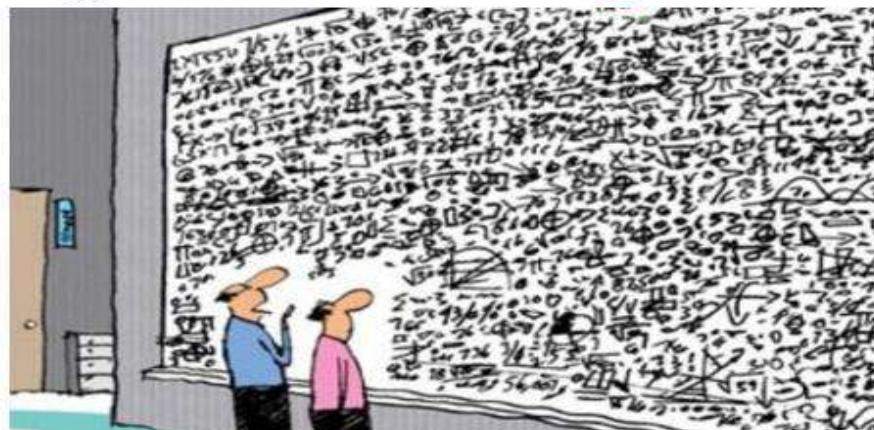
ATARI 800XL – 64KB memory
HDD → magnetic tape



- Space complexity is still important in the field of embedded computing (handheld computer based equipment like cell phones, palm devices, etc.)

Time Complexity

- Is the algorithm “**fast enough**” for my needs
- How much longer will the algorithm take if I increase the amount of data it must process
- Given a set of algorithms that accomplish the same thing, which is the **right** one to choose



Degrees of Complexity

An algorithm shows different degrees of complexity when exposed to different configuration of inputs.

Best-Case Analysis

Pure Optimism. In the most ideal case, the minimum number of primitive operations for any input size n (with n not equal to 1).

Average-Case Analysis

Being Practical. The maximum number of primitive operations performed on an input of size n (with n not equal to infinity).

Worst-Case Analysis

Pure Pessimism. In the worst case, the weighted average number of primitive operations on input size n .

Asymptotic Analysis

- Asymptotic analysis is the process of calculating the running time of an algorithm in mathematical units to find the program's limitations, or run-time performance.
- Asymptotic analysis defines the mathematical boundation of an algorithm run-time.
- We want to know is how long these algorithms take.
- Asymptotic Analysis is the evaluation of the performance of an algorithm in terms of just the input size n .
- Give an idea of the limiting behavior of a function.
- Determine the best case, worst case and average case time required to execute a given function.
- Focus on how fast a function grows with the input size.
- We call this the rate of growth of the running time.

Tilde Approximations

- Tilde notation is used when we want to make a simple approximation of a complex function.
- Assume that the lower order terms contributes relatively small and thus considers only leading terms.
- Use tilde notation to develop simpler approximate expressions.
- Throw away low-order terms that complicate formulas.
- Simply drops the lower order terms.
- Tilde notation (\sim) defines both the upper bound and lower bound of an algorithm.
- It gives a sharper approximation of an algorithm.
- In asymptotic analysis we are only interested to know about the growth of a function for a larger value of n .

Tilde Approximations II

Tilde Notation (\sim)

Tilde notation is used when we want to make a simple approximation of a complex function. It simply drops the lower order terms. It is denoted by $\sim g(n)$. We write $\sim f(n)$ to represent any function that when divided by $f(n)$ approaches 1 as n grows. We write $g(n) \sim f(n)$ to indicate that $g(n)/f(n)$ approaches 1 as n grows. Mathematically, we can express it as :

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

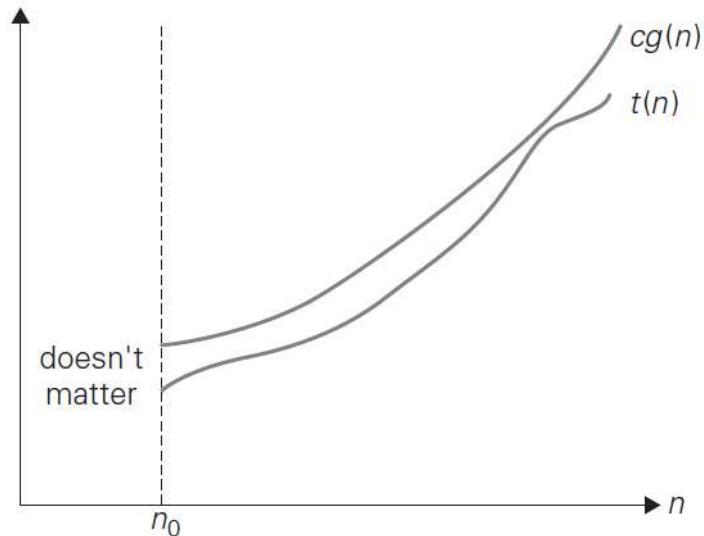
Asymptotic notations and basic efficiency classes

Informally, $O(g(n))$ is the set of all functions with a lower or same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity). Thus, to give a few examples, the following assertions are all true:

$$n \in O(n^2), \quad 100n + 5 \in O(n^2), \quad \frac{1}{2}n(n - 1) \in O(n^2).$$

$$n^3 \notin O(n^2), \quad 0.00001n^3 \notin O(n^2), \quad n^4 + n + 1 \notin O(n^2).$$

O-notation



DEFINITION A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n , i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \quad \text{for all } n \geq n_0.$$

As an example, let us formally prove one of the assertions made in the introduction: $100n + 5 \in O(n^2)$. Indeed,

$$100n + 5 \leq 100n + n \text{ (for all } n \geq 5\text{)} = 101n \leq 101n^2.$$

Thus, as values of the constants c and n_0 required by the definition, we can take 101 and 5, respectively.

Properties of Big-O

Lemma

Let $f, g, h : \mathbb{N} \rightarrow \mathbb{R}^+$. Then:

1. For every constant $c > 0$, if $f \in O(g)$ then $cf \in O(g)$.
2. For every constant $c > 0$, if $f \in O(g)$ then $f \in O(cg)$.
3. If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$ then $f_1 + f_2 \in O(g_1 + g_2)$.
4. If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$ then $f_1 + f_2 \in O(\max(g_1, g_2))$.
5. If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$ then $f_1 \cdot f_2 \in O(g_1 \cdot g_2)$.
6. If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$.
7. Every polynomial of degree $l \geq 0$ is in $O(n^l)$.
8. For any $c > 0$ in \mathbb{R} , we have $\lg(n^c) \in O(\lg(n))$.
9. For every constant $c, d > 0$, we have $\lg^c(n) \in O(n^d)$.
10. For every constant $c > 0$ and $d > 1$, we have $n^c \in O(d^n)$.
11. For every constant $0 \leq c \leq d$, we have $n^c \in O(n^d)$.

Example

Example. Show that

$$57n^3 + 4n^2 \cdot \lg^5(n) + 17n + 498 \in O(n^3)$$

by appealing to Lemma 1.

$$\lg^5(n) \in O(n) \quad \because 9$$

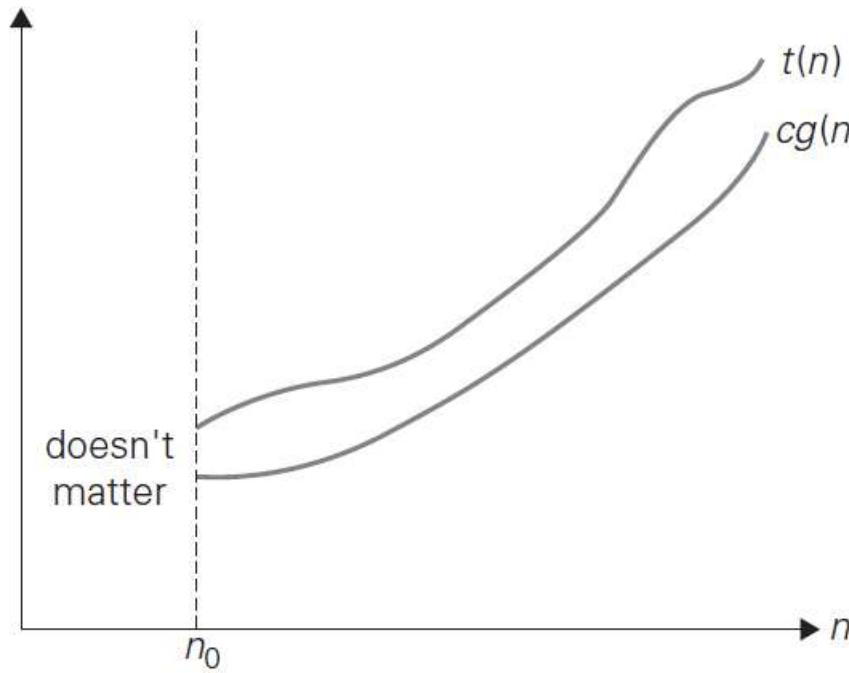
$$4n^2 \cdot \lg^5(n) \in O(4n^3) \quad \because 5$$

$$57n^3 + 4n^2 \cdot \lg^5(n) + 17n + 498 \in O(57n^3 + 4n^3 + 17n + 498) \quad \because 3$$

$$57n^3 + 4n^3 + 17n + 498 \in O(n^3) \quad \because 7$$

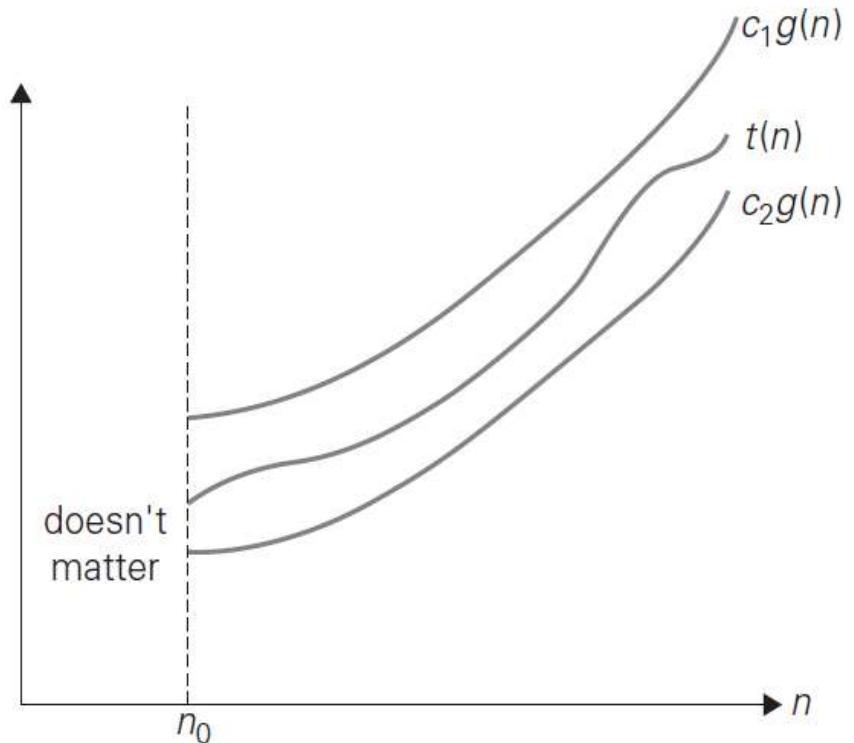
$$57n^3 + 4n^2 \cdot \lg^5(n) + 17n + 498 \in O(n^3) \quad \because 6$$

Ω and Θ notations



$$t(n) \in \Omega(g(n)).$$

$$t(n) \geq cg(n) \quad \text{for all } n \geq n_0.$$



$$t(n) \in \Theta(g(n)).$$

$$c_2g(n) \leq t(n) \leq c_1g(n) \quad \text{for all } n \geq n_0.$$

Examples

1. $5n^3 + 88n = \Theta(n^3)$

2. $2 + \sin(\lg n) = \Theta(1)$

3. $n! = \Theta(n^{n+1/2} e^{-n})$.

Consequence of Stirling's Approximation.

4. For all $a, b > 1$, $\log_a n = \Theta(\log_b n)$.

Consequence of the relation $\log_b a = \frac{\log_c a}{\log_c b}$.

From now on, we will be “neutral” and write $\Theta(\log n)$, without specifying the base of the logarithm.

Example

1. $n^n = \Omega(n!)$

2. $2^n = \Omega(n^{10})$.

Question. OK, we have seen that $\log_a n$ is a Big-Theta of $\log_b n$.

But is $2^{\log_a n}$ a Big-Theta of $2^{\log_b n}$?

No! Recall that $a^{\log_b c} = c^{\log_b a}$ for all $a, b, c > 0$.

Using this relation, we have $2^{\log_a n} = n^{\log_a 2}$ and $2^{\log_b n} = n^{\log_b 2}$.

If $a \neq b$, we have two different powers of n , but $n^c = \Theta(n^d)$ only if $c = d$.

Revision

Logarithms.

$\log_2 n$ is sometimes written $\lg n$, and $\log_e n$ is sometimes written $\ln n$.

Recall the following useful facts. Let $a, b, c > 0$.

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

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b a = \frac{1}{\log_a b}$$

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

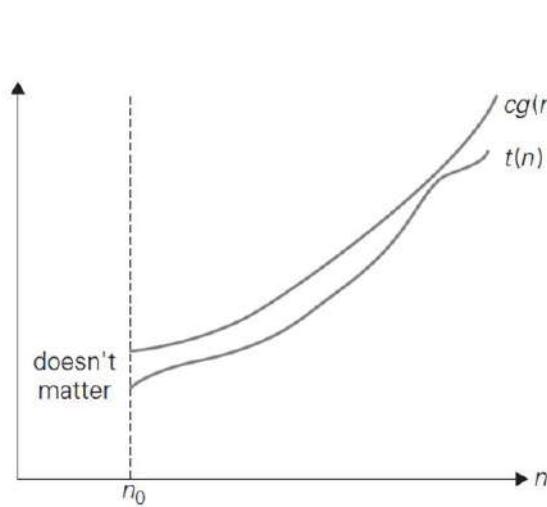
A form of Stirling's approximation.

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

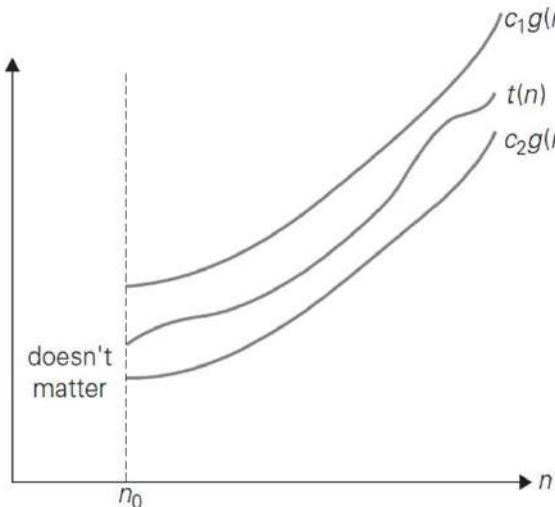
Comparing Orders of Growth

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

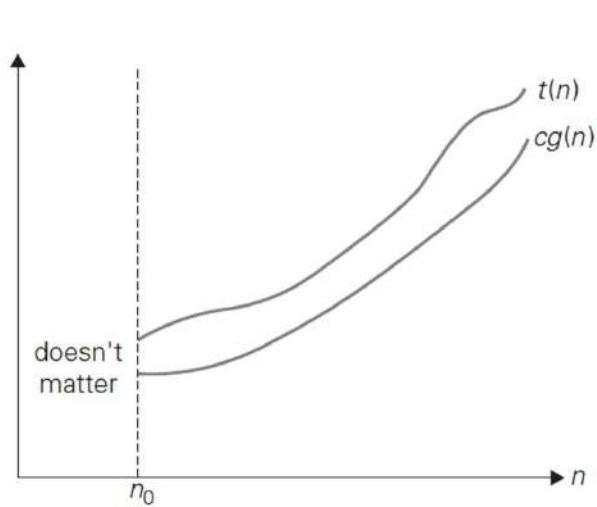
Note that the first two cases mean that $t(n) \in O(g(n))$, the last two mean that $t(n) \in \Omega(g(n))$, and the second case means that $t(n) \in \Theta(g(n))$.



$t(n) \in O(g(n))$



$t(n) \in \Theta(g(n))$.

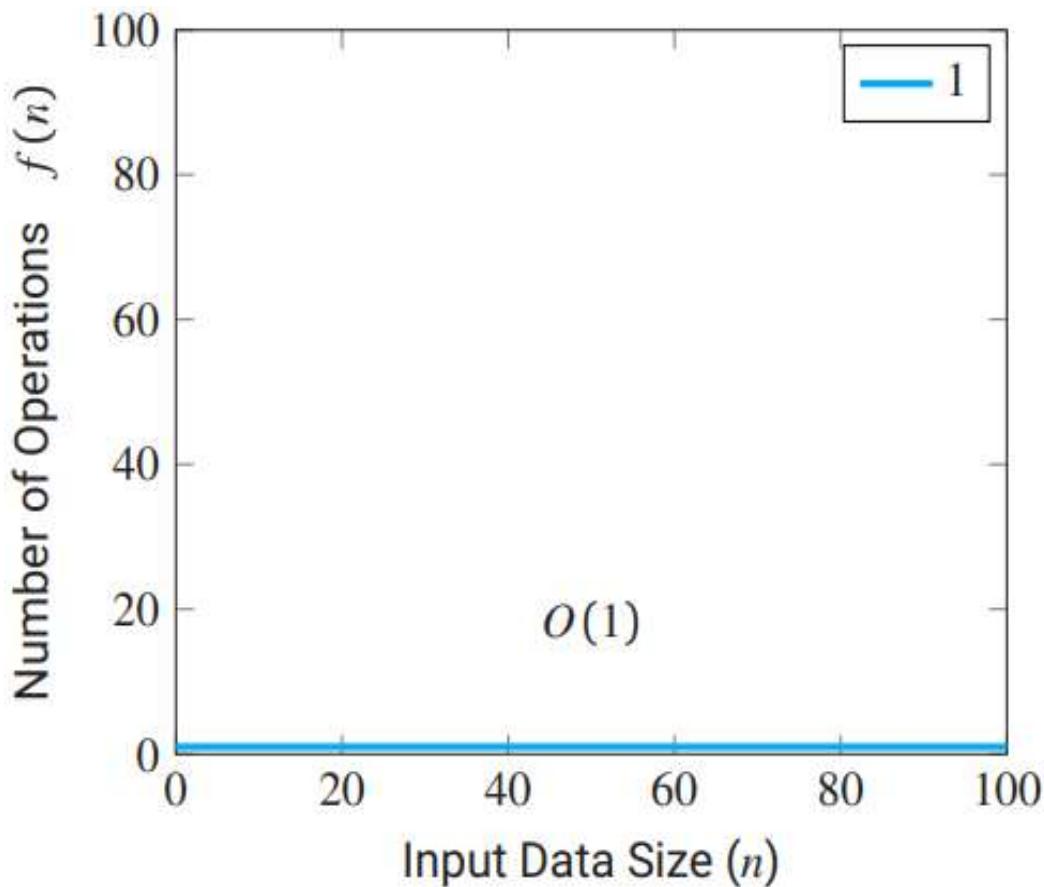


$t(n) \in \Omega(g(n))$.

Big-O Analysis

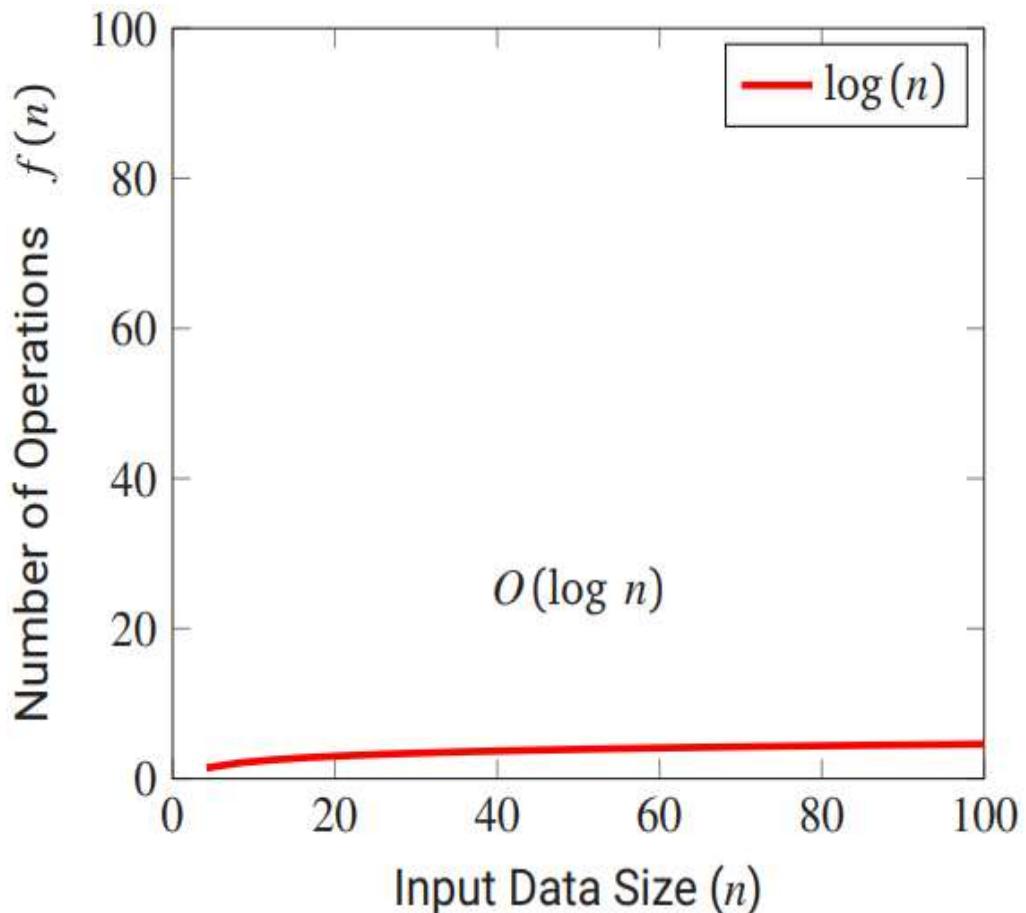
- Given the input of size n , Big- O describes an upper bound on number of relevant, primitive operations algorithm needs to perform.
- Big- O can be used to describe space complexity as well as time complexity, or any other complexity.
- This notation is basically used to describe the asymptotic upper bound.
- The time complexity of an algorithm is commonly expressed using Big- O notation.
- Big- O notation describes the execution time required or the spaced used by an algorithm.
- Big- O actually represents how fast things can go really bad.
- Big- O notation specifically describes the worst-case scenario.

Constant Time Complexity



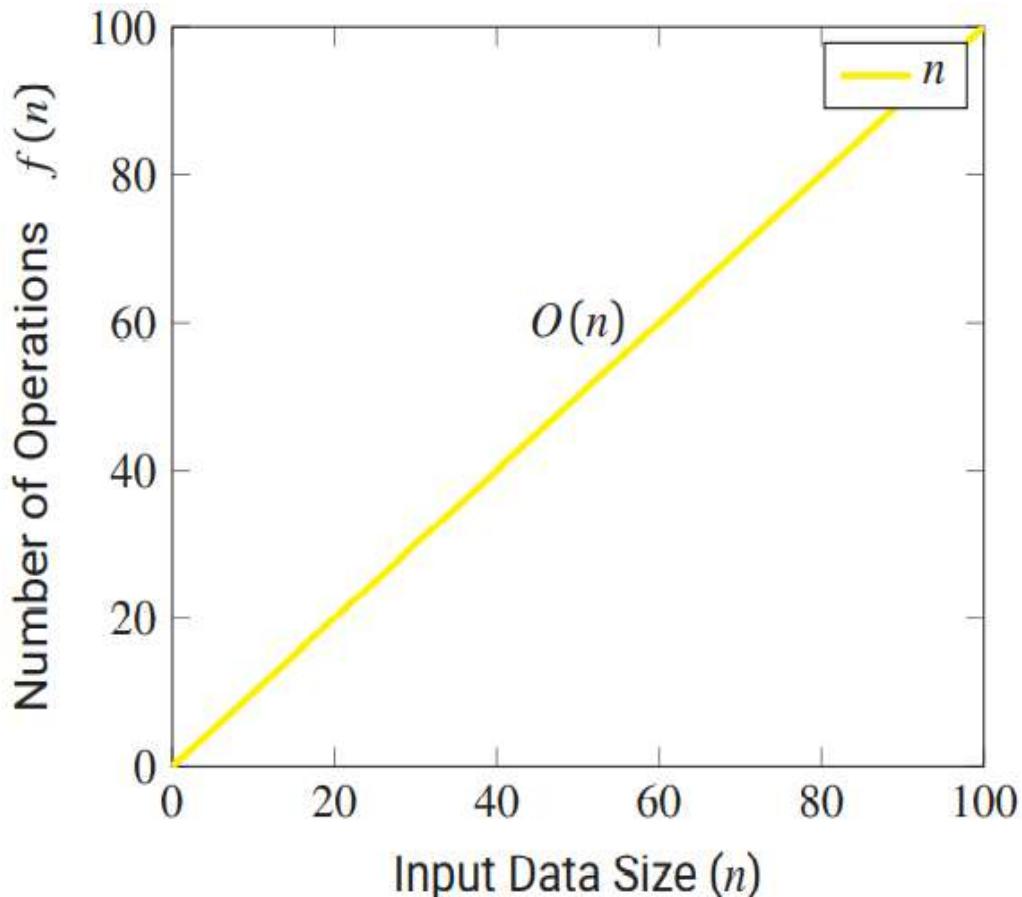
- Constant Time Complexity $O(1)$.
- Constant growth rate.
- Fastest time complexity.
- Constant-time method is order 1.
- Not dependent on the input size n .
- Irrespective of the input size n .
- The runtime will always be the same.

Logarithmic Time Complexity



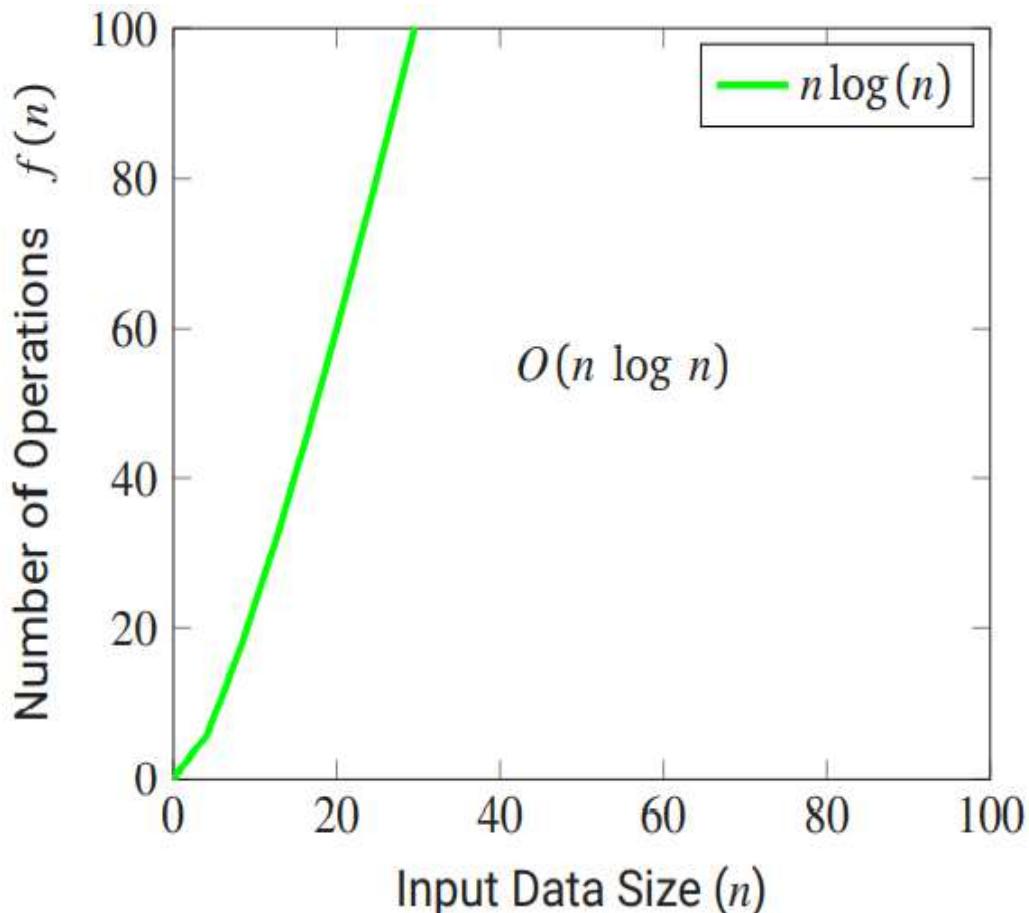
- Time Complexity $O(\log n)$.
- Logarithmic order complexity.
- Considered highly efficient.
- The ratio of the number of operations to the size of the input decreases and tends to zero when n increases.
- Execution time increases, but at a decreasing rate.
- Commonly found in operations on binary trees or search.

Linear Time Complexity



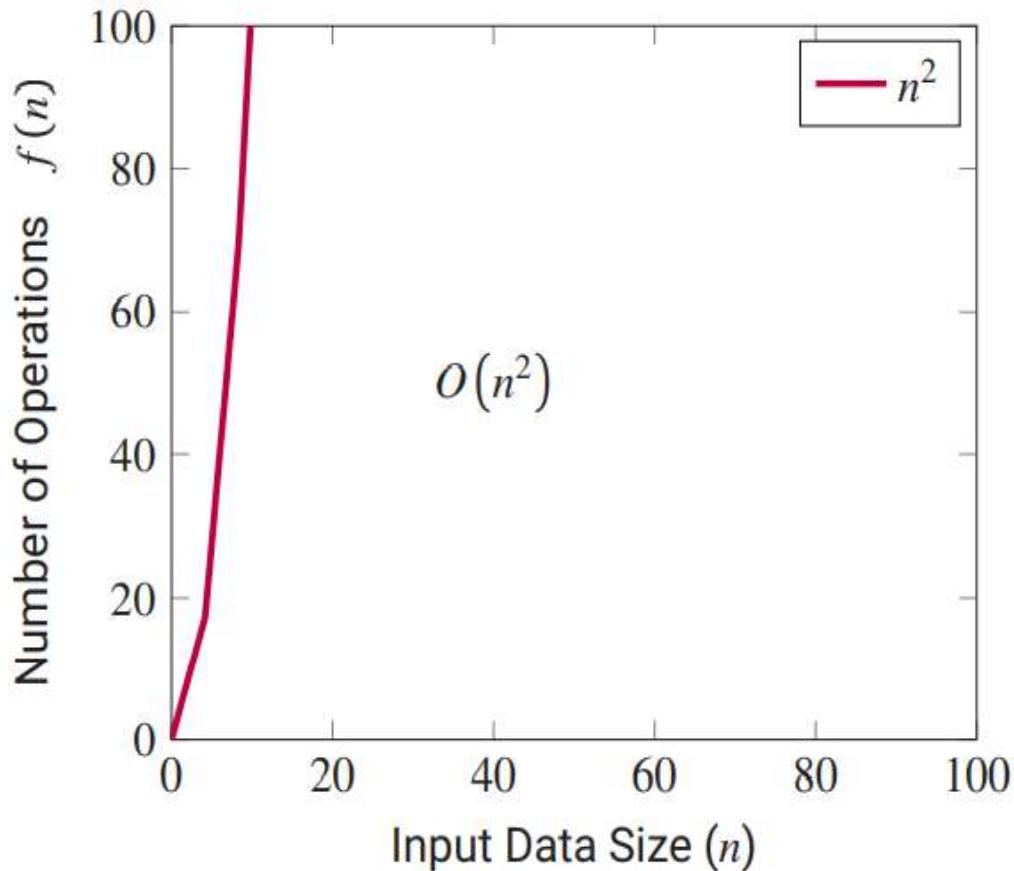
- Time Complexity $O(n)$.
- The running time increases at most linearly with the length of the input.
- Takes proportionally longer to complete as the input grows.
- Number of elements and number of steps are linearly dependent.
- Best possible time complexity to sequentially read its entire input.

Linearithmic Time Complexity



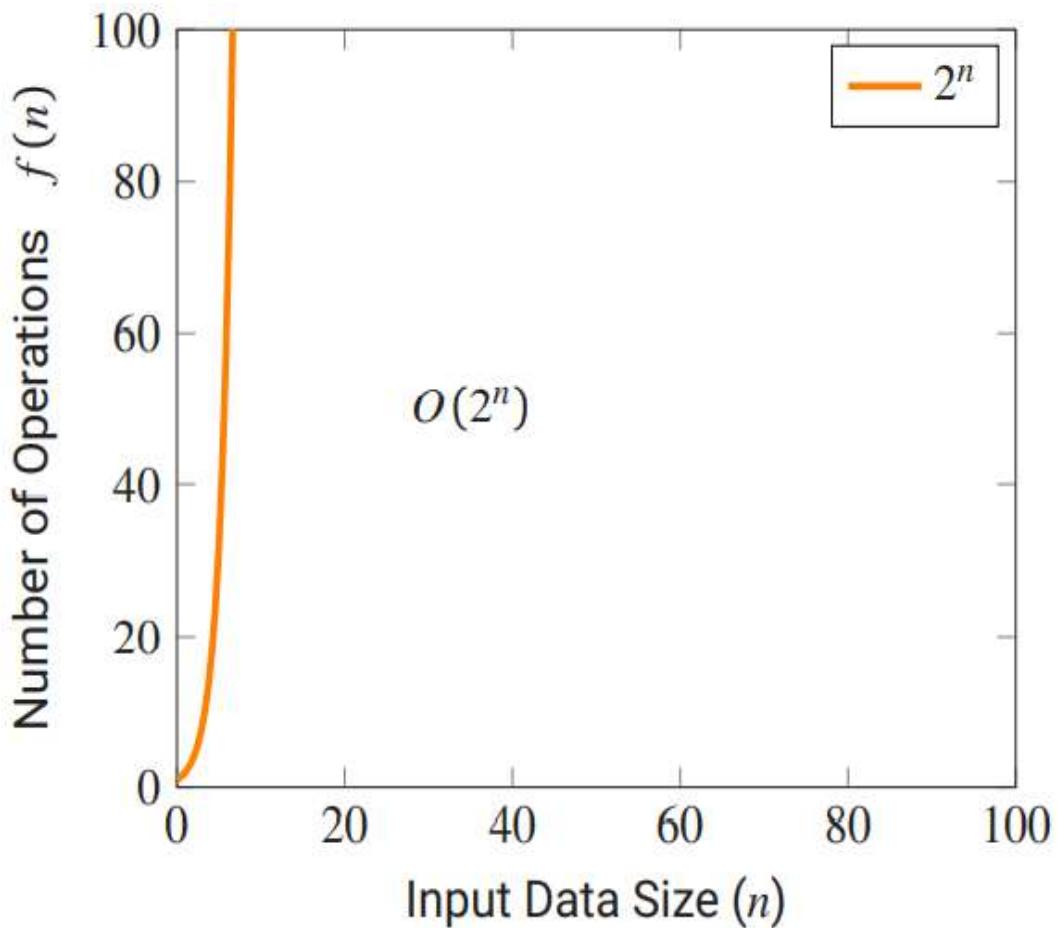
- Time Complexity $O(n \log n)$.
- Takes an amount of time between linear and polynomial.
- The worst of the best complexity.
- A moderate complexity that floats around linear time.
- Slightly slower than a linear algorithm.
- Still much better than a quadratic algorithm.

Quadratic Time Complexity



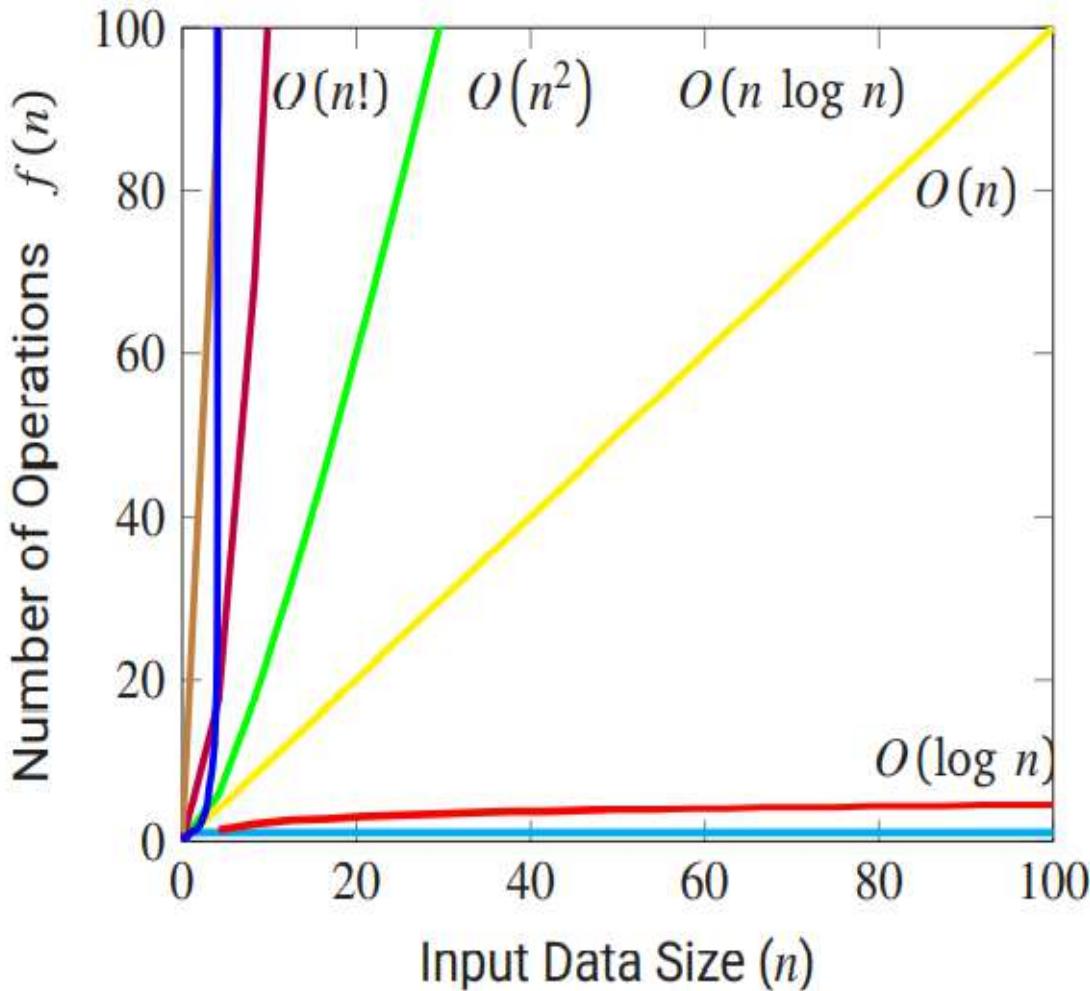
- Time Complexity $O(n^2)$.
- Growth rate of n^2 .
- Performance is directly proportional to the squared size of the input.
- If the input is size 2, it will do four operations.
- With quadratic complexity execution time increases at an increasing rate.
- It is slower than logarithmic time.

Exponential Time Complexity



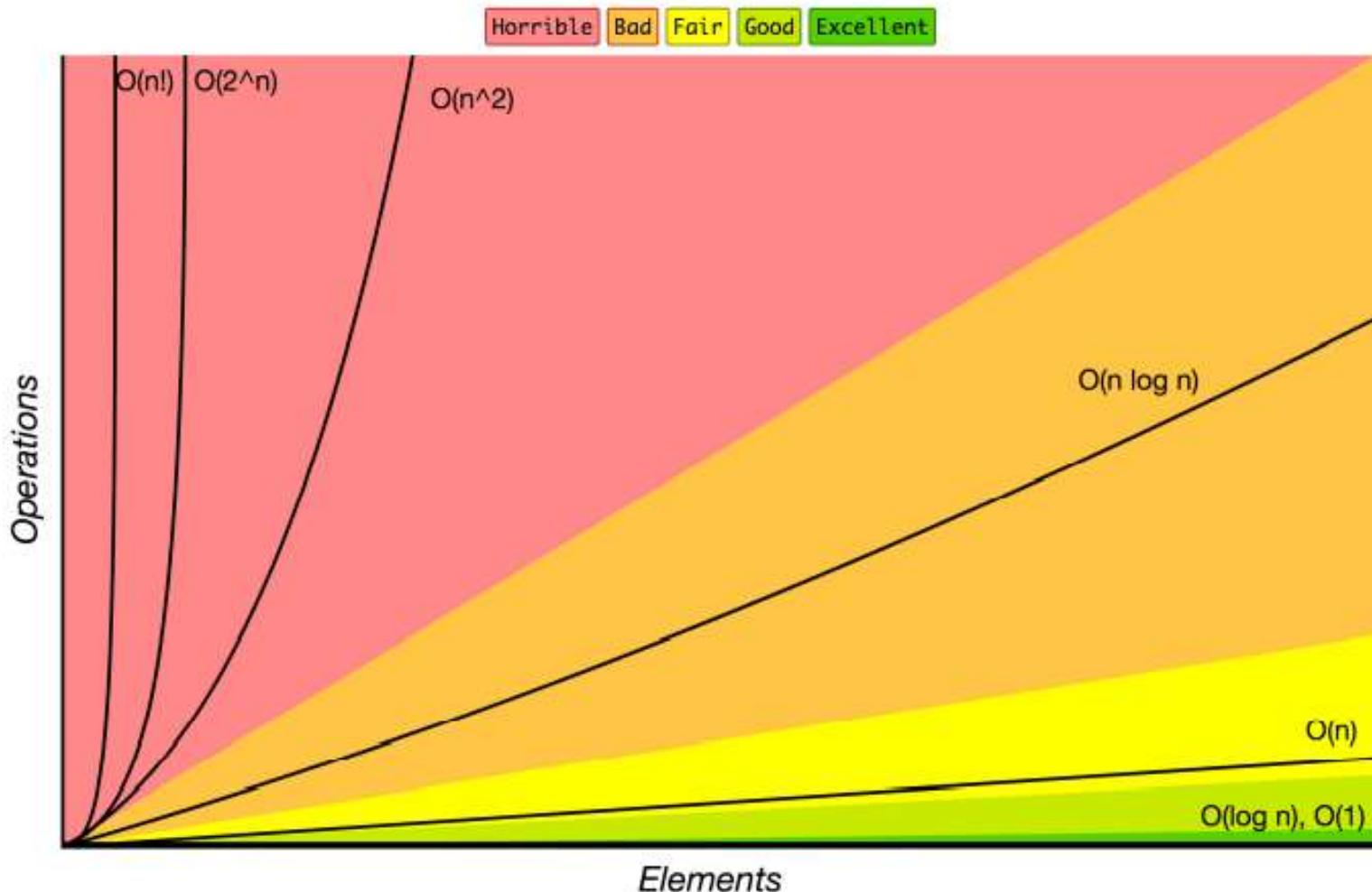
- Time Complexity $O(2^n)$.
- Denotes an algorithm whose growth doubles anytime an input unit increases by 1.
- Exponential time complexity is usually seen in Brute-Force algorithms.
- When the input is big enough, resource consumption will quickly approach infinity.

Order of Growth



- $O(1)$: constant
- $O(\log n)$: logarithmic
- $O(n)$: linear
- $O(n \log n)$: linearithmic
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(2^n)$: exponential
- $O(n!)$: factorial
- $O(n^c)$: polynomial

Order of Growth II



Order of Growth and Time

Some order magnitude on a specific (not generalizable) example:

- $O(1) \implies T(n) = 1 \text{ second.}$
- $O(\log n) \implies T(n) = (1 \cdot \log 16) / (\log 8) = 4/3 \text{ seconds.}$
- $O(n) \implies T(n) = (1 \cdot 8) = 2 \text{ seconds.}$
- $O(n \log n) \implies T(n) = (1 \cdot 16 \cdot \log 16) / (8 \cdot \log 8) = 3 \text{ seconds.}$
- $O(n^2) \implies T(n) = (1 \cdot 16^2) / 8^2 = 4 \text{ seconds.}$
- $O(n^3) \implies T(n) = (1 \cdot 16^3) / 8^3 = 8 \text{ seconds.}$
- $O(2^n) \implies T(n) = (1 \cdot 2^{16}) / 2^8 = 256 \text{ seconds.}$

Big-O vs Tilde

Big-O Notation

- It generally defines the upper bound of an algorithm.
- It defines the worst case.
- It shows the order of growth of algorithms.

Tilde Notation

- It defines both the upper bound and lower bound of an algorithm.
- It mostly defines the average case.
- It simply drops the lower order terms.

Proving Tilde Complexity

$$8n + 24 \sim 8n$$

$$40n + 24 \sim 40n$$

$$8n + 48 \sim 8n$$

$$2n + 56 \sim 2n$$

$$n^2 + 32n + 24 \sim n^2$$

$$4n^2 + 32n + 24 \sim 4n^2$$

$$6n^3 + 20n + 16 \sim 6n^3$$

$$6n^3 + 100n^{4/3} + 16 \sim 6n^3$$

$$6n^3 + 17n^2 \log(n) + 7n \sim 6n^3$$

Proving Big-O Complexity

$$t(n) \in O(g(n))$$

$$t(n) \leq cg(n) \quad \text{for all } n \geq n_0.$$

Using the definition of big-*O* notation:

- $7n - 2$ is $O(n)$

need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq c \cdot n$ for $n \geq n_0$.

This is true for $c = 7$ and $n_0 = 1$.

- $3n^3 + 20n^2 + 5$ is $O(n^3)$

need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$.

This is true for $c = 4$ and $n_0 = 21$.

- $3\log(n) + 5$ is $O(\log n)$

need $c > 0$ and $n_0 \geq 1$ such that $3\log(n) + 5 \leq c \cdot \log n$ for $n \geq n_0$.

This is true for $c = 8$ and $n_0 = 2$.

Tilde vs Order of Growth

Function	Tilde Approximation	Order of Growth
$n^3/6 - n^2/2 + n/3$	$\sim n^3/6$	$O(n^3)$
$n^2/2 - n/2$	$\sim n^2/2$	$O(n^2)$
$\log n + 5$	$\sim \log n$	$O(\log n)$
3	~ 3	$O(1)$

Elementary Operations

- Primitive Operations \Leftrightarrow Elementary Operations.
- A primitive operation is a basic step that can be performed in constant time.
- Including: mathematical constants and arithmetic operations.
- Including: assigning a value to a variable, as well as comparisons and tests.
- Including: evaluating an expression.
- Including: indexing into an array.
- Including: calling a method, returning from a method.
- Easily identifiable and independent from the programming language.
- Non-elementary operations: have a unit cost and bit costs.
- Non-elementary operations: for and while loop.
- Non-elementary operations: nested loops, simple recursive functions.

Elementary Operations Count

- examine a piece of code and predict the number of instructions to be executed
 - e.g. for each instruction predict how many times each will be encountered as the code runs

Inst #	Code	F.C.
1	for (int i=0; i< n ; i++)	n+1
2	{ cout << i;	n
3	p = p + i;	n
	}	—

totaling the counts produces the F.C. (frequency count)

3n+1

discarding constant terms produces : $3n$

clearing coefficients : n

Big O = O(n)

picking the most significant term: n

Elementary Operations Count II

Inst	Code	F.C.	F.C.
#	for (int i=0; i< n ; i++)	$n+1$	$n+1$
1	for (int j=0 ; j < n; j++)	$n(n+1)$	n^2+n
2	{ cout << i;	$n*n$	n^2
3	p = p + i;	$n*n$	n^2
4	}		<hr/>
			$3n^2+2n+1$

discarding constant terms produces :

$$3n^2+2n$$

clearing coefficients : n^2+n

$$\text{Big O} = O(n^2)$$

picking the most significant term: n^2

Examples of Algorithm Complexity I

```
int function(int n)
{ if (n>1)
  { return 3*function(n/2) + 1; }
}
```

```
int count=0;
for (int i=n; i>0 ; i/=2)
{ count++; }
```

```
int i=1;
while (i<n)
{ sum=sum+1;
  i=i*2;
}
```

$O(\log n)$

Examples of Algorithm Complexity II

```
void function(int n)
{
    int i,j,k;
    int count=0;

    for (i=n/2 ; i<=n ; i++)
    {
        for (j=1; j<=n ; j=j*2)
        {
            for (k=1 ; k=n; k=k*2)
                { count++; }
        }
    }
}
```

$$O(n \log n)$$

Examples of Algorithm Complexity III

```
void function(int n)
{
    int x=0;
    for (int i=1 ; i<n ; i++)
    {
        for (int j=1 ; j<=i ; j++)
        {
            for (int k=1 ; k<=j ; k++)
            {
                x=x+1;
            }
        }
    }
}
```

$$O(n^3)$$

Examples of Algorithm Complexity IV

```
int function{int n}
{
    int a=0;
    int b=0;

    if (n==0)
    {
        return 1;
    }
    else
    {
        a=function(n-1);
        b=function(n-1);
        return a+b;
    }
}
```

$$O(2^n)$$

Time Measurement Method

```
#include <chrono>
using namespace std::chrono;

int64_t millisSinceEpoch(void)
{
    steady_clock::duration dur{steady_clock::now().time_since_epoch()};
    return duration_cast<milliseconds>(dur).count();
}

int64_t microsSinceEpoch(void)
{
    steady_clock::duration dur{steady_clock::now().time_since_epoch()};
    return duration_cast<microseconds>(dur).count();
}
```

Time Measurement Method II

```
#include <chrono>
using namespace std;

auto start = chrono::steady_clock::now();
/* call your functions here */
auto end = chrono::steady_clock::now();

cout << "Microsecond: " <<
chrono::duration_cast<chrono::microseconds>(end-start).count()
<< endl;

cout << "Second: " <<
chrono::duration_cast<chrono::seconds>(end-start).count()
<< endl;
```

Count vs. Time

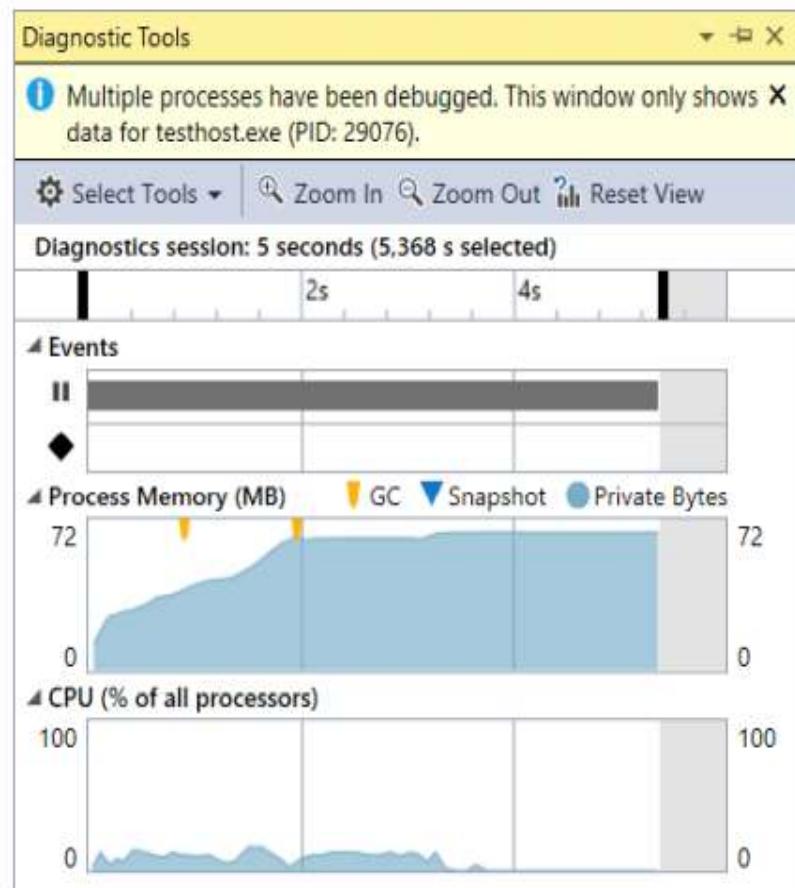
Step Count Method

The operation-count method of estimating time complexity omits accounting for the time spent on all but the chosen operations relates to the problem size. A step is any computation unit that is independent of the problem size. The success of this method depends on the ability to identify the operations that contribute most to the time complexity.

Measure Execution Time

The running time of an algorithm for a specific input depends on the number of operations executed. We measure the running time of a program or an algorithm in term of CPU (time) usage. To measure the execution time, we make a measurement in two different time instants on the clock cycles and find the difference. This depends on the machine, compiler as well as the code. We have somewhat mitigated the machine-dependency of CPU time.

Memory Usage



When you start debugging in Visual Studio by selecting *Debug > Start Debugging*, or pressing *F5*, the Diagnostic Tools window appears by default.

To open it manually, select *Debug > Windows > Show Diagnostic Tools*. The Diagnostic Tools window shows information about events, process memory, and CPU usage.

Memory Usage II

sizeof()

The `sizeof()` predefined built-in function tell us the size in bytes of whatever datatype is specified and occupied the memory. This function (also named operator) is machine dependent, and returns different values according to the specific hardware. For instance, what can take 2 bytes in most PC will take 4 bytes for mainframes and 16 bytes for supercomputers. This function is working for primitives and user-defined datatypes.

```
int a;  
int b=sizeof(a);
```

Recap

- Algorithms have different types of voracity for consuming time and memory.
- Assess algorithms with time and space complexity analysis.
- Calculate time complexity by finding the exact $T(n)$ function, the number of operations performed by an algorithm.
- Express time complexity using the Big-O notation.
- Perform simple time complexity analysis of algorithms using this notation.
- Calculating $T(n)$ is not necessary for inferring the Big-O complexity of an algorithm.
- The cost of running exponential algorithms explode and not runnable for big inputs.
- If an algorithm is too slow, would optimizing the algorithm or using a supercomputer help?

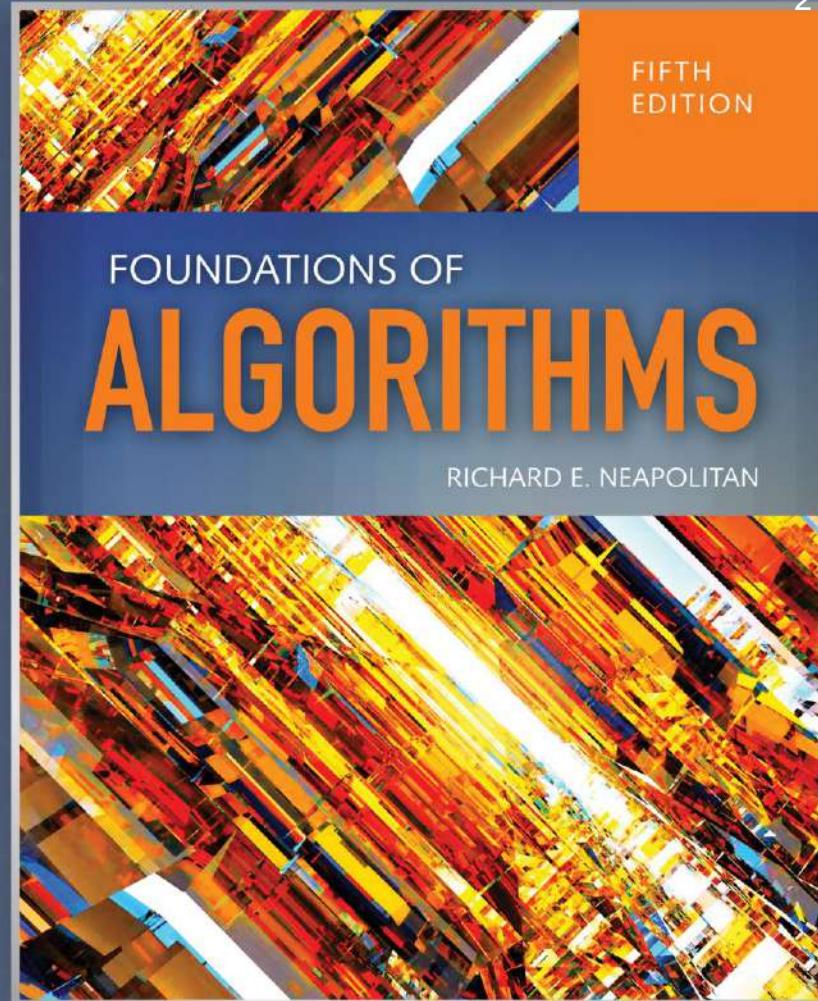
Basic algorithmic techniques

When faced with a new algorithmic problem, one should consider applying one of the following approaches:

- **Divide-and-conquer** :: divide the problem into two subproblems, solve each problem separately and merge the solutions
- **Dynamic programming** :: express the solution of the original problem as a recursion on solutions of similar smaller problems. Then instead of solving only the original problem, solve all sub-problems that can occur when the recursion is unravelled, and combine their solutions
- **Greedy approach** :: build the solution of an optimization problem one piece at a time, optimizing each piece separately
- **Inductive approach** :: express the solution of the original problem based on the solution of the same problem with one fewer item; a special case of dynamic programming and similar to the greedy approach

Divide and Conquer

Chapter 2





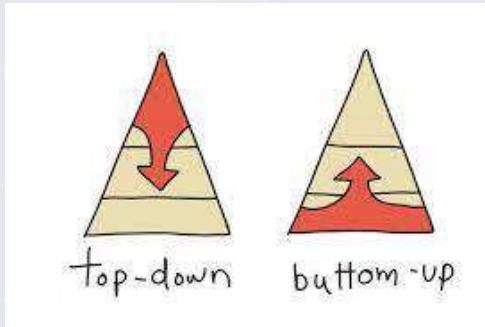
Objectives

- Describe the **divide-and-conquer approach** to solving problems
- Apply the divide-and-conquer approach to solve a problem
- Determine when the divide-and-conquer approach is an appropriate solution approach
- Determine complexity analysis of divide and conquer algorithms
- Contrast worst-case and average-case complexity analysis
- **Master Method** to analyze recursive algorithms

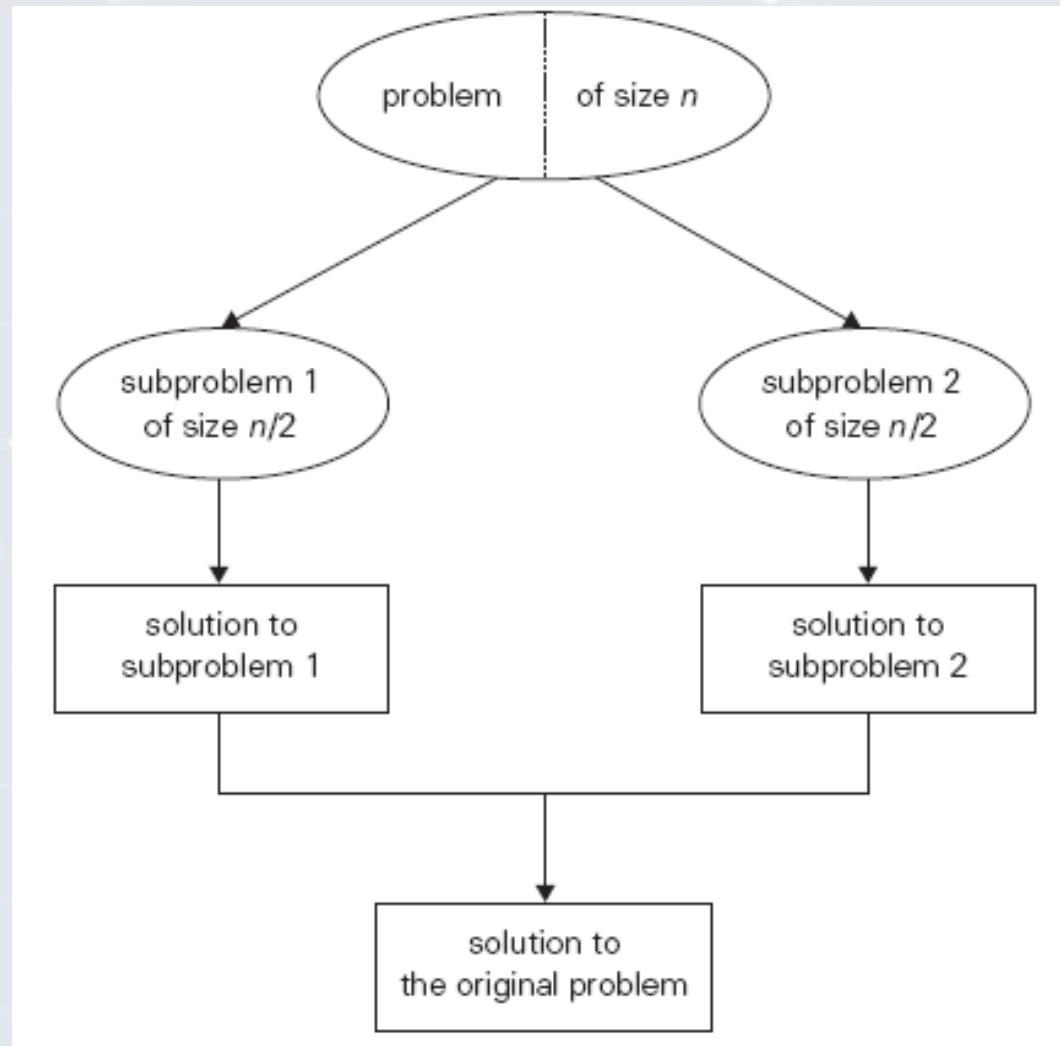
Battle of Austerlitz

December 2, 1805

- Napoleon split the Austro-Russian Army and was able to conquer 2 weaker armies
- Divide an instance of a problem into 2 or more smaller instances
- Top-down approach



Divide-and-Conquer Technique



0	4	7	10	14	23	45	47	53
---	---	---	----	----	----	----	----	----

Binary Search

- Locate key x in an array of size n sorted in non-decreasing order
- Compare x with the middle element – if equal, done – quit. Else
 - *Divide* the Array into two sub-arrays approximately half as large
 - If x is smaller than the middle item, select left sub-array
 - If x is larger than the middle item, select right sub-array



Binary Search

- *Conquer* (solve) the sub-array: Is x in the sub-array using recursion until the sub-array is sufficiently small?
- *Obtain* the solution to the array from the solution to the sub-array

Binary Search (Recursive)

Problem: Determine whether x is in the sorted array S of size n .

Inputs: positive integer n , sorted (nondecreasing order) array of keys S indexed from 1 to n , a key x .

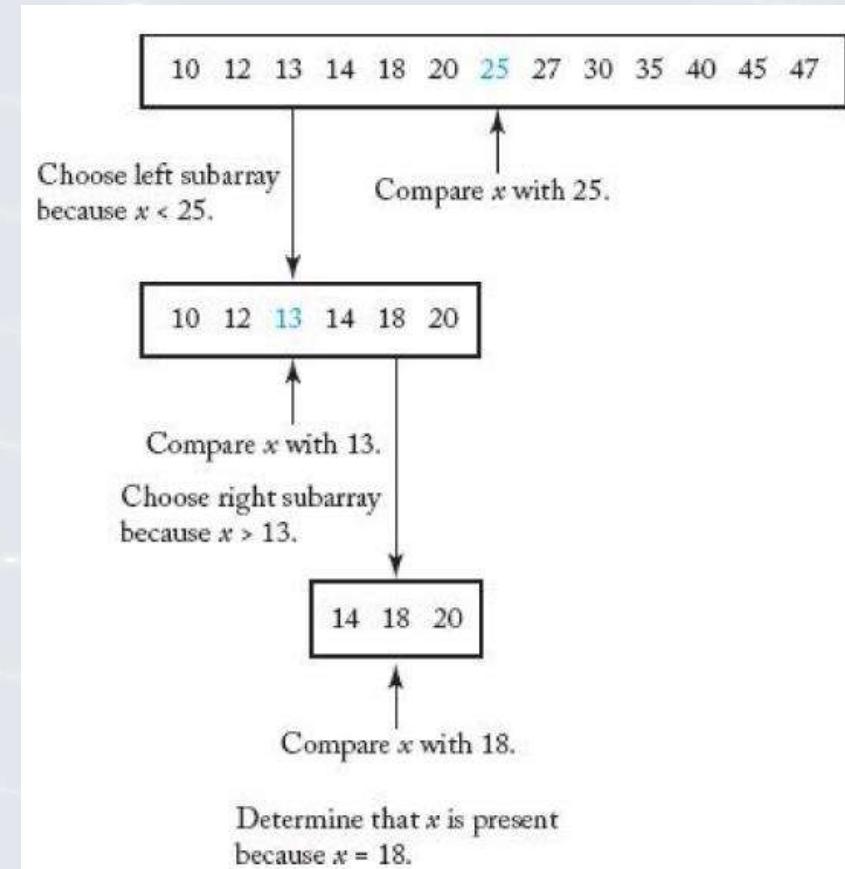
Outputs: *location*, the location of x in S (0 if x is not in S).

$x = 18$

```
index location (index low, index high)
{
    index mid;

    if (low > high)
        return 0;
    else {
        mid = ⌊(low + high)/2⌋;
        if (x == S[mid])
            return mid;
        else if (x < S[mid])
            return location(low, mid - 1);
        else
            return location(mid + 1, high);
    }
}
```

locationout = *location* (1 , n);



Worst Case Complexity Analysis

Basic operation: the comparison of x with $S[mid]$.

Input size: n , the number of items in the array.

- n is a power of 2 and $x > S[n]$
- $W(n) = W(n/2) + 1$ for $n > 1$ and n power of 2
- $W(1) = 1$
- $W(n/2)$ = the number of comparisons in the recursive call
 - 1 comparison at the top level
- Example B1 in Appendix B:
 - $W(n) = \lg n + 1$
- n not a power of 2
 - $W(n) = \lfloor \lg n \rfloor + 1 \in \Theta(\lg n)$

Example B.1

Consider the recurrence

$$\begin{cases} t_n = t_{n/2} + 1 & \text{for } n > 1, n \text{ a power of 2} \\ t_1 = 1 & \end{cases}$$

Merge sort [CLRS 2.3.1]

Merge sort is a divide-and-conquer algorithm.

Informal description:

It sorts a subarray $A[p \dots r) := A[p \dots r - 1]$

Divide by splitting it into subarrays $A[p \dots q)$ and $A[q \dots r)$
where $q = \lfloor (p + r)/2 \rfloor$.

Conquer by recursively sorting the subarrays.

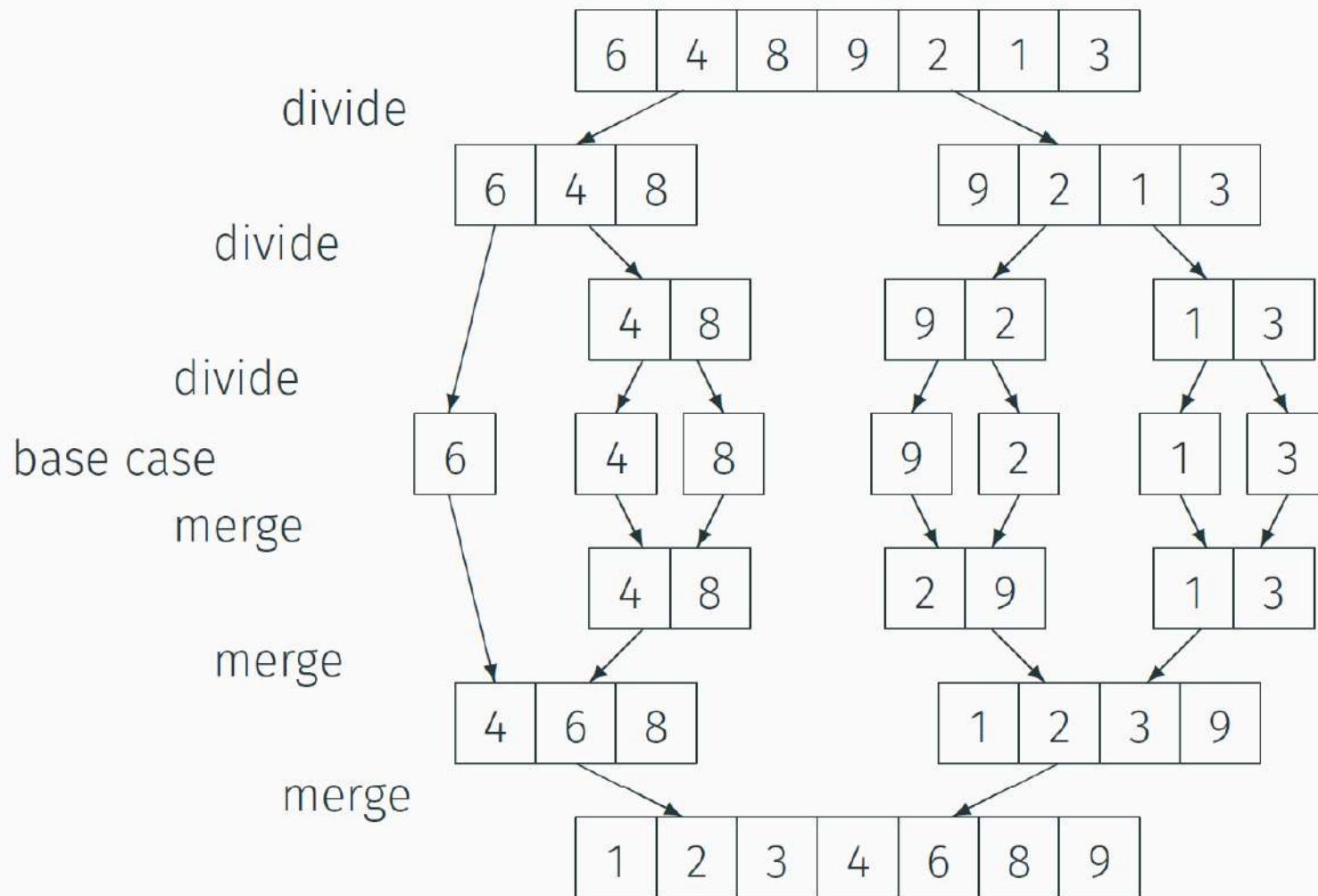
Recursion stops when the subarray contains only one element.

Combine by merging the sorted subarrays $A[p \dots q)$ and $A[q \dots r)$ into a single sorted array, using a procedure called `MERGE(A, p, q, r)`.

`MERGE` compares the two smallest elements of the two subarrays and copies the smaller one into the output array.

This procedure is repeated until all the elements in the two subarrays have been copied.

Example



Pseudocode for MERGE-SORT

MERGE-SORT(A, p, r)

Input: An integer array A with indices $p < r$.

Output: The subarray $A[p .. r)$ is sorted in non-decreasing order.

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

Initial call: MERGE-SORT($A, 1, n + 1$)

Merge

Input: Array A with indices p, q, r such that

- $p < q < r$
- Subarrays $A[p..q)$ and $A[q..r)$ are both sorted.

Output: The two sorted subarrays are merged into a single sorted subarray in $A[p..r)$.

Pseudocode for MERGE

MERGE(A, p, q, r)

```
1   $n_1 = q - p$ 
2   $n_2 = r - q$ 
3  Create array  $L$  of size  $n_1 + 1$ 
4  Create array  $R$  of size  $n_2 + 1$ 
5  for  $i = 1$  to  $n_1$ 
6     $L[i] = A[p + i - 1]$ 
7  for  $j = 1$  to  $n_2$ 
8     $R[j] = A[q + j - 1]$ 
9   $L[n_1 + 1] = \infty$ 
10  $R[n_2 + 1] = \infty$ 
```

```
11  $i = 1$ 
12  $j = 1$ 
13 for  $k = p$  to  $r - 1$ 
14   if  $L[i] \leq R[j]$ 
15      $A[k] = L[i]$ 
16      $i = i + 1$ 
17   else  $A[k] = R[j]$ 
18      $j = j + 1$ 
```

Running time of MERGE

- The first two **for** loops take $\Theta(n_1 + n_2) = \Theta(n)$ time, where $n = r - p$.
- The last **for** loop makes n iterations, each taking constant time, for $\Theta(n)$ time.
- Total time: $\Theta(n)$.

Remark

- The test in line 14 is left-biased, which ensures that MERGE-SORT is a *stable* sorting algorithm: if $A[i] = A[j]$ and $A[i]$ appears before $A[j]$ in the input array, then in the output array the element pointing to $A[i]$ appears to the left of the element pointing to $A[j]$.

Characteristics of merge sort

- The worst-case running time of MERGE-SORT is $\Theta(n \log n)$, much better than the worst-case running time of INSERTION-SORT, which was $\Theta(n^2)$.
(see next slides for the explicit analysis of MERGE-SORT).
- MERGE-SORT is stable, because MERGE is left-biased.
- MERGE and therefore MERGE-SORT is not in-place: it requires $\Theta(n)$ extra space.
- MERGE-SORT is not an online-algorithm: the whole array A must be specified before the algorithm starts running.

Analysing divide-and-conquer algorithms [CLRS 2.3.2]

We often use a **recurrence** to express the running time of a divide-and-conquer algorithm.

Let $T(n)$ = running time on a problem of size n .

- If n is small (say $n \leq \ell$), use constant-time brute force solution.
- Otherwise, we divide the problem into a subproblems, each $1/b$ the size of the original.
- Let the time to divide a size- n problem be $D(n)$.
- Let the time to combine solutions (back to that of size n) be $C(n)$.

We get the recurrence

$$T(n) = \begin{cases} c & \text{if } n \leq \ell \\ a T(n/b) + D(n) + C(n) & \text{if } n > \ell \end{cases}$$

Example: MERGE-SORT

For simplicity, assume $n = 2^k$.

For $n = 1$, the running time is a constant c .

For $n \geq 2$, the time taken for each step is:

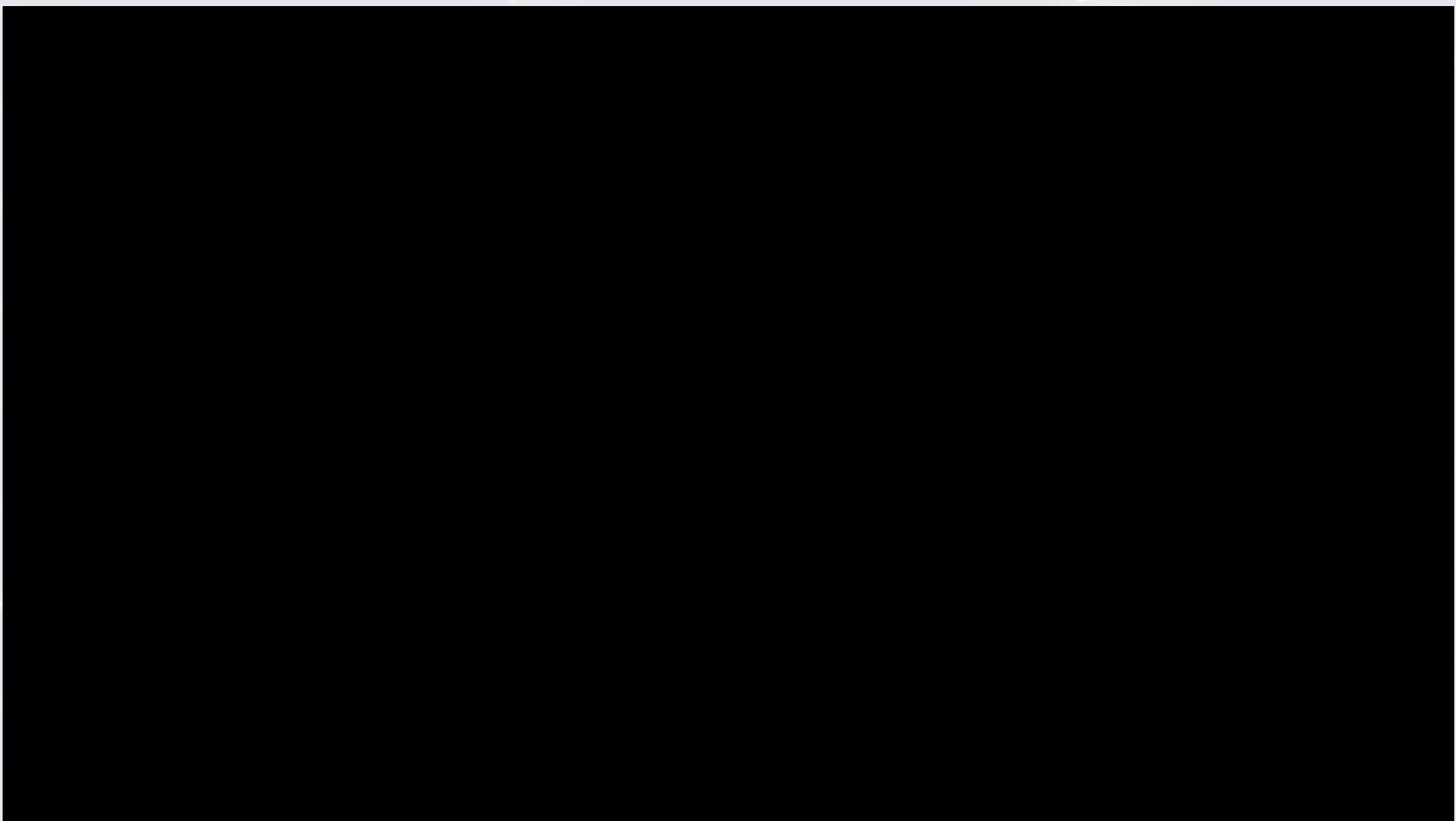
- **Divide:** Compute $q = (p + r)/2$; so, $D(n) = \Theta(1)$.
- **Conquer:** Recursively solve 2 subproblems, each of size $n/2$; so, $2T(n/2)$.
- **Combine:** MERGE two arrays of size n ; so, $C(n) = \Theta(n)$.

More precisely, the recurrence for MERGE-SORT is

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

where the function $f(n)$ is bounded as $d' n \leq f(n) \leq d n$ for suitable constants $d, d' > 0$.

Worst-Case Analysis Mergesort

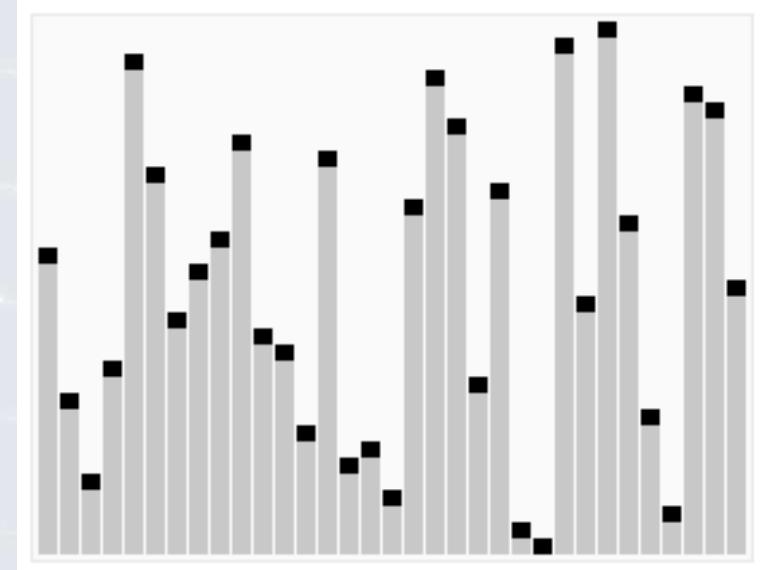


<https://www.youtube.com/watch?v=dENca26N6V4>

© Comstock Images/age fotostock. Copyright © 2014 by Jones & Bartlett Learning, LLC an Ascend Learning Company
www.jblearning.com

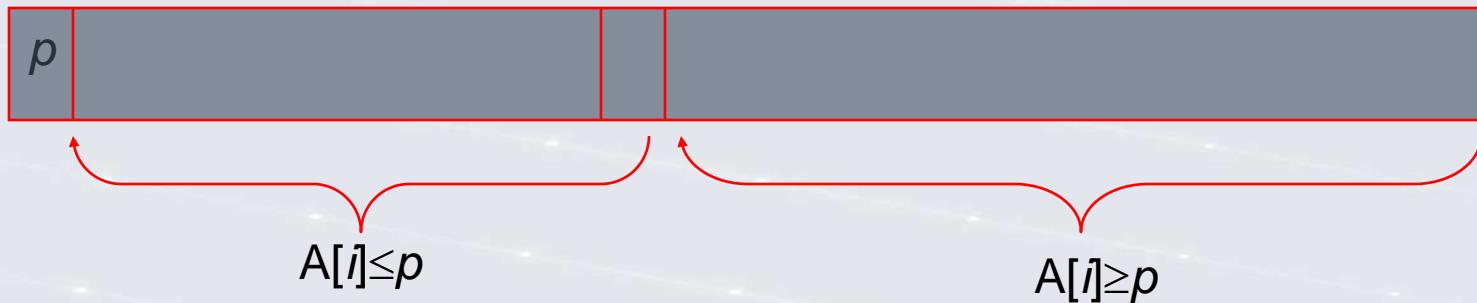
QuickSort

- Normally considered the best general purpose sort algorithm
- Implemented in many standard libraries
 - Array.Sort in C# /.net
 - Array.Sort in Java
 - qsort (C standard library)*



Quicksort

- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first s positions are smaller than or equal to the pivot and all the elements in the remaining $n-s$ positions are larger than or equal to the pivot



- Exchange the pivot with the last element in the first (i.e., \leq) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

QuickSort

Example 2.3

Suppose the array contains these numbers in sequence:

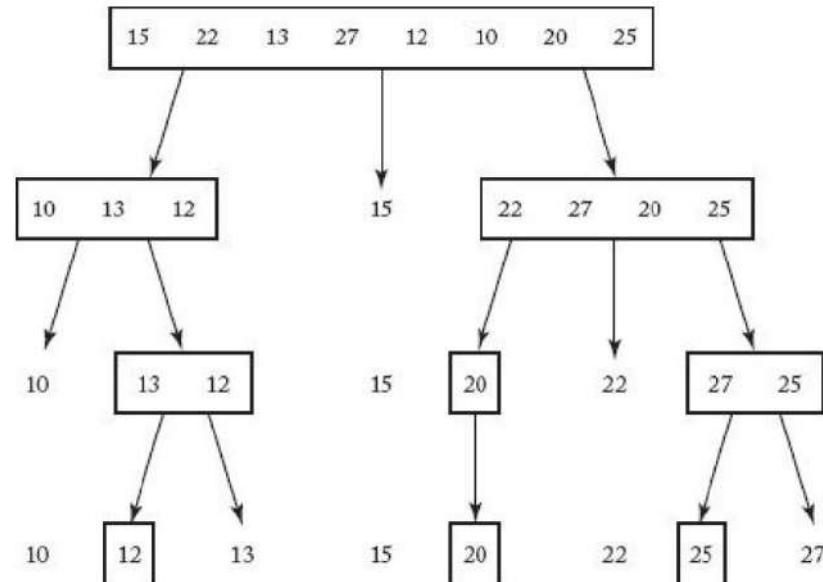
Pivot item
 ↓
 15 22 13 27 12 10 20 25

1. Partition the array so that all items smaller than the pivot item are to the left of it and all items larger are to the right:

Pivot item
 ↓
 10 13 12 15 22 27 20 25
 All smaller All larger

2. Sort the subarrays:

Pivot item
 ↓
 10 12 13 15 20 22 25 27
 Sorted Sorted



Quicksort

Problem: Sort n keys in nondecreasing order.

23

Inputs: positive integer n , array of keys S indexed from 1 to n .

Outputs: the array S containing the keys in nondecreasing order.

```
quicksort(1, n);
```

```
void quicksort(index low, index high)
{
    index pivotpoint;

    if (high > low){
        partition(low, high, pivotpoint);
        quicksort(low, pivotpoint - 1);
        quicksort(pivotpoint + 1, high);
    }
}
```

Partition

Problem: Partition the array S for Quicksort.

Inputs: two indices, low and $high$, and the subarray of S indexed from low to $high$.

Outputs: $pivotpoint$, the pivot point for the subarray indexed from low to $high$.

```
void partition(index low, index high,
               index& pivotpoint)
{
    index i, j;
    keytype pivotitem;

    pivotitem = S[low];
    j = low;
    for (i = low + 1; i <= high; i++)
        if (S[i] < pivotitem){
            j++;
            exchange S[i] and S[j];
        }
    pivotpoint = j;
    exchange S[low] and S[pivotpoint];
}
```



Basic operation

- Comparison of $S[i]$ with pivot item
- Input size n

Worst-Case Complexity Analysis of Quicksort

- Array is sorted in non-decreasing order
- Array is repeatedly sorted into an empty sub-array which is less than the pivot and a sub-array of $n-1$ containing items greater than pivot
- If there are k keys in the current sub-array, $k-1$ key comparisons are executed

Worst-Case Complexity Analysis of Quicksort

- $T(n)$ is specified because analysis is for the every-case complexity for the class of instances already sorted in non-decreasing order
- $T(n) = \text{time to sort left sub-array} + \text{time to sort right sub-array} + \text{time to partition}$
- $T(n) = T(0) + T(n-1) + n - 1$
- $T(n) = T(n - 1) + n - 1$ for $n > 0$
- $T(0) = 0$
- From B16
 - $T(n) = n(n-1)/2$

Example B.16

We solve the recurrence

$$\begin{aligned}t_n - t_{n-1} &= n - 1 && \text{for } n > 0 \\t_0 &= 0\end{aligned}$$



Worst Case

- At least $n(n-1)/2$
- Use induction to show it is the worst case
 - $W(n) \leq n(n-1)/2$

the worst-case time complexity is given by

$$W(n) = \frac{n(n-1)}{2} \in \Theta(n^2).$$

Average-Case Time Complexity of Quicksort

- Value of the pivot point is equally likely to be any of the numbers from 1 to n
- Average obtained is the average sorting time when every possible ordering is sorted the same number of times
- $A_n = \frac{1}{n} \sum_{p=1}^n (A(p-1) + A(n-p)) + n - 1$

$$A(n) = \sum_{p=1}^n \frac{1}{n} (A(p-1) + A(n-p)) + n - 1$$

- 1/n is the probability pivot point is p
- Solving equation and using B22
 - $A(n) \in \Theta(n^* \lg n)$**

Analysis of Quicksort

29

- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n^2)$
- Average case: random arrays — $\Theta(n \log n)$

- Improvements:
 - better pivot selection: median of three partitioning
 - switch to insertion sort on small subfiles
 - elimination of recursion

These combine to 20-25% improvement

- Considered the method of choice for internal sorting of large files ($n \geq 10000$)

Multiplication of Large Integers

30

Consider the problem of multiplying two (large) n -digit integers represented by arrays of their digits such as:

$$A = 12345678901357986429 \quad B = 87654321284820912836$$

The grade-school algorithm:

$$\begin{array}{r} a_1 \ a_2 \dots \ a_n \\ b_1 \ b_2 \dots \ b_n \\ \hline (d_{10}) \ d_{11} \ d_{12} \dots \ d_{1n} \\ (d_{20}) \ d_{21} \ d_{22} \dots \ d_{2n} \\ \dots \dots \dots \dots \dots \\ (d_{n0}) \ d_{n1} \ d_{n2} \dots \ d_{nn} \\ \hline \end{array}$$

Efficiency: n^2 one-digit multiplications



Multiplications of Large Integers

$$\underbrace{567,832}_{6 \text{ digits}} = \underbrace{567}_{3 \text{ digits}} \times 10^3 + \underbrace{832}_{3 \text{ digits}}$$

$$\underbrace{9,423,723}_{7 \text{ digits}} = \underbrace{9423}_{4 \text{ digits}} \times 10^3 + \underbrace{723}_{3 \text{ digits}}$$

In general, if n is the number of digits in the integer u , we will split the integer into two integers, one with $\lceil n/2 \rceil$ and the other with $\lfloor n/2 \rfloor$, as follows:

$$\underbrace{u}_{n \text{ digits}} = \underbrace{x}_{\lceil n/2 \rceil \text{ digits}} \times 10^m + \underbrace{y}_{\lfloor n/2 \rfloor \text{ digits}}$$

With this representation, the exponent m of 10 is given by

$$m = \left\lfloor \frac{n}{2} \right\rfloor.$$

If we have two n -digit integers

$$\begin{aligned} &= (567 \times 10^3 + 832) (9423 \times 10^3 + 723) \\ &= 567 \times 9423 \times 10^6 + (567 \times 723 + 9423 \times 832) \\ &\quad \times 10^3 + 832 \times 723 \end{aligned}$$

their product is given by

$$\begin{aligned} uv &= (x \times 10^m + y)(w \times 10^m + z) \\ &= xw \times 10^{2m} + (xz + wy) \times 10^m + yz. \end{aligned}$$

First Divide-and-Conquer Algorithm

32

A small example: $A * B$ where $A = 2135$ and $B = 4014$

$$A = (21 \cdot 10^2 + 35), \quad B = (40 \cdot 10^2 + 14)$$

$$\begin{aligned} \text{So, } A * B &= (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14) \\ &= 21 * 40 \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + 35 * 14 \end{aligned}$$

In general, if $A = A_1 A_2$ and $B = B_1 B_2$ (where A and B are n -digit, A_1, A_2, B_1, B_2 are $n/2$ -digit numbers),

$$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$$

Recurrence for the number of one-digit multiplications $M(n)$:

$$M(n) = 4M(n/2), \quad M(1) = 1$$

Solution: $M(n) = n^2$

Large Integer Multiplication

33

Problem: Multiply two large integers, u and v .

Inputs: large integers u and v .

Outputs: $prod$, the product of u and v .

```
large_integer prod (large_integer u, large_integer v)
{
    large_integer x, y, w, z;
    int n, m;

    n = maximum( number of digits in u, number of digits in v)
    if (u == 0 || v == 0)
        return 0;
    else if (n <= threshold)
        return u × v obtained in the usual way;
    else{
        m = ⌊n/2⌋;
        x = u divide  $10^m$ ; y = u rem  $10^m$ ;
        w = v divide  $10^m$ ; z = v rem  $10^m$ ;
        return prod(x,w) ×  $10^{2m}$  + (prod(x,z) + prod(w,y)) ×  $10^m$  + prod(y,z);
    }
}
```

Second Divide-and-Conquer Algorithm

34

$$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$$

The idea is to decrease the number of multiplications from 4 to 3:

$$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2,$$

i.e., $(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2$,
which requires only 3 multiplications at the expense of (4-1) extra add/sub.

Recurrence for the number of multiplications $M(n)$:

$$M(n) = 3M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$

Large Integer Multiplication 2

Problem: Multiply two large integers, u and v .

35

Inputs: large integers u and v .

Outputs: $prod2$, the product of u and v .

```
large_integer prod2 (large_integer u, large_integer v)
{
    large_integer x, y, w, z, r, p, q;
    int n, m;

    n = maximum( number of digits in u, number of digits in v );
    if ( u == 0 || v == 0 )
        return 0;
    else if ( n <= threshold )
        return u × v obtained in the usual way;
    else {
        m = ⌊n/2⌋;
        x = u divide  $10^m$ ; y = u rem  $10^m$ ;
        w = v divide  $10^m$ ; z = v rem  $10^m$ ;
        r = prod2(x + y, w + z);
        p = prod2(x, w);
        q = prod2(y, z);
        return p ×  $10^{2m}$  + (r - p - q) ×  $10^m$  + q;
    }
}
```

Strassen's Matrix Multiplication

Strassen observed [1969] that the product of two matrices can be computed as follows:

Suppose we want the product C of two 2×2 matrices, A and B . That is,

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

Strassen determined that if we let

$$m_1 = (a_{11} + a_{22})(b_{11} + b_{22})$$

$$m_2 = (a_{21} + a_{22})b_{11}$$

$$m_3 = a_{11}(b_{12} - b_{22})$$

$$m_4 = a_{22}(b_{21} - b_{11})$$

$$m_5 = (a_{11} + a_{12})b_{22}$$

$$m_6 = (a_{21} - a_{11})(b_{11} + b_{12})$$

$$m_7 = (a_{12} - a_{22})(b_{21} + b_{22}),$$

the product C is given by

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}.$$

Strassen's Matrix Multiplication

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix} \quad B = \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix}.$$

$$\begin{array}{c} \xleftarrow{n/2} \\ \uparrow \\ \left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array} \right] \end{array} = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right] \times \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array} \right]$$

$$M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$

$$\begin{aligned} &= \left(\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \right) \times \left(\begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} \right) \\ &= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix}. \end{aligned}$$

$$\begin{array}{c} \xleftarrow{2} \\ \uparrow \\ \left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array} \right] \end{array} = \left[\begin{array}{c|c|c|c} 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 1 & 2 & 3 \\ \hline 4 & 5 & 6 & 7 \end{array} \right] \times \left[\begin{array}{c|c|c|c} 8 & 9 & 1 & 2 \\ \hline 3 & 4 & 5 & 6 \\ \hline 7 & 8 & 9 & 1 \\ \hline 2 & 3 & 4 & 5 \end{array} \right]$$

$$= \begin{bmatrix} 3 \times 17 + 5 \times 7 & 3 \times 10 + 5 \times 9 \\ 11 \times 17 + 13 \times 7 & 11 \times 10 + 13 \times 9 \end{bmatrix} = \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix}.$$

Strassen

Problem: Determine the product of two $n \times n$ matrices where n is a power of 2.

Inputs: an integer n that is a power of 2, and two $n \times n$ matrices A and B .

Outputs: the product C of A and B .

```
void strassen (int n
                 $n \times n$ -matrix A,
                 $n \times n$ -matrix B,
                 $n \times n$ -matrix& C)
{
    if ( $n \leq threshold$ )
        compute  $C = A \times B$  using the standard algorithm;
    else{
        partition  $A$  into four submatrices  $A_{11}$ ,  $A_{12}$ ,  $A_{21}$ ,  $A_{22}$ ;
        partition  $B$  into four submatrices  $B_{11}$ ,  $B_{12}$ ,  $B_{21}$ ,  $B_{22}$ ;
        compute  $C = A \times B$  using Strassen's method;
        // example recursive call:
        // strassen( $n/2$ ,  $A_{11} + A_{22}$ ,  $B_{11} + B_{22}$ ,  $M_1$ );
    }
}
```

Analysis of Strassen's Algorithm

If n is not a power of 2, matrices can be padded with zeros.

Number of multiplications:

$$M(n) = 7M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$ vs. n^3 of brute-force alg.

Algorithms with better asymptotic efficiency are known but they are even more complex.

Summary

40

Recursive approach

$$\text{Write } x = 10^{n/2}a + b \quad y = 10^{n/2}c + d$$

[where a,b,c,d are n/2 – digit numbers]

So :

$$x \cdot y = 10^n ac + 10^{n/2}(ad + bc) + bd \quad (*)$$

Algorithm#1 : recursively compute ac,ad,bc,bd, then compute (*) in the obvious way.

$T(n)$ = maximum number of operations this algorithm needs to multiply two n-digit numbers

Recurrence : express $T(n)$ in terms of running time of recursive calls.

Base Case : $T(1) \leq$ a constant.

For all $n > 1$: $T(n) \leq 4T(n/2) + O(n)$

Work done by recursive calls

Work done here

Algorithm #2 (Gauss) : recursively compute ac , bd , $(a+b)(c+d)^{(3)}$ [recall $ad+bc = (3) - (1) - (2)$]

New Recurrence :

Base Case : $T(1) \leq$ a constant

For all $n > 1$: $T(n) \leq 3T(n/2) + O(n)$

Work done here

Work done by recursive calls

Solving recurrence equations

We will consider three methods for solving recurrence equations:

1. Guess-and-test (called the substitution method in [CLRS])
2. Recursion tree
3. Master Theorem
4. By changing variables

Guess-and-test [CLRS 4.3]

- Guess an expression for the solution. The expression can contain constants that will be determined later.
- Use induction to find the constants and show that the solution works.

Let us apply this method to MERGE-SORT.

The recurrence of MERGE-SORT implies that there exist two constants $c, d > 0$ such that

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

Guess. There is some constant $a > 0$ such that $T(n) \leq an \lg n$ for all $n \geq 2$ that are powers of 2.

Let's test it!

Solving the MERGE-SORT recurrence by guess-and-test

Test. For $n = 2^k$, by induction on k .

Base case: $k = 1$

$$T(2) = 2c + 2d \leq a2 \lg 2 \quad \text{if } a \geq c + d$$

Inductive step: assume $T(n) \leq an \log n$ for $n = 2^k$.

Then, for $n' = 2^{k+1}$ we have:

$$\begin{aligned} T(n') &\leq 2a \frac{n'}{2} \lg \left(\frac{n'}{2}\right) + d n' \\ &= an' \lg n' - an' \lg 2 + d n' \\ &\leq an' \lg n' \quad \text{if } a \geq d \end{aligned}$$

In summary: choosing $a \geq c + d$ ensures $T(n) \leq an \log n$,
and thus $T(n) = O(n \log n)$.

A similar argument can be used to show that $T(n) = \Omega(n \log n)$.
Hence, $T(n) = \Theta(n \log n)$.

The recursion tree [CLRS 4.4]

Guess-and-test is great, but how do we guess the solution?
One way is to use the **recursion tree**,
which exposes successive unfoldings of the recurrence.

The idea is well exemplified in the case of MERGE-SORT.
The recurrence is

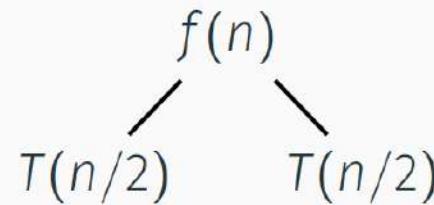
$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + f(n) & \text{if } n > 1 \end{cases}$$

where the function $f(n)$ satisfies the bounds $d' n \leq f(n) \leq d n$, for suitable constants $d, d' > 0$.

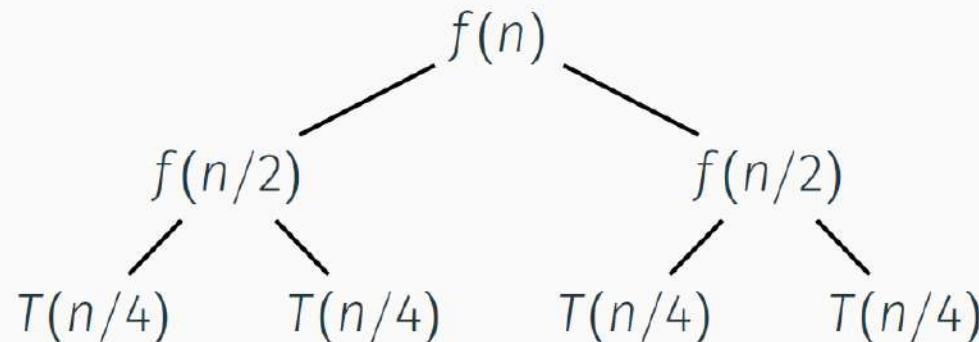
Unfolding the recurrence of MERGE-SORT

Assume $n = 2^k$ for simplicity.

First unfolding: cost of $f(n)$ plus cost of two subproblems of size $n/2$

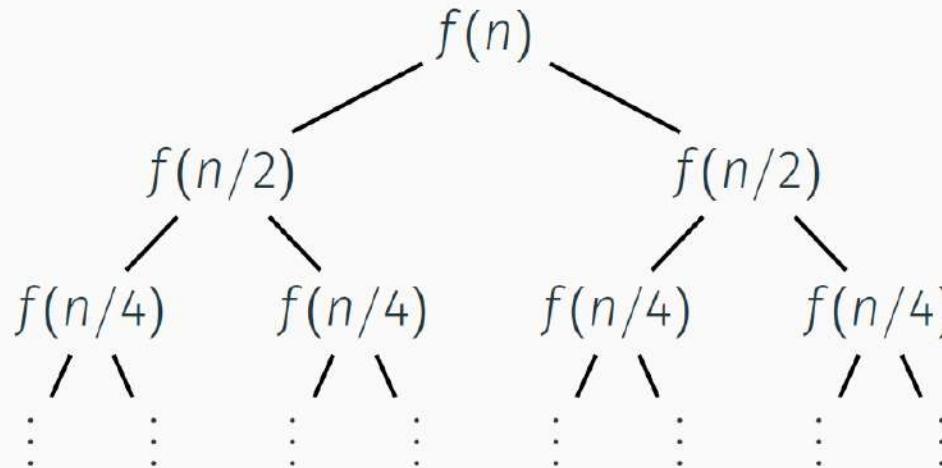


Second unfolding: for each size- $n/2$ subproblem, cost of $f(n/2)$ plus cost of two subproblems of size $n/4$ each.



Unfolding the recurrence of MERGE-SORT (cont'd)

Continue unfolding, until the problem size (= node label) gets down to 1:



In total, there are $\lg n + 1$ levels.

- Level 0 (root) has cost $C_0(n) = f(n)$.
- Level 1 has cost $C_1(n) = 2f(n/2)$.
- Level 2 has cost $C_2(n) = 4f(n/4)$.
- For $l < \lg n$, level l has cost $C_l(n) = 2^l f(n/2^l)$.
Note that, since $d' n \leq f(n) \leq d n$, we have $d' n \leq C_l(n) \leq d n$.
- The last level (consisting of n leaves) has cost cn .

Analysing MERGE-SORT with the recursion tree

The total cost of the algorithm is the sum of the costs of all levels:

$$T(n) = \sum_{l=0}^{\lg n - 1} C_l(n) + c n .$$

Using the relation $d' n \leq C_l(n) \leq dn$ for $l < \lg n$, we obtain the bounds

$$d' n \lg n + c n \leq T(n) \leq dn \lg n + cn .$$

Hence, $T(n) = \Theta(n \log n)$.

The Master Method

48

Cool Feature : a “black box” for solving recurrences.

Assumption : all subproblems have equal size.

The Master Method – recurrence format

49

1. Base Case : $T(n) \leq$ a constant for all sufficiently small n
2. For all larger n :

$$T(n) \leq aT(n/b) + O(n^d)$$

where

a = number of recursive calls (≥ 1)

b = input size shrinkage factor (> 1)

d = exponent in running time of “combine step” (≥ 0)

[a,b,d independent of n]

The Master Method

50

Base doesn't matter (only changes leading constants)

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad (\text{Case 1}) \\ O(n^d) & \text{if } a < b^d \quad (\text{Case 2}) \\ O(n^{\log_b a}) & \text{if } a > b^d \quad (\text{Case 3}) \end{cases}$$

Base matters

a = number of recursive calls (≥ 1)

b = input size shrinkage factor (> 1)

d = exponent in running time of “combine step” (≥ 0)

The Master Method

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \text{ (Case 1)} \\ O(n^d) & \text{if } a < b^d \text{ (Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \text{ (Case 3)} \end{cases}$$

Where are the respective values of a, b, d for a binary search of a sorted array, and which case of the Master Method does this correspond to?

- 1, 2, 0 [Case 1]
- 1, 2, 1 [Case 2]
- 2, 2, 0 [Case 3]
- 2, 2, 1 [Case 1]

Merge Sort

$$\left. \begin{array}{l} a = 2 \\ b = 2 \\ d = 1 \end{array} \right\} b^d = a \Rightarrow \text{Case 1}$$

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

Integer Multiplication Algorithm # 1

$$\left. \begin{array}{l} a = 4 \\ b = 2 \\ d = 1 \end{array} \right\} b^d = 2 < a \text{ (Case 3)}$$

$$\begin{aligned} \Rightarrow T(n) &= O(n^{\log_b a}) = O(n^{\log_2 4}) \\ &= O(n^2) \end{aligned}$$

Strassen's Matrix Multiplication Algorithm

$$\left. \begin{array}{l} a = 7 \\ b = 2 \\ d = 2 \end{array} \right\} b^d = 4 < a \text{ (Case 3)}$$

$$\Rightarrow T(n) = O(n^{\log_2 7}) = O(n^{2.81})$$

\Rightarrow beats the naïve iterative algorithm !

The Master Method

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \text{ (Case 1)} \\ O(n^d) & \text{if } a < b^d \text{ (Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \text{ (Case 3)} \end{cases}$$

Where are the respective values of a, b, d for a binary search of a sorted array, and which case of the Master Method does this correspond to?

- 1, 2, 0 [Case 1] $a = b^d \Rightarrow T(n) = O(n^d \log n) = O(n \log n)$
- 1, 2, 1 [Case 2]
- 2, 2, 0 [Case 3]
- 2, 2, 1 [Case 1]

Merge Sort

$$\left. \begin{array}{l} a = 2 \\ b = 2 \\ d = 1 \end{array} \right\} b^d = a \Rightarrow \text{Case 1}$$

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

Integer Multiplication Algorithm # 1

$$\left. \begin{array}{l} a = 4 \\ b = 2 \\ d = 1 \end{array} \right\} b^d = 2 < a \text{ (Case 3)}$$

$$\begin{aligned} \Rightarrow T(n) &= O(n^{\log_b a}) = O(n^{\log_2 4}) \\ &= O(n^2) \end{aligned}$$

Strassen's Matrix Multiplication Algorithm

$$\left. \begin{array}{l} a = 7 \\ b = 2 \\ d = 2 \end{array} \right\} b^d = 4 < a \text{ (Case 3)}$$

$$\Rightarrow T(n) = O(n^{\log_2 7}) = O(n^{2.81})$$

=> beats the naïve iterative algorithm !

The Master Method – proof Case 1

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \text{ (Case 1)} \\ O(n^d) & \text{if } a < b^d \text{ (Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \text{ (Case 3)} \end{cases}$$

Assume : recurrence is

I. $T(1) \leq c$

II. $T(n) \leq aT(n/b) + cn^d$

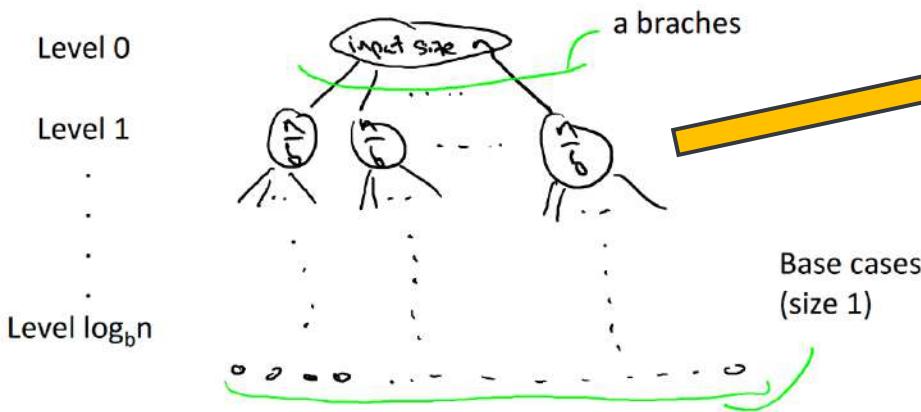
(For some constant c)

And n is a power of b .

(general case is similar, but more tedious)

Idea : generalize MergeSort analysis.
(i.e., use a recursion tree)

The Recursion Tree



© Comstock Images/age foto

Work at a Single Level

Total work at level j [ignoring work in recursive calls]

$$\leq a^j \cdot c \cdot \underbrace{\left(\frac{n}{b^j}\right)^d}_{\substack{\# \text{ of level-}j \\ \text{subproblems}}} = cn^d \cdot \left(\frac{a}{b^d}\right)^j \underbrace{\left(\frac{n}{b^j}\right)^d}_{\substack{\text{Work per level-}j \text{ subproblem} \\ \text{Size of each level-}j \text{ subproblem}}}$$

Total Work

Summing over all levels $j = 0, 1, 2, \dots, \log_b n$:

$$\text{Total work} \leq cn^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j$$

a = rate of subproblem proliferation (RSP)
 b^d = rate of work shrinkage (RWS)
 (per subproblem)

The Master Method – proof Case 1

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad (\text{Case 1}) \\ O(n^d) & \text{if } a < b^d \quad (\text{Case 2}) \\ O(n^{\log_b a}) & \text{if } a > b^d \quad (\text{Case 3}) \end{cases}$$

Total Work

Summing over all levels $j = 0, 1, 2, \dots, \log_b n$:

$$\text{Total work} \leq cn^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j$$

a = rate of subproblem proliferation (RSP)
 b^d = rate of work shrinkage (RWS)
 (per subproblem)

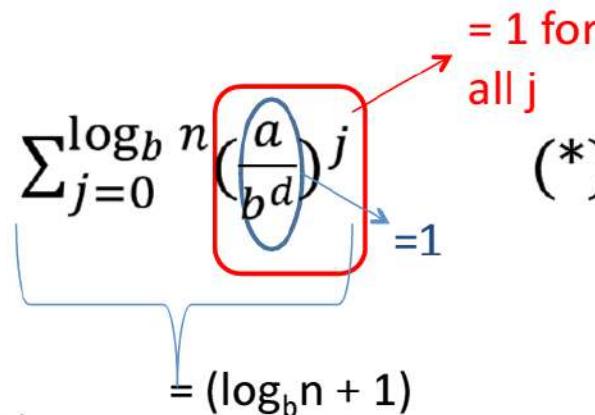
Total work: $\leq cn^d \times \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j \quad (*)$

If $a = b^d$, then

$$(*) = cn^d (\log_b n + 1)$$

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

[end Case 1]



$$\sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j = 1 \text{ for all } j$$

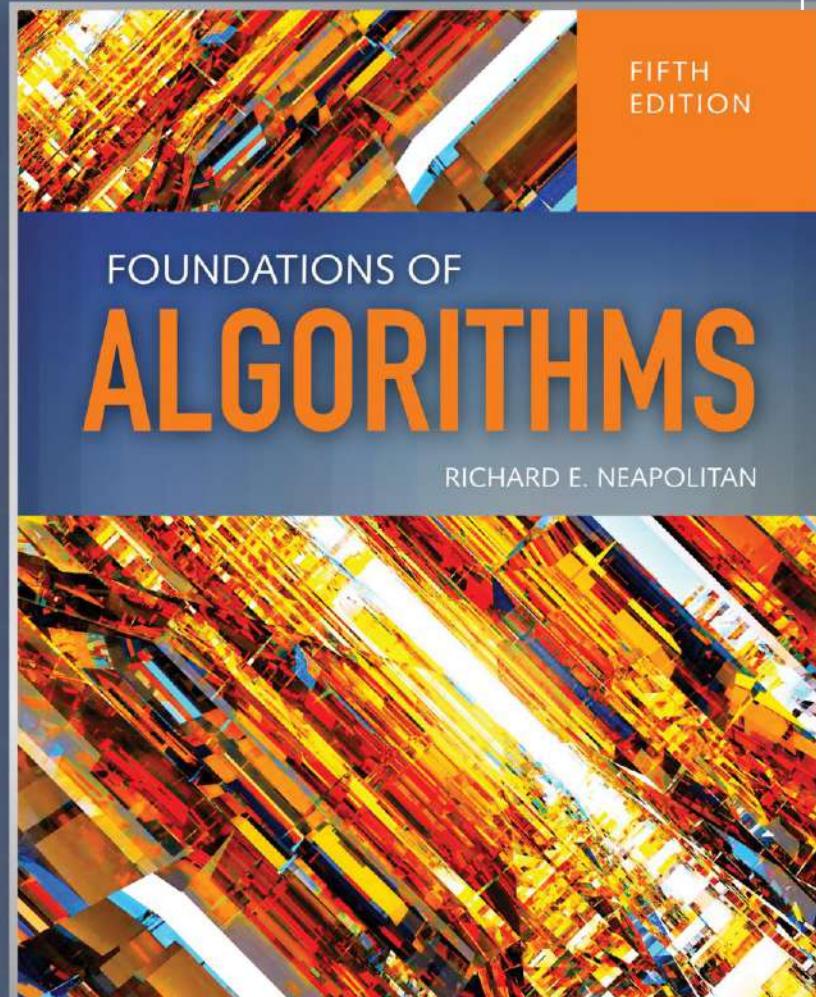
$$= (\log_b n + 1)$$

When NOT to use Divide-and-Conquer

- Determining Thresholds
 - What is the optimal threshold value of n ? ($n < 591$, $n < 128$)
- If possible we should avoid divide-and-conquer in following
 - An instance of size n is divided into two or more instances each almost of size n
 - LEADS TO EXPONENTIAL-TIME ALGORITHM
 - An instance of size n is divided into almost n instances of size n/c , where c is a constant
 - LEADS TO $n^{\Theta(\lg n)}$ ALGORITHM

Dynamic Programming

Chapter 3





Objectives

- Describe the Dynamic Programming Technique
- Contrast the Divide and Conquer and Dynamic Programming approaches to solving problems
- Identify when dynamic programming should be used to solve a problem
- Define the Principle of Optimality
- Apply the Principle of Optimality to solve Optimization Problems



Divide and Conquer

- Top-down approach to problem solving
- Blindly divide problem into smaller instances and solve the smaller instances
- Technique works efficiently for problems where smaller instances are unrelated
- Inefficient solution to problems where smaller instances are related
- Recursive solution to the Fibonacci sequence

Dynamic Programming

- Dynamic Programming is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems
- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS
- “Programming” here means “planning”
- Main idea:
 - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
 - solve smaller instances once
 - record solutions in a table
 - extract solution to the initial instance from that table
- Iterative solution to the Fibonacci Sequence

Example 1: Fibonacci numbers



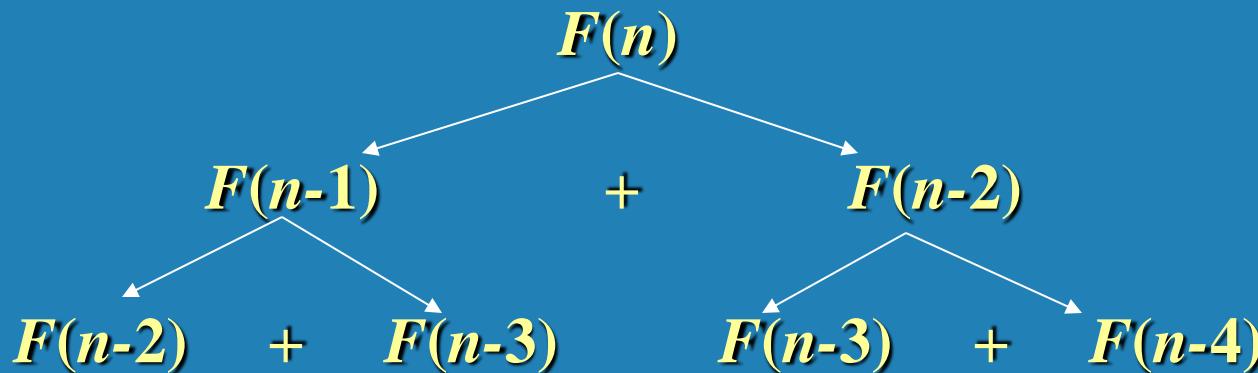
- Recall definition of Fibonacci numbers:

$$F(n) = F(n-1) + F(n-2)$$

$$F(0) = 0$$

$$F(1) = 1$$

- Computing the n^{th} Fibonacci number recursively (top-down):



Example 1: Fibonacci numbers (cont.)



Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

$$F(0) = 0$$

$$F(1) = 1$$

$$F(2) = 1+0 = 1$$

...

$$F(n-2) =$$

$$F(n-1) =$$

$$F(n) = F(n-1) + F(n-2)$$

0	1	1	...	$F(n-2)$	$F(n-1)$	$F(n)$
---	---	---	-----	----------	----------	--------

Efficiency:

- time

- space

Steps to develop a dynamic programming algorithm

1. Establish a recursive property that gives the solution to an instance of the problem
2. Compute the value of an optimal solution in a bottom-up fashion by solving smaller instances first

Divide-and-Conquer

- It partitions the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
- A divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems.

Dynamic Programming

- Dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems.
- It solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

The change-making problem [DPV, Exercise 6.17]

Change-Making Problem

Input: Positive integers $1 = x_1 < x_2 < \dots < x_n$ and v

Task: Given an unlimited supply of coins of denominations x_1, \dots, x_n , find the minimum number of coins needed to sum up to v .

Key question of dynamic programming: *What are the subproblems?*

For $0 \leq u \leq v$, compute the minimum number of coins needed to make value u , denoted as $C[u]$

For $u = v$, $C[u]$ is the solution of the original problem.

Optimal substructure: for $u \geq 1$, one has

$$C[u] = 1 + \min\{ C[u - x_i] : 1 \leq i \leq n \wedge u \geq x_i \}.$$

$C[u]$ can be computed from the values of $C[u']$ with $u' < u$.

Pseudocode for the Change-Making Problem

CHANGE-MAKING($x_1, \dots, x_n; v$)

Input: Positive integers $1 = x_1 < x_2 < \dots < x_n$ and v

Output: Minimum number of coins needed to sum up to v

```
1  $C[0] = 0$ 
2 for  $u = 1$  to  $v$ 
3    $C[u] = 1 + \min\{ C[u - x_i] : 1 \leq i \leq n \wedge u \geq x_i \}$ 
4 return  $C[v]$ 
```

Running time analysis

The array $C[1..v]$ has length v , and each entry takes $O(n)$ time to compute. Hence running time is $O(nv)$.

The Binomial Coefficient

Binomial coefficients are coefficients of the binomial formula:

$$(a + b)^n = C(n,0)a^n b^0 + \dots + C(n,k)a^{n-k} b^k + \dots + C(n,n)a^0 b^n$$

$$C(n,k) = \binom{n}{k} = \frac{n!}{k!(n-k)!} \quad \text{for } 0 \leq k \leq n.$$

For values of n and k that are not small, we cannot compute the binomial coefficient directly from this definition because $n!$ is very large even for moderate values of n . In the exercises we establish that

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \text{ or } k = n. \end{cases} \quad (3.1)$$

Recurrence: $C(n,k) = C(n-1,k) + C(n-1,k-1)$ for $n > k > 0$

$$C(n,0) = 1, \quad C(n,n) = 1 \quad \text{for } n \geq 0$$

Algorithm 3.1 Binomial Coefficient

11

Binomial Coefficient Using Divide-and-Conquer

Problem: Compute the binomial coefficient.

Inputs: nonnegative integers n and k , where $k \leq n$.

Outputs: bin , the binomial coefficient $\binom{n}{k}$.

```
int bin (int n, int k)
{
    if (k == 0 || n == k)
        return 1;
    else
        return bin(n-1, k - 1)+bin(n - 1, k);
}
```

Number of terms computed by recursive bin

Like Algorithm 1.6 (nth Fibonacci Term, Recursive), this algorithm is very inefficient. In the exercises you will establish that the algorithm computes

$$2\binom{n}{k} - 1$$

terms to determine $\binom{n}{k}$.

“

**It's hardware that
makes a machine fast.
It's software that makes
a fast machine slow.**

Doug Bruce



Dynamic Programming Solution to the Binomial Coefficient Problem

The steps for constructing a dynamic programming algorithm for this problem are as follows:

1. Establish a recursive property. This has already been done in Equality 3.1. Written in terms of B , it is

$$B\binom{i}{j} = B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & 0 < j < i \\ 1 & j = 0 \quad \text{or} \quad j = i. \end{cases}$$

2. Solve an instance of the problem in a *bottom-up* fashion by computing the rows in B in sequence starting with the first row.

- At each iteration, the values needed for that iteration have already been computed

Example

14

Compute $B[4][2] = \binom{4}{2}$.

Compute row 0: {This is done only to mimic the algorithm exactly.}

{The value $B[0][0]$ is not needed in a later computation.}

$$B[0][0] = 1$$

Compute row 1:

$$B[1][0] = 1$$

$$B[1][1] = 1$$

Compute row 2:

$$B[2][0] = 1$$

$$B[2][1] = B[1][0] + B[1][1] = 1 + 1 = 2$$

$$B[2][2] = 1$$

Compute row 3:

$$B[3][0] = 1$$

$$B[3][1] = B[2][0] + B[2][1] = 1 + 2 = 3$$

$$B[3][2] = B[2][1] + B[2][2] = 2 + 1 = 3$$

Compute row 4:

$$B[4][0] = 1$$

$$B[4][1] = B[3][0] + B[3][1] = 1 + 3 = 4$$

$$B[4][2] = B[3][1] + B[3][2] = 3 + 3 = 6$$

	0	1	2	3	4	j	k
0	1						
1		1	1				
2			1	2	1		
3				1	3	3	1
4					1	4	6
							4

A diagram showing the computation of a matrix element $B[i-1][j-1]$. An arrow points from $B[i-1][j-1]$ to $B[i-1][j]$, which then points to $B[i][j]$.

Algorithm 3.2

15

Binomial Coefficient Using Dynamic Programming

Problem: Compute the binomial coefficient.

Inputs: nonnegative integers n and k , where $k \leq n$.

$$\binom{n}{k}.$$

Outputs: $bin2$, the binomial coefficient

```
int bin2 (int n, int k)
{
    index i, j;
    int B[0..n][0..k];

    for (i = 0; i <= n; i++)
        for (j = 0; j <= minimum(i, k); j++)
            if (j == 0 || j == i)
                B[i][j] = 1;
            else
                B[i][j] = B[i-1][j-1] + B[i-1][j];
    return B[n][k];
}
```

Algorithm 3.2 Binomial Coefficient using Dynamic Programming

- The work done by bin2 as a function of n and k

for-*j* loop. The following table shows the number of passes for each value of *i*:

<i>i</i>	0	1	2	3	...	<i>k</i>	<i>k</i> + 1	...	<i>n</i>
Number of passes	1	2	3	4	...	<i>k</i> + 1	<i>k</i> + 1	...	<i>k</i> + 1

The total number of passes is therefore given by

$$1 + 2 + 3 + 4 + \dots + k + \underbrace{(k+1) + (k+1) \dots + (k+1)}_{n-k+1 \text{ times}}.$$

Applying the result in Example A.1 in [Appendix A](#), we find that this expression equals

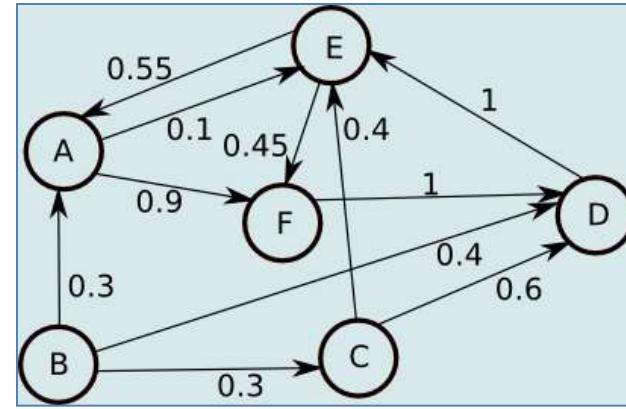
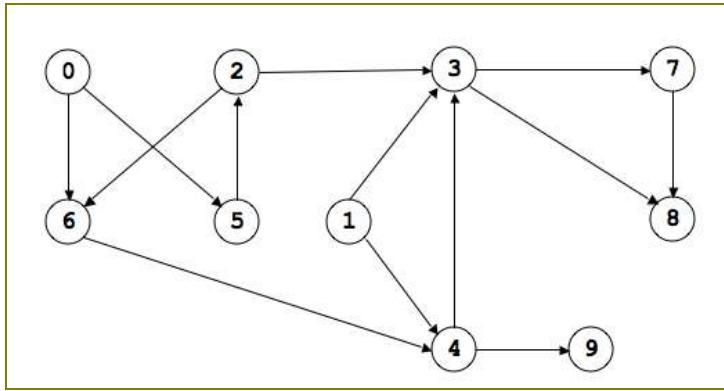
$$\frac{k(k+1)}{2} + (n-k+1)(k+1) = \frac{(2n-k+2)(k+1)}{2} \in \Theta(nk).$$



Optimization Problem

- Multiple candidate solutions
- Candidate solution has a value associated with it
- Solution to the instance is a candidate solution with an optimal value
- Minimum/Maximum

Graphs – an introduction

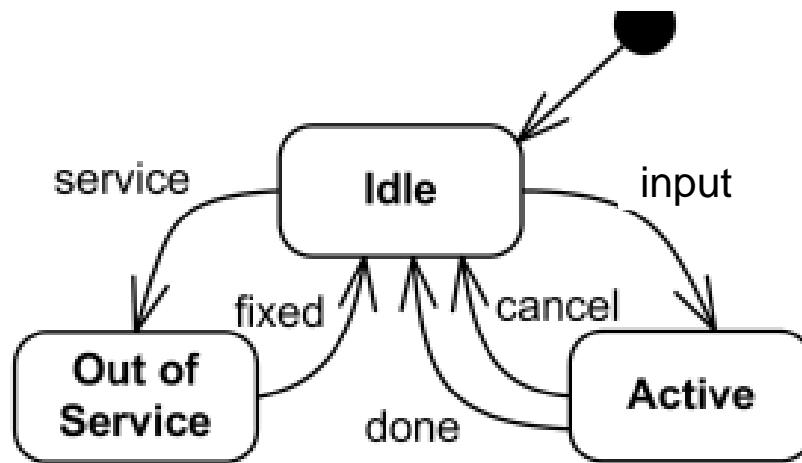


Aims

- General Terminology
 - Vertices, Edges, Connectedness, Path, Cycle etc.
- SubTypes
 - DiGraphs, Trees
- Representation techniques
 - Adjacency List, Adjacency Matrix

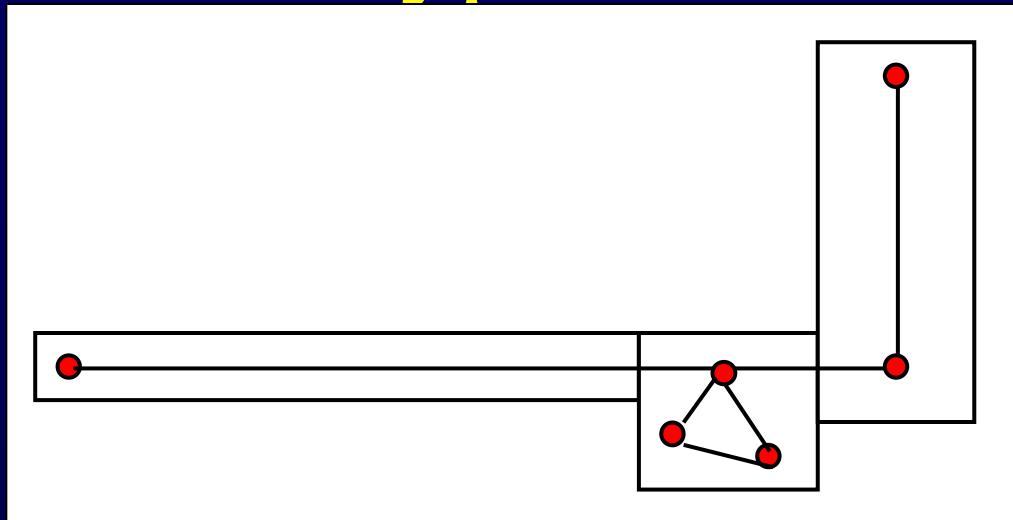
Introduction

- Graph Theory – discrete mathematics
- Used to represent relationships between objects, E.g.
 - route plan (places and connections), network, pipeline.
 - Cheapest way to lay cables between towns
 - state machines (states, transitions)



- Game - board states (positions of pieces, possible moves)
 - Plan a strategy to win a game – e.g. find quickest checkmate

Waypoints

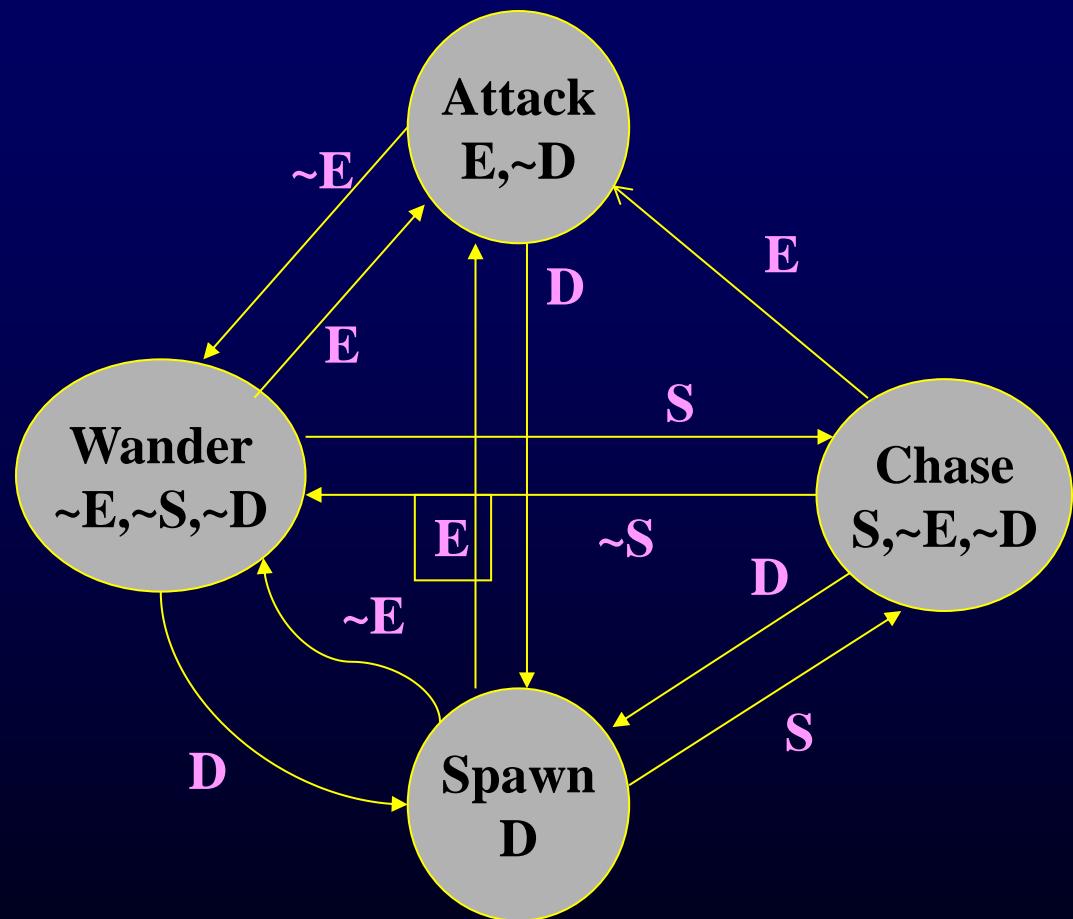


- Points in the graph/map.
- The points are connected to form a graph.
- To find a path through this map, you get to the closest point and then follow the lines to the target.

Finite State Machine - FSM

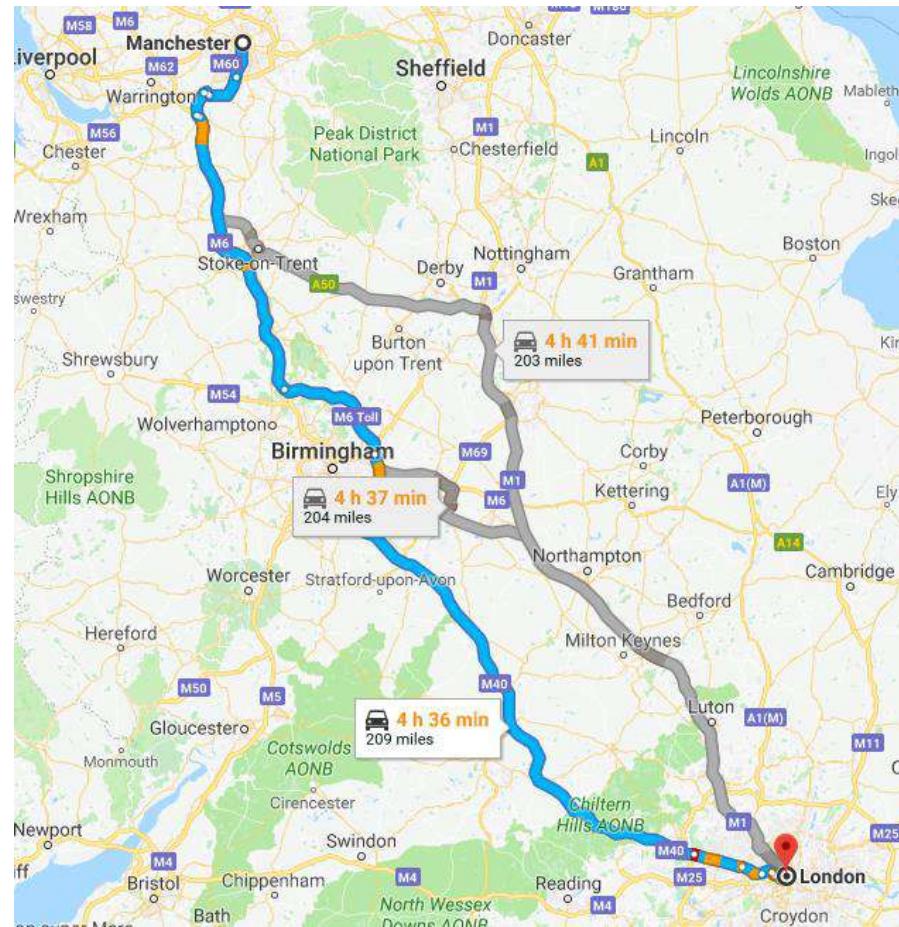


Example FSM



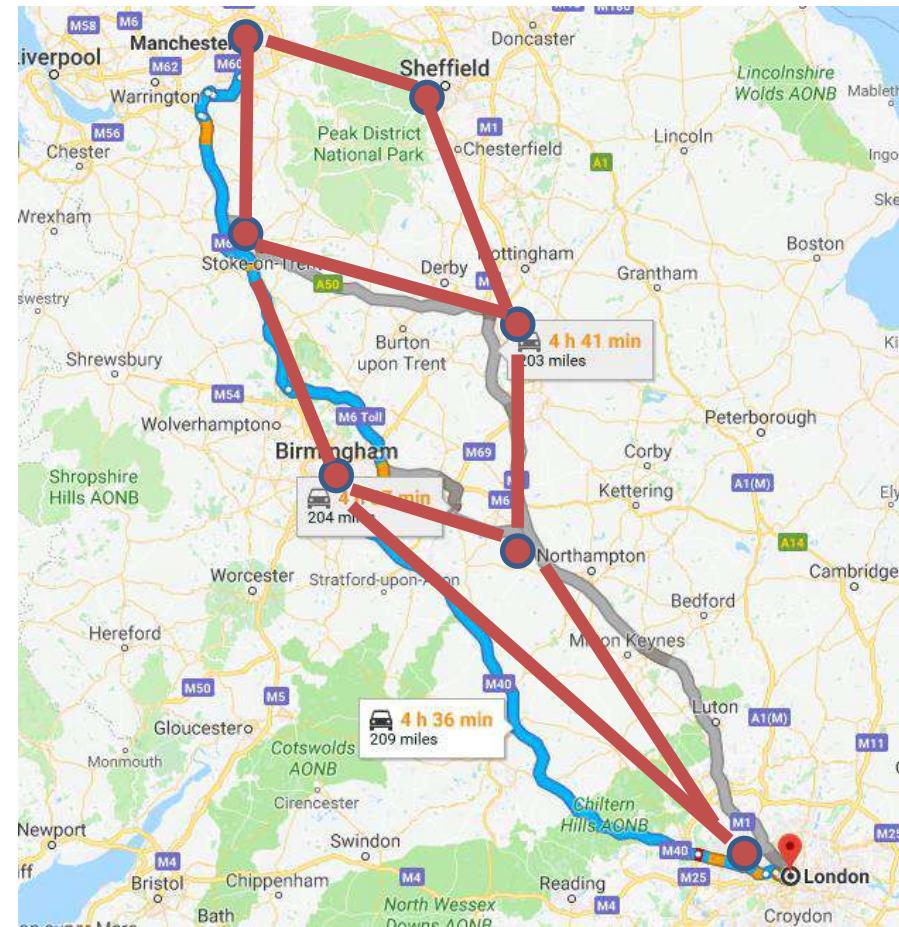
- States:
 - E : enemy in sight
 - S : sound audible
 - D : dead
- Events:
 - E : see an enemy
 - S : hear a sound
 - D : die
- Action performed:
 - On each transition
 - On each update in some states (e.g. attack)

Fastest Route



google map

Fastest Route



google map

Graph: a set of points (cities) and lines connecting the points (roads)

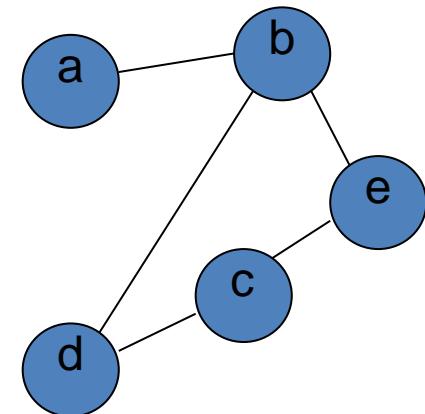
Graphs and Terminology

- Graph G is the data structure specified by the pair $G = \langle V, E \rangle$
 - V set of **vertices** (or nodes)
 - E set of (**unordered**) pairs on V called **edges** (or arcs) :
 - $(E \subset V \times V)$

e.g.

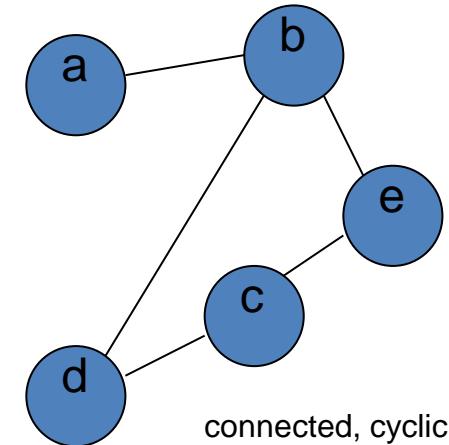
$$G_1 = \langle a, b, c, d, e, (a,b), (b,e), (e,c), (c,d), (b,d) \rangle$$

- **Connectedness**
 - If $(a,b) \in E$ then there is an edge between a and b ;
 - a and b are **adjacent**;
 - a and b are **connected**;
 - there is a **path** between a and b .

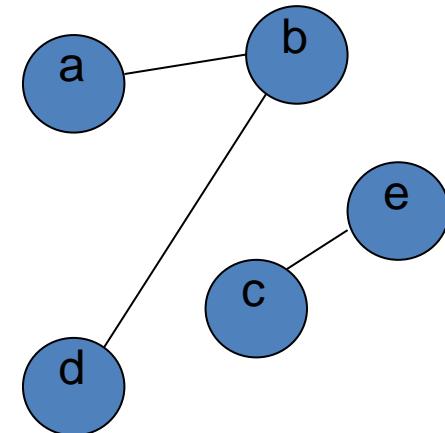


Graphs and Terminology

- If there's a path from **a** to **b**, and a path from **b** to **e**, then there is a path from **a** to **e**.
 - vertices on path must be distinct (unless first = last)
- A graph G is **connected** if there is a **path** between any given pair of vertices.
 - Otherwise its an **unconnected** graph
- A path from a vertex to at least one other node and back to itself is a **closed path** or **cycle**.
- A graph G with at least one **cycle** is a **cyclic** graph.
 - Otherwise **acyclic**



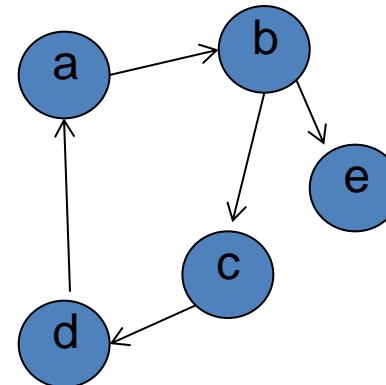
connected, cyclic



unconnected

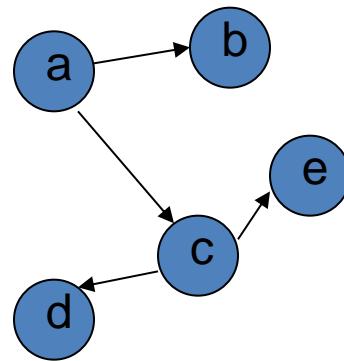
Directed graph (digraph)

- A **Directed graph** G is the data structure specified by the pair $G = \langle V, E \rangle$
 - V is the set of **vertices** (or nodes)
 - E is the set of **ordered** pairs on V
 - ($E \subset V \times V$ is a binary relation on V)
- If $(a,b) \in E$ then there is an edge from a to b
 - but not necessarily an edge from b to a.
- On a diagram (digraph) an edge is represented by an arrow.

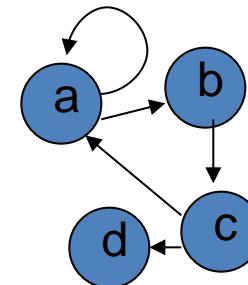


Directed graph (digraph)

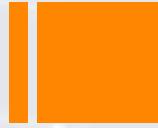
- Draw digraphs
 1. $G1 = \langle a, b, c, d, e, (a, b), (a, c), (c, d), (c, e) \rangle$
 2. $G2 = \langle a, b, c, d, (a, a), (a, b), (b, c), (c, a), (c, d) \rangle$
and comment on them.



- Unconnected - no directed path between e,d
- Weakly connected – undirected paths for all pairs of nodes
- Acyclic
- Binary Tree
- Longest path = 3 nodes



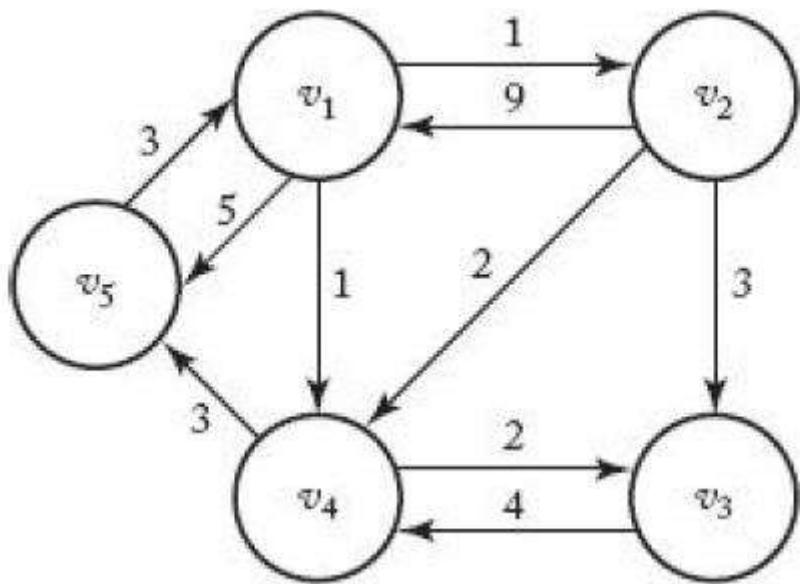
- Connected
- Cyclic
- Trivial path
- No longest path - cyclic



Shortest Path Problem

- **Optimization Problem**
- Candidate Solution: path from one vertex to another
- Value of candidate solution: length of the path
- Optimal value – minimum length
- Possible multiple shortest paths

Weighted Directed Graph



Simple path – never passes through the same vertex twice

Shortest path must be simple path !

Here are three simple paths from v_1 to v_3

$$\text{length}[v_1, v_2, v_3] = 1 + 3 = 4$$

$$\text{length } [v_1, v_4, v_3] = 1 + 2 = 3$$

$$\text{length } [v_1, v_2, v_4, v_3] = 1 + 2 + 2 = 5$$

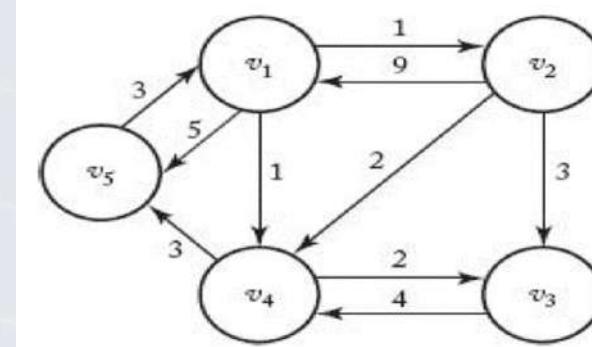
$[v_1, v_4, v_3]$ is the **shortest path** from v_1 to v_3



Brute Force

- For every vertex, determine lengths of all paths from that vertex to every other vertex and compute minimum lengths
- Complete Graph G
 - $(n-2)!$

Adjacency Matrix M



- $W[i,j]$ = weight of the path from $v_i \rightarrow v_j$ if there is an edge
- $W[i,j] = \infty$ if there is no edge from $v_i \rightarrow v_j$
- $W[i,j] = 0$ if $i = j$

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

D

Dynamic Programming Solution to³³ the all-pairs shortest path

- n vertices in the graph
- Create a sequence of $n+1$ arrays D^k where $0 \leq k \leq n$, where
 - $D^k [i,j] = \text{length of the shortest path from } v_i \text{ to } v_j \text{ using}$ only vertices in the set $\{v_1, v_2, \dots, v_k\}$ as intermediate vertices
- $D^n [i,j] = \text{length of the shortest path from } v_i \text{ to } v_j$
- $D^0 [i,j] = \text{the weight on the edge from } v_i \text{ to } v_j$
- We have established

$$D^0 = W \text{ and } D^n = D$$

Dynamic Programming Steps

To determine D from W , we need only find a way to obtain D^N from D^0 using the following two steps:

1. Establish a recursive property to compute D^k from $D^{(k-1)}$
2. Solve an instance of the problem bottom-up by repeating the process (in step 1) for $k=1$ to n . This creates the sequence:

$$\begin{array}{c} D^0, D^1, D^2, \dots, D^N \\ W \qquad \qquad \qquad D \end{array}$$

We accomplish step 1 by considering 2 cases...



Establish a recursive Property

Two Cases to consider (details following two slides)

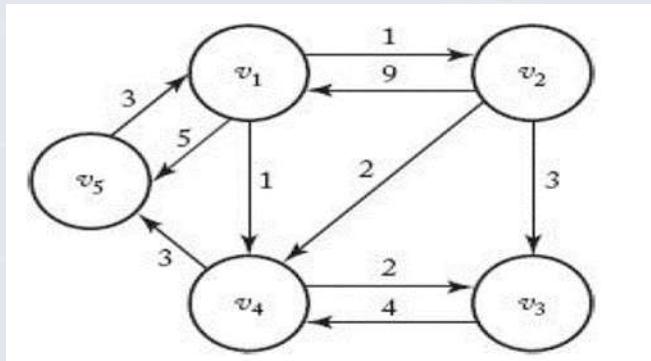
$D^k[i,j] = \text{minimum } (\text{case 1, case 2})$

$= \text{minimum } (D^{(k-1)}[i,j] , D^{(k-1)}[i,k] + D^{(k-1)}[k,j])$

Case 1

- At least one shortest path from v_i to v_j uses only vertices in set $\{v_1, v_2, \dots, v_k\}$ as the intermediate vertex does not use v_k

Then $D^k [i,j] = D^{(k-1)} [i,j]$



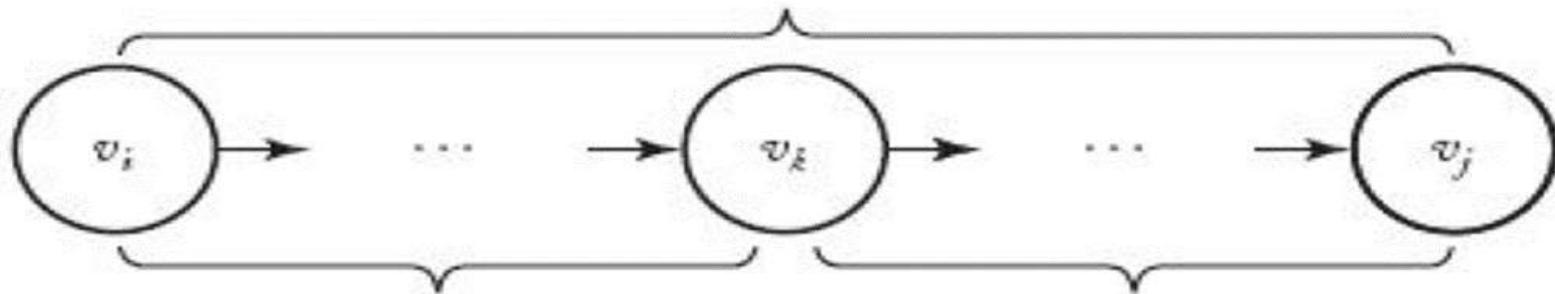
Example

$D^5[1,3] = D^4[1,3] = 3$, because when we include vertex v_5 , the shortest path from v_1 to v_3 is still $[v_1, v_4, v_3]$.

Case 2

All shortest paths from v_i to v_j uses only vertices in set $\{v_1, v_2, \dots, v_k\}$ as intermediate vertices do use v_k

A shortest path from v_i to v_j using only vertices in $\{v_1, v_2, \dots, v_k\}$



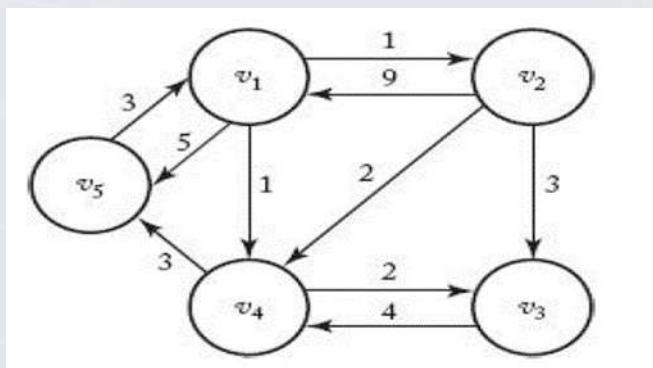
A shortest path from v_i to v_k using only vertices in $\{v_1, v_2, \dots, v_k\}$

A shortest path from v_k to v_j using only vertices in $\{v_1, v_2, \dots, v_k\}$

- Path = $v_i, \dots, v_k, \dots, v_j$ where v_i, \dots, v_k consists only of vertices in $\{v_1, v_2, \dots, v_{k-1}\}$ as intermediates: **Cost of path = $D^{(k-1)}[i,k]$**
- And where v_k, \dots, v_j consists only of vertices in $\{v_1, v_2, \dots, v_{k-1}\}$ as intermediates: **Cost of path = $D^{(k-1)}[k,j]$**

Case 2, cont.

- Path = $v_i, \dots, v_k, \dots, v_j$ where v_i, \dots, v_k consists only of vertices in $\{v_1, v_2, \dots, v_{k-1}\}$ as intermediates: **Cost of path = $D^{(k-1)}[i,k]$**
- And where v_k, \dots, v_j consists only of vertices in $\{v_1, v_2, \dots, v_{k-1}\}$ as intermediates: **Cost of path = $D^{(k-1)}[k,j]$**
- Therefore $D^{(k)}[i,j] = D^{(k-1)}[i,k] + D^{(k-1)}[k,j]$



Example

$$D^2[5,3] = 7 = 4+3 = D^1[5,2] + D^1[2,3]$$

Floyd's Algorithm for Shortest Paths – Algorithm 3.3

Floyd's Algorithm for Shortest Paths

Problem: Compute the shortest paths from each vertex in a weighted graph to each of the other vertices. The weights are nonnegative numbers.

Inputs: A weighted, directed graph and n , the number of vertices in the graph. The graph is represented by a two-dimensional array W , which has both its rows and columns indexed from 1 to n , where $W[i][j]$ is the weight on the edge from the i th vertex to the j th vertex.

Outputs: A two-dimensional array D , which has both its rows and columns indexed from 1 to n , where $D[i][j]$ is the length of a shortest path from the i th vertex to the j th vertex.

```
void floyd (int n
            const number W[][],
            number D[][])
{
    index i, j, k;
    D = W;
    for (k = 1; k <= n; k++)
        for (i = 1; i <= n; i++)
            for (j = 1; j <= n; j++)
                D[i][j] = minimum(D[i][j], D[i][k] + D[k][j]);
}
```

Basic operation: The instruction in the **for**- j loop.
Input size: n , the number of vertices in the graph.

$$T(n) = n \times n \times n = n^3 \in \Theta(n^3).$$



Does Dynamic Programming Apply to all Optimization Problems?

- **No**
- The Principle of Optimality
 - An optimal solution to an instance of a problem always contains optimal solution to all substances
- Shortest Paths Problem
 - If v_k is a node on an optimal path from v_i to v_j then the sub-paths v_i to v_k and v_k to v_j are also optimal paths

Chained-Matrix Multiplication

Suppose we want to multiply a 2×2 matrix times a 3×4 matrix as follows:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 83 \end{bmatrix}$$

In general, to multiply an $i \times j$ matrix times a $j \times k$ matrix using the standard method, it is necessary to do

$$i \times j \times k \text{ elementary multiplications.}$$

Consider the multiplication of the following four matrices:

$$\begin{array}{ccccccccc} A & \times & B & \times & C & \times & D \\ 20 \times 2 & & 2 \times 30 & & 30 \times 12 & & 12 \times 8 \end{array}$$



$A(B(CD))$	$30 \times 12 \times 8 + 2 \times 30 \times 8 + 20 \times 2 \times 8 = 3,680$
$(AB)(CD)$	$20 \times 2 \times 30 + 30 \times 12 \times 8 + 20 \times 30 \times 8 = 8,880$
$A((BC)D)$	$2 \times 30 \times 12 + 2 \times 12 \times 8 + 20 \times 2 \times 8 = 1,232$
$((AB)C)D$	$20 \times 2 \times 30 + 20 \times 30 \times 12 + 20 \times 12 \times 8 = 10,320$
$(A(BC))D$	$2 \times 30 \times 12 + 20 \times 2 \times 12 + 20 \times 12 \times 8 = 3,120$



Chained-Matrix Multiplication

- Optimal order for chained-matrix multiplication dependent on array dimensions
- Consider all possible orders and take the minimum: $t_n > 2^{n-2}$
- Principle of Optimality applies
- Develop Dynamic Programming Solution

Chained-Matrix Multiplication

Suppose we have the following six matrices:

$$\begin{array}{ccccccccc}
 A_1 & \times & A_2 & \times & A_3 & \times & A_4 & \times & A_5 & \times & A_6 \\
 5 \times 2 & & 2 \times 3 & & 3 \times 4 & & 4 \times 6 & & 6 \times 7 & & 7 \times 8 \\
 d_0 & d_1 & d_1 & d_2 & d_2 & d_3 & d_3 & d_4 & d_4 & d_5 & d_5 & d_6
 \end{array}$$

To multiply A_2 , A_5 , and A_6 , we have the following two orders and numbers of elementary multiplications:

$$\begin{aligned}
 (A_4 A_5) A_6 \text{ Number of multiplications} &= d_3 \times d_4 \times d_5 + d_3 \times d_5 \times d_6 \\
 &= 4 \times 6 \times 7 + 4 \times 7 \times 8 = 392
 \end{aligned}$$

$$\begin{aligned}
 A_4 (A_5 A_6) \text{ Number of multiplications} &= d_4 \times d_5 \times d_6 + d_3 \times d_4 \times d_6 \\
 &= 6 \times 7 \times 8 + 4 \times 6 \times 8 = 528
 \end{aligned}$$

Therefore,

$$M[4][6] = \min(392, 528) = 392.$$

$M[i,j] = \text{minimum number of multiplications needed to multiply } A_i \text{ through } A_j$

Chained-Matrix Multiplication

- Principle of Optimality applies

The optimal order for multiplying six matrices must have one of these factorizations:

1. $A_1 (A_2 A_3 A_4 A_5 A_6)$
2. $(A_1 A_2) (A_3 A_4 A_5 A_6)$
3. $(A_1 A_2 A_3) (A_4 A_5 A_6)$
4. $(A_1 A_2 A_3 A_4) (A_5 A_6)$
5. $(A_1 A_2 A_3 A_4 A_5) A_6$

number of multiplications for the k th factorization is the minimum number needed to obtain each factor plus the number needed to multiply the two factors. This means that it equals

$$M[1][k] + M[k+1][6] + d_0 d_k d_6.$$

We have established that

$$M[1][6] = \underset{1 \leq k \leq 5}{\text{minimum}}(M[1][k] + M[k+1][6] + d_0 d_k d_6).$$

$$M[i][j] = \underset{i \leq k \leq j-1}{\text{minimum}}(M[i][k] + M[k+1][j] + d_{i-1} d_k d_j), \text{ if } i < j.$$

$$M[i][i] = 0.$$

When multiplying n matrices, then for
 $1 \leq i \leq j \leq n$



Chained-Matrix Multiplication

The steps in the dynamic programming algorithm follow
Compute diagonal 0:

$$M[i][i] = 0 \quad \text{for } 1 \leq i \leq 6.$$

Compute diagonal 1:

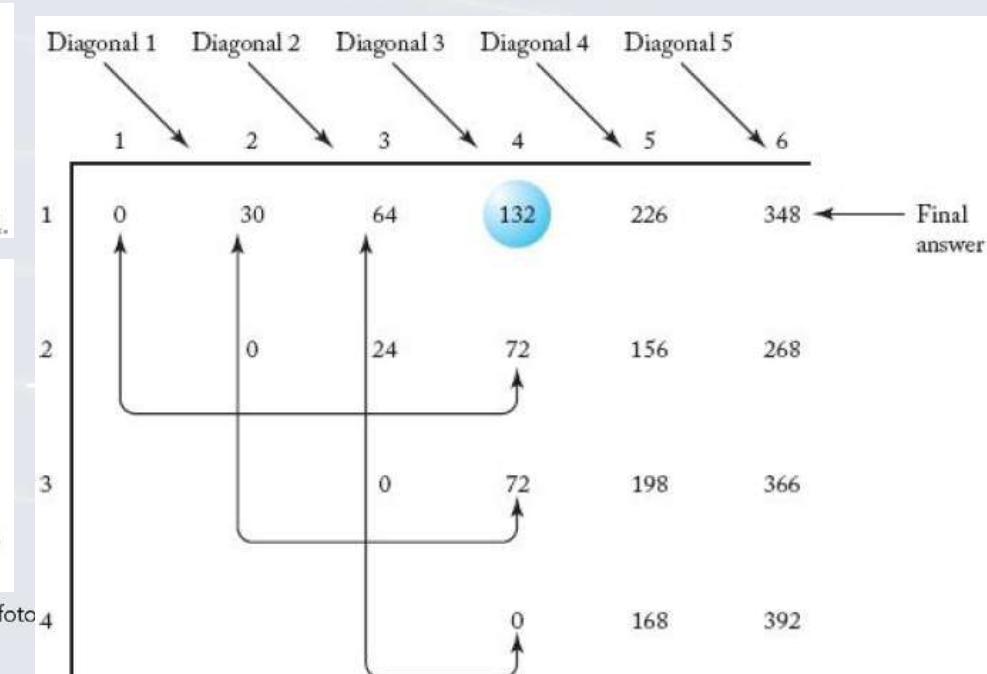
$$\begin{aligned} M[1][2] &= \min_{1 \leq k \leq 1} (M[1][k] + M[k+1][2] + d_0 d_k d_2) \\ &= M[1][1] + M[2][2] + d_0 d_1 d_2 \\ &= 0 + 0 + 5 \times 2 \times 3 = 30. \end{aligned}$$

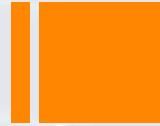
Compute diagonal 2:

$$\begin{aligned} M[1][3] &= \min_{1 \leq k \leq 2} (M[1][k] + M[k+1][3] + d_0 d_k d_3) \\ &= \min(M[1][1] + M[2][3] + d_0 d_1 d_3, \\ &\quad M[1][2] + M[3][3] + d_0 d_2 d_3) \\ &= \min(0 + 24 + 5 \times 2 \times 4, 30 + 0 + 5 \times 3 \times 4) = 64. \end{aligned}$$

Compute diagonal 3:

$$\begin{aligned} M[1][4] &= \min_{1 \leq k \leq 3} (M[1][k] + M[k+1][4] + d_0 d_k d_4) \\ &= \min(M[1][1] + M[2][4] + d_0 d_1 d_4, \\ &\quad M[1][2] + M[3][4] + d_0 d_2 d_4, \\ &\quad M[1][3] + M[4][4] + d_0 d_3 d_4) \\ &= \min(0 + 72 + 5 \times 2 \times 6, 30 + 72 + 5 \times 3 \times 6, \\ &\quad 64 + 0 + 5 \times 4 \times 6) = 132. \end{aligned}$$



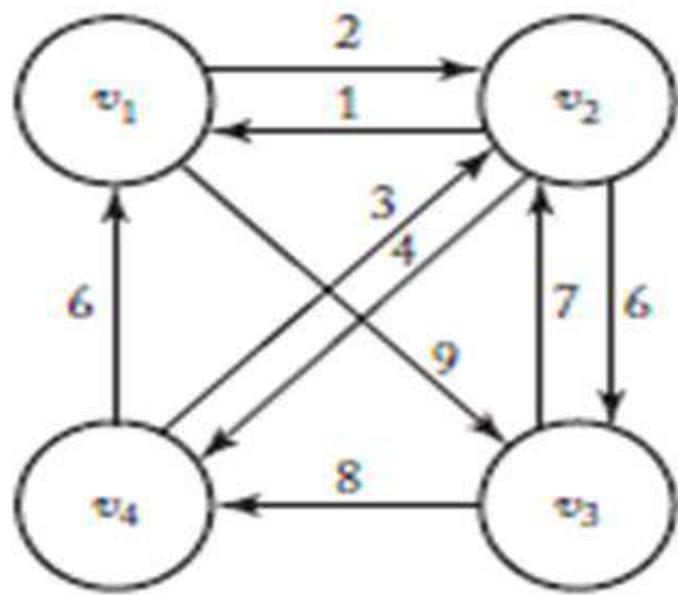


Traveling Salesperson Problem

- Sales trip – n cities
- Each city connects to some of the other cities by a road
- Minimize travel time – determine a shortest route that starts at the salesperson's home city, visits each city once, and ends at home city
- Represent instance of the problem with a weighted graph

What is
**THE TRAVELING SALESMAN
PROBLEM?**

Example



Tour (Hamiltonian circuit) in a directed graph – path from a vertex to itself that passes through each of the other vertices only once

Optimal tour in a weighted, directed graph – path of minimum length

$$\text{length } [v_1, v_2, v_3, v_4, v_1] = 22$$

$$\text{length } [v_1, v_3, v_2, v_4, v_1] = 26$$

$$\text{length } [v_1, v_3, v_4, v_2, v_1] = 21$$

Dynamic Programming Algorithm for Traveling Salesperson Problem

- $\Theta(n^{2^n})$
- Inefficient solution using dynamic programming
- Problem NP-Complete

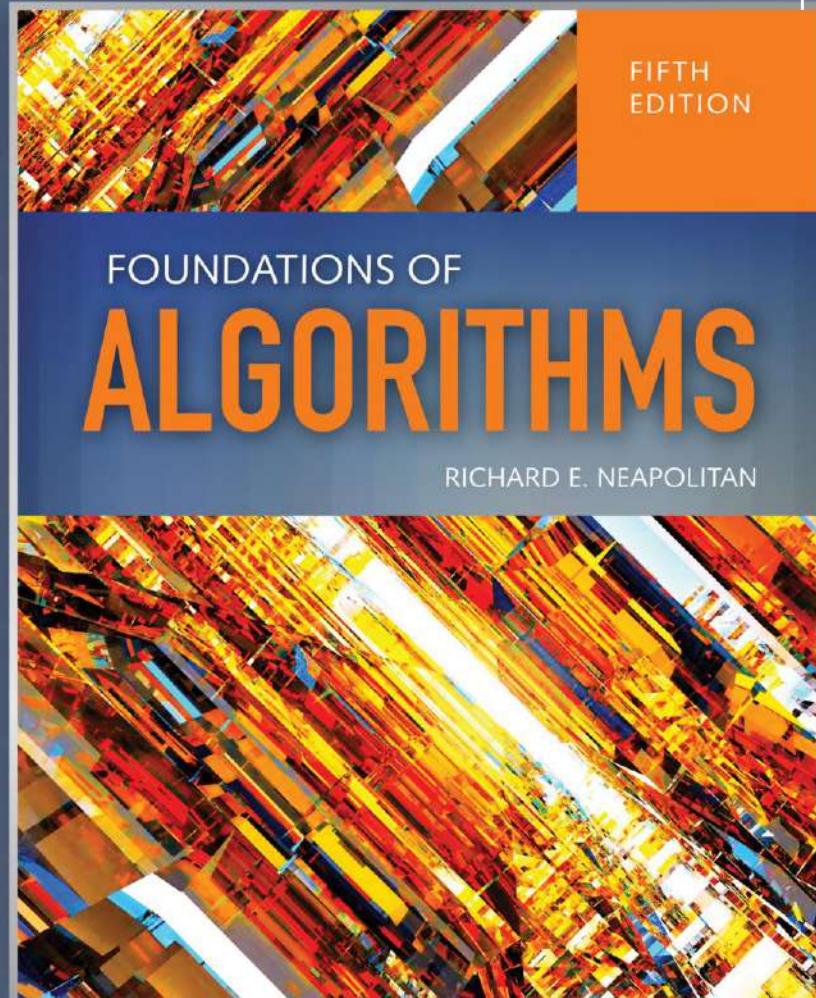


Dynamic Programming vs Divide-and-Conquer

- DP is an *optimization* technique and is applicable only to problems with *optimal substructure*.
D&C is not normally used to solve optimization problems.
- Both DP and D&C split the problem into parts, find solutions to the parts, and combine them into a solution of the larger problem.
 - In D&C, the subproblems are *significantly smaller* than the original problem (e.g. half of the size, as in MERGE-SORT) and “do not overlap” (i.e. they do not share sub-subproblems).
 - In DP, the subproblems are not significantly smaller and are overlapping.
- In D&C, the dependency of the subproblems can be represented by a tree. In DP, it can be represented by a directed path from the smallest to the largest problem (or, more accurately, by a *directed acyclic graph*, as we will see later in the course).

The Greedy Approach

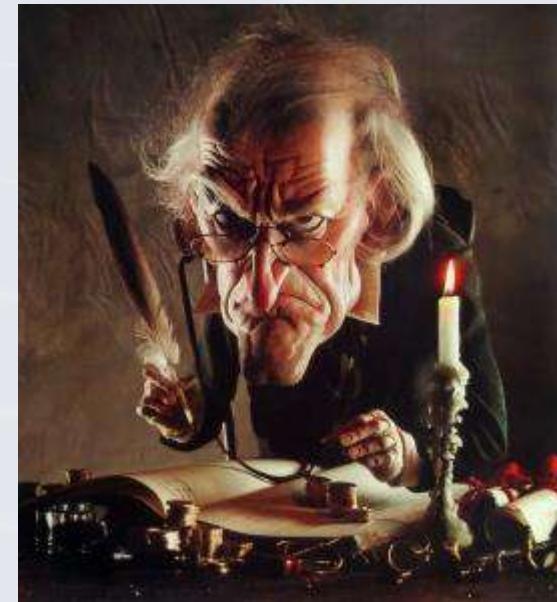
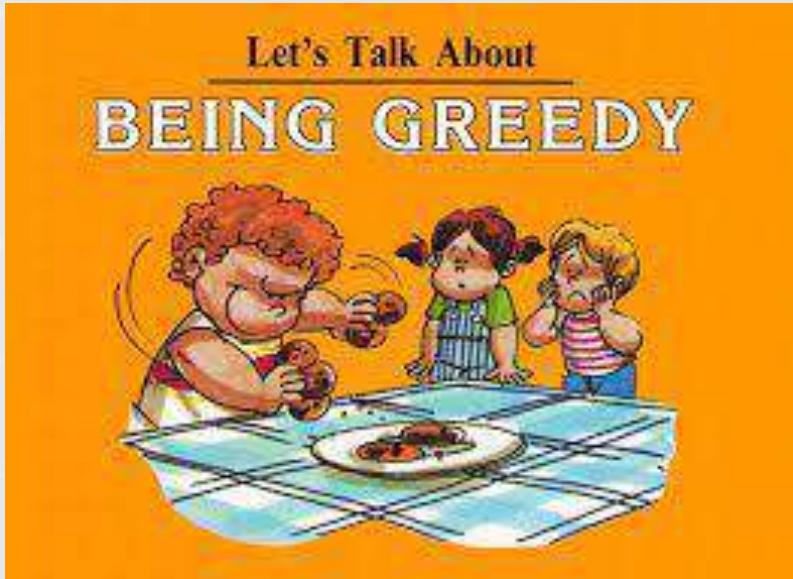
Chapter 4





Objectives

- Describe the Greedy Programming Technique
- Contrast the Greedy and Dynamic Programming approaches to solving problems
- Identify when greedy programming should be used to solve a problem
- Prove/disprove greedy algorithm produces optimal solution
- Solve optimization problems using the greedy approach



	Greedy Approach	Dynamic Programming Approach
Approach	In Greedy Technique, we make a decision/choice that seems best at a particular moment hoping it would lead to an optimal solution.	In Dynamic Programming, we make decisions/choices by keeping in mind the current state and the solution to the previously solved subproblems in order to calculate the optimal solution.
Optimality	There is no guarantee that we will always arrive at a solution using the Greedy Technique.	There is a guarantee of arriving at a solution because this technique considers all possible cases and arrives at the most optimal solution.
Memory	It is more efficient in terms of memory as it doesn't look back or update the previously calculated values.	It requires a table for updating solutions at each step and looks back at previously calculated values, increasing its memory complexity
Time Complexity	Greedy approach is comparatively faster than Dynamic Programming Approach	Dynamic Programming is generally slower than the Greedy Approach.

- The **goal** is to produce a **globally optimal solution**
- Optimal – must be proven

Greedy Technique

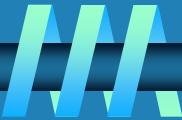


Constructs a solution to an *optimization problem* piece by piece through a sequence of choices that are:

- *feasible*
- *locally optimal*
- *irrevocable*

For some problems, yields an optimal solution for every instance.
For most, does not but can be useful for fast approximations.

Applications of the Greedy Strategy



- Optimal solutions:

- change making for “normal” coin denominations
- minimum spanning tree (MST)
- single-source shortest paths
- simple scheduling problems
- Huffman codes

- Approximations:

- traveling salesman problem (TSP)
- knapsack problem
- other combinatorial optimization problems



Greedy Algorithm

- ***Selection procedure:*** Choose the next item to add to the solution set according to the greedy criterion satisfying the locally optimal consideration
- ***Feasibility Check:*** Determine if the new set is feasible by determining if it is possible to complete this set to provide a solution to the problem instance
- ***Solution Check:*** Determine whether the new set produced is a solution to the problem instance.

Change-Making Problem



Given unlimited amounts of coins of denominations $d_1 > \dots > d_m$, give change for amount n with the least number of coins

Example: $d_1 = 25\text{c}$, $d_2 = 10\text{c}$, $d_3 = 5\text{c}$, $d_4 = 1\text{c}$ and $n = 48\text{c}$

Greedy solution:

Greedy solution is

- optimal for any amount and “normal” set of denominations
- may not be optimal for arbitrary coin denominations

Make Change Algorithm

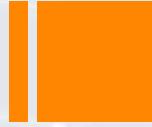
```
▪ while (there are more coins and the instance is not solved)
  {
    grab the largest coin;
    if (adding the coin makes the change exceed amount owed)
    {
      reject coin;
    } else
    {
      add the coin to the change;
    }
    if (total value of the change equals the amount owed)
    {
      the instance is solved;
    }
  }
```

$d_1 = 25c$, $d_2 = 10c$, $d_3 = 5c$,
 $d_4 = 1c$ and $n = 48c$

1. 25 (added 25, sum 25)
2. 10 (rejected 25, sum 25)
3. 10 (added 10, sum 35)
4. 10 (added 10, sum 45)
5. 10 (rejected 10, sum 45)
6. 5 (rejected 5, sum 45)
7. 1 (added 1, sum 46)
8. 1 (added 1, sum 47)
9. 1 added 1, sum 48)

10. result

$$25 + 10 + 10 + 1 + 1 + 1$$



Optimal Solution? Prove

- Set of coins finite – {H,Q,D,N,P}
- Brute force, show greedy algorithm produces an optimal solution to be made for \$.01 - \$.50
- Any amount of change > \$.50 would be a multiple of what was shown (use induction)
- Include a 12-cent coin: coins .50, .25, .12, .10, .05, .01
 - Produce \$.16 in change: not optimal

Spanning Tree

- Assume: Connected, weighted, undirected graph G
- **Spanning tree** of a connected graph G: a connected acyclic subgraph of G that includes all of G's vertices
- **Minimum spanning tree (MST)** of a weighted, connected graph G: a spanning tree of G of minimum total weight

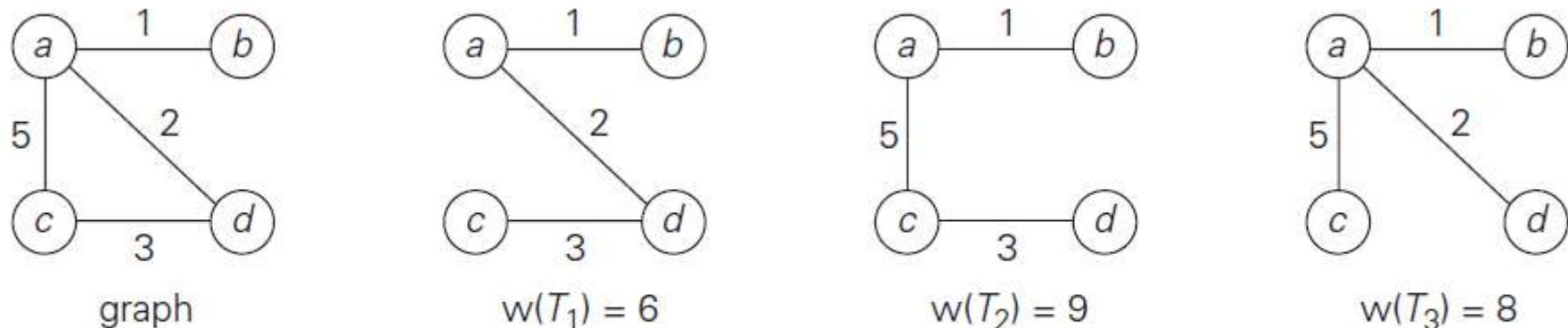
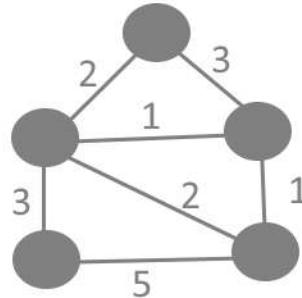


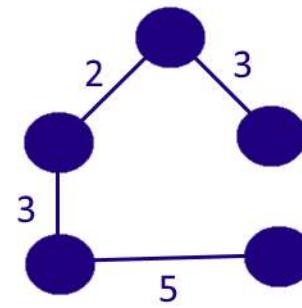
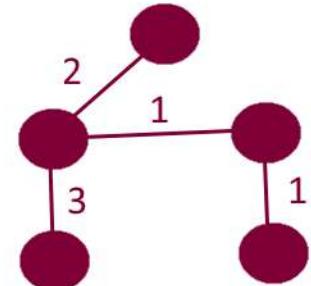
FIGURE 9.2 Graph and its spanning trees, with T_1 being the minimum spanning tree.

Minimum Spanning Tree for G

- Let $G = (V, E)$
- Let T be a spanning tree for G : $T = (V, F)$ where $F \subseteq E$
- Find T such that the sum of the weights of the edges in F is minimal



Graph

Spanning Tree
Cost = 13Minimum Spanning
Tree, Cost = 7

PRIMS ALGORITHM VERSUS KRUSHAL ALGORITHM

PRIMS ALGORITHM

A greedy algorithm that finds a minimum spanning tree for a weighted undirected graph

Generates the minimum spanning tree starting from the root vertex

Selects the root vertex

Selects the shortest edge connected to the root vertex

KRUSHAL ALGORITHM

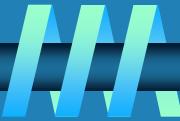
A minimum spanning tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest

Generates the minimum spanning tree starting from the least weighted edge

Selects the shortest edge

Selects the next shortest edge

Prim's MST algorithm



- Start with tree T_1 consisting of one (any) vertex and “grow” tree one vertex at a time to produce MST through a series of expanding subtrees T_1, T_2, \dots, T_n
- On each iteration, construct T_{i+1} from T_i by adding vertex not in T_i that is closest to those already in T_i (this is a “greedy” step!)
- Stop when all vertices are included

High-Level - Prim's Algorithm

15

```
F = Ø;                                // Initialize set of edges  
Y = {v1};                            // to empty.  
while (the instance is not solved){  
    select a vertex in V - Y that is      // selection procedure and  
    nearest to Y;                      // feasibility check  
    add the vertex to Y;  
    add the edge to F;  
    if (Y == V)                          // solution check  
        the instance is solved;  
}
```

Prim's MST Algorithm

```

void prim (int n,
           const number W[ ][],
           set_of_edges& F)
{
    index i, vnear;
    number min;
    edge e;
    index nearest[2..n];
    number distance[2..n];

    F = Ø;
    for (i = 2; i <= n; i++){
        nearest[i] = 1;
        distance[i] = W[1][i];
    }

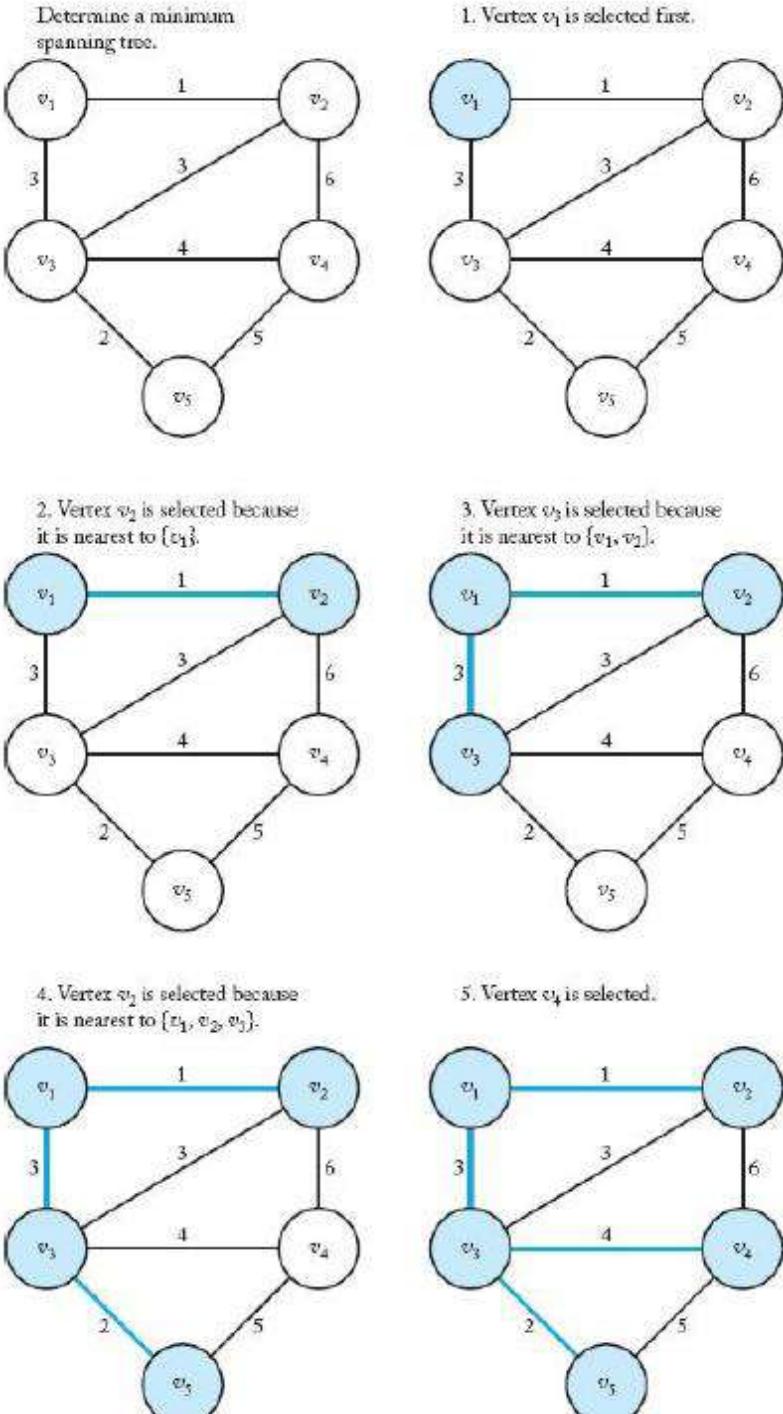
    repeat (n - 1 times){
        min = ∞;
        for (i = 2; i <= n; i++)
            if (0 ≤ distance[i] < min) {
                min = distance[i];
                vnear = i;
            }
        e = edge connecting vertices indexed
            by vnear and nearest[vnear];
        add e to F;
        distance[vnear] = -1;
        for (i = 2; i <= n; i++)
            if (W[i][vnear] < distance[i]){
                distance[i] = W[i][vnear];
                nearest[i] = vnear;
            }
    }
}

```

// For all vertices, initialize v_1
 // to be the nearest vertex in
 // Y and initialize the *distance*
 // from Y to be the weight
 // on the edge to v_1 .

// Add all $n - 1$ vertices to Y .

// Check each vertex for
 // being nearest to Y .



Every-Case Time Complexity of Prim's Algorithm 4.1

17

- Input Size: n (the number of vertices)
- Basic Operation: Two loops with $n - 1$ iterations inside a repeat loop
- Proof by induction that this construction actually yields MST
- Needs priority queue for locating the closest fringe vertex
- Time complexity:

$$T(n) = 2(n - 1)(n - 1) \in \Theta(n^2)$$

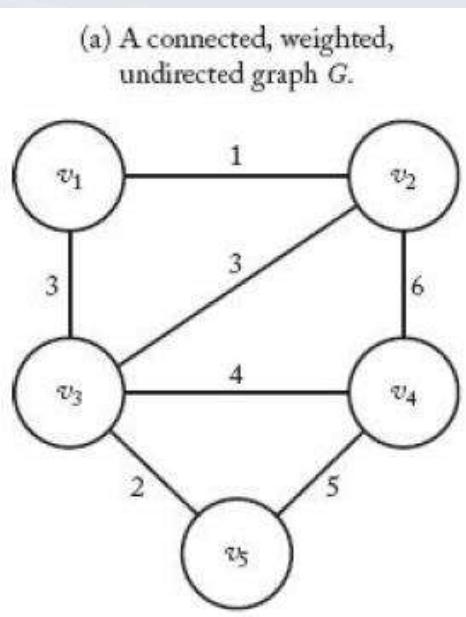
- Efficiency
 - $O(n^2)$ for weight matrix representation of graph and array implementation of priority queue
 - $O(m \log n)$ for adjacency list representation of graph with n vertices and m edges and min-heap implementation of priority queue

Spanning Tree Produced by Prim's Algorithm **Minimal?**

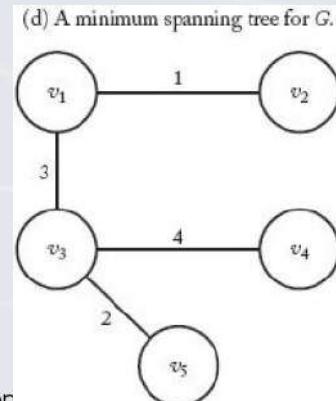
- **Dynamic Programming Algorithm** – show principle of optimality applies
- **Greedy Algorithm** – easier to develop – **must formally prove optimal solution always produced**
- Two parts to proof:
 - Lemma 4.1
 - Theorem 4.1

Promising

- Undirected graph $G = (V, E)$
- Subset $F \subseteq E$
- F is called *promising* if edges can be added to it to form a **minimum spanning tree**



Subset $\{(v1, v2), (v1, v3)\}$ is promising
 Subset $\{(v2, v4)\}$ is not promising





Lemma 4.1

Let:

$G = (V, E)$ connected weighted, undirected graph

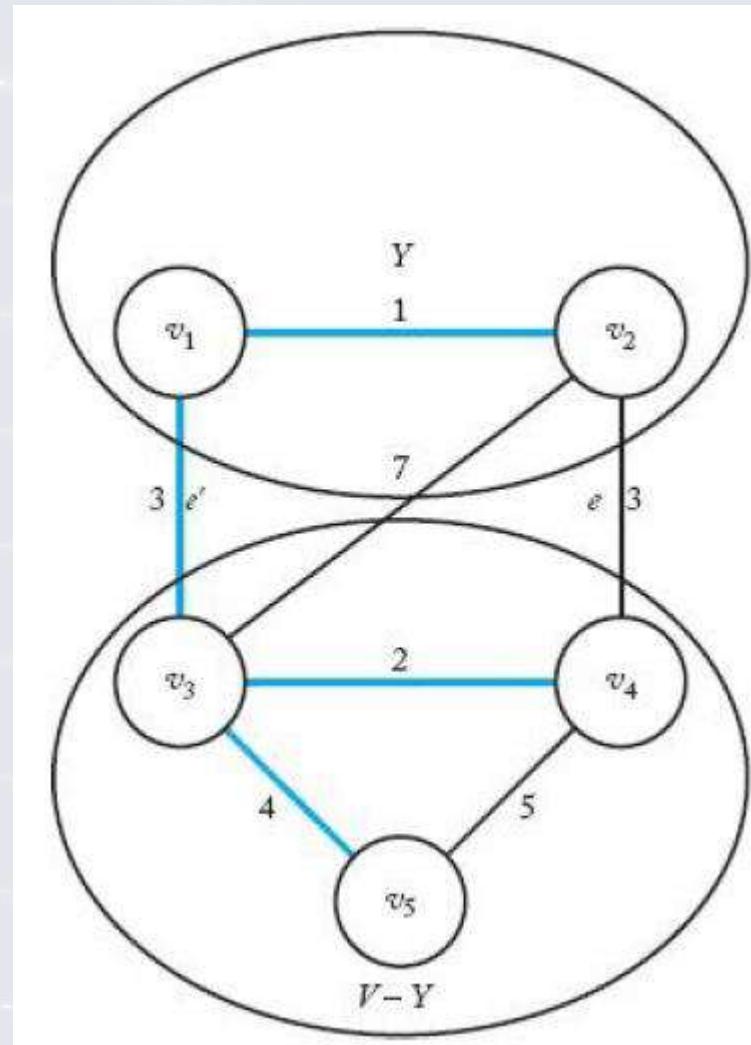
$F \subseteq E$ be promising

$Y \subseteq V$ be the set of vertices connected by edges in F

If e is a minimum edge connecting some $v_y \in Y$ to $v_x \in V - Y$, then $F \cup \{e\}$ is promising

Proof Lemma 4.1

- F is promising – must be a minimum spanning tree (v, F') such that $F \subseteq F'$
- if $e \in F'$, $F \cup \{e\} \subseteq F' \Rightarrow F \cup \{e\}$ is promising (proof complete)
- If $e \notin F'$, $F' \cup \{e\}$ must contain a cycle containing $e \Rightarrow$ there must be another $e' \in F'$ in the cycle connecting some $v_x \in Y$ to $v_y \in V - Y$
- Cycle disappears if $F' \cup \{e\} - \{e'\} \Rightarrow$ spanning
- Since e is minimum, $e \leq e' \Rightarrow F' \cup \{e\} - \{e'\}$ must be a spanning tree
- $F \cup \{e\} \subseteq F' \cup \{e'\} - \{e'\}$
- Since F connects only vertices in Y , $e' \notin F$
- i.e. e was selected
 - adding e does not create a cycle \Rightarrow **$F \cup \{e\}$ is promising**





Theorem 4.1 – Proof by Induction

Prim's Algorithm always produces a minimum spanning tree

- Induction Base: $F = \text{empty set}$ is promising
- Induction Hypothesis: Assume after i iterations of the repeat loop, F is promising
- Induction Step: Show $F \cup \{e\}$ is promising where e is the edge selected in the next iteration
 - Since e was selected, e is a minimum weight edge connecting a vertex in Y to a vertex in $V - Y$
 - By Lemma 4.1, $F \cup \{e\}$ is promising

Kruskal's Minimum Spanning Tree Algorithm

- Sort the edges in nondecreasing order of lengths
- “Grow” tree one edge at a time to produce MST through a series of expanding forests F_1, F_2, \dots, F_{n-1}
- On each iteration, add the next edge on the sorted list unless this would create a cycle. (If it would, skip the edge.)

High-Level Kruskal's MST Algorithm

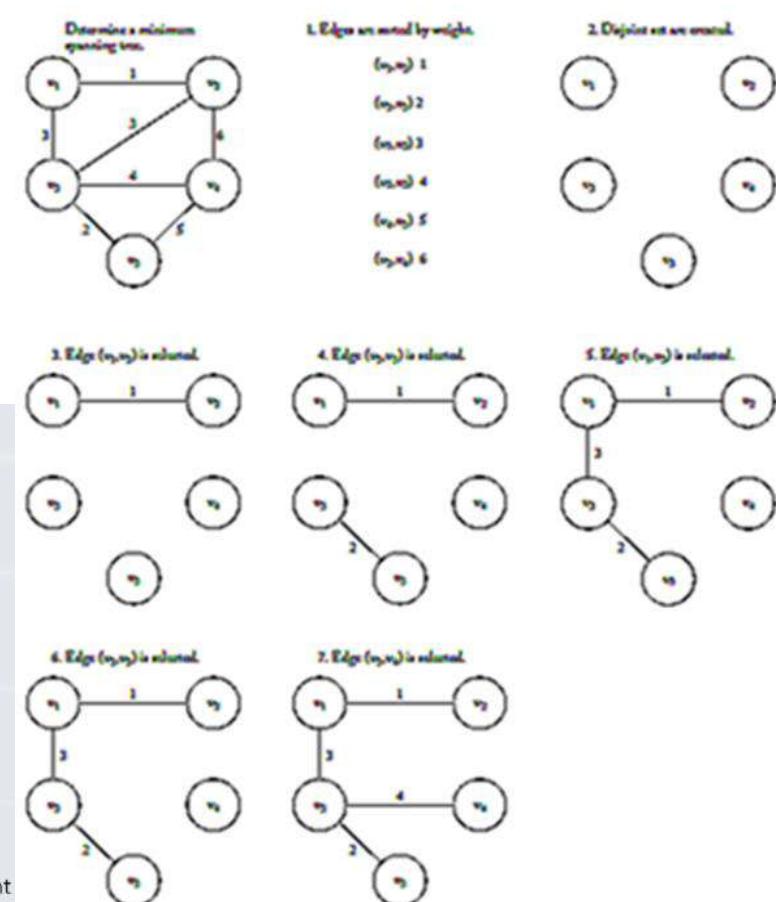
```
F = Ø;                                // Initialize set of
                                         // edges to empty.
create disjoint subsets of V, one for each
vertex and containing only that vertex;
sort the edges in E in nondecreasing order;
while (the instance is not solved){
    select next edge;                  // selection procedure
    if (the edge connects two vertices in // feasibility check
        disjoint subsets){
        merge the subsets;
        add the edge to F;
    }
    if (all the subsets are merged)      // solution check
        the instance is solved;
}
```

```

void kruskal (int n, int m,
              set_of_edges E,
              set_of_edges& F)
{
    index i, j;
    set_pointer p, q;
    edge e;
    Sort the m edges in E by weight in nondecreasing order;
    F = ∅;
    initial(n); // Initialize n disjoint sets
    while (number of edges in F is less than n - 1){
        e = edge with least weight not yet considered;
        i, j = indices of vertices connected by e;
        p = find(i);
        q = find(j);
        if (!equal(p, q)){
            merge(p, q);
            add e to F;
        }
    }
}

```

- $\text{initial}(n) \in \Theta(n)$
- $p = \text{find}(i)$ sets p to point at the set containing index i
 - $\text{find} \in \Theta(\lg m)$ where m is the depth of the tree representing the disjoint data sets
- $\text{merge}(p,q)$ merges 2 sets into 1 set
 - $\text{merge} \in \Theta(c)$ where c is a constant
- $\text{equal}(p,q)$ where p and q point to sets returns true if p and q point to the same set
 - $\text{equal} \in \Theta(c)$ where c is a constant





Worst-case Time Complexity

Kruskal

- Basic operation: a comparison instruction
- Input size: n , number of vertices
 m , number of edges



3 considerations of Kruskal

1. Time to sort edges: $W(m) \in \Theta(m \lg m)$
2. Time in the while loop $W(m) \in \Theta(m \lg m)$
3. Time to initialize n disjoint data sets: $T(n) \in \Theta(n)$
while loop: manipulation of disjoint data sets
 - Worst case, every edge is considered
 $W(m,n) \in \Theta(m \lg m)$

To connect n nodes requires at least $n-1$ edges:

$$m \geq n-1$$

$$G \text{ fully connected } m = n(n-1)/2 \in \Theta(n^2)$$

$$W(m,n) \in \Theta(n^2 \lg n^2) = \Theta(n^2 2\lg n) = \Theta(n^2 \lg n)$$



Spanning Tree Produced by Kruskal's Algorithm Minimal?

- Lemma 4.2
- Theorem 4.2

Lemma 4.2

- Let $G = (V, E)$ is a connected, weighted, undirected graph
- F is a promising subset of E
- Let e be an edge of minimum weight in $E - F$

then

- $F \cup \{e\}$ has no cycles
- $F \cup \{e\}$ is promising

Proof of Lemma 4.2 is similar to the proof of Lemma 4.1

Theorem 4.2

Kruskal's Algorithm always produces a minimum spanning tree

Proof: Use induction to show the set F is promising after each iteration of the repeat loop

- Induction base: $F = \emptyset$ empty set is promising
- Induction hypothesis: assume after the i th iteration of the repeat loop, the set of edges F selected so far is promising
- Induction step: Show $F \cup \{e\}$ is promising where e is the selected edge in the $(i+1)$ th iteration
- e selected in the next iteration, it has a minimum weight
- e connects vertices in disjoint sets
- Because e is selected, it is minimum and connects two vertices in disjoint sets
- By Lemma 4.2 $F \cup \{e\}$ is promising



Prim vs Kruskal

- **Sparse graph**
 - m close to $n - 1$
 - Kruskal $\Theta(n \lg n)$ faster than Prim
- **Highly connected graph**
 - Kruskal $\Theta(n^2 \lg n)$
 - Prim's faster

Shortest paths – Dijkstra's algorithm

Single Source Shortest Paths Problem: Given a weighted connected graph G , find shortest paths from source vertex s to each of the other vertices

Dijkstra's algorithm: Similar to Prim's MST algorithm, with a different way of computing numerical labels: Among vertices not already in the tree, it finds vertex u with the smallest sum

$$d_v + w(v,u)$$

where

v is a vertex for which shortest path has been already found on preceding iterations (such vertices form a tree)

d_v is the length of the shortest path from source to v

$w(v,u)$ is the length (weight) of edge from v to u





Notes on Dijkstra's algorithm

- Doesn't work for graphs with negative weights
- Applicable to both undirected and directed graphs
- Efficiency
 - $O(|V|^2)$ for graphs represented by weight matrix and array implementation of priority queue
 - $O(|E|\log|V|)$ for graphs represented by adj. lists and min-heap implementation of priority queue
- Don't mix up Dijkstra's algorithm with Prim's algorithm!

Coding Problem

Coding: assignment of bit strings to alphabet characters

Codewords: bit strings assigned for characters of alphabet

Two types of codes:

- fixed-length encoding (e.g., ASCII)
- variable-length encoding (e.g., Morse code)

Prefix-free codes: no codeword is a prefix of another codeword

Problem: If frequencies of the character occurrences are known, what is the best binary prefix-free code?

Huffman codes

- Any binary tree with edges labeled with 0's and 1's yields a prefix-free code of characters assigned to its leaves
- Optimal binary tree minimizing the expected (weighted average) length of a codeword can be constructed as follows

Huffman's algorithm

Initialize n one-node trees with alphabet characters and the tree weights with their frequencies.

Repeat the following step $n-1$ times: join two binary trees with smallest weights into one (as left and right subtrees) and make its weight equal the sum of the weights of the two trees.

Mark edges leading to left and right subtrees with 0's and 1's, respectively.

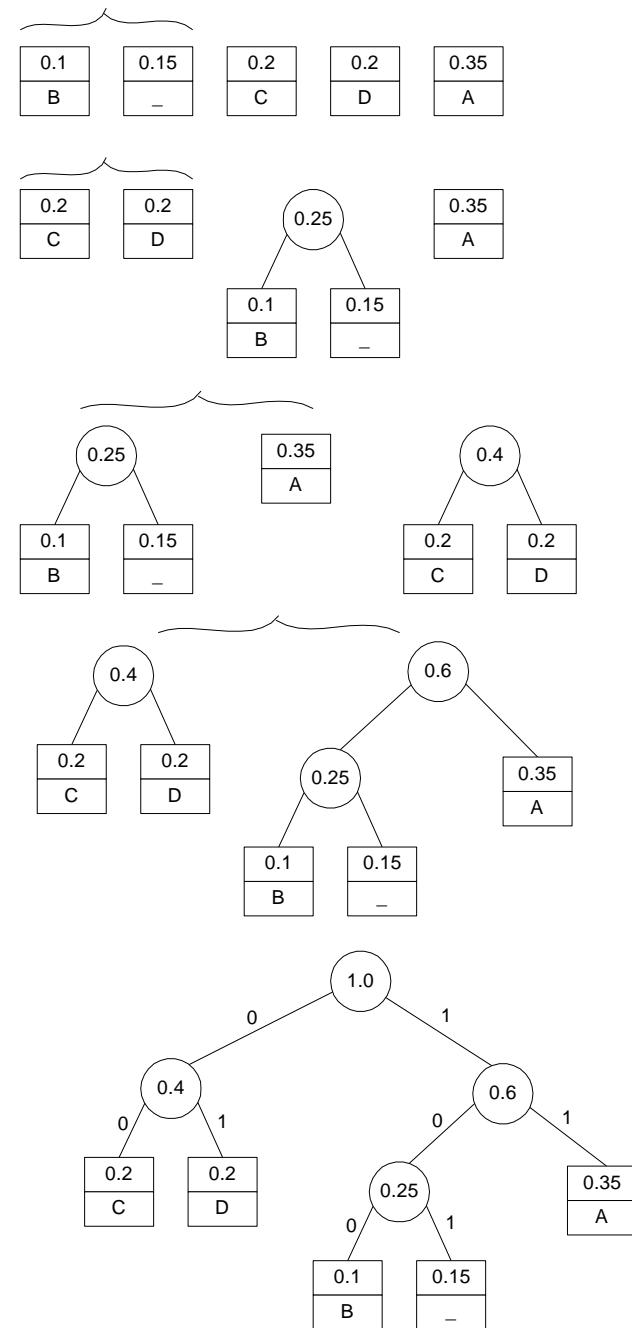
Example

character	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

average bits per character: 2.25

for fixed-length encoding: 3

compression ratio: $(3-2.25)/3 \times 100\% = 25\%$





Greedy vs Dynamic

- Both solve optimization problems
- Shortest Path
 - Floyd – all pairs dynamic
 - Dijkstra – single source greedy
- Greedy algorithms usually simpler
- Greedy algorithms do not always produce optimal solution – must formally prove
- Dynamic Programming – show principle of optimality applies



0-1 Knapsack Problem

- Thief breaks into jewelry store carrying a knapsack in which to place stolen items
- Knapsack has a weight capacity W
- Knapsack will break if weight of stolen items exceeds W
- Each item has a value
- Thief's dilemma is to maximize the total value of items stolen while not exceeding the total weight capacity W

0-1 Knapsack Problem

Suppose there are n items. Let

$$S = \{item_1, item_2, \dots, item_n\}$$

w_i = weight of $item_i$

p_i = profit of $item_i$

W = maximum weight the knapsack can hold,

where w , p , and W are positive integers. Determine a subset A of S such that

$$\sum_{item_i \in A} p_i \quad \text{is maximized subject to} \quad \sum_{item_i \in A} w_i \leq W.$$



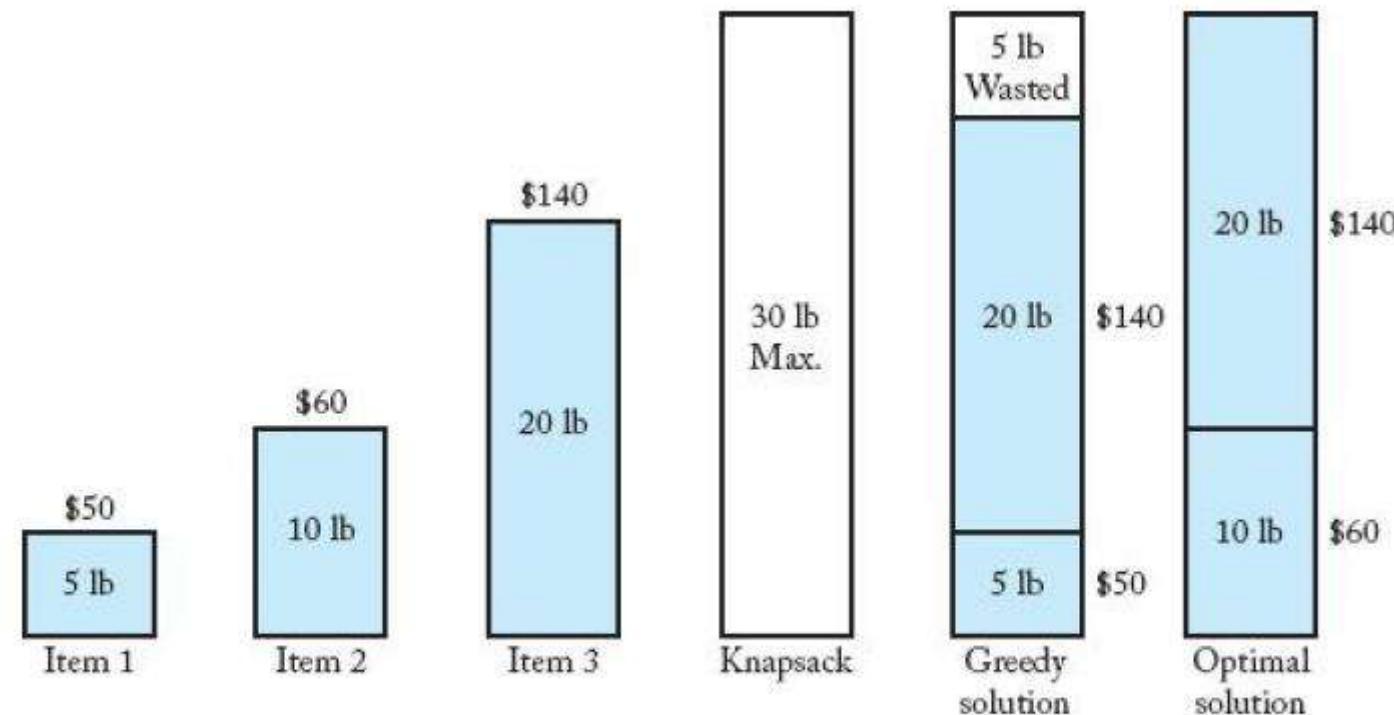
Brute Force Solution

- Consider all subsets of the n items
- Discard subsets whose total weight exceeds W
- Of the remaining, take the one with maximum profit
- 2^n subsets of a set containing n items

Greedy Strategy

43

- Steal items with the largest profit first – stealing in non-increasing order according to profit
- It Can easily be shown by example **greedy strategy does not always produce an optimal solution**



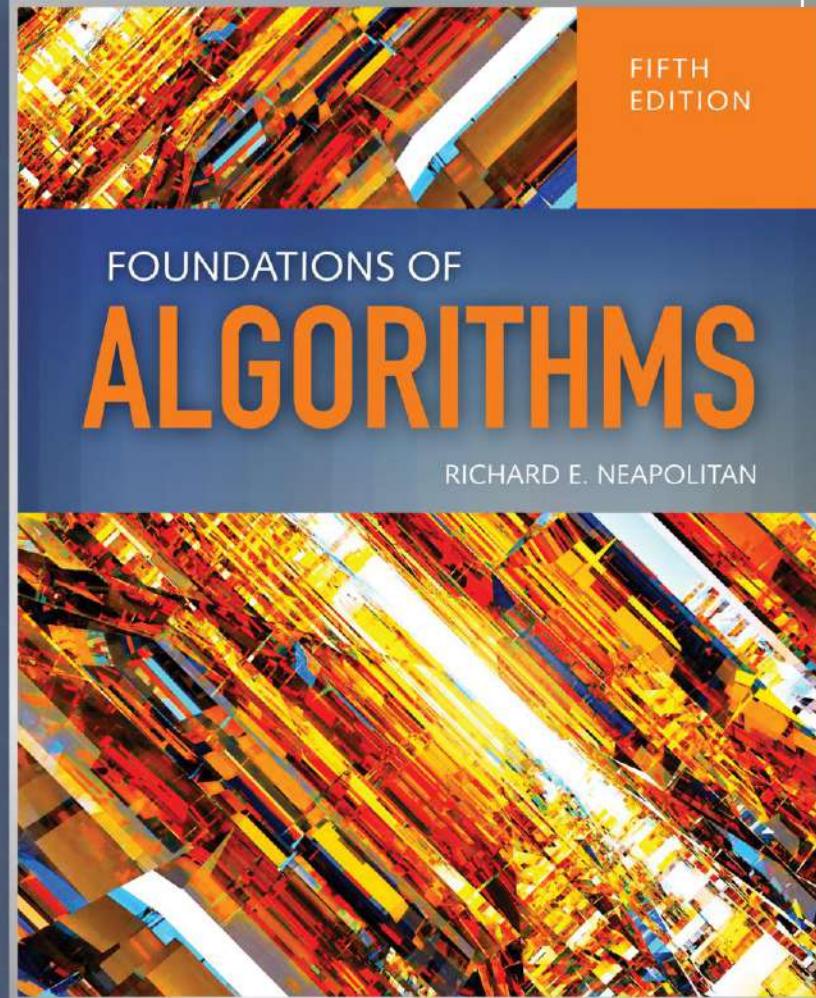
$$item_1 : \frac{\$50}{5} = \$10$$

$$item_2 : \frac{\$60}{10} = \$6$$

$$item_3 : \frac{\$140}{20} = \$7.$$

Backtracking

Chapter 5



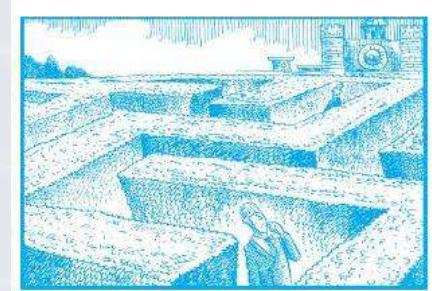


Objectives

- Describe the backtrack programming technique
- Determine when the backtracking technique is an appropriate approach to solving a problem
- Define a **state space tree** for a given problem
- Define when a node in a state space tree for a given problem is **promising/non-promising**
- Create an algorithm to **prune** a state space tree
- Create an algorithm to apply the backtracking technique to solve a given problem

Finding Your Way Thru a Maze

- Follow a path until a dead end is reached
- Go back until reaching a fork
- Pursue another path
- Suppose there were signs indicating path leads to dead end?
- Sign positioned near beginning of path – time savings enormous
- Sign positioned near end of path – very little time saved



Sudoku

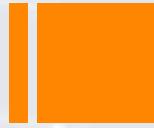
						1	2
			3	5			
		6			7		
7				3			
		4		8			
1							
		1	2				
8					4		
5				6			

6	7	3	8	9	4	5	1	2
9	1	2	7	3	5	4	8	6
8	4	5	6	1	2	9	7	3
7	9	8	2	6	1	3	5	4
5	2	6	4	7	3	8	9	1
1	3	4	5	8	9	2	6	7
4	6	9	1	2	8	7	3	5
2	8	7	3	5	6	1	4	9
3	5	1	9	4	7	6	2	8



Solving Sudoku

- Solving Sudoku puzzles involves a form of an exhaustive search of possible configurations.
- However, exploiting constraints to rule out certain possibilities for certain positions enables us to prune the search to the point that people can solve Sudoku by hand.
- Backtracking is the key to implementing exhaustive search programs correctly and efficiently.



Backtracking

- **Backtracking** is a systematic method to iterate through all possible configurations of a search space. It is a general algorithm which must be customized for each application.
- We model our solution as a vector $a = (a_1; a_2; \dots; a_n)$, where each element a_i is selected from a finite ordered set S_i .
- Such a vector might represent an arrangement where a_i contains the i th element of the permutation. Or the vector might represent a given subset S , where a_i is true if and only if the i th element of the universe is in S .
- Can be used to solve NP-Complete problems such as 0-1 Knapsack **more efficiently**

Backtracking vs Dynamic Programming

- Dynamic Programming – **subsets of a solution are generated**
- Backtracking – **Technique for deciding that some subsets need not be generated**
- Backtracking is used to solve problems in which a **sequence of objects is chosen from a specified set** so that the sequence satisfies some **criterion**.
- Efficient for many large instances of a problem (but not all)

Procedure called by passing root at the top level

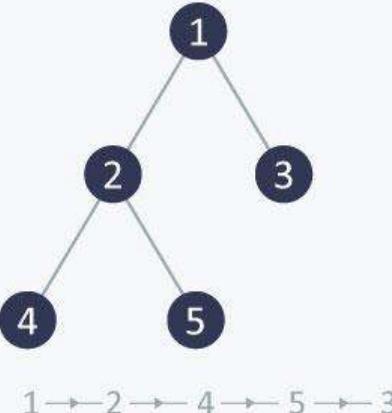
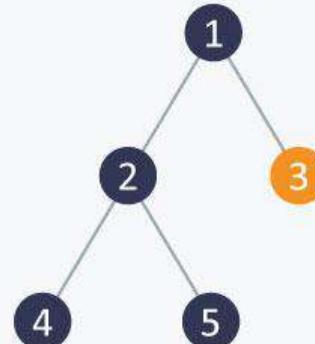
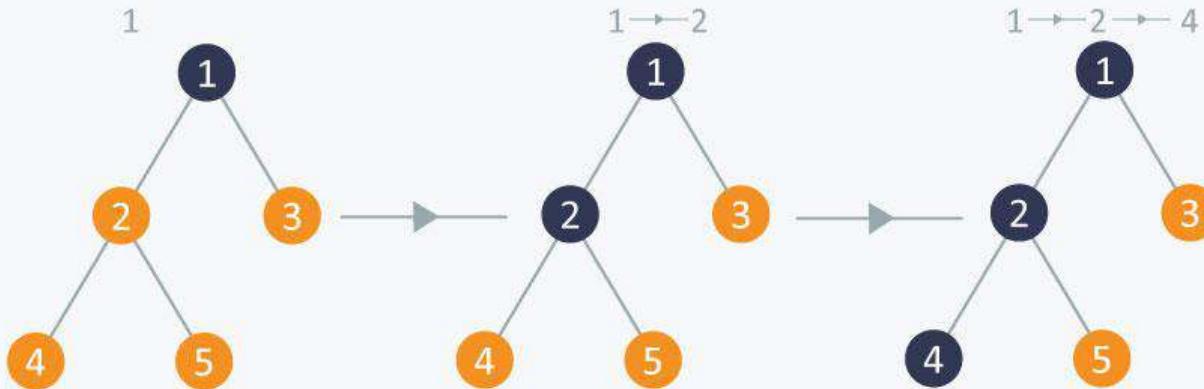
Backtracking is a modified **depth-first search** of a rooted tree

```
void depth_first_tree_search(node v)
{
    node u;
    visit v

    for( each child u of v)
        depth_first_tree_search(u);
}
```

DFS Example

DFS





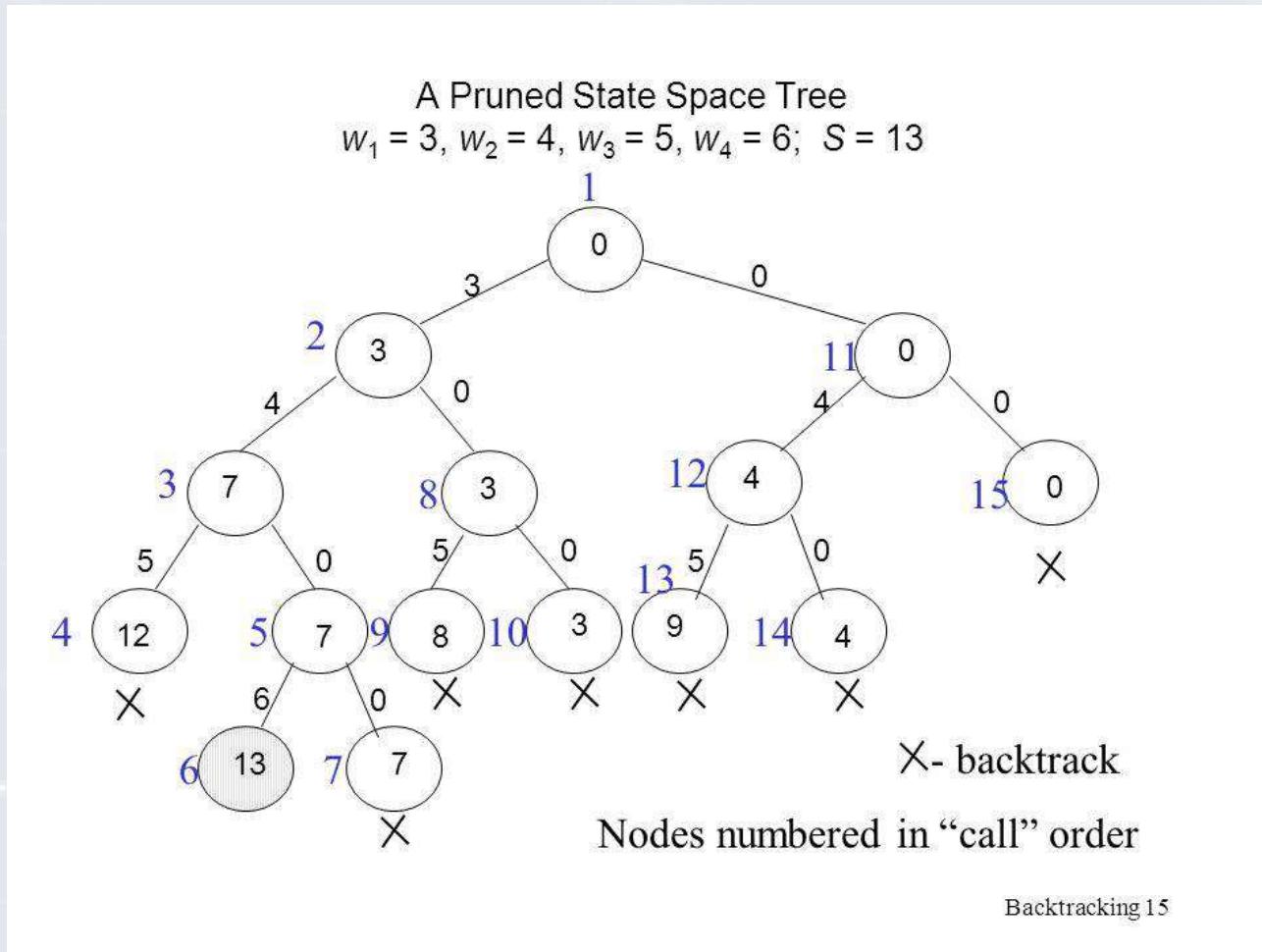
Backtracking Procedure

- After determining a node cannot lead to a solution, backtrack to the node's parent and proceed with the search on the next child
- **Non-promising node:** when the node is visited, it is determined the node cannot lead to a solution
- **Promising node:** may lead to a solution
- **Backtracking**
 - DFS of state space tree
 - **Pruning state space tree:** if a node is determined to be non-promising, back track to its parent

Backtracking Procedure

11

- Pruned State Space Tree: sub-tree consisting of visited nodes





N-Queen Problem

- Goal: **position n queens on a n x n board such that no two queens threaten each other**
 - No two queens may be in the same row, column, or diagonal
- **Sequence:** n positions where queens are placed
- **Set:** n^2 positions on the board
- **Criterion:** no two queens threaten each other



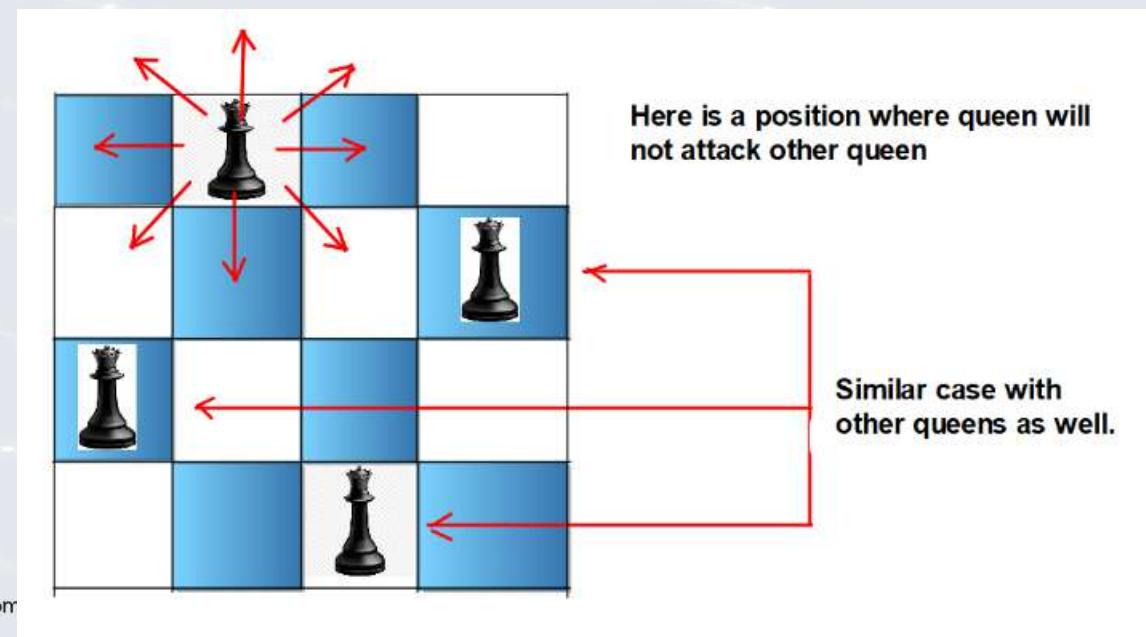
e.g. 4-Queen Problem Solution

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



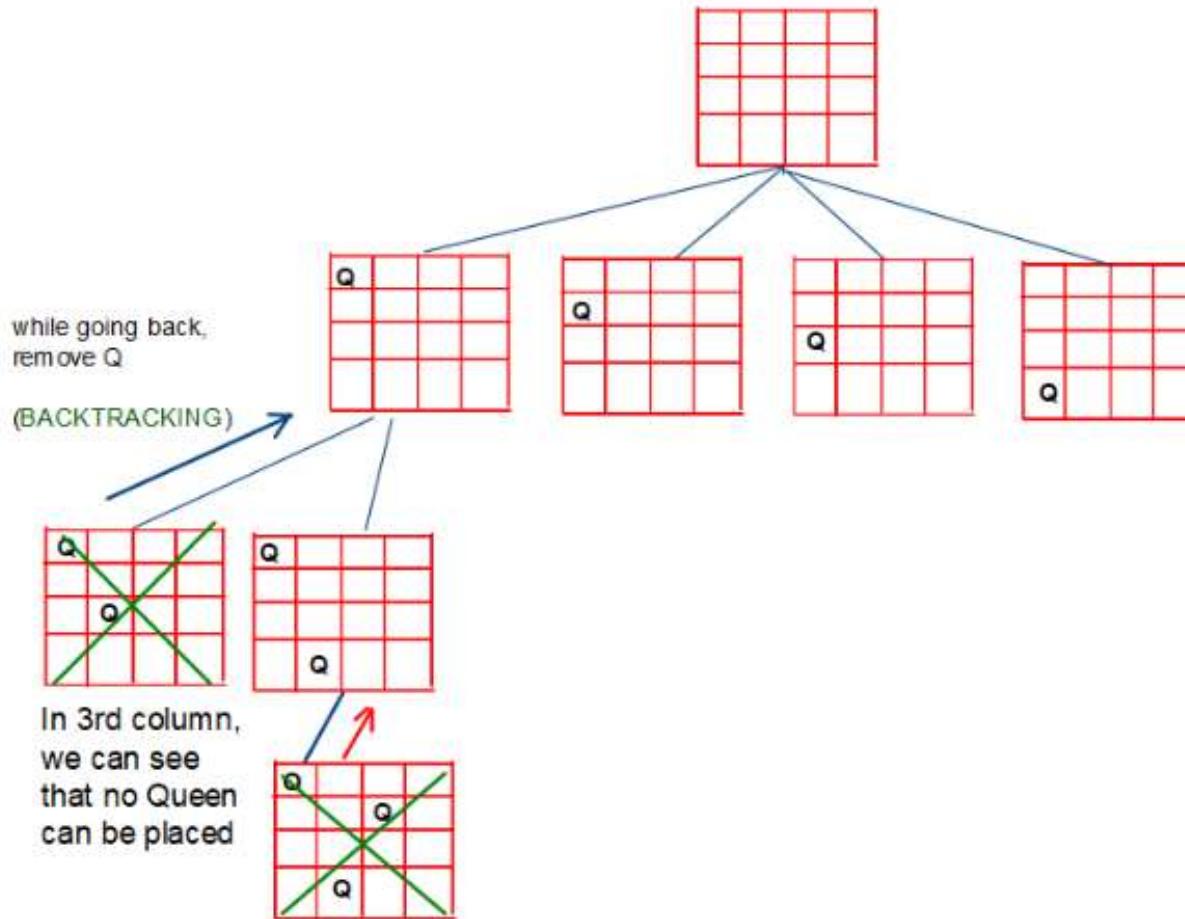
4-Queen Problem

- Assign each queen a different row
- Check which column combinations yield solutions
- Each queen can be in any one of four columns: $4 \times 4 \times 4 \times 4 = 256$



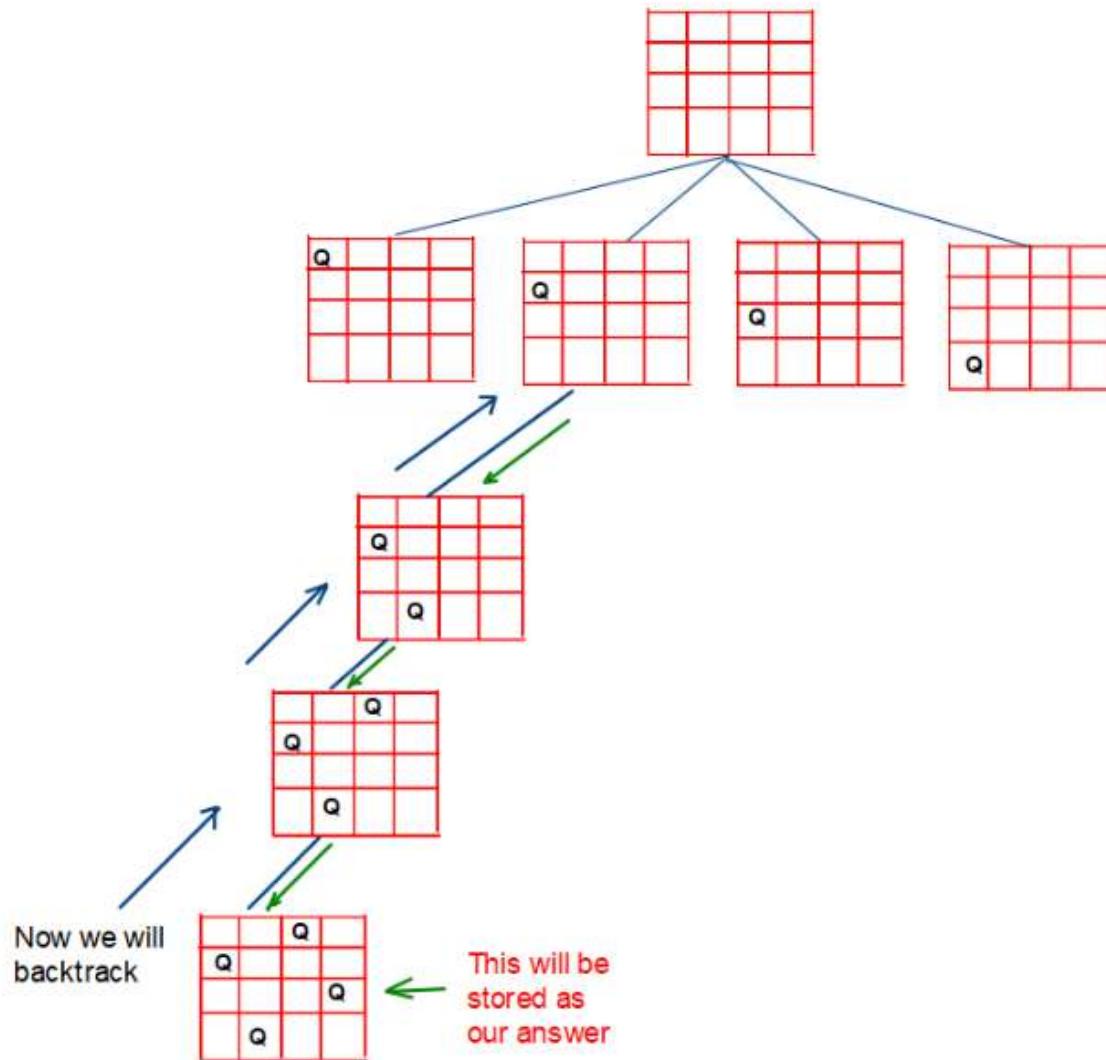
1st position: This is the position where we can see no possible arrangement is found where all queens can be placed since, at the 3rd column, the Queen will be killed at all possible positions of row.

15



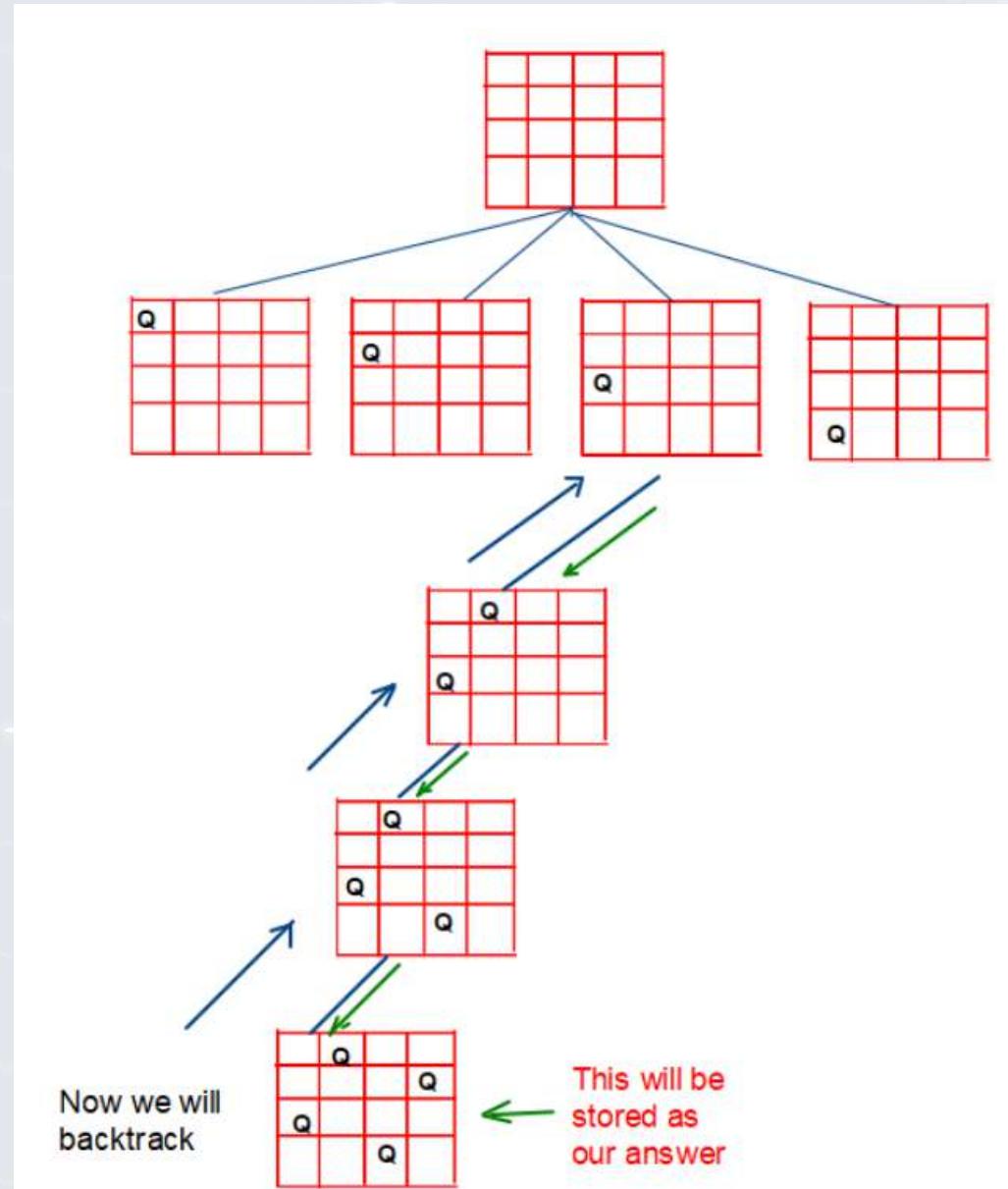
2nd position: One of the correct possible arrangements is found. So we will store it as our answer.

16

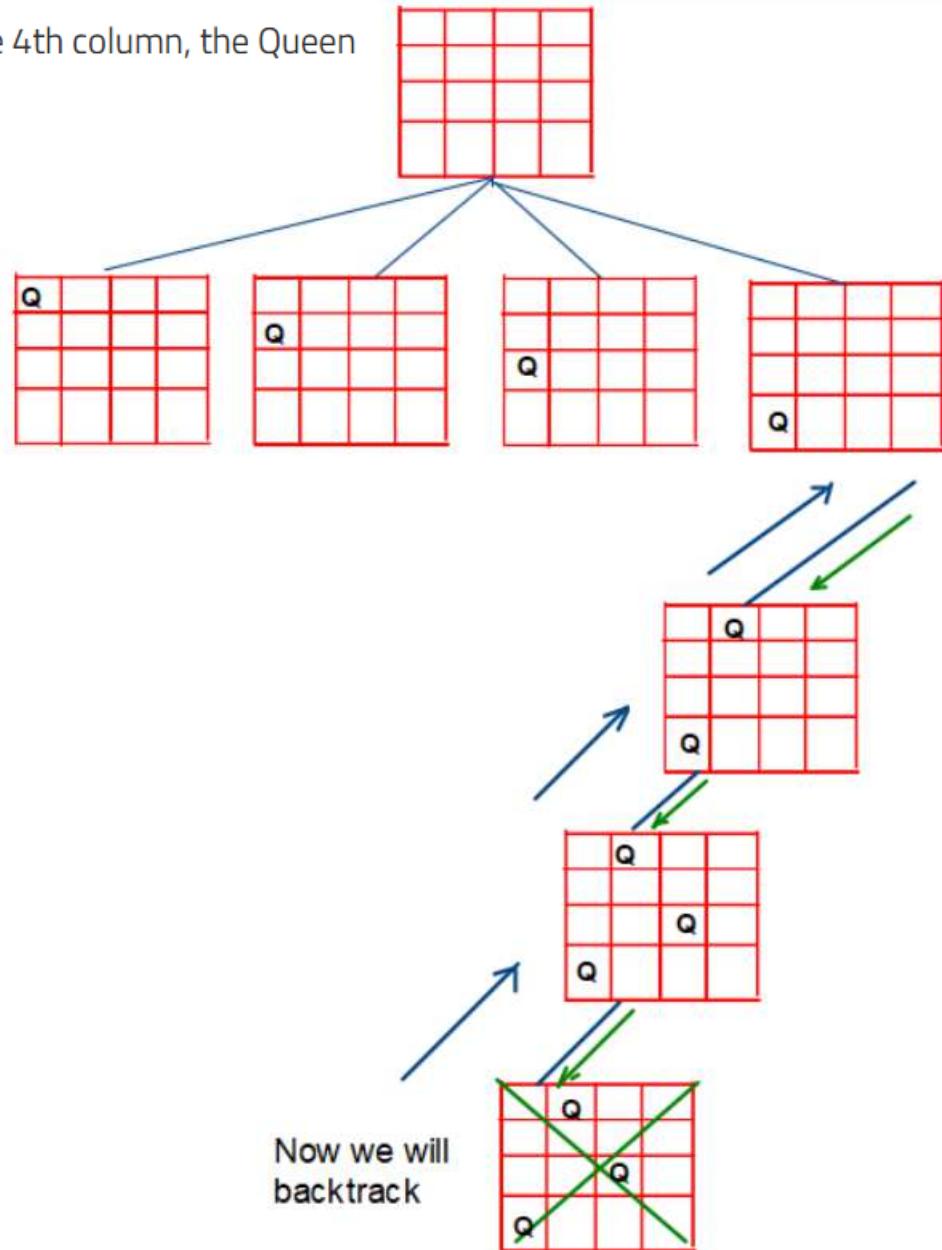


3rd position: One of the correct possible arrangements is found. So we will store it as our answer.

17



4th position: This is the position where we can see no possible arrangement is found where all queens can be placed since, at the 4th column, the Queen will be killed at all possible positions of row.



Construct State-Space Tree – Candidate Solutions

- Root – start node
- Column choices for first queen stored at level-1 nodes
- Column choices for second queen stored at level-2 nodes
- Etc.
- Path from root to a leaf is candidate solution
- Check each candidate solution in sequence starting with left-most path



Pruning

- DFS of state space tree
- Check to see if each node is **promising**
- **If a node is non-promising, backtrack to node's parent**
- **Pruned state space tree** – subtree consisting of visited nodes
- Promising function – application dependent
- Promising function n-queen problem: returns false if a node and any of the node's ancestors place queens in the same column or diagonal

A general algorithm for Backtracking

21

```
void checknode (node v)
{
    node u;
    if (promising(v))
        if (there is a solution at v)
            write the solution;
        else
            for (each child u of v)
                checknode(u);
}
```

- The root of SST is passed to **checknode** at the top level
- Check node – promising ?
Yes → solution
- No solution → visit children of the node

- Function **promising()** is different in each application of backtracking
- **Promising function for the algorithm**

checknode()

```
void checknode (node v)
{
    node u;

    if (promising(v))
        if (there is a solution at v)
            write the solution;
        else
            for (each child u of v)
                checknode(u);
}
```

- **Backtracking algorithm for the n-Queens Problem**
- State space tree implicit – **tree not actually created**
 - Values in the current branch under investigation kept track of
- Algorithm **inefficient**
- Checks node promising after passing it to the procedure
- Activation records checked and pushed onto the stack for non-promising nodes

expand()

- Improves efficiency
- Check to see if node is promising before passing the node

```
void expand(node v)
{
    node u;

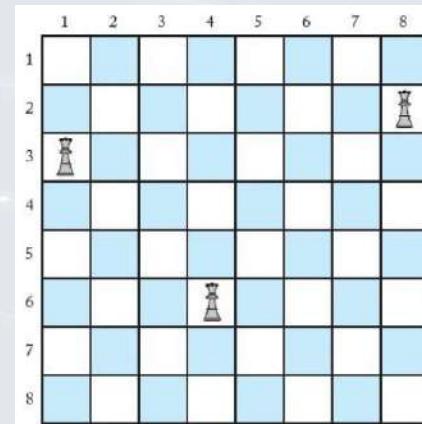
    for (each child u of v)
        if (promising(u))
            if (there is a solution at u)
                write the solution;
            else
                expand(u);
}
```

```
void checknode (node v)
{
    node u;

    if (promising(v))
        if (there is a solution at v)
            write the solution;
        else
            for (each child u of v)
                checknode(u);
}
```

Algorithm 5.1 Backtracking Algorithm for N-Queens Problem

- All solutions to N-Queens problem
- The promising function must check whether two queens are in the same column or diagonal.
- Let **col(i)** be the column where the queen in the i th row is located
- Promising function:
 - **2 queens on the same row?**
 $\text{col}(i) == \text{col}(k)$
 - **2 queens on the same diagonal?**
 $\text{col}(i) - \text{col}(k) == i - k$
 $\text{|| } \text{col}(i) - \text{col}(k) == k - i$



$$\text{col}(6) - \text{col}(3) = 4 - 1 = 3 = 6 - 3$$

$$\text{col}(6) - \text{col}(2) = 4 - 8 = -4 = 2 - 6$$

Algorithm 5.1 Backtracking Algorithm for N-Queens Problem

25

```
void queens (index i)
{
    index j;

    if (promising(i))
        if (i == n)
            cout << col[1] through col [n];
        else
            for (j = 1; j <= n; j++){
                col[i + 1] = j;
                queens(i + 1);
            }
}

bool promising (index i)
{
    index k;
    bool switch;

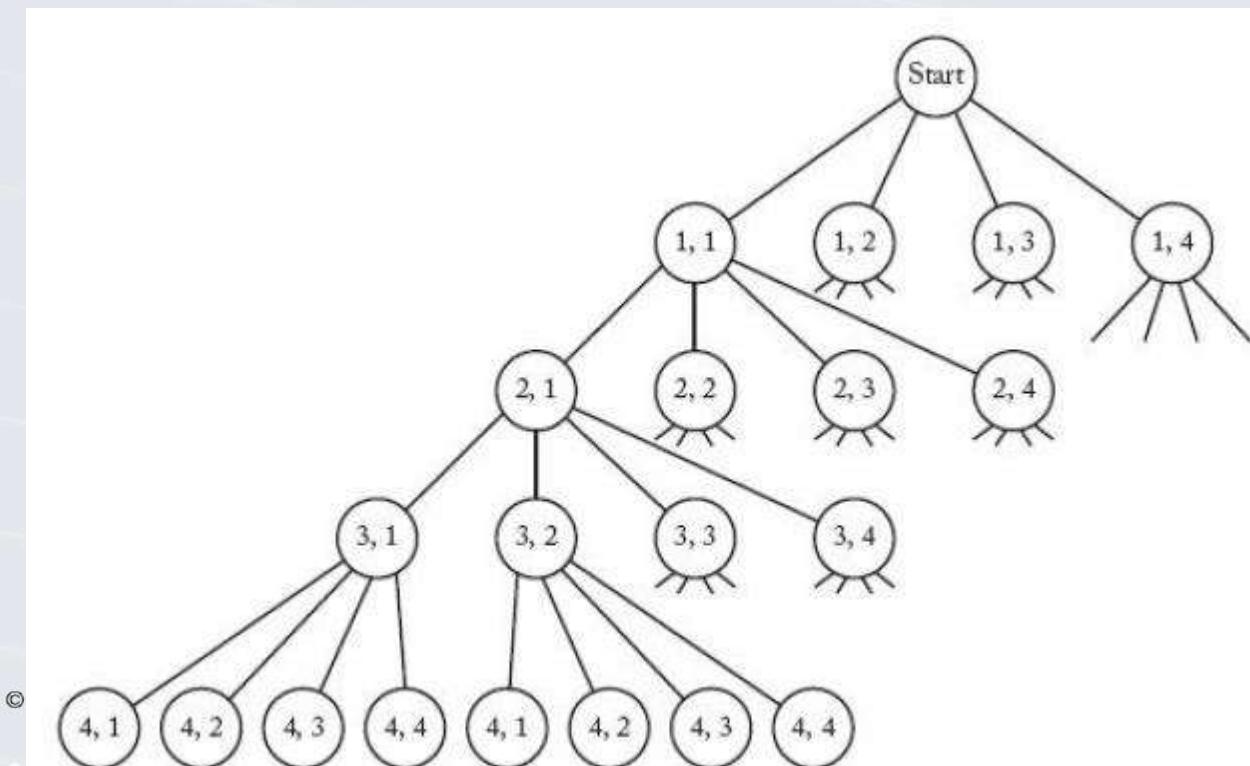
    k = 1;
    switch = true;           // Check if any queen threatens
    while (k < i && switch){ // queen in the ith row.
        if (col[i] == col[k] || abs(col[i] - col[k]) == i - k)
            switch = false;
        k++;
    }
    return switch;
}
```

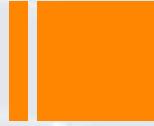
If **n** and **col** are defined as global variables,
then the top-level call to **queens()** would be
queens (0);

How many solutions does this algorithm produce?

Analysis of queens theoretically difficult

- Upper bound on the number of nodes checked in the pruned state space tree by counting the number of nodes in the entire state space tree:
 - 1 node level 0
 - n nodes level 1
 - n^2 nodes level 2
 - ...
 - N^n nodes level n





Total Number of nodes

$$1 + n + n^2 + n^3 + \cdots + n^n = \frac{n^{n+1} - 1}{n - 1}.$$

This equality is obtained in Example A.4 in [Appendix A](#). For the instance in which $n = 8$, the state space tree contains

$$\frac{8^{8+1} - 1}{8 - 1} = 19,173,961 \text{ nodes.}$$

This analysis is of limited value because the whole purpose of backtracking is to avoid checking many of these nodes.

Backtracking savings by executing code and counting nodes checked

- Algorithm1 – DFS of state space tree without backtracking
- Algorithm2 – checks no two queens in the same row or same column

n	Number of Nodes Checked by Algorithm 1†	Number of Candidate Solutions Checked by Algorithm 2‡	Number of Nodes Checked by Backtracking	Number of Nodes Found Promising by Backtracking
4	341	24	61	17
8	19,173,961	40,320	15,721	2057
12	9.73×10^{12}	4.79×10^8	1.01×10^7	8.56×10^5
14	1.20×10^{16}	8.72×10^{10}	3.78×10^8	2.74×10^7

*Entries indicate numbers of checks required to find all solutions.

†Algorithm 1 does a depth-first search of the state space tree without backtracking.

‡Algorithm 2 generates the $n!$ candidate solutions that place each queen in a different row and column.



Sum-of-Subsets Problem

- Let $S = \{s_1, s_2, \dots, s_n\}$
- Let W be a positive integer
- Find every $S' \subseteq S$ such that

$$\sum_{s \in S'} s = W$$



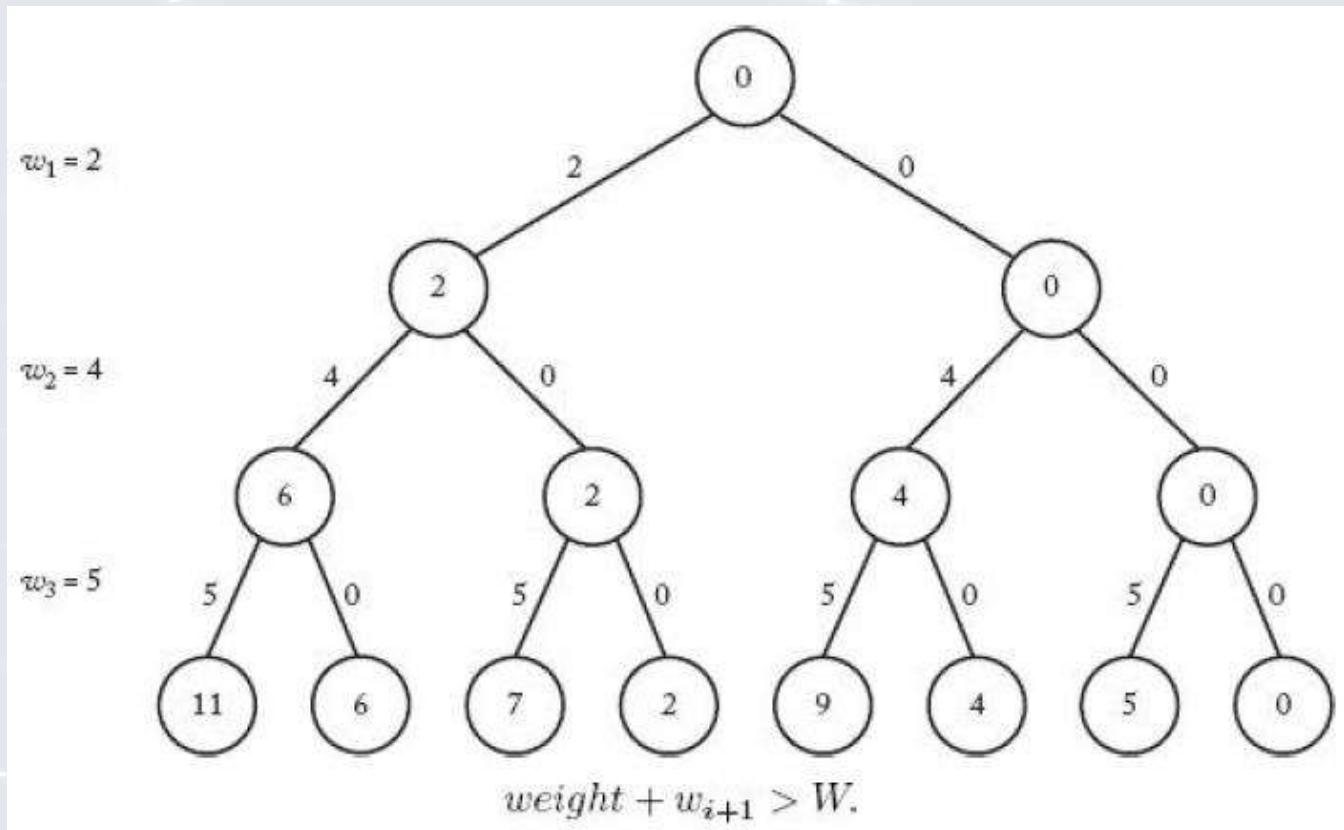
Example

- $S = \{w_1 = 5, w_2 = 6, w_3 = 10, w_4 = 11, w_5 = 16\}$ and $W = 21$
- Solutions:
 - $\{w_1, w_2, w_3\} : 5 + 6 + 10 = 21$
 - $\{w_1, w_5\} : 5 + 16 = 21$
 - $\{w_3, w_4\} : 10 + 11 = 21$

the solutions are $\{w_1, w_2, w_3\}$, $\{w_1, w_5\}$, and $\{w_3, w_4\}$.

Example 5.3

- $n = 3$
- $W = 6$
- $w_1 = 2$
- $w_2 = 4$
- $w_3 = 5$





Prune the Tree

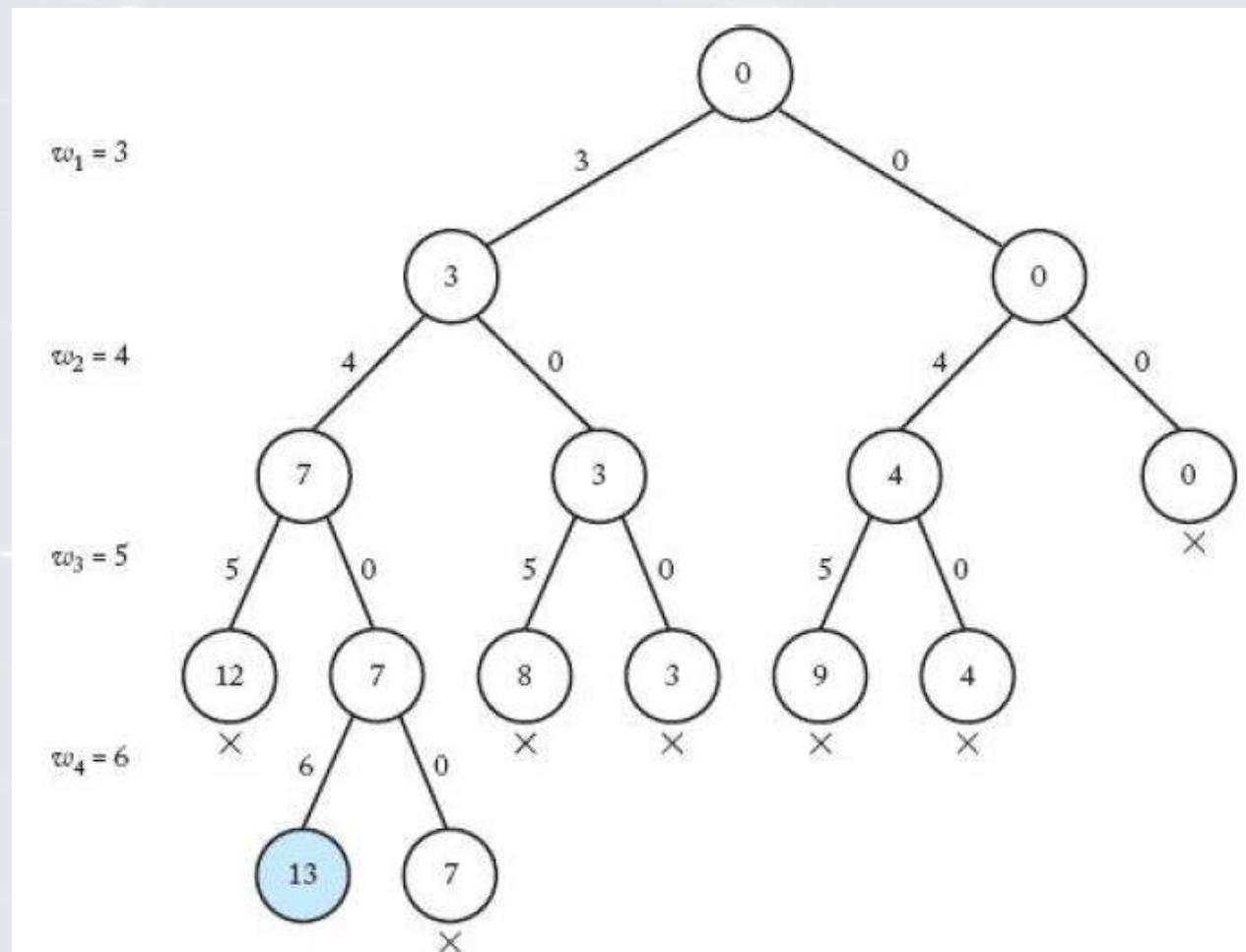
- Sort weights in non-decreasing order before search
- Let **weight** be the **sum of weights** that have been included up to a node at level i: node at level I is non-promising if **weight + w_{i+1} > W**
- Let *total* be the total weight of the remaining weights at a given node.
 - A node is non-promising if **weight + total < W**

Example 5.4

Figure 5.9 shows the pruned state space tree when backtracking is used with $n = 4$, $W = 13$, and

$$w_1 = 3 \quad w_2 = 4 \quad w_3 = 5 \quad w_4 = 6.$$

The only solution is found at the node shaded in color. The solution is $\{w_1, w_2, w_3\}$. The nonpromising nodes are marked with crosses. The nodes containing 12,



The Backtracking Algorithm for the Sum-of-Subsets Problem

Problem: Given n positive integers (weights) and a positive integer W , determine all combinations of the integers that sum to W .

Inputs: positive integer n , sorted (nondecreasing order) array of positive integers w indexed from 1 to n , and a positive integer W .

Outputs: all combinations of the integers that sum to W .

```

void sum_of_subsets (index i,
                     int weight, int total)
{
    if (promising(i))
        if (weight == W)
            cout << include[1] through include[i];
        else{
            include[i + 1] = "yes";           // Include w[i + 1].
            sum_of_subsets(i + 1, weight + w[i + 1], total - w[i + 1]);
            include[i + 1] = "no";           // Do not include w[i + 1].
            sum_of_subsets(i + 1, weight, total - w[i + 1]);
        }
    }

bool promising (index i);
{
    return (weight + total >= W) && (weight == W || weight + w[i + 1] <= W);
}

```

If n , w , W and **include** variables were defined globally, the top-level call to **sum_of_subsets()** would be as follows:

$sum_of_subsets(0, 0, total);$

where initially

st

$$total = \sum_{j=1}^n w[j].$$

Algorithm 5.4 Backtracking Algorithm for Sum-of-Subsets

- Total number of nodes in the state space searched by Algorithm 5.4

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

- Worst case could be better – i.e. for every instance, only a small portion of the state space tree is search

- **Not the case**

- For each n , it is possible to construct an instance for which algorithm visits exponentially large number of nodes – even if only seeking one solution
- **The algorithm can be efficient for many large instances.**

Optimization problem

- General algorithm of backtracking in the case of optimization problem

```
void checknode (node v)
{
    node u;
    if (value(v) is better than best)
        best = value(v);
    if (promising(v))
        for (each child u of v)
            checknode(u);
}
```

best = value of the best solution so far

value(v) = value of the solution at the node v

node is promising only **if we should expand to its children** (different from the previous algorithms)

profit is the sum of the profits of the items included up to the node

weight is the sum of the weights of those items

1. initialise variables **bound** and **totweight** to **profit** and **weight**, respectively
2. until we get to an item that, if grabbed, would bring **totweight** above **W**
greedily grab items, adding their **profits** to **bound** and their **weights** to **totweight**

Suppose the node is at level i , and the node at level k is the one that would bring the sum of the weights above W . Then

$$\text{totweight} = \text{weight} + \sum_{j=i+1}^{k-1} w_j, \quad \text{and}$$

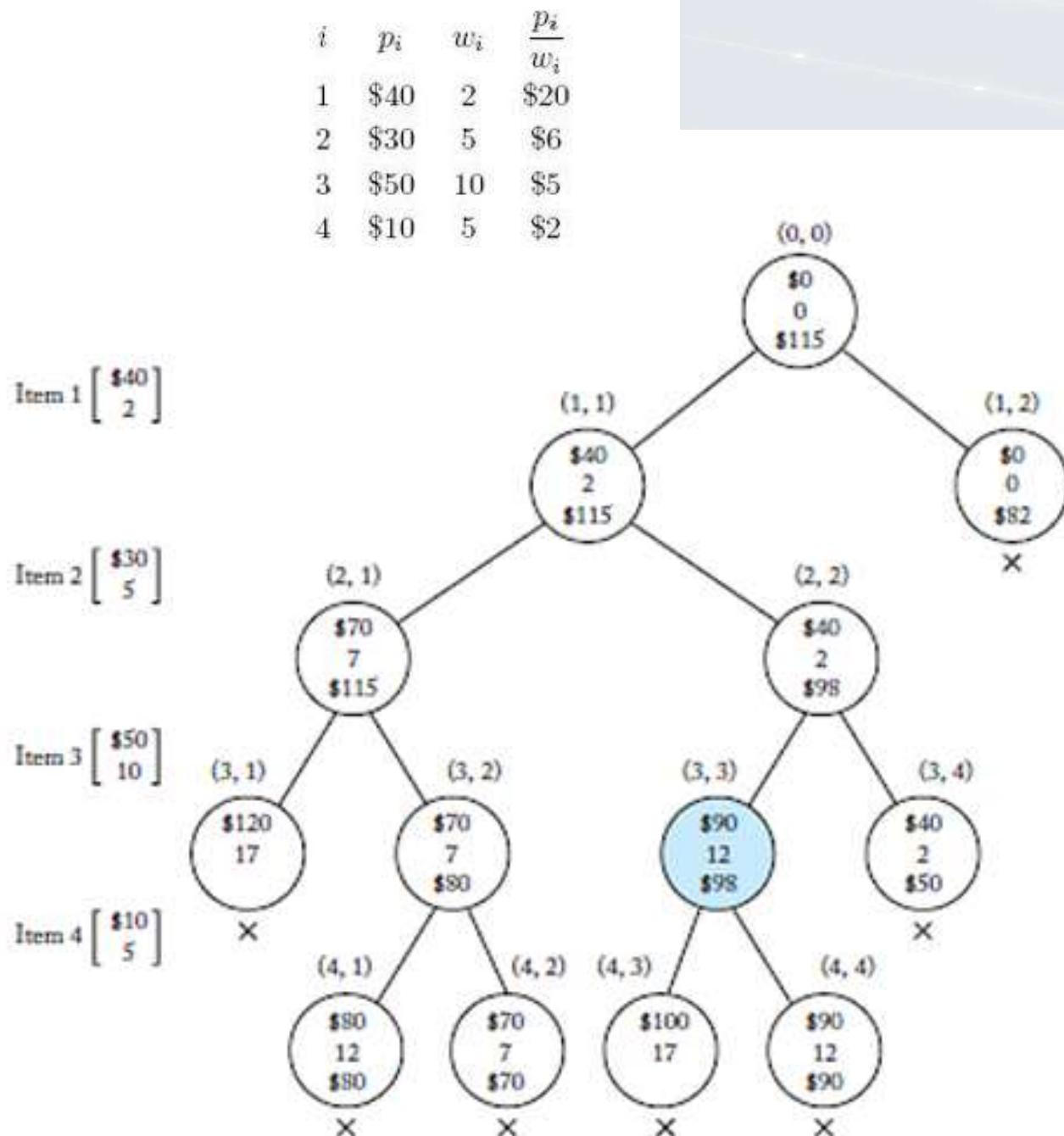
$$\text{bound} = \underbrace{\left(\text{profit} + \sum_{j=i+1}^{k-1} p_j \right)}_{\text{Profit from first } k-1 \text{ items taken}} + \underbrace{\frac{(W - \text{totweight})}{\text{Capacity available for } k\text{th item}}}_{\times} \underbrace{\frac{p_k}{w_k}}_{\text{Profit per unit weight for } k\text{th item}}$$

If maxprofit is the value of the profit in the best solution found so far, then a node at level i is nonpromising if

$$\text{bound} \leq \text{maxprofit}.$$

Suppose that $n = 4$, $W = 16$, and we have the following:

38



0-1 Knapsack Problem

- Optimization problem
- Best solution so far

The state space tree in the 0-1 Knapsack problem is the same as that in the Sum-of-Subsets problem. As shown in Section 5.4, the number of nodes in that tree is

$$2^{n+1} - 1.$$

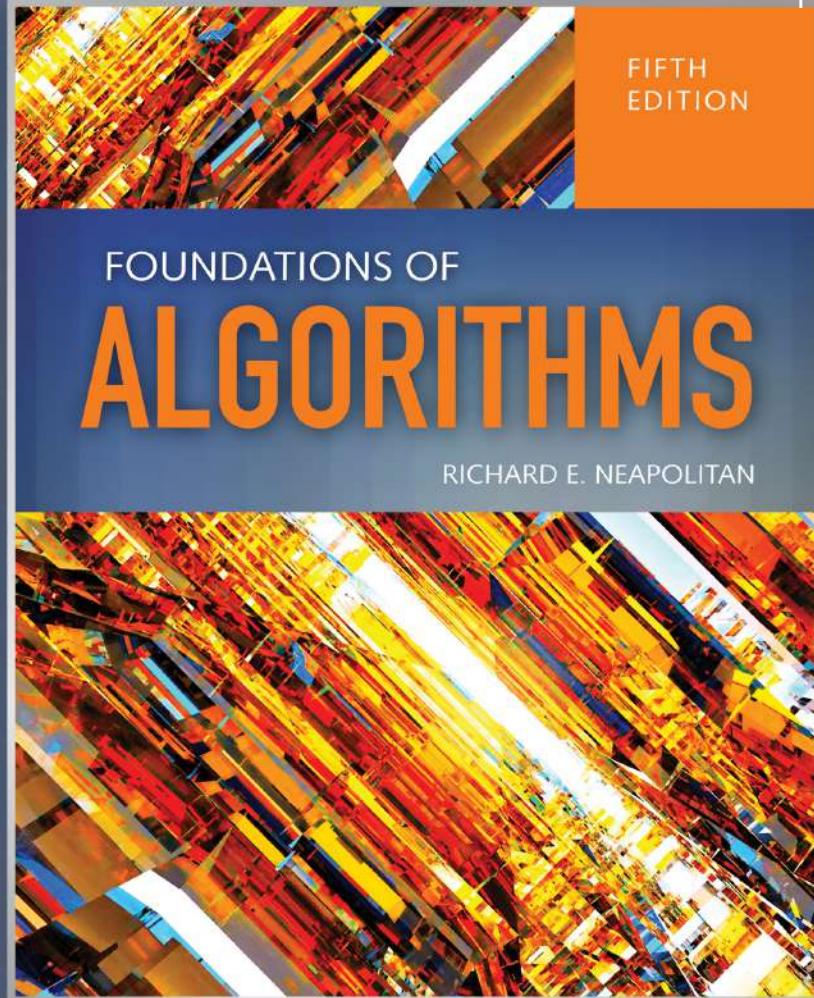
Algorithm 5.7 checks all nodes in the state space tree for the following instances. For a given n , let $W = n$, and

$$\begin{aligned} p_i &= 1 & w_i &= 1 & \text{for } 1 \leq i \leq n-1 \\ p_n &= n & w_n &= n. \end{aligned}$$

The optimal solution is to take only the n th item, and this solution will not be found until we go all the way to the right to a depth of $n - 1$ and then go left.

Comparing the Dynamic Programming Algorithm and the Backtracking Algorithm for the 0-1 Knapsack Problem

- dynamic programming algorithm for the 0-1 Knapsack problem is in $O(\min(2^n, nW))$
- In the worst case, the backtracking algorithm checks $\Theta(2^n)$ nodes
- **it is difficult to analyse theoretically the relative efficiencies of the two algorithms**
- Horowitz and Sahni (1978) found that the backtracking algorithm is **usually more efficient** than the dynamic programming algorithm.



Branch-and-Bound

Chapter 6



Objectives

- Describe the branch-and-bound technique for solving optimization problems
- Contrast the branch-and-bound technique for solving problems with the backtracking technique
- Identify when it is appropriate to use the branch-and-bound technique to solve a particular problem.
- Apply the branch-and bound technique to solve the N-Queen Problem



Objectives

- Apply the branch-and-bound technique to solve the 0-1 Knapsack Problem
- Apply the branch-and-bound technique to solve the Traveling Salesperson Problem

BACKTRACKING VERSUS BRANCH AND BOUND

4

BACKTRACKING

An algorithm for finding all solutions to some computational problems, notably constraint satisfaction problems that incrementally builds candidates to the solutions

Finds the solution to the overall issue by finding a solution to the first subproblem and then recursively solving other subproblems based on the solution of the first issue

More efficient

BRANCH AND BOUND

An algorithm for discrete and combinatorial optimization problems and mathematical optimization

Solves a given problem by dividing it into at least two new restricted subproblems

Less efficient

Visit www.PEDIAA.com



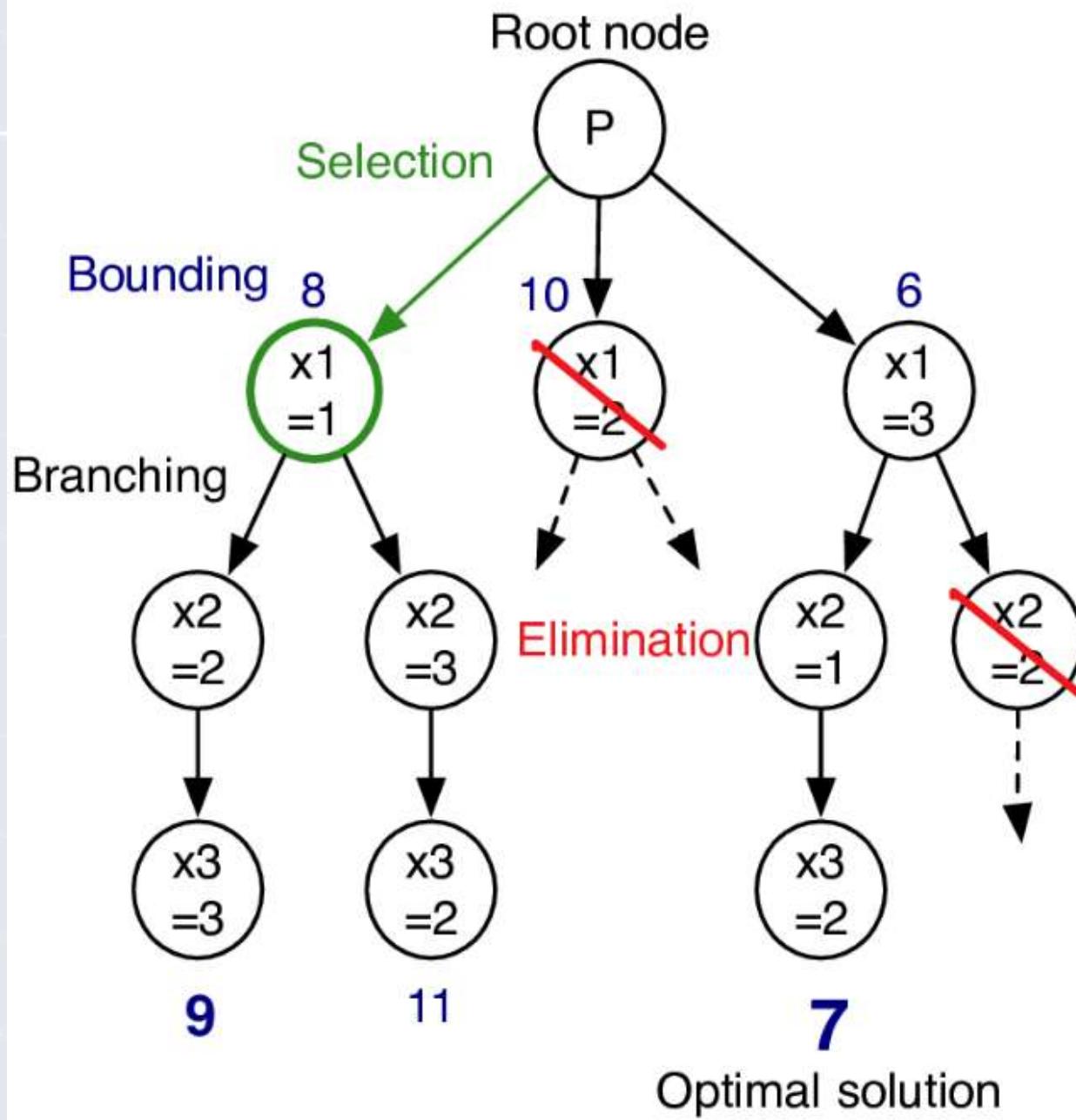
Branch-and-Bound Design Strategy

- Branch-and-bound design strategy is **similar to backtracking**
- **State space tree** used to solve problem
- Difference between branch-and-bound and backtracking:
 - 1. branch-and-bound is **not limited to a particular tree traversal**
 - 2. branch-and-bound is used **only for optimization problems**

Branch- and-Bound Design Strategy

- A branch-and-bound algorithm computes a number (**bound**) at a node to determine whether the node is promising.
- The number is a **bound on the value of the solution** that could be obtained by expanding beyond the node.
- If that bound is no **better** than the value of the best solution found so far, the node is **nonpromising**.
- Otherwise, it is **promising**.

- Note: by “better” we mean smaller or larger, depending on the problem



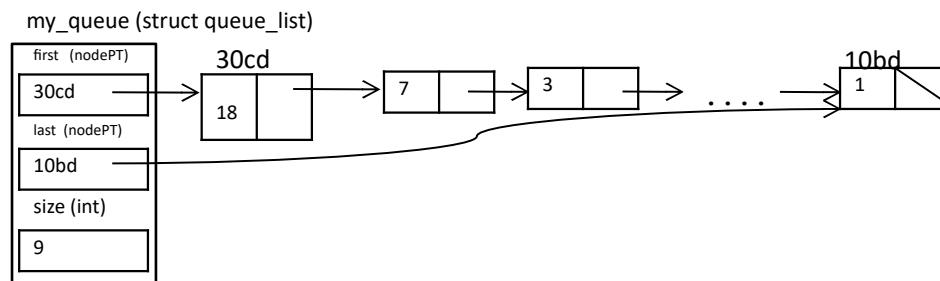
FIFO Queues

- *First-in first-out (FIFO)* queues.
- Examples of usage of FIFO queues:
 - Program execution:
 - Requests for access to memory, disk, network...
 - Resource allocation:
 - Forwarding network traffic in network switches and routers.
 - Search algorithms.
 - E.g. part of **BFS in Graphs**, level-order traversal for trees.
- Main operations:
 - **put** - inserts an item at the end of the queue.
 - **get** - removes the item from the head of the queue.
- 2 implementations for FIFO queues: **single linked list & array**

List Implementation for FIFO Queues

- A FIFO queue is essentially a list.
- **put(&queue, item)** inserts that item at the **end** of the list. - O(1)
 - Assumption: the list data type contains a pointer to the last element.
- **get(&queue)** removes (and returns) the item at the **beginning** of the list. - O(1)

```
typedef struct node * nodePT;  
  
struct queue_list {  
    nodePT first;  
    nodePT last;  
    int size;  
};
```





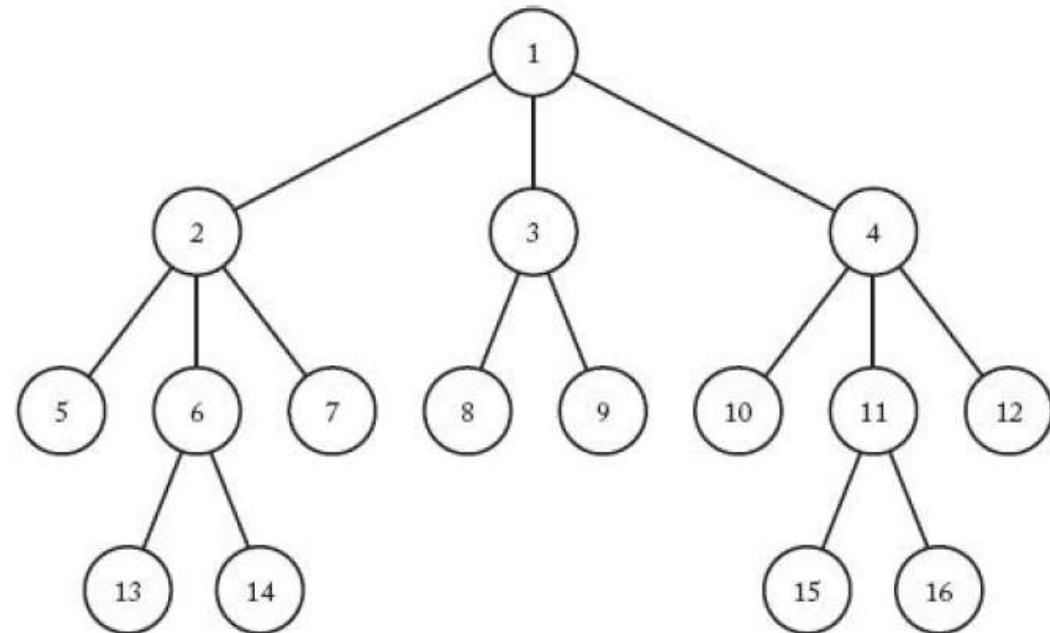
Breadth-first Tree Search

- Use Queue - FIFO
- Visit root first
- Visit all nodes at level 1 next
- Visit all nodes at level 2 next . . .
- Visit all nodes at level n

```

void breadth-first-tree-search (tree T);
{
    queue_of_node Q;
    node u, v,
    initialize(Q);                                // Initialize Q to be empty.
    v = root of T;
    visit v;
    enqueue(Q, v);
    while (! empty(Q)) {
        dequeue(Q, v);
        for (each child u of v){
            visit u;
            enqueue(Q, u);
        }
    }
}

```





0-1 Knapsack Problem

- **First solution:** Breadth-First Search with Branch-and-Bound Pruning
- Let ***weight*** and ***profit*** be the total weight and total profit of the items that have been included up to a node

Breadth-First Search with Branch-and-Bound Pruning¹³

- Initialize ***totweight*** to ***weight*** and ***bound*** to ***profit***
- Greedily grab items
 - Add item's weight to ***totweight*** and items ***profit*** to ***bound***
- Until an item's weight would bring ***totweight*** above ***W***
- Grab the fraction of the item allowed by available weight and add ***profit*** of the fraction to ***bound***

Determine if a mode is promising

- A node is **non-promising** if this bound is $\leq \maxprofit$
– the value of the best solution found up to that point
- A node is also **non-promising** if $\text{weight} \geq W$

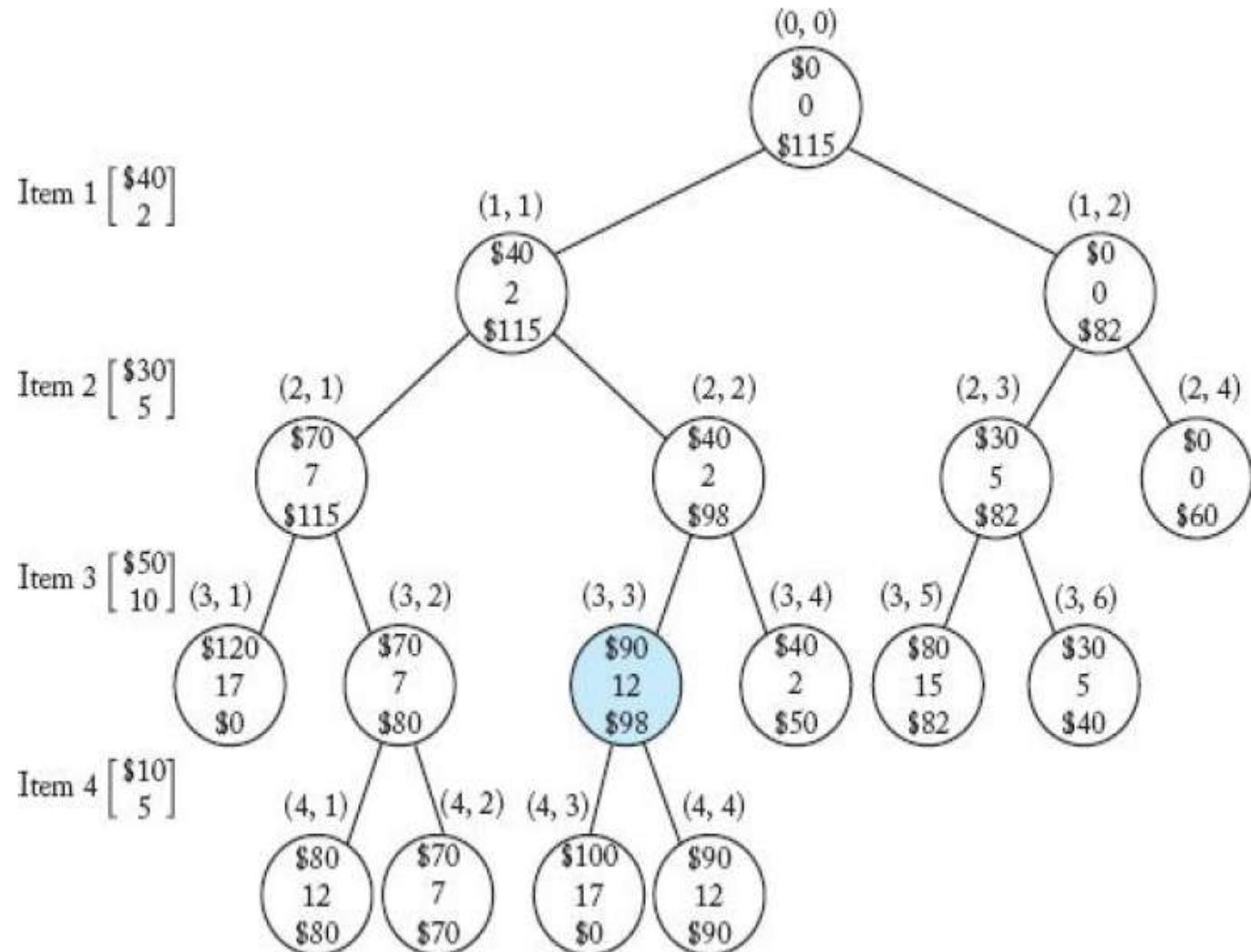
Exercise 5.6. That is, $n = 4$, $W = 16$, and we have the following:

i	p_i	w_i	$\frac{p_i}{w_i}$
1	\$40	2	\$20
2	\$30	5	\$6
3	\$50	10	\$5
4	\$10	5	\$2

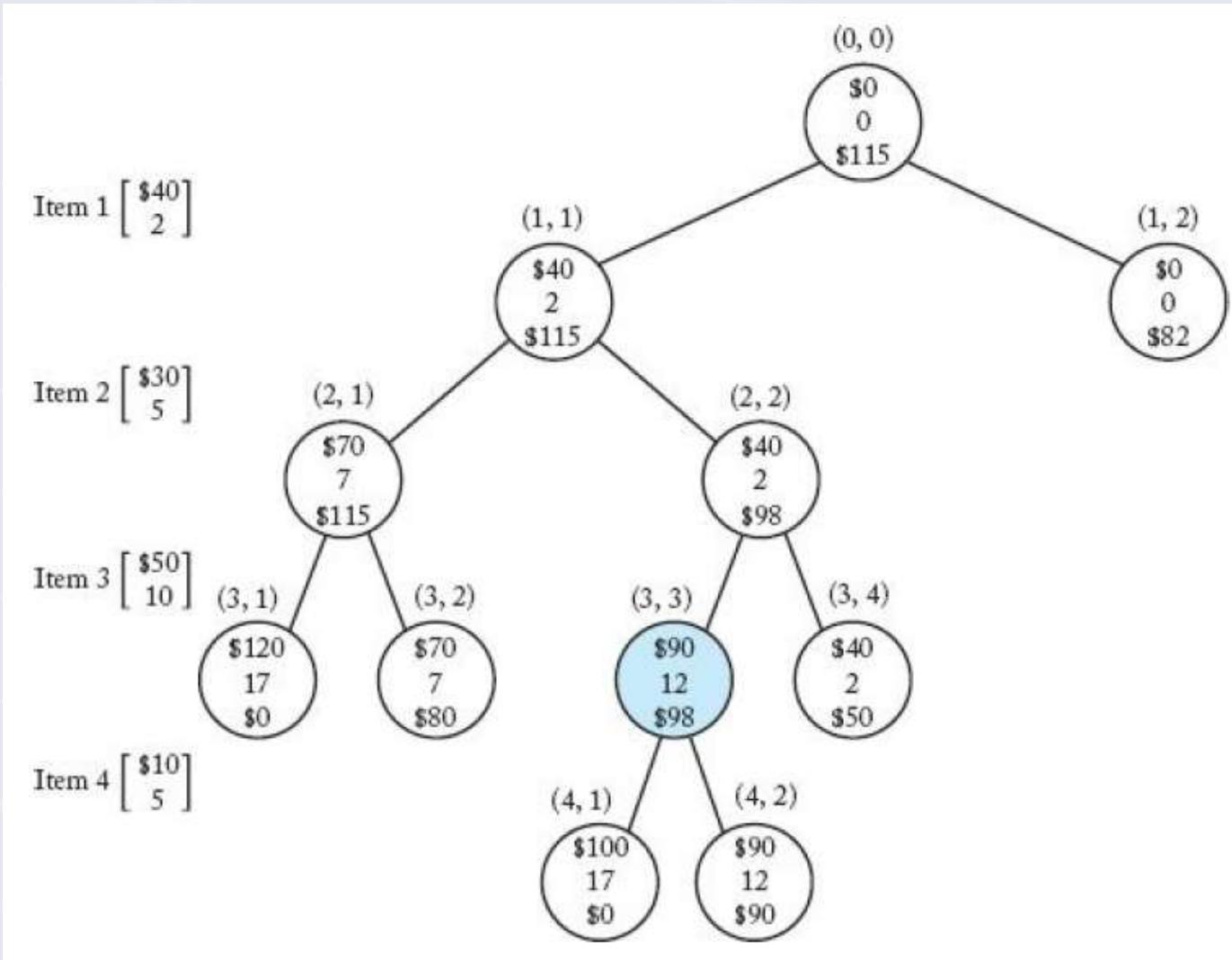
The pruned state space tree produced using breadth-first search with branch-and-bound pruning

$n = 4, W = 16$

i	p_i	w_i	$\frac{p_i}{w_i}$
1	\$40	2	\$20
2	\$30	5	\$6
3	\$50	10	\$5
4	\$10	5	\$2



The pruned state space tree produced using best-first search with branch-and-bound pruning





Summary Branch-and-Bound

- State space pruning
- **Breadth-first traversal** of state space tree
- Calculate a **bound** for a node
 - Bound best possible solution obtainable by expanding this node
 - Choose technique for calculating bound that produces best solution
- If bound of a node not better than the best solution found so far or current node conditions exceed problem instance limits, do not expand the node



Summary Branch and Bound

- BFS branch-and-bound pruning, decision on visiting node's children is made at time node is visited. **Maxprofit** can change from time decision is made to visit until node is actually visited wasting time

Use bound to Improve Search

- After visiting all of the children of a given node, expand the one with the best bound
- Determine promising unexpanded node with greatest bound
- Order determined by best bound rather than pre-determined
- **Use priority queue for implementation**

Traveling Salesperson Problem

- **Goal:** find an optimal tour
 - Starts at a given city
 - Visits every city exactly once
 - Returns to the starting city
- Such that the total distance travelled is minimal



Definitions

- A **walk** in a network is a finite sequence of edges such that the end vertex of one edge is the start vertex of the next
 - A **tour** is a walk which visits every vertex, returning to its starting vertex.
 - A tour which visits every vertex exactly once is a **Hamiltonian cycle**.
 - The Travelling Salesman Problem involves finding a **tour of minimum total weight**.
-
- In the classical problem, each vertex must be visited **exactly** once before returning to the start.
 - In the practical problem, each vertex must be visited **at least** once before returning to the start.

Figure 6.4 Adjacency matrix representation of a graph that has an edge from every vertex to every other vertex (left), and the nodes in the graph and the edges in an optimal tour (right).

22

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0

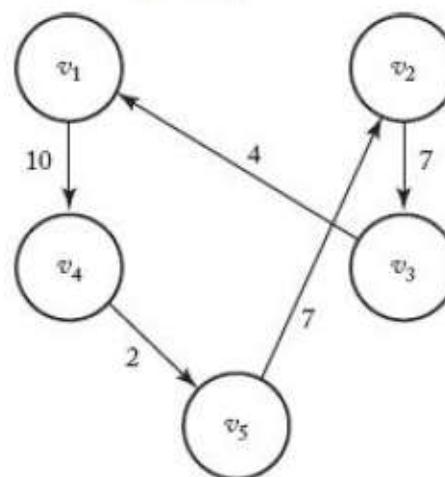
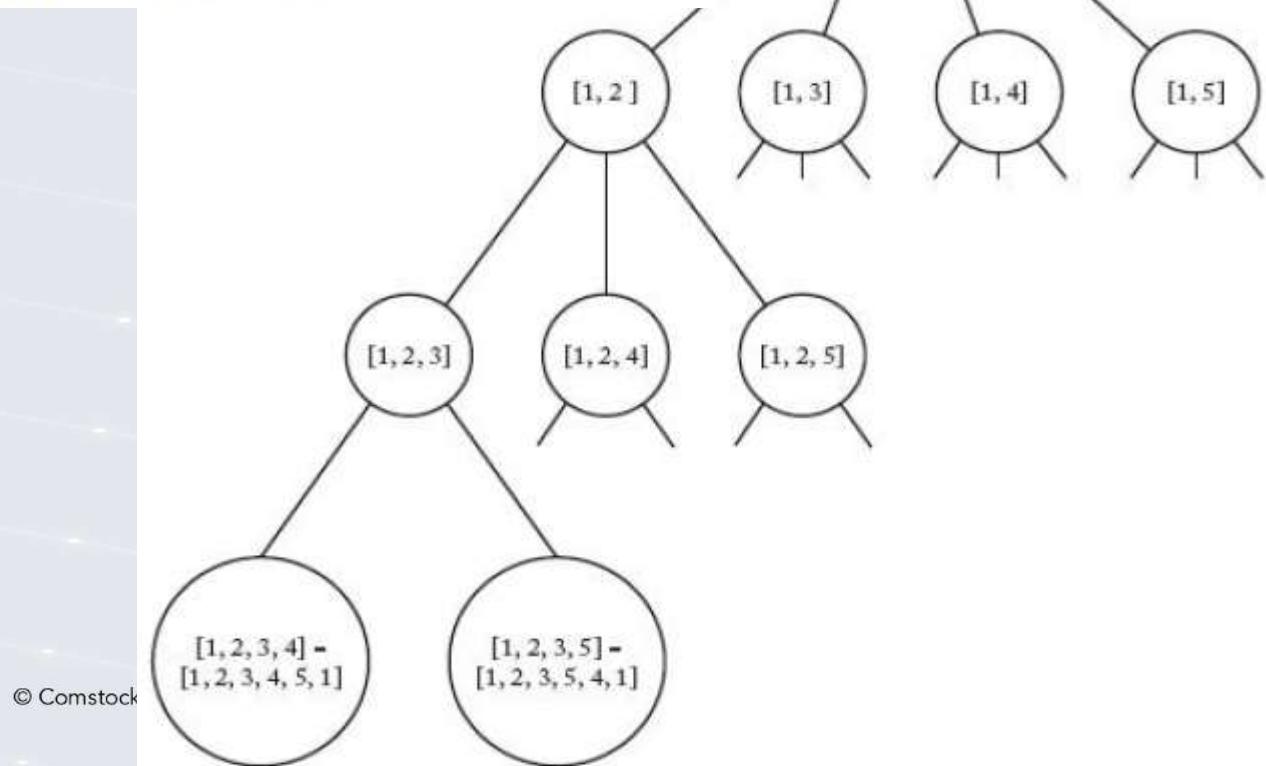


Figure 6.5 A state space tree for an instance of the Traveling Salesperson problem in which there are five vertices. The indices of the vertices in the partial tour are stored at each node.





Bounds

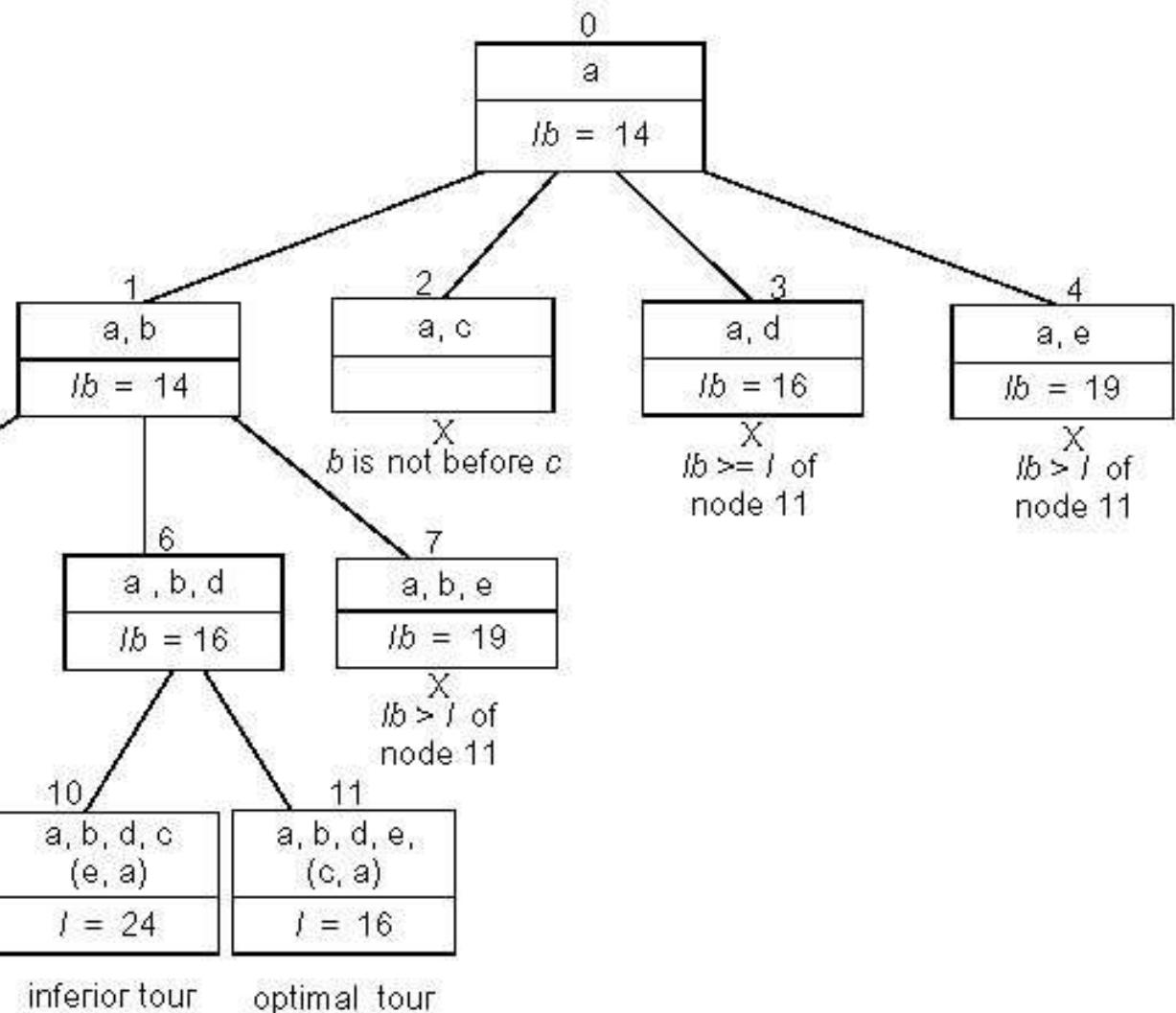
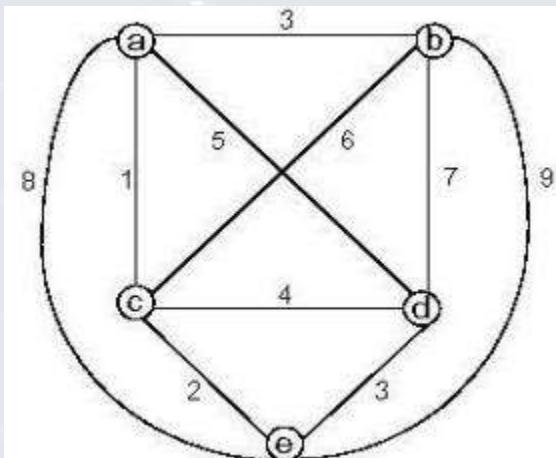
- There is **no quick algorithm** for solving The Travelling Salesman Problem, in most practical situations it is good enough to find a good solution, which **may not be optimal**.
- We calculate '**upper bounds**' and '**lower bounds**' and use these to trap the optimal solution.



Bound

- **Determine lower bound on the length of any tour** obtainable by expanding beyond given node
- Adjacency Matrix to represent distances between cities
- **Length of an edge** taken when leaving a vertex \geq **length of the shortest edge** leaving that vertex
- **Bound** = cost of traveling the sub-path to arrive at node i plus the cost of the minimum sub-path to travel from the last vertex in the path back to vi
- A node is **non-promising** if **bound** \geq **minlength**

Example: Traveling Salesman Problem



Assignment Problem



Select one element in each row of the cost matrix C so that:

- no two selected elements are in the same column
- the sum is minimized

Example

	Job 1	Job 2	Job 3	Job 4
Person a	9	2	7	8
Person b	6	4	3	7
Person c	5	8	1	8
Person d	7	6	9	4

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

Lower bound: Any solution to this problem will have total cost at least: $2 + 3 + 1 + 4$ (or $5 + 2 + 1 + 4$)

First two levels of the state-space tree

job 1	job 2	job 3	job 4	
person <i>a</i>	9	2	7	8
person <i>b</i>	6	4	3	7
person <i>c</i>	5	8	1	8
person <i>d</i>	7	6	9	4

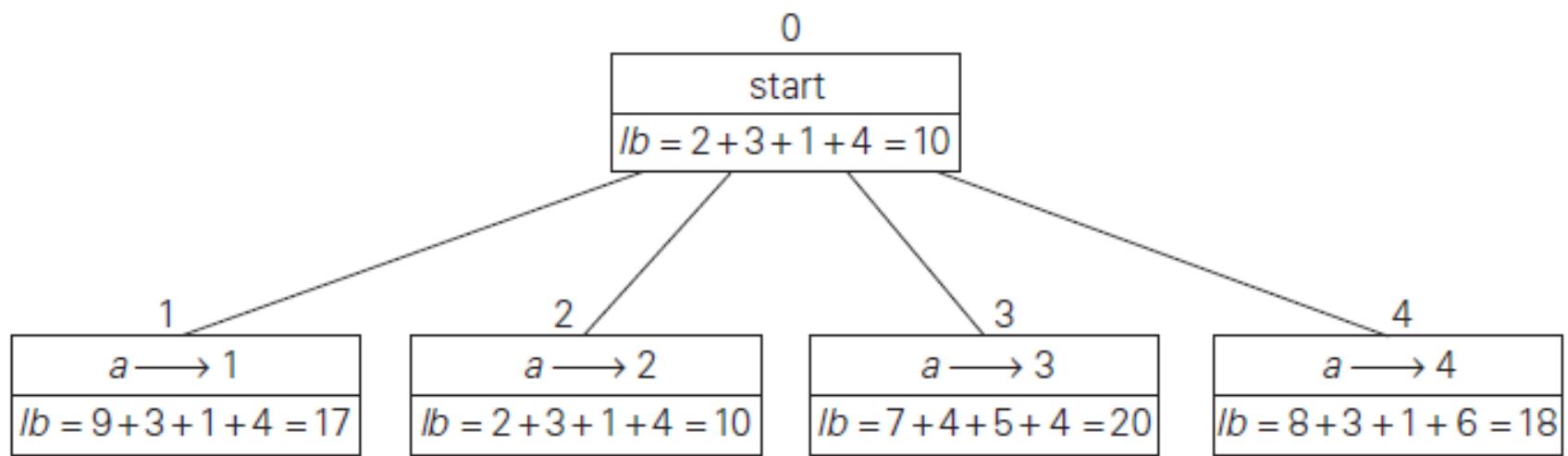


FIGURE 12.5 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person *a* and the lower bound value, *lb*, for this node.

First three levels of the state-space tree



job 1	job 2	job 3	job 4	
person <i>a</i>	9	2	7	8
person <i>b</i>	6	4	3	7
person <i>c</i>	5	8	1	8
person <i>d</i>	7	6	9	4

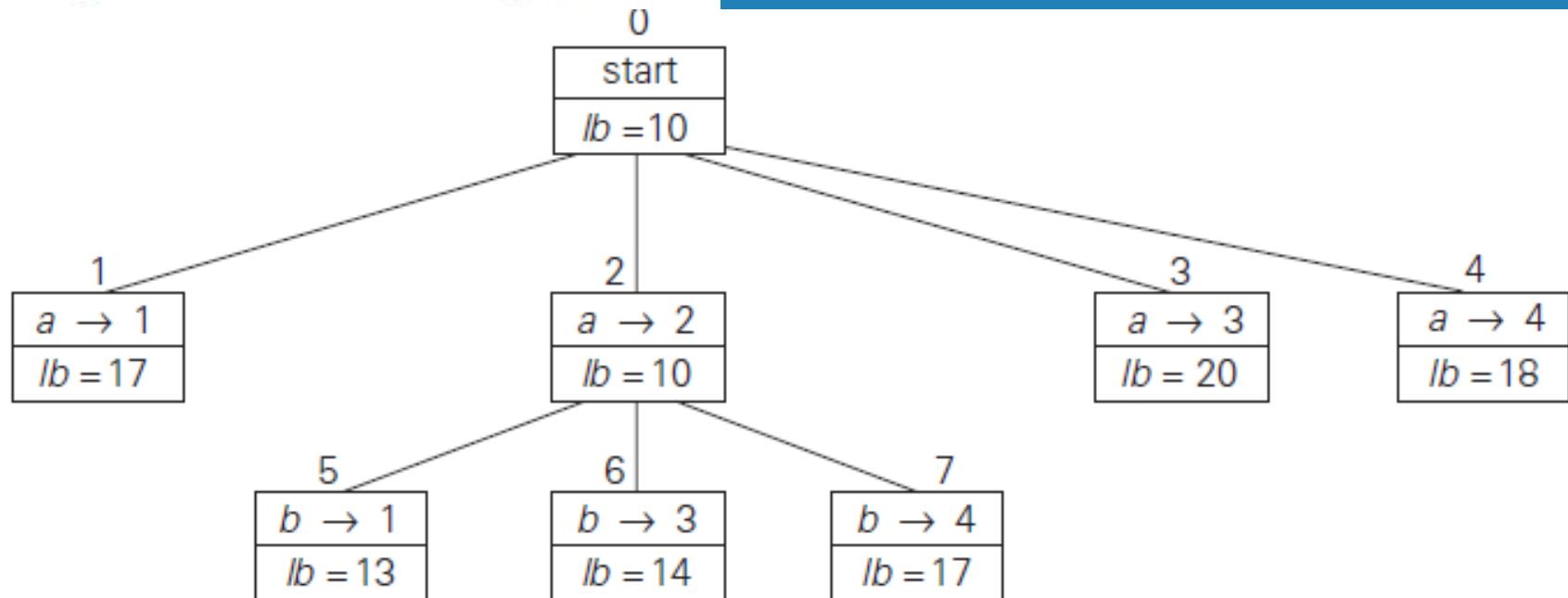
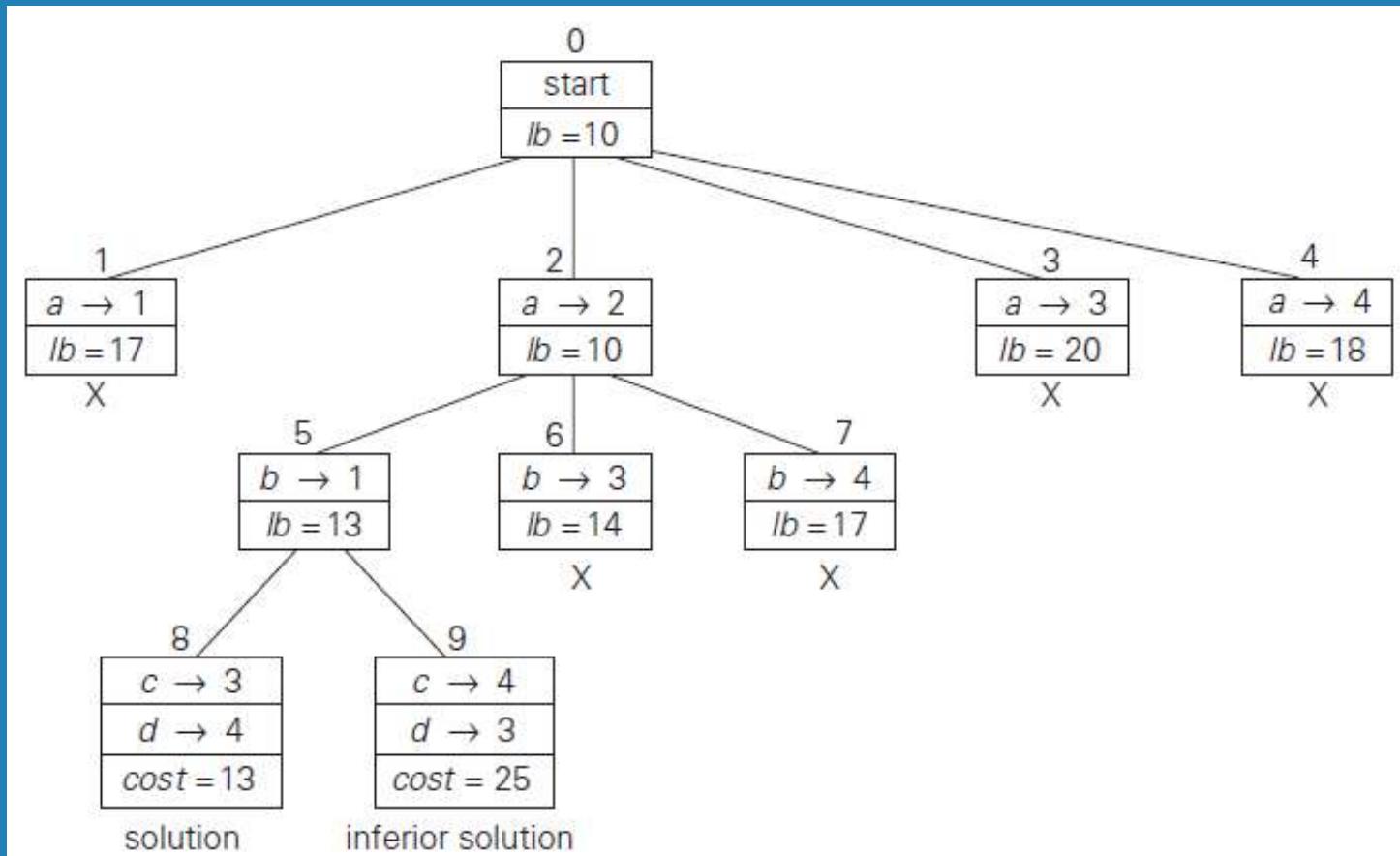


FIGURE 12.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm.

Complete state-space tree



job 1	job 2	job 3	job 4	
9	2	7	8	person <i>a</i>
6	4	3	7	person <i>b</i>
5	8	1	8	person <i>c</i>
7	6	9	4	person <i>d</i>



Lower Bounds



Lower bound: an estimate on a minimum amount of work needed to solve a given problem

Examples:

- number of comparisons needed to find the largest element in a set of n numbers
- number of comparisons needed to sort an array of size n
- number of comparisons necessary for searching in a sorted array
- number of multiplications needed to multiply two n -by- n matrices

Lower Bounds (cont.)



- Lower bound can be
 - an exact count
 - an efficiency class (Ω)
- Tight lower bound: there exists an algorithm with the same efficiency as the lower bound

Problem	Lower bound	Tightness
sorting	$\Omega(n \log n)$	yes
searching in a sorted array	$\Omega(\log n)$	yes
element uniqueness	$\Omega(n \log n)$	yes
n -digit integer multiplication	$\Omega(n)$	unknown
multiplication of n -by- n matrices	$\Omega(n^2)$	unknown

Methods for Establishing Lower Bounds



- ➊ trivial lower bounds
- ➋ information-theoretic arguments (decision trees)
- ➌ adversary arguments
- ➍ problem reduction

Trivial Lower Bounds

Trivial lower bounds: based on counting the number of items that must be processed in input and generated as output



Examples

- finding max element
- polynomial evaluation
- sorting
- element uniqueness
- Hamiltonian circuit existence

Conclusions

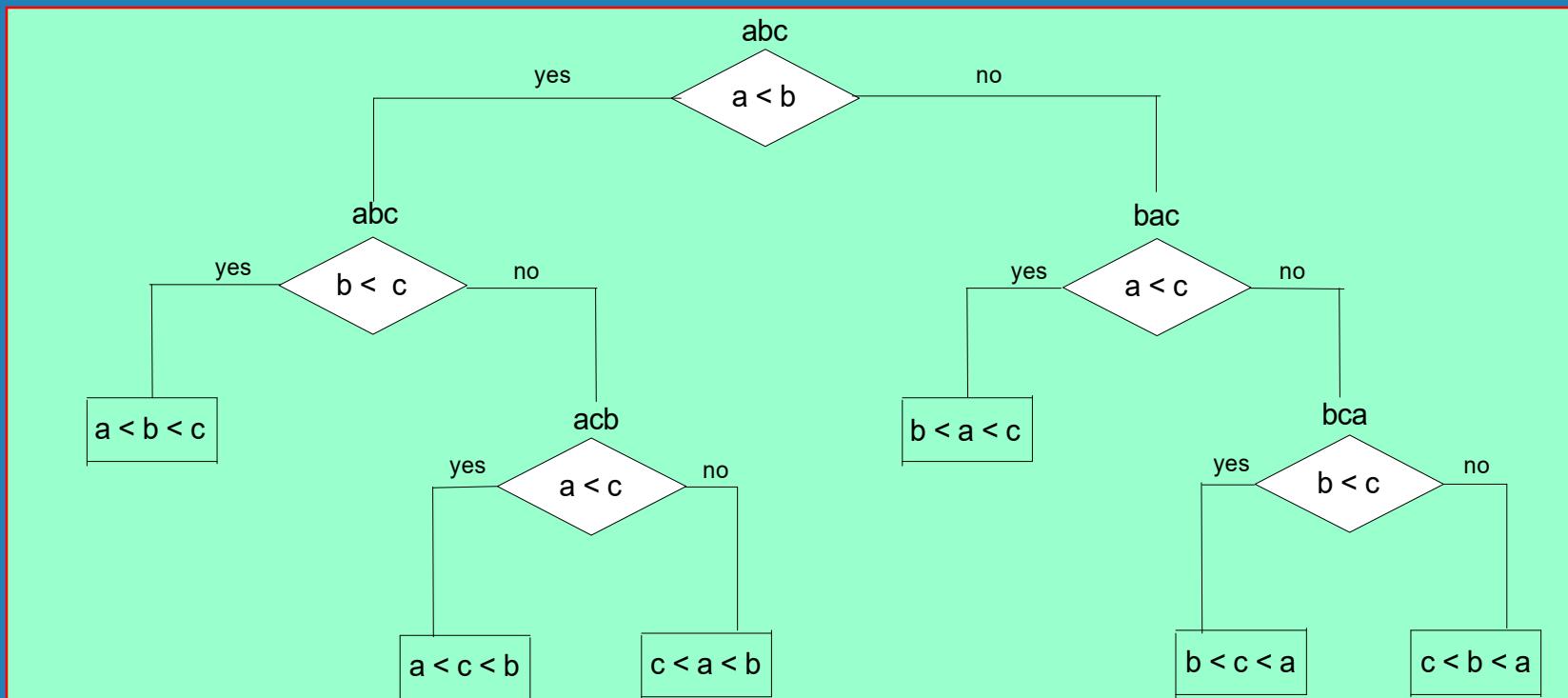
- may and may not be useful
- be careful in deciding how many elements must be processed

Decision Trees

Decision tree — a convenient model of algorithms involving comparisons in which:

- internal nodes represent comparisons
- leaves represent outcomes

Decision tree for 3-element insertion sort



Decision Trees and Sorting Algorithms



- Any comparison-based sorting algorithm can be represented by a decision tree
- Number of leaves (outcomes) $\geq n!$
- Height of binary tree with $n!$ leaves $\geq \lceil \log_2 n! \rceil$
- Minimum number of comparisons in the worst case $\geq \lceil \log_2 n! \rceil$ for any comparison-based sorting algorithm
- $\lceil \log_2 n! \rceil \approx n \log_2 n$
- This lower bound is tight (mergesort)

Adversary Arguments



Adversary argument: a method of proving a lower bound by playing role of adversary that makes algorithm work the hardest by adjusting input

Example 1: “Guessing” a number between 1 and n with yes/no questions

Adversary: Puts the number in a larger of the two subsets generated by last question

Example 2: Merging two sorted lists of size n

$$a_1 < a_2 < \dots < a_n \text{ and } b_1 < b_2 < \dots < b_n$$

Adversary: $a_i < b_j$ iff $i < j$

Output $b_1 < a_1 < b_2 < a_2 < \dots < b_n < a_n$ requires $2n-1$ comparisons of adjacent elements

Lower Bounds by Problem Reduction



Idea: If problem P is at least as hard as problem Q , then a lower bound for Q is also a lower bound for P .

Hence, find problem Q with a known lower bound that can be reduced to problem P in question.

Example: P is finding MST for n points in Cartesian plane
 Q is element uniqueness problem (known to be in $\Omega(n \log n)$)

Problem Types: Optimization and Decision



- ➊ **Optimization problem:** find a solution that maximizes or minimizes some objective function
- ➋ **Decision problem:** answer yes/no to a question

Many problems have decision and optimization versions.

E.g.: traveling salesman problem

- ➊ *optimization:* find Hamiltonian cycle of minimum length
- ➋ *decision:* find Hamiltonian cycle of length $\leq m$

Decision problems are more convenient for formal investigation of their complexity.

Transform and Conquer



This group of techniques solves a problem by a *transformation* to

- ➊ a simpler/more convenient instance of the same problem (*instance simplification*)
- ➋ a different representation of the same instance (*representation change*)
- ➌ a different problem for which an algorithm is already available (*problem reduction*)

Instance simplification - Presorting



Solve a problem's instance by transforming it into another simpler/easier instance of the same problem

Presorting

Many problems involving lists are easier when list is sorted, e.g.

- ➊ searching
- ➋ computing the median (selection problem)
- ➌ checking if all elements are distinct (element uniqueness)

Also:

- ➊ Topological sorting helps solving some problems for dags.
- ➋ Presorting is used in many geometric algorithms.

How fast can we sort ?



Efficiency of algorithms involving sorting depends on efficiency of sorting.

Theorem (see Sec. 11.2): $\lceil \log_2 n! \rceil \approx n \log_2 n$ comparisons are necessary in the worst case to sort a list of size n by any comparison-based algorithm.

Note: About $n \log_2 n$ comparisons are also sufficient to sort array of size n (by mergesort).

Searching with presorting



Problem: Search for a given K in $A[0..n-1]$

Presorting-based algorithm:

Stage 1 Sort the array by an efficient sorting algorithm

Stage 2 Apply binary search

Efficiency: $\Theta(n \log n) + O(\log n) = \Theta(n \log n)$

Good or bad?

**Why do we have our dictionaries, telephone directories, etc.
sorted?**

Element Uniqueness with presorting



- ➊ Presorting-based algorithm

Stage 1: sort by efficient sorting algorithm (e.g. mergesort)

Stage 2: scan array to check pairs of adjacent elements

Efficiency: $\Theta(n \log n) + O(n) = \Theta(n \log n)$

- ➋ Brute force algorithm

Compare all pairs of elements

Efficiency: $O(n^2)$

- ➌ Another algorithm? Hashing

Instance simplification – Gaussian Elimination

Given: A system of n linear equations in n unknowns with an arbitrary coefficient matrix.

Transform to: An equivalent system of n linear equations in n unknowns with an upper triangular coefficient matrix.

Solve the latter by substitutions starting with the last equation and moving up to the first one.

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2$$

$$a_{1,1}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1$$

$$a_{22}x_2 + \dots + a_{2n}x_n = b_2$$



Gaussian Elimination (cont.)



The transformation is accomplished by a sequence of elementary operations on the system's coefficient matrix (which don't change the system's solution):

for $i \leftarrow 1$ to $n-1$ do

 replace each of the subsequent rows (i.e., rows $i+1, \dots, n$) by a difference between that row and an appropriate multiple of the i -th row to make the new coefficient in the i -th column

 of that row 0

Example of Gaussian Elimination



Solve $2x_1 - 4x_2 + x_3 = 6$
 $3x_1 - x_2 + x_3 = 11$
 $x_1 + x_2 - x_3 = -3$

Gaussian elimination

$$\begin{array}{cccc|c} 2 & -4 & 1 & 6 \\ 3 & -1 & 1 & 11 & \text{row2} - (3/2)\text{*row1} \\ 1 & 1 & -1 & -3 & \text{row3} - (1/2)\text{*row1} \end{array}$$

$$\begin{array}{cccc|c} 2 & -4 & 1 & 6 \\ 0 & 5 & -1/2 & 2 & \\ 0 & 3 & -3/2 & -6 & \text{row3} - (3/5)\text{*row2} \end{array}$$

$$\begin{array}{cccc|c} 2 & -4 & 1 & 6 \\ 0 & 5 & -1/2 & 2 & \\ 0 & 0 & -6/5 & -36/5 & \end{array}$$

Backward substitution

$$\begin{aligned} x_3 &= (-36/5) / (-6/5) = 6 \\ x_2 &= (2 + (1/2)*6) / 5 = 1 \end{aligned}$$

$$x_1 = (6 + 4*1)/2 = 2$$

Pseudocode and Efficiency of Gaussian Elimination



Stage 1: Reduction to an upper-triangular matrix

for $i \leftarrow 1$ to $n-1$ do

 for $j \leftarrow i+1$ to n do

 for $k \leftarrow i$ to $n+1$ do

$A[j, k] \leftarrow A[j, k] - A[i, k] * A[j, i] / A[i, i]$ //improve!

Stage 2: Back substitutions

for $j \leftarrow n$ downto 1 do

$t \leftarrow 0$

 for $k \leftarrow j+1$ to n do

$t \leftarrow t + A[j, k] * x[k]$

$x[j] \leftarrow (A[j, n+1] - t) / A[j, j]$

Efficiency: $\Theta(n^3) + \Theta(n^2) = \Theta(n^3)$



Problem Reduction



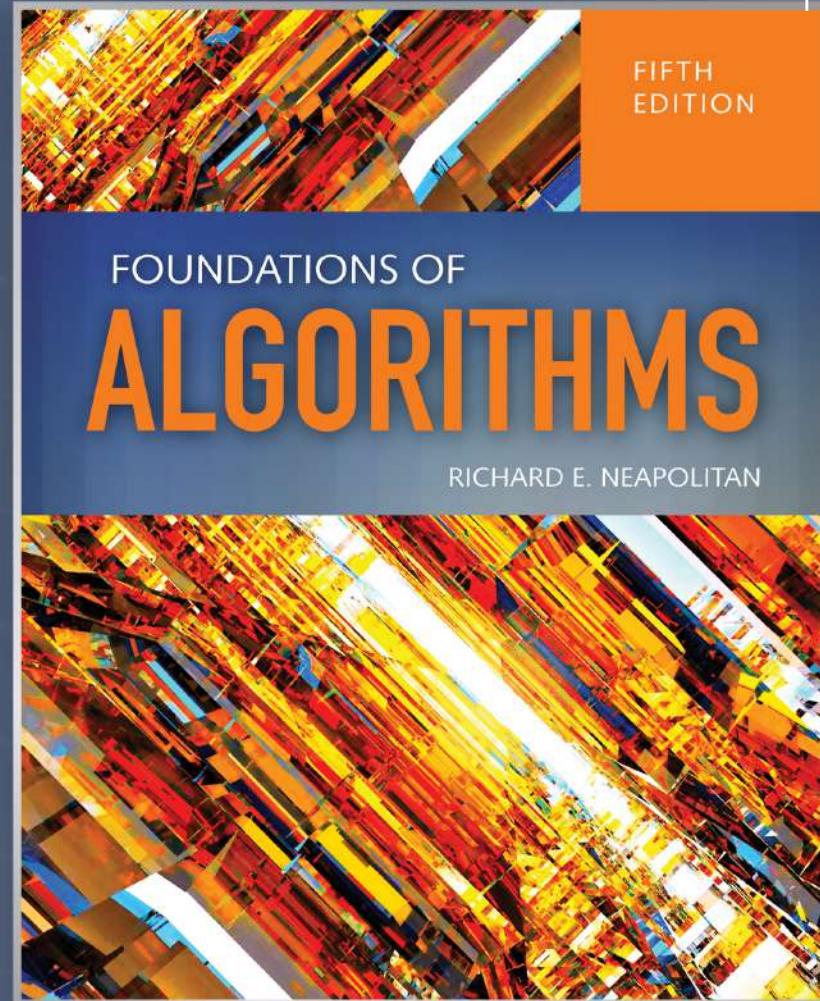
This variation of transform-and-conquer solves a problem by transforming it into different problem for which an algorithm is already available.

To be of practical value, the combined time of the transformation and solving the other problem should be smaller than solving the problem as given by another method.

Examples of Solving Problems by Reduction



- ➊ computing $\text{lcm}(m, n)$ via computing $\text{gcd}(m, n)$
- ➋ counting number of paths of length n in a graph by raising the graph's adjacency matrix to the n -th power
- ➌ transforming a maximization problem to a minimization problem and vice versa (also, min-heap construction)
- ➍ linear programming
- ➎ reduction to graph problems (e.g., solving puzzles via state-space graphs)



Introduction to Computational Complexity: The Sorting Problem

Chapter 7



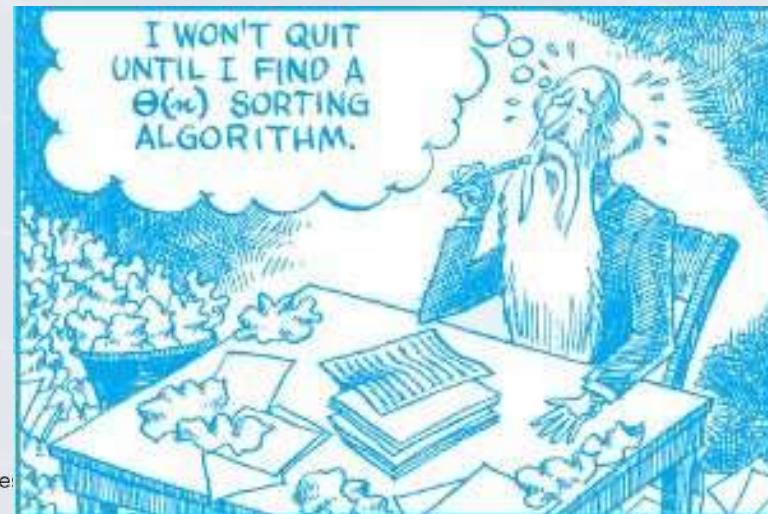
Objectives

- Use computational complexity analysis to determine lower bounds on sorting algorithms
- Analyze algorithms that sort only by comparison of keys
- Use computational complexity analysis to determine lower bounds of quadratic time on the class of sorting algorithms that remove one inversion per comparison
- **Analyze the class of $\Theta(n \lg n)$ sorting algorithms**

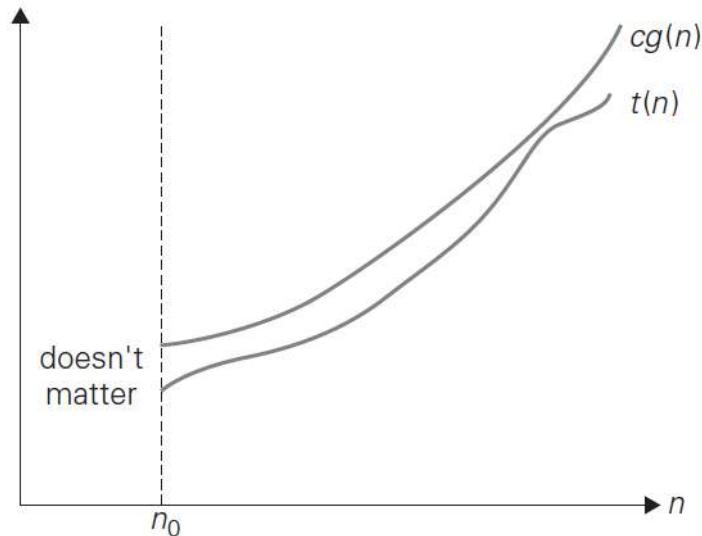


Computational Complexity

- Study of all possible algorithms that solve a given problem
- Determine a lower bound on the efficiency of all algorithms for a given problem
- Problem analysis as opposed to algorithm analysis



O-notation



DEFINITION A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n , i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

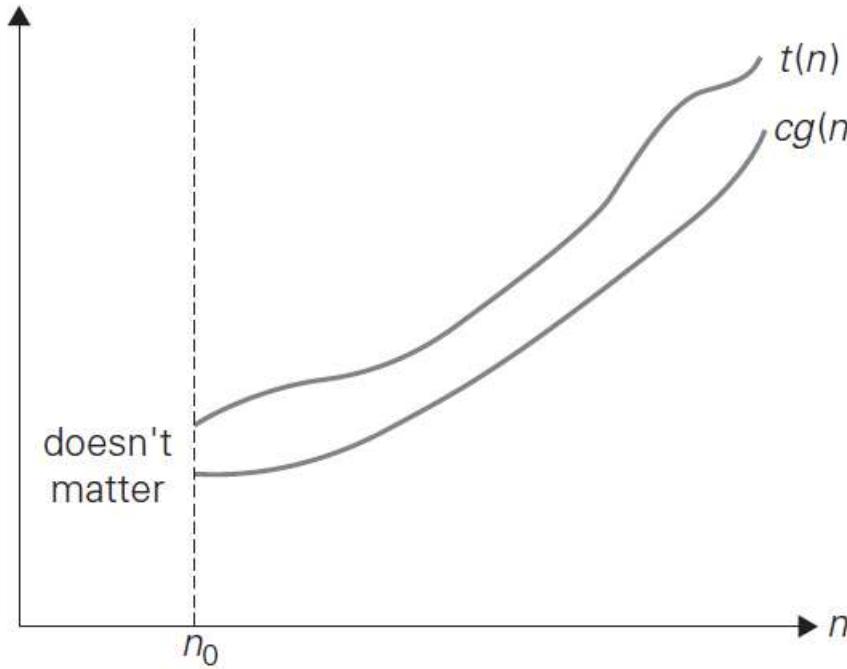
$$t(n) \leq cg(n) \quad \text{for all } n \geq n_0.$$

As an example, let us formally prove one of the assertions made in the introduction: $100n + 5 \in O(n^2)$. Indeed,

$$100n + 5 \leq 100n + n \text{ (for all } n \geq 5\text{)} = 101n \leq 101n^2.$$

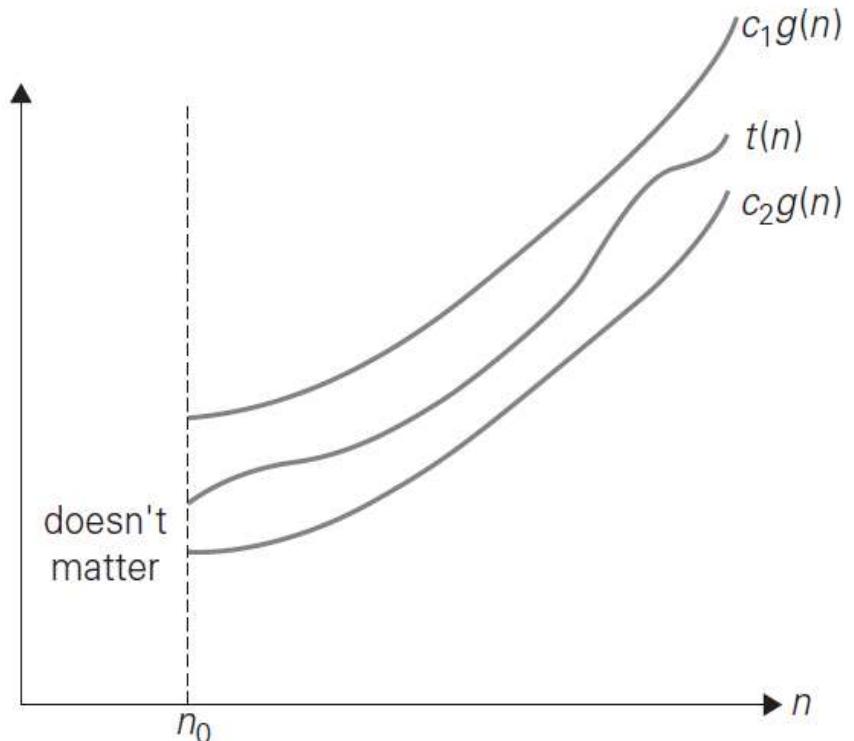
Thus, as values of the constants c and n_0 required by the definition, we can take 101 and 5, respectively.

Ω and Θ notations



$$t(n) \in \Omega(g(n)).$$

$$t(n) \geq cg(n) \quad \text{for all } n \geq n_0.$$



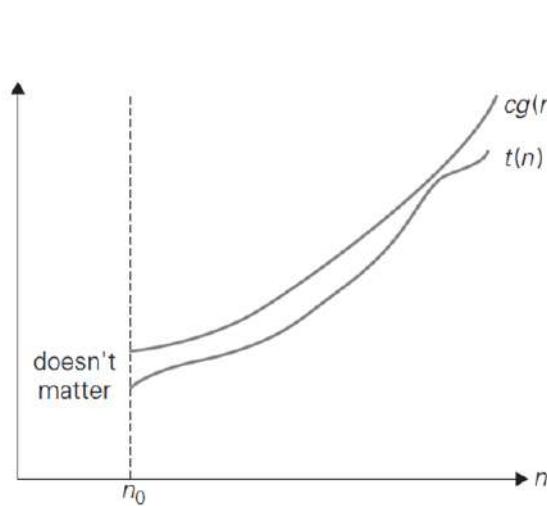
$$t(n) \in \Theta(g(n)).$$

$$c_2g(n) \leq t(n) \leq c_1g(n) \quad \text{for all } n \geq n_0.$$

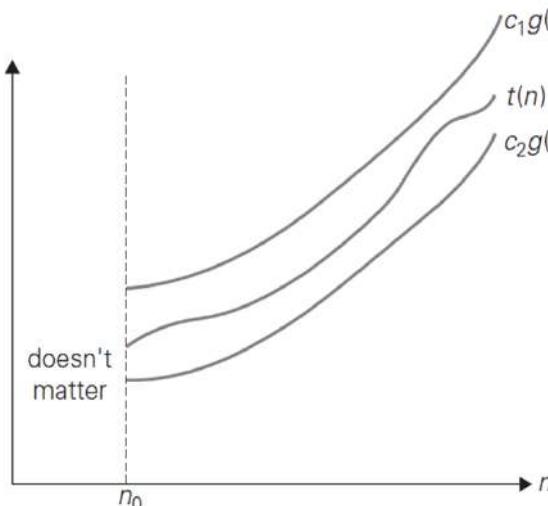
Comparing Orders of Growth

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

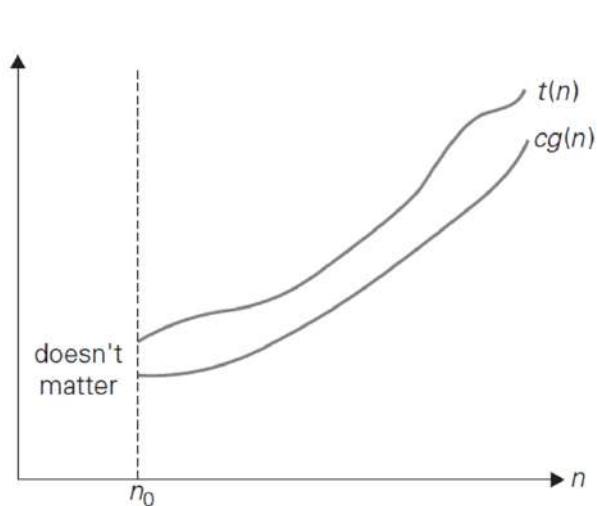
Note that the first two cases mean that $t(n) \in O(g(n))$, the last two mean that $t(n) \in \Omega(g(n))$, and the second case means that $t(n) \in \Theta(g(n))$.



$t(n) \in O(g(n))$



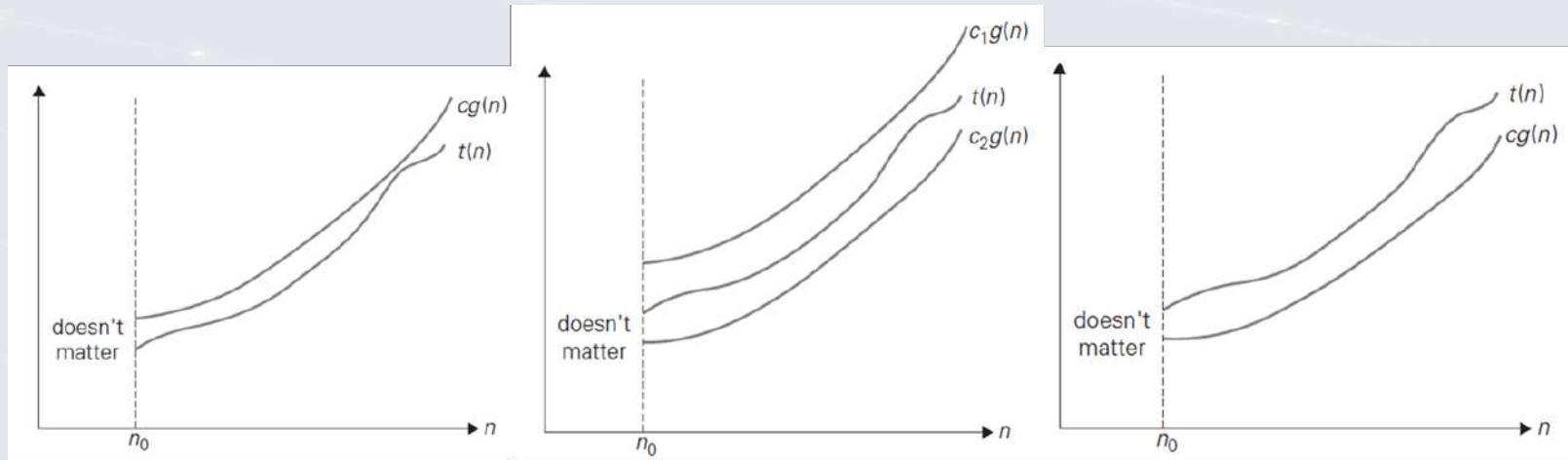
$t(n) \in \Theta(g(n))$.



$t(n) \in \Omega(g(n))$.

e.g. Matrix Multiplication

- Computational complexity analysis has determined a lower bound on the efficiency as $\Omega(n^2)$
- Does not mean it is possible to create an algorithm $\Theta(n^2)$
- It means it is impossible to create an algorithm better than $\Theta(n^2)$
- Best algorithm to date: $\Theta(n^{2.38})$



$$t(n) \in O(g(n))$$

© 2014 by

$$t(n) \in \Theta(g(n)).$$

ight © 2014 by

$$t(n) \in \Omega(g(n)).$$

end Learning Company
www.jblearning.com



e.g Matrix Multiplication

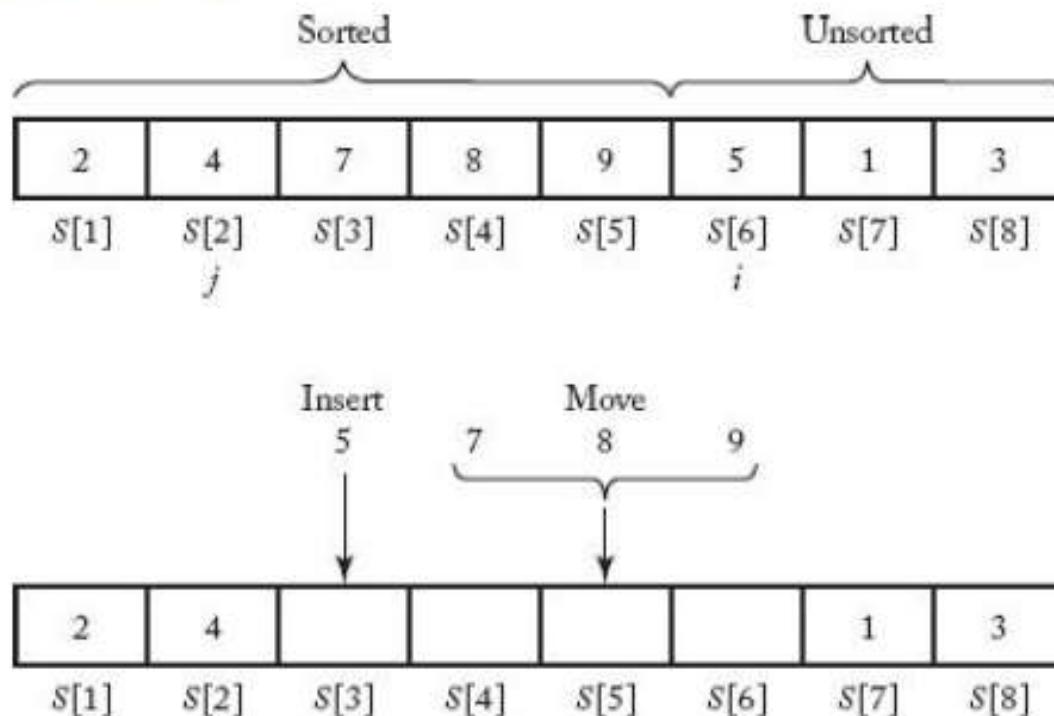
- Proceed? Two directions:
 - Try to find a more efficient algorithm
 - Use computational complexity analysis to find a higher lower bound



Sort

- Re-arrange records according to a key field
- Algorithms that sort by comparison of keys can **compare 2 keys to determine which is larger** and can copy keys
- **Cannot do other operations on them**

Figure 7.1 An example illustrating what Insertion Sort does when $i = 6$ and $j = 2$ (top). The array before inserting, and (bottom) the insertion step.



Insertion Sort – Algorithm 7.1

```
void insertionsort (int n, keytype S[])
{
    index i, j;
    keytype x;
    for (i = 2; i <= n; i++){
        x = S[i];
        j = i - 1;
        while (j > 0 && S[j] > x){
            S[j + 1] = S[j];
            j--;
        }
        S[j + 1] = x;
    }
}
```

- **Insert records** in an existing sorted array – arranging cards being dealt one at a time
- Assume keys in first $i-1$ array slots are sorted
- Let x be the key in the i th slot
- Compare x with $S[i-1]$, $S[i-2]$, . . . Until $S[j] < x$
- Move $S[j+1]$ through $S[i-1]$ to $S[j+2]$ through $S[i]$
- Set $S[j+1]$ to x
- Repeat for $i = 2$ to n

Worst-case Time Complexity Analysis of Number of Comparisons of Keys

- **Basic Operation:** Comparison of $S[j]$ with x
- **Input size:** n , the number of keys to be sorted
- Assume short-circuit evaluation
- Prior to loop execution, j set to $i-1$
- j decremented at each loop iteration
- $j > 0$ clause of the while expression becomes FALSE after $i-1$ iterations of the while loop
- While loop executes from $i = 2$ to n

```
void insertionsort (int n, keytype S[])
{
    index i, j;
    keytype x;
    for (i = 2; i <= n; i++){
        x = S[i];
        j = i - 1;
        while (j > 0 && S[j] > x){
            S[j + 1] = S[j];
            j--;
        }
        S[j + 1] = x;
    }
}
```

Total number of comparisons is at most

comparison of $S[j]$ with x is not done when j is 0. Therefore, this comparison is done at most $i-1$ times for a given i . Because i ranges in value from 2 to n , the total number of comparisons is at most

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

Worst-case behavior of Insertion Sort

- Keys in original array are in non-increasing order
- At position $i+1$, $S[i+1] < S[j]$ for $1 \leq j \leq i$
- Positions $S[1] \dots S[i]$ sorted
- While loop will exit after i iterations because $S[j] > x$ will always be true

$$W(n) = \frac{n(n-1)}{2}.$$



Space usage of Insertion Sort

- **In-place sort**
- When **the extra space is a constant** (that is, when it does not increase with n , the number of keys to be sorted), the algorithm is called an **in-place sort**.

Table 7.1 analysis summary for exchange, insertion, and selection sorts

Algorithm	Comparisons of Keys	Assignments of Records	Extra Space Usage
Exchange Sort	$T(n) = \frac{n^2}{2}$	$W(n) = \frac{3n^2}{2}$ $A(n) = \frac{3n^2}{4}$	In-place
Insertion Sort	$W(n) = \frac{n^2}{2}$	$W(n) = \frac{n^2}{2}$ $A(n) = \frac{n^2}{4}$	In-place
Selection Sort	$T(n) = \frac{n^2}{2}$	$T(n) = 3n$	In-place

*Entries are approximate.

Lower Bounds Sort by Comparison of Keys

- Insertion Sort, Exchange Sort, Selection Sort
- Worst case input of size n contains n distinct keys
- $n!$ different orderings
- **Permutation:** $[k_1, k_2, \dots, k_n]$ k_i is the integer at the i th position
 - E.g. $[3, 2, 1]$ $k_1 = 3, k_2 = 2, k_3 = 1$
- **Inversion** in a permutation (k_i, k_j) pair such that $i < j$ and $k_i > k_j$
- $[3, 2, 4, 1, 6, 5]$
 - Inversions: $(3,2), (3,1), (2,1), (6,5), (4,1)$
- **This means that the task of sorting n distinct keys is the removal of all inversions in permutation**

Theorem 7.1

Any algorithm that sorts n distinct keys only by comparisons of keys and removes at most one inversion after each comparison must in the worst case do at least

$$\frac{n(n - 1)}{2} \text{ comparisons of keys}$$

and, on the average, do at least

$$\frac{n(n - 1)}{4} \text{ comparisons of keys.}$$

Proof Theorem 7.1

- To establish the result for the **worst case**, we need only show that there is a permutation with $n(n - 1) / 2$ inversions because when that permutation is the input, the algorithm will have to remove that many inversions and therefore do at least that many comparisons.
 - At most one inversion is removed with each comparison => $n(n-1)/2$ comparisons
 - Show $[n, n-1, n-2, \dots, 3, 2, 1]$ is such a permutation
 - $n-1$ inversions with n
 - $n-2$ inversions with $n-1$
- To establish the result for the **average case**, we pair the permutation $[kn, kn-1, \dots, k1]$ with the permutation $[k1, k2, \dots, kn]$. This permutation is called the **transpose** of the original permutation.
 - Let r and s be integers between 1 and n such that $s > r$.
 - Given a permutation, the pair (s, r) is an inversion in either the permutation or its transpose but not in both. Showing that there are $n(n - 1) / 2$ such pairs of integers between 1 and n is left as an exercise. This means that a permutation and its transpose have exactly $n(n - 1) / 2$ inversions between them. So the average number of inversions in a permutation and its transpose is

$$\frac{1}{2} \times \frac{n(n - 1)}{2} = \frac{n(n - 1)}{4}.$$
 - Therefore, if we consider all permutations equally probable for the input, the average number of inversions in the input is also $n(n - 1) / 4$. Because we assumed that the algorithm removes at most one inversion after each comparison, on average it must do at least this many comparisons to remove all inversions and thereby sort the input.

Algorithms addressed by Theorem 7.1

- Insertion Sort
- Selection Sort
- Exchange Sort

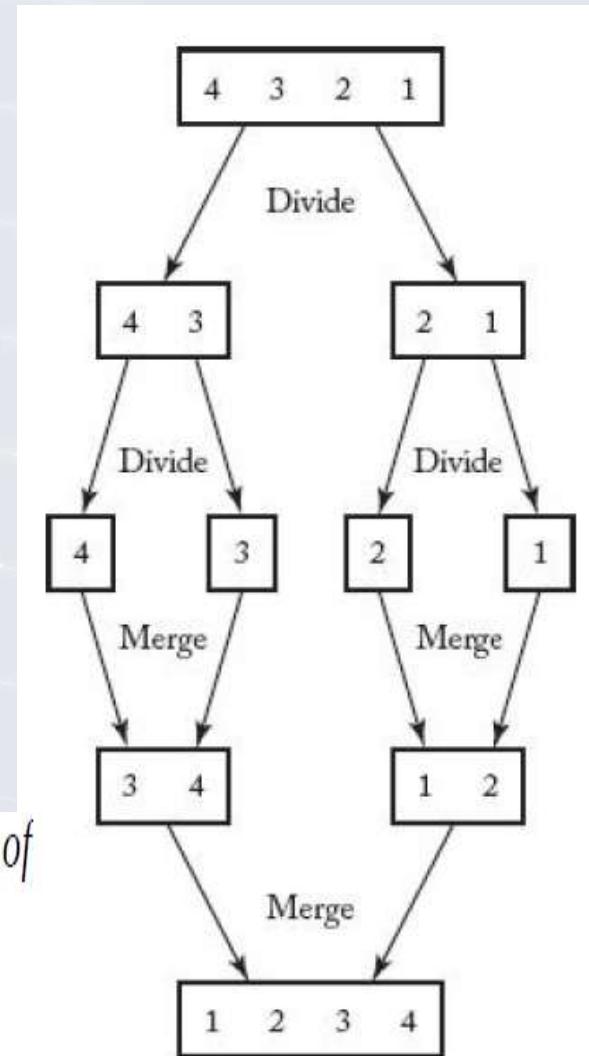
Mergesort

- Input in reverse order: $S = [4,3,2,1]$
- 3 and 1 are compared and 1 placed in output array
 - Inversions (3,1) and (4,1) removed
- 3 and 2 are compared and 2 placed in output array
 - Inversions (3,2) and (4,2) removed
- $W(n) = n \lg n - (n - 1) \in \Theta(n \lg n)$

Recall that the worst-case time complexity of Mergesort's number of comparisons of keys is given by

$$W(n) = n \lg n - (n - 1)$$

when n is a power of 2, and in general, it is in $\Theta(n \lg n)$.



Extra space usage

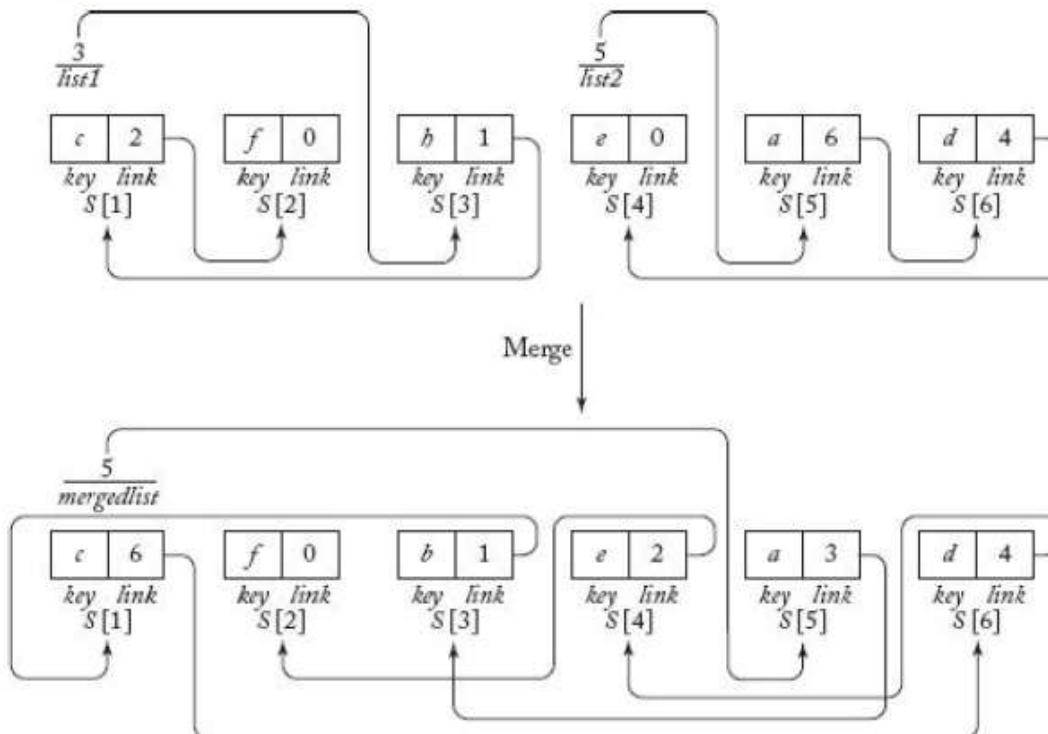
- Stack grows to a **depth of $\lceil \lg n \rceil$**
- Space for additional array of records dominates
- Every-case extra space usage is **$\Theta(n)$**

the algorithm is sorting the first subarray, the values of mid , $mid+1$, low , and $high$ need to be stored in the stack of activation records. Because the array is always split in the middle, this stack grows to a depth of $\lceil \lg n \rceil$. The space for the additional array of records dominates, which means that in every case the extra space usage is in $\Theta(n)$ records. By “in $\Theta(n)$ records” we mean that the number of records is in $\Theta(n)$.

Improvements to Mergesort

- Dynamic Programming version $T(n) \sim n \lg n$
- Linked List version

Figure 7.3 Merging using links. Arrows are used to show how the links work. The keys are letters to avoid confusion with indices.



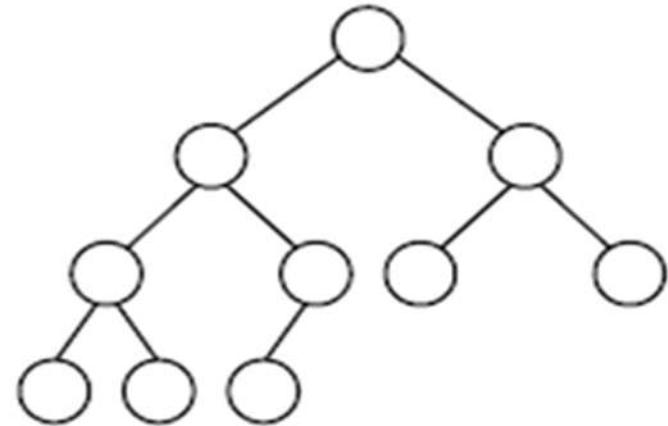
```
struct node
{
    keytype key;
    index link;
};
```



Definitions

- **Tree**: connected, acyclic graph
- **Depth of a node**: number of edges in the unique path from the root to that node
- **Depth d of a tree** is the maximum depth of all nodes in the tree
- **Leaf** is any node with no children
- **Internal node** is any node that has at least one child

Binary Tree



- **Complete Binary Tree**

- All internal nodes have two children
- All leaves have depth d

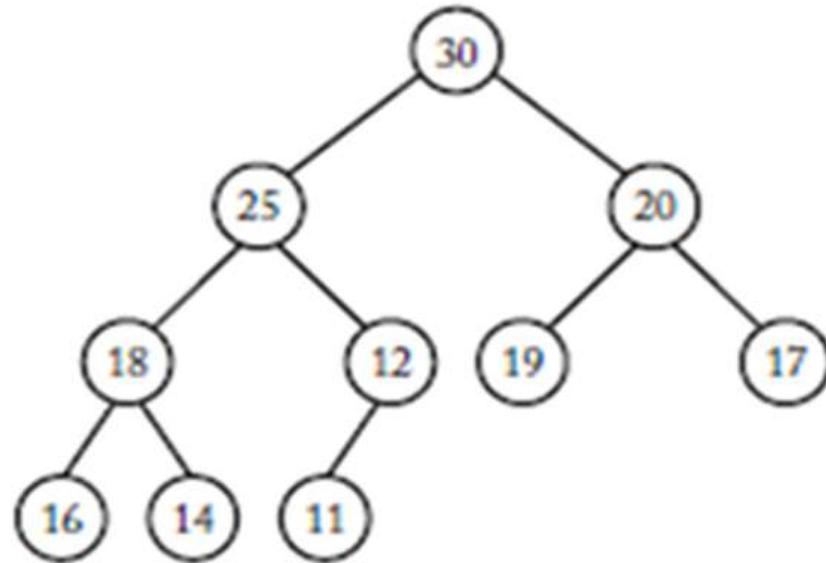
- **Essentially Complete Binary Tree**

- A complete binary tree down to depth of $d-1$
- Nodes with depth d are as far to the left as possible



Heap: Essentially complete binary tree such that:

- Values stored at the nodes come from an ordered set
- **Heap Property:** value stored at each node is \geq the values stored at its children





Heapsort

- In-place sort
- $\Theta(n \lg n)$
- Main idea:
 - Arrange keys to be sorted in a heap
 - Repeatedly remove the key stored at the root while maintaining the heap property
 - Removes keys in non-decreasing order
 - As keys removed, placed in array starting in nth entry down to the first position (reverse order)
 - Array will be sorted in non-decreasing order



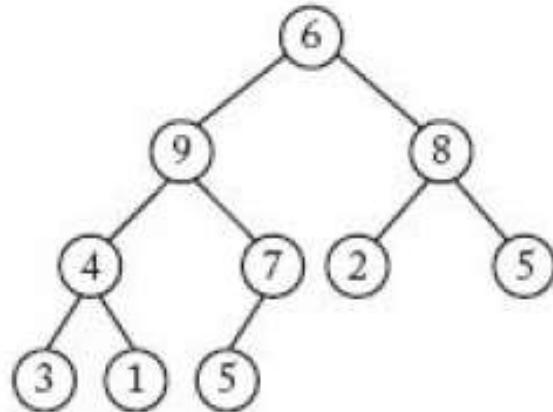
Restoring the heap property:

- Remove key at root
- Replace key at root with key stored at the bottom node (far right leaf) and deleting bottom node (decrement heap size)
- Sift new root down the heap until heap property restored
 - Compare key at root with larger of the keys of the root's children
 - If key at root is smaller, exchange keys

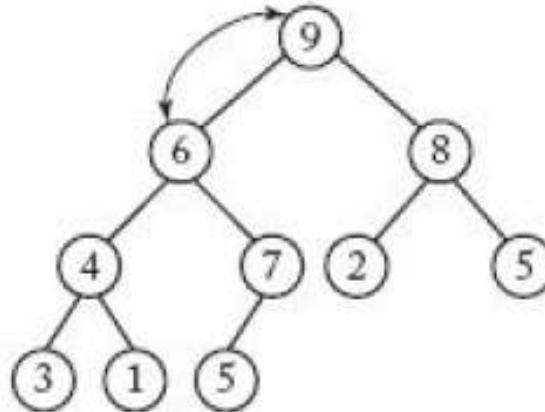
Restoring the heap property

- Repeat process down the tree until the key at node is not smaller than the larger of the children

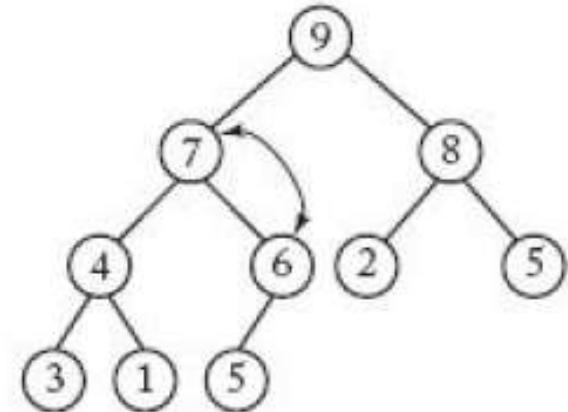
(a) Not a heap



(b) Keys 6 and 9 swapped



(c) Keys 6 and 7 swapped



Array implementation of heap

- Root stored at $S[1]$
- Let i be the index of a given node m
 - $2i =$ index of m 's left child
 - $2i+1 =$ index of m 's right child

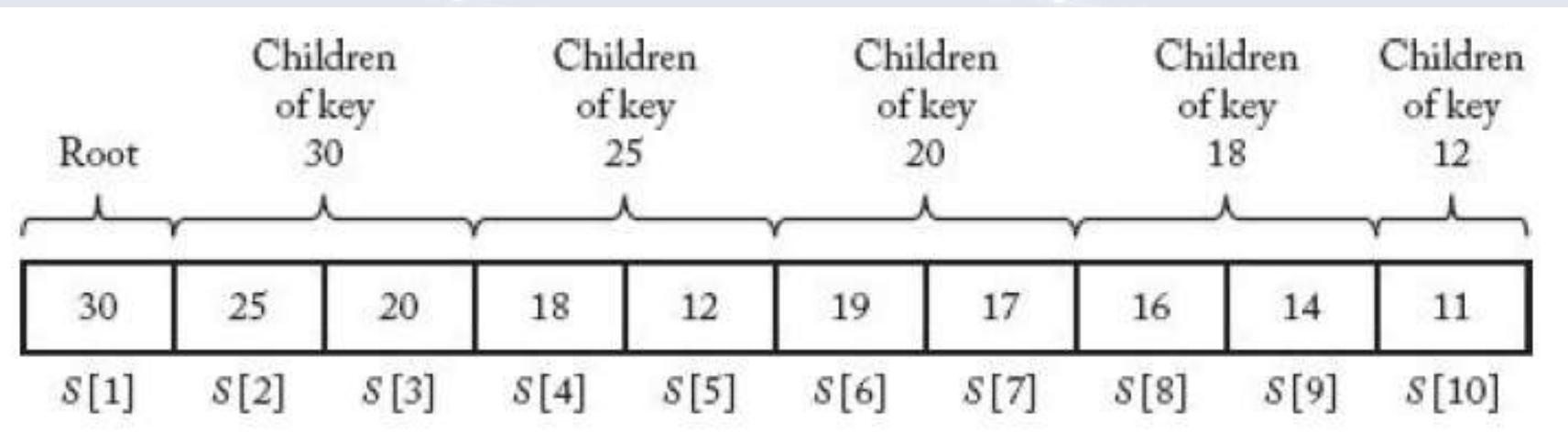
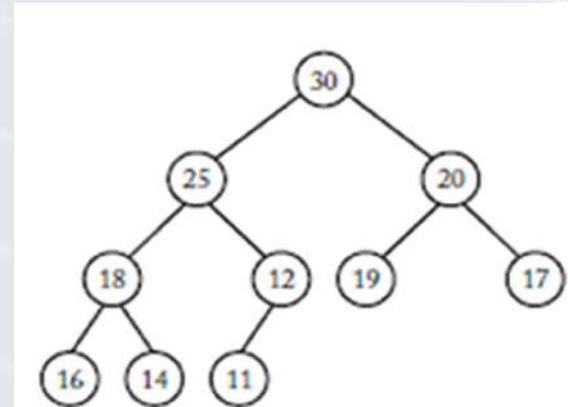
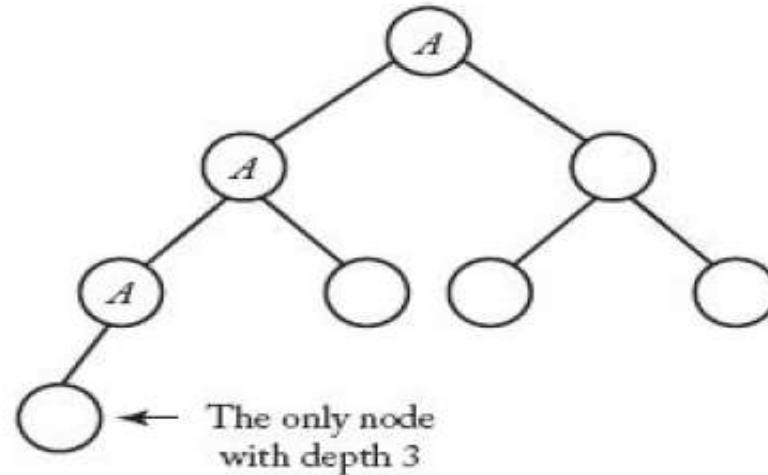


Figure 7.9 An illustration using $n = 8$, showing that if an essentially complete binary tree has n nodes and n is a power of 2, then the depth d of the tree is $\lg n$, there is one node with depth d , and that node has d ancestors. The three ancestors of that node are marked “A.”



Depth	Number of Nodes with this Depth	Greatest Number of Nodes that a Key Would Be Sifted
0	2^0	$d - 1$
1	2^1	$d - 2$
2	2^2	$d - 3$
:	:	:
j	2^j	$d - j - 1$
:	:	:
$d - 1$	2^{d-1}	0

Heapsort – Algorithm 7.5 Worst-Case Time Complexity Analysis of Number of Comparisons of Keys

- **Basic instruction:** Comparison of Keys in procedure *siftdown()*
- **Input size:** n
- **Makeheap():**
 - Upper bound on total # nodes all keys will be sifted through $n - 1$
 - For each pass of while loop in *siftdown()*, 2 comparisons of keys
 - Number of comparisons of keys done by *makeheap()* is at most $2(n - 1)$
- Analysis of remove keys $2n \lg n - 4n + 4$
- Combine analysis of *makeheap()* and *removekeys()*

$$2(n - 1) + 2n \lg n - 4n + 4 = 2(n \lg n - n + 1) \approx 2n \lg n$$

when n is a power of 2. Showing that there is a case in which we have this number of comparisons is left as an exercise. Therefore, for n a power of 2,

$$W(n) \approx 2n \lg n \in \Theta(n \lg n).$$

Extra space for heapsort

- Heap sort is **an in-place algorithm** - $\Theta(1)$
- **Its typical implementation is not stable**, but can be made stable
- Typically 2-3 times slower than well-implemented [QuickSort](#). The reason for slowness is a lack of locality of reference.

Advantages of heapsort:

- **Efficiency** – The time required to perform Heap sort increases logarithmically while other algorithms may grow exponentially slower as the number of items to sort increases. This sorting algorithm is very efficient.
- **Memory Usage** – Memory usage is minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work
- **Simplicity** – It is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion



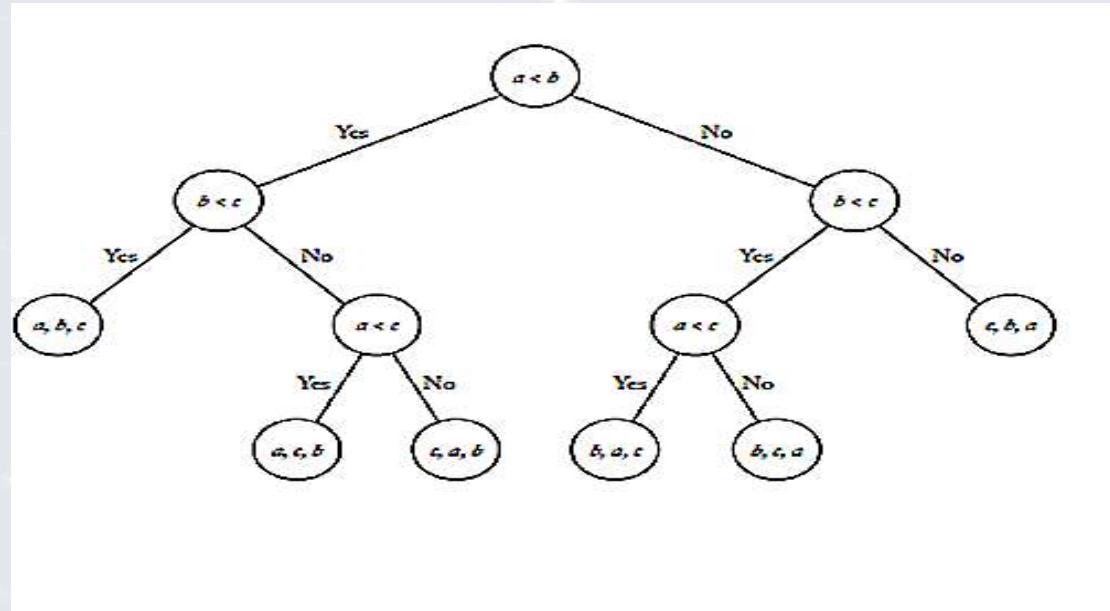
Lower Bounds for Sorting only by comparison of keys

- **Mergesort and Heapsort:** $\theta(n \lg n)$
- Substantially better than $\theta(n^2)$
- Can it be improved?
- **Show** that sorting by comparison a faster algorithm **cannot** be developed

Consider the following algorithm for sorting three keys.

```
void sortthree (keytype S[])
{
    keytype a, b, c;
    a = S[1]; b = S[2]; c = S[3];
    if (a < b)
        if (b < c)
            S = a, b, c;           // This means S[1] = a; S[2] = b; S[3] = c;
    else if (a < c)
        S = a, c, b;
    else
        S = c, a, b;
    else if (b < c)
        if (a < c)
            S = b, a, c;
        else
            S = b, c, a;
    else
        S = c, b, a;
}
```

Decision Tree



This tree is called a ***decision tree*** because at each node a decision must be made as to which node to visit next. Decision tree is **pruned** if every leaf can be reached from the root by making a consistent sequence of decisions



Lemma 7.1

- To every deterministic algorithm for sorting n distinct keys there corresponds a pruned, valid, binary decision tree containing **exactly $n!$ leaves**

Proof Outline:

- All keys distinct: result of a comparison is < or >
- Each node has at most two children – binary tree
- $n!$ leaves
 - $n!$ different inputs that contain n distinct keys
 - Decision tree is valid only if it has a leaf for every input
 - Decision tree has $n!$ leaves
- Unique path in the tree for each of the $n!$ different inputs
- Every leaf in a pruned decision tree must be reachable
- Tree can have no more than $n!$ leaves. Therefore, the tree has exactly $n!$ leaves

Lemma 7.2

The worst-case number of comparisons done by a decision tree is equal to its depth.

- Proof Outline
 - Number of comparisons done by a decision tree is the number of internal nodes on the path followed for the input
 - Number of internal nodes is the same as the length of the path
 - Worst case number of comparisons done by the decision tree is the length of the longest path to a leaf (depth of the decision tree)

Lemma 7.3

If m is the number of leaves in a binary tree and d is the depth: $d \geq \lceil \lg m \rceil$

- **Proof by Induction**

- Induction Base: complete binary tree depth 0: $2^0 = 1$
- Induction Hypothesis: Assume for the complete binary tree with depth d : $2^d = m$
- Induction step: show that for the complete binary tree with depth $d+1$, $2^{d+1} = m'$ where m' is the number of leaves



Theorem 7.2

- Any deterministic algorithm that sorts n distinct keys only by comparisons of keys must in the worst case do at least $\lceil \lg(n!) \rceil$ comparison of keys
- Proof:
 - Lemma 7.1
 - Lemma 7.3
 - Lemma 7.2



Lemma 7.4

- For any positive integer n ,

$$\lg(n!) \geq n \lg n - 1.45n$$

Proof: The proof requires knowledge of integral calculus. We have

$$\begin{aligned}\lg(n!) &= \lg[n(n-1)(n-2)\cdots(2)1] \\&= \sum_{i=2}^n \lg i && \{\text{because } \lg 1 = 0\} \\&\geq \int_1^n \lg x \, dx = \frac{1}{\ln 2} (n \ln n - n + 1) \geq n \lg n - 1.45n.\end{aligned}$$



Theorem 7.3

- Any deterministic algorithm that sorts n distinct keys only by comparison of keys must in the worst case do at least

$\lceil n \lg n - 1.45n \rceil$ comparisons of keys

- Proof follows from Theorem 7.2 and Lemma 7.4



Sorting by Distribution - Radix sort

- Keys non-negative integers
- Keys represented in some base
- All keys have the same number of digits
- Radix Sort – based on old card sorting machines
- Radix – any number of alphabet base



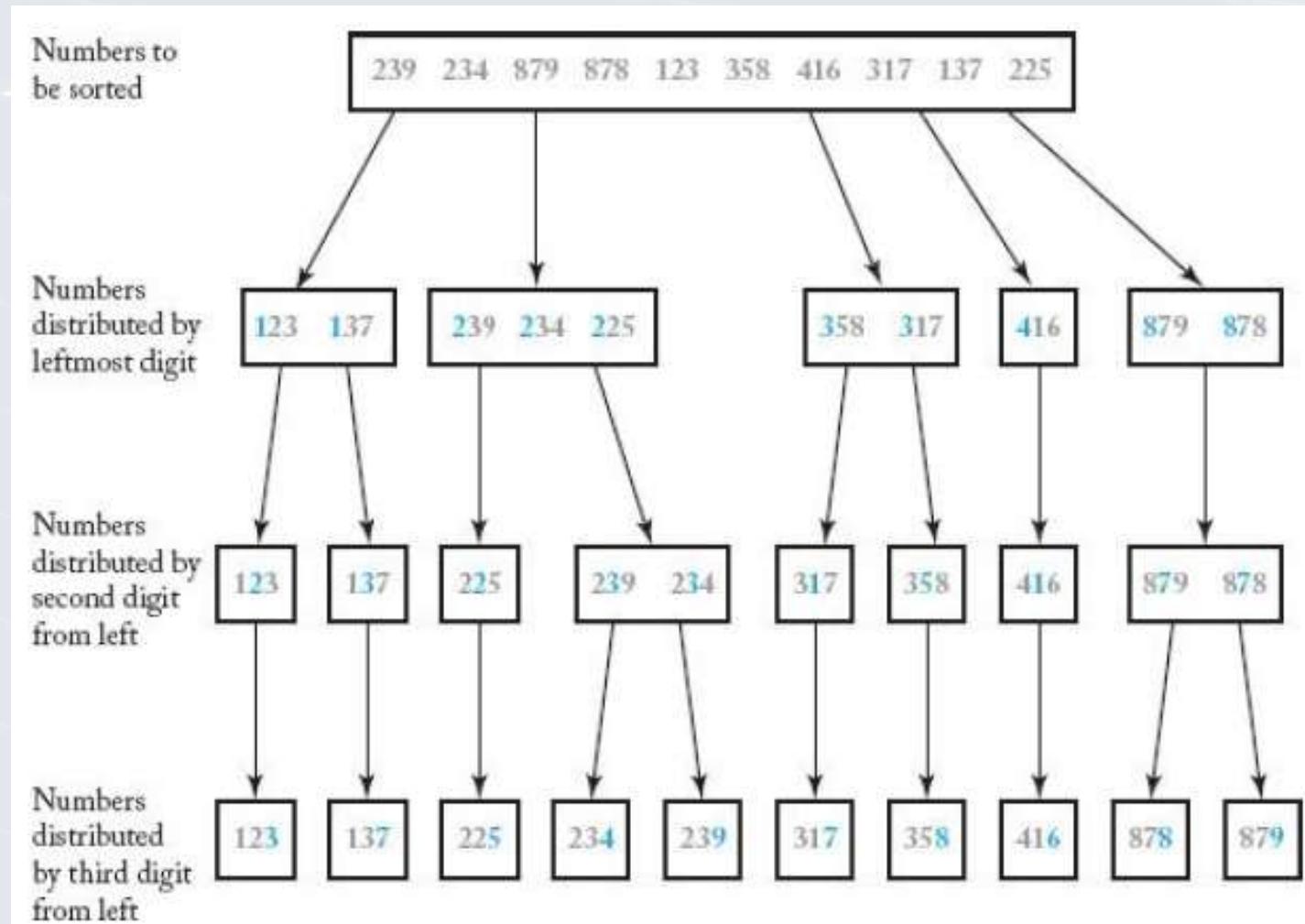
Distribute the keys into piles

- Number of piles equals the number base (radix)
- Inspect keys from right to left (lsb -> msb)
- Place a key into a pile corresponding to the digit currently being inspected
- Each pass: if 2 keys are to be placed in the same pile, the key coming from the left-most pile (previous pass) is placed to the left of the other key

Distribute the keys into piles

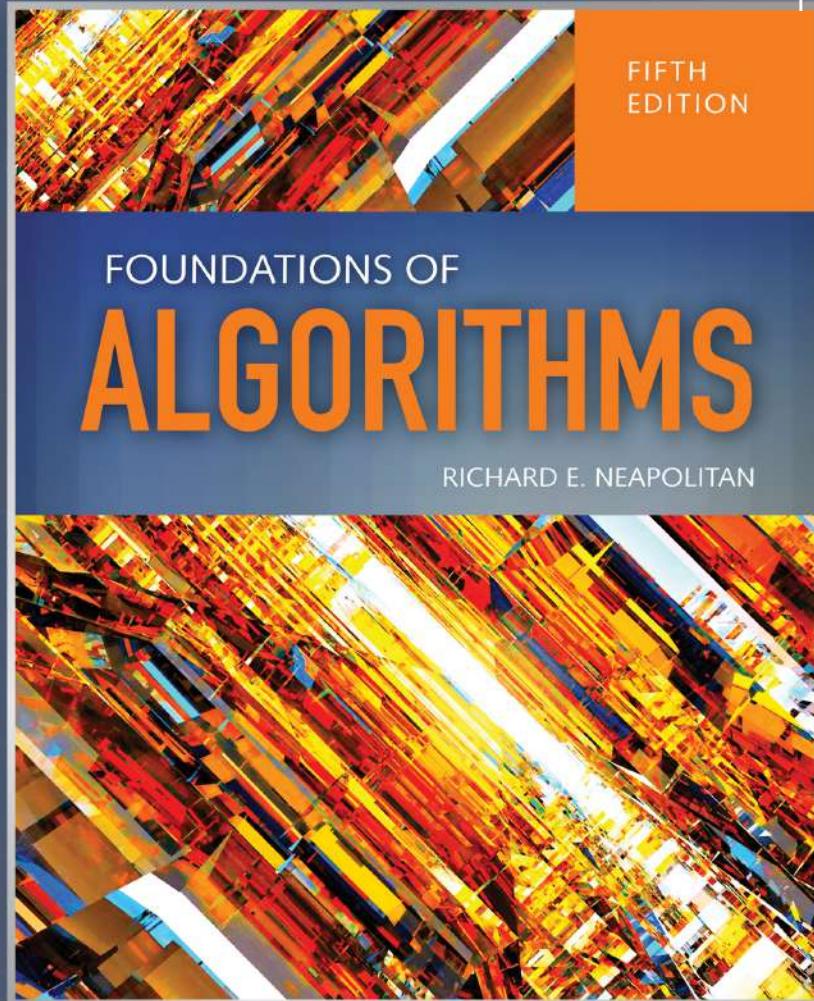
- Implementation:
 - Piles represented by a linked list
 - After each pass, keys removed from each list pile and merged into single linked list
 - Next pass, single linked list traversed and keys placed in appropriate piles based on the digit being sorted

Sorting by distribution while inspecting the digits from left to right



More Computational Complexity: The Searching Problem

Chapter 8





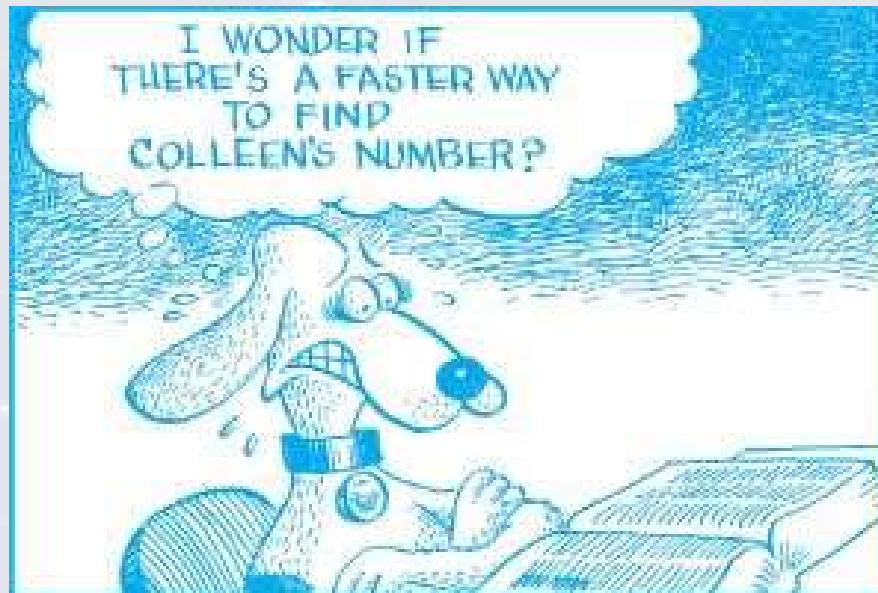
Objectives

- Establish a lower bound on searching by comparison of keys of n distinct keys for key x .
- Prove binary search is optimal
- Apply interpolation search to evenly distributed data
- Differentiate between static and dynamic searching
- Establish the average binary search tree search time



Objectives

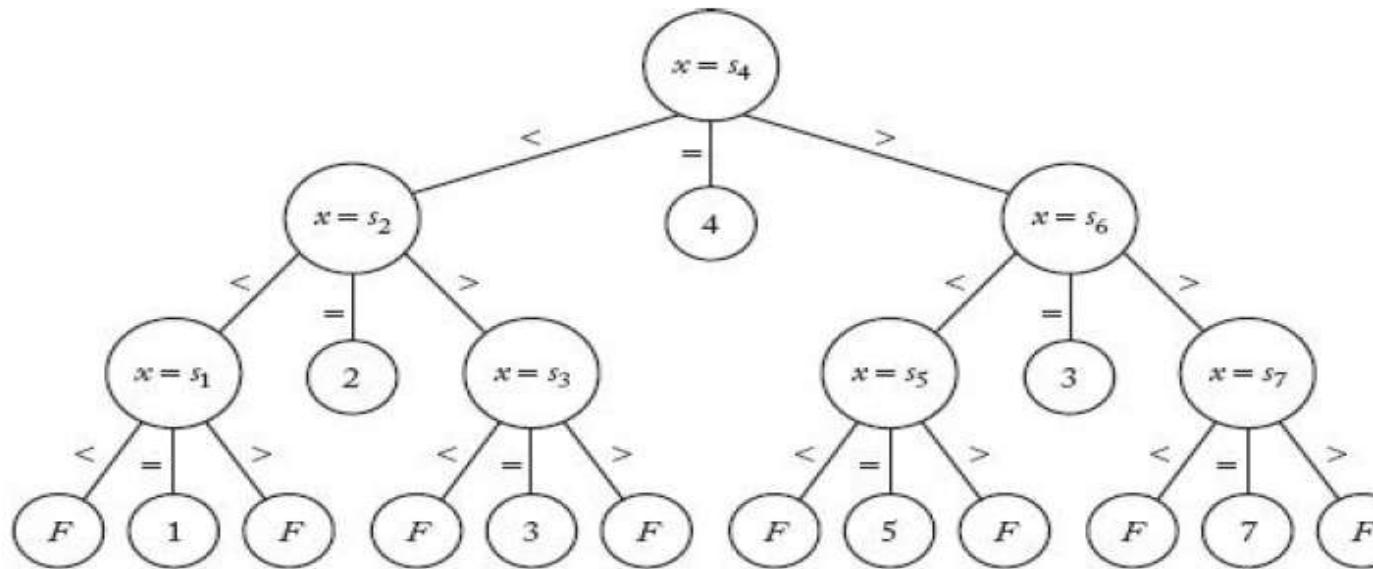
- Determine lower bounds on finding largest and smallest keys
- Define a linear-time algorithm to find the k^{th} smallest key in an array of n distinct keys



Binary Search

- Algorithm 2.1
- $W(n) = \lfloor \lg n \rfloor + 1$
- Establish binary search algorithm is optimal
- Establish a lower bounds for searching only by comparison of keys

Figure 8.1 The decision tree corresponding to Binary Search when searching seven keys.





Decision Tree Binary Search

- **Every algorithm that searches for key x in an array of n keys has a corresponding pruned, valid decision tree**
- Leaf represents a point where algorithm stops and reports an **index of x** or **failure**
- **Every internal node represents a comparison**
- **Valid** if for each possible outcome, there is a path from the root to a leaf reporting that outcome
- **Pruned** if every leaf is reachable



Pruned, valid Decision Tree

- 3 different results: $>$, $=$, $<$
- Decision node can have at most three children
- **Equality** is a leaf returning index
- At most 2 children can be comparison nodes
- Therefore, # comparisons is a **binary tree**



Establish a lower bound on the number of comparisons

- Every node reachable: **worst case comparison nodes on longest path**
- Number of nodes on the longest path from the root to a leaf in the binary tree consisting of comparison of nodes
- **Number of comparisons** is the **depth+1**



Lemma 8.1

- Establish a lower bound on the depth of the binary tree consisting of comparison nodes

**If n is the number of nodes in a binary tree
and d is the depth:**

$$d \geq \lfloor \lg n \rfloor$$



Proof Lemma 8.1

- At most one root, At most 2 nodes with depth 1, At most 2^2 nodes with depth 2, . . . , At most 2^d nodes with depth d

- $n \leq 1 + 2 + 2^2 + \dots + 2^d$

- A3:

$$\Rightarrow n \leq 2^{d+1} - 1$$

$$\Rightarrow n < 2^{d+1}$$

$$\sum_{i=0}^n 2^i = 2^{n+1} - 1$$

- Take lg of both sides

$$\lg n < \lg 2^{d+1}$$

$$\Rightarrow \lg n < (d+1)\lg 2 = d+1$$

$$\Rightarrow \lfloor \lg n \rfloor \leq d$$



Lemma 8.2

- To be a **pruned, valid decision tree** for searching n distinct keys for a key x , **the binary tree consisting of the comparison nodes must contain at least n nodes**

Steps of Proof by Contradiction

Let s_i for $i \in \langle 1..n \rangle$ be the values of the n keys.

1. Show that every s_i must be in at least one comparison node
2. Show that since every s_i must be in a comparison node there must be n comparison nodes



Theorem 8.1

Any deterministic algorithm that **searches for a key x** in an array of **n distinct keys** only by **comparisons of keys** must in **the worst case** do at least

$\lfloor \lg n \rfloor + 1$ **comparisons of keys**

Proof Theorem 8.1

- Given a **pruned, valid decision tree** for the algorithm that searches n distinct keys for a key x
- Worst-case number of comparisons is the number of nodes on the longest path from the root to a leaf in the binary tree of comparison nodes: $d + 1$
- By Lemma 8.2, **the binary tree has at least n nodes**
- By Lemma 8.1, $d \geq \lfloor \lg n \rfloor$

Interpolation Search

- an **improvement** over Binary Search for instances, where the values in a sorted array are uniformly distributed.
- interpolation constructs new data points within the range of a discrete set of known data points.
- Binary Search always goes to the middle element
- **interpolation search may go to different locations according to the value of the key being searched.**
- For example, if the value of the key is closer to the last element, the interpolation search is likely to start search toward the end side.

Interpolation Search

- Given an application, data is evenly distributed in the search array
- Instead of starting the search at mid, make the decision based on the value of x
- Use linear interpolation to determine where x should be located

$$mid = low + \left\lfloor \frac{x - S[low]}{S[high] - S[low]} \times (high - low) \right\rfloor.$$

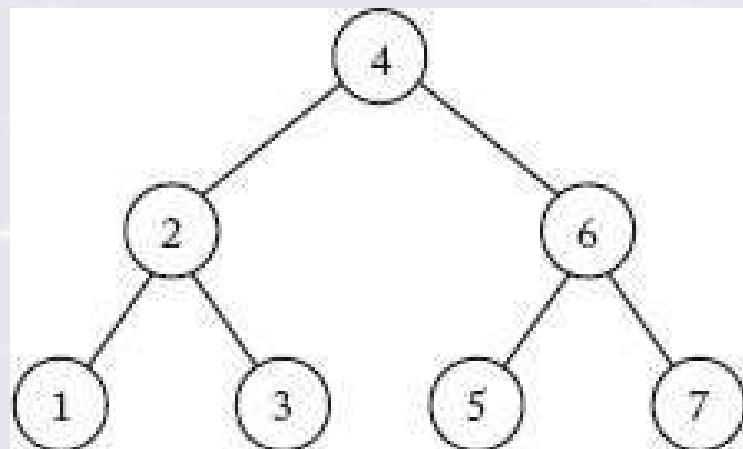
For example, if $S[1] = 4$ and $S[10] = 97$, and we were searching for $x = 25$,

$$mid = 1 + \left\lfloor \frac{25 - 4}{97 - 4} \times (10 - 1) \right\rfloor = 1 + \lfloor 2.032 \rfloor = 3.$$

Volatility of data

- **Static Searching:** records are added to the file one at a time. Once file established, records never added or deleted.
 - Array appropriate storage/search structure
- **Dynamic Searching:** records frequently added/deleted (e.g. airline reservation system)
 - Binary Search Tree

Binary Search Tree



- Tree of items from an ordered set such that
 - **Each node contains one key**
 - **Keys in the left subtree** of a given node are less than or equal to the key in that node
 - **Keys in the right subtree** of a given node are greater than or equal to the key in that node

In-order traversal

- Visit all nodes in the left subtree using in-order traversal
- Visit root
- Visit all nodes in the right subtree using in-order traversal
- **Produces sorted order of keys**

InOrder(root) visits nodes in the following order:

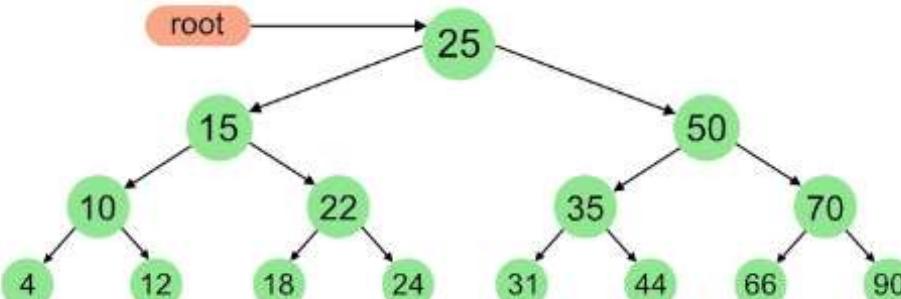
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Theorem 8.3

Assume all inputs are equally probable and that the search key x is equally probable to be any of the n keys, the average search time over all inputs containing n distinct keys using binary search trees is given approximately by $A(n) \approx 1.38 \lg n$

- Theorem 8.3 **does not mean** the average for a given input is $1.38 \lg n$
- Theorem 8.3 **is the average search time over all inputs containing n keys input**

Selection Problem

20

Problem: Find the largest key in the array S of size n .

Inputs: positive integer n , array of keys S indexed from 1 to n .

Outputs: variable $large$, whose value is the largest key in S .

```
void find_largest (int n,
                   const keytype S[] ,
                   keytype& large)
{
    index i;

    large = S[1];
    for (i = 2; i <= n; i++)
        if (S[i] > large)
            large = S[i];
}
```

Clearly, the number of comparisons of keys done by the algorithm is given by

$$T(n) = n - 1.$$

Theorem 8.7: Any deterministic algorithm that can find **the largest of n keys** in every possible input only by comparisons of keys must in every case do at least

$n-1$ comparisons of keys

© Comstock Images/age fotostock. Copyright © 2014 by Jones & Bartlett Learning, LLC an Ascend Learning Company

www.jblearning.com

- Proof by contradiction

Find the smallest and largest key

Inputs: positive integer n , array of keys S indexed from 1 to n .

Outputs: variables $small$ and $large$, whose values are the smallest and largest keys in S .

```
void find_both (int n,
                const keytype S[],
                keytype& small,
                keytype& large)

{
    index i;

    small = S[1];
    large = S[1];
    for (i = 2; i <= n; i++)
        if (S[i] < small)
            small = S[i];
        else if (S[i] > large)
            large = S[i];
}
```



Find the smallest and largest key

- Worst case is when $S[1]$ is the smallest key, second comparison is done for all i
- $W(n) = 2(n-1)$
- Improve?



Find the smallest and largest key

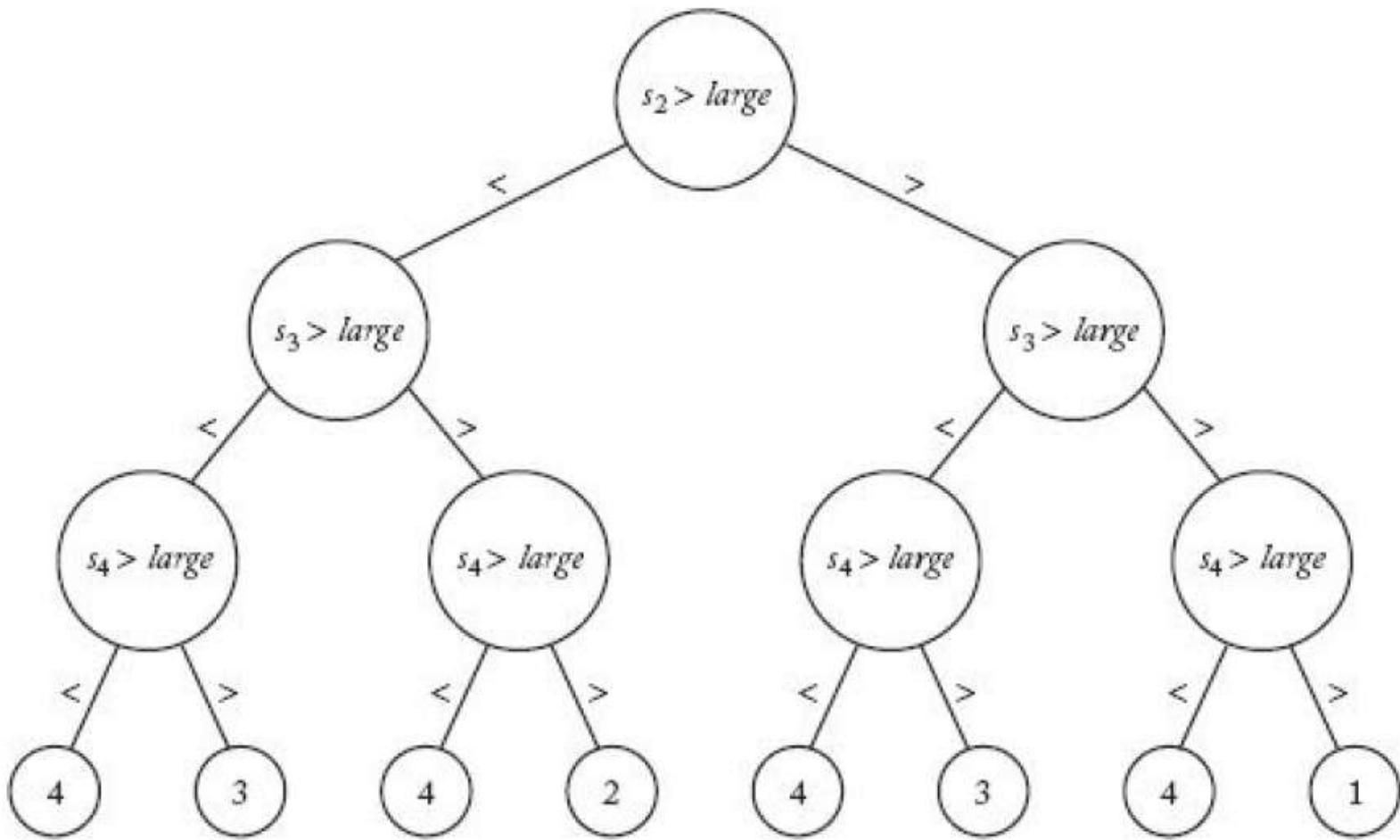
- Worst case is when $S[1]$ is the smallest key, second comparison is done for all i
- $W(n) = 2(n-1)$
- Improve?

but we can. The trick is to pair the keys and find which key in each pair is smaller. This can be done with about $n/2$ comparisons. We can then find the smallest of all the smaller keys with about $n/2$ comparisons and the largest of all the larger keys with about $n/2$ comparisons. In this way, we find both the smallest and largest keys with only about $3n/2$ total comparisons. An algorithm for this method follows. The algorithm assumes that n is even.

Find smallest and largest keys by pairing keys

- Assume n is even
- Pair i_1 and i_2 , i_3 and i_4 , . . . , i_j and i_{j+1} , . . . , i_{n-1} and i_n
- Find the largest of the pair: i_l and compare it with the largest so far
- Compare the other key of the pair with the smallest so far
- Iterate loop by 2 instead of 1
- n is even: $T(n) = 3n/2 - 2$
- n is odd: $T(n) = 3n/2 - 3/2$

Figure 8.9 The decision tree corresponding to Algorithm 8.2 when $n = 4$.





Theorem 8.8

Any deterministic algorithm that can find both **the smallest and the largest of keys** in every possible input only by **comparisons of keys** must in the worst case do at least:

$$3n/2 - 2 \text{ if } n \text{ is even}$$

$$3n/2 - 3/2 \text{ if } n \text{ is odd}$$



Algorithm 8.5

- **Find the k^{th} smallest key in an array S of n distinct keys**
- Could sort the array in $n\lg(n)$ time and take the k^{th} smallest
- Use Algorithm 2.7, partition from quicksort
- **Recursively partition** the left sub-array if k is less than pivotpoint
- **Recursively partition** the right sub-array if k is greater than pivotpoint



Algorithm 8.5

- $W(n) = n(n-1)/2$
- $A(n) \in \Theta(n)$
- Can quadratic time worst case be prevented?



Pivotpoint

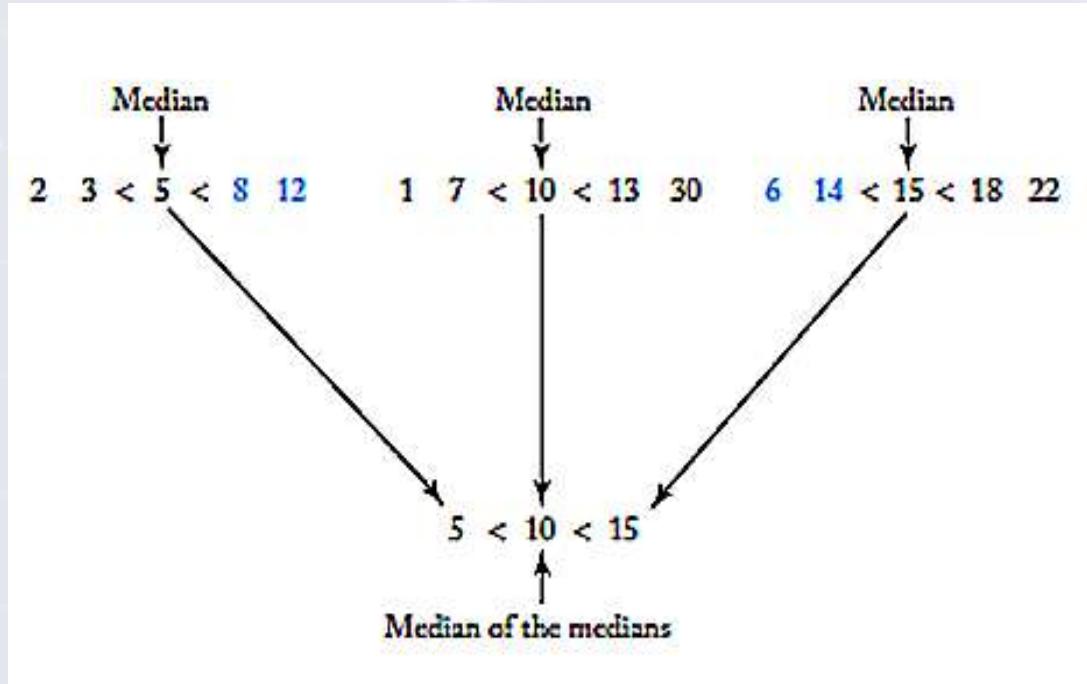
- Split array in the middle
- Input cut in half each recursive call
- Median of n distinct keys (precise if n is odd)
- Half of the keys smaller
- Half of the keys larger

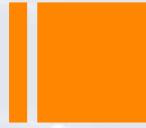


Assume n is an odd multiple of 5

- Divide n keys into $n/5$ groups of keys each containing 5 keys
- Find the median of each of the groups directly
- The median of 5 items can be found with 6 comparisons (see exercises)
- Call selection to find the median of the $n/5$ medians
- Median of $n/5$ medians not necessarily the median of the n elements but will be close
- pivotitem is the median of the medians
- Partition around pivotitem – optimal pivotpoint

Figure 8.12





Keys n is an odd multiple of 5

- Number of keys known to be on one side of the median of the medians: $\frac{1}{2}[n - 1 - 2(n/5 - 1)]$
- Number of keys that could be on either side of the median of the medians: $2(n/5 - 1)$
- At most $\frac{1}{2}[n - 1 - 2(n/5 - 1)] + 2(n/5 - 1) = 7n/10 - 3/2$ keys on one side of the median of the medians

Worst-Case Time Complexity (Selection Using the Median)

- Basic operation: comparison of $S[i]$ with pivotitem in partition2
- Input size: n
- Recurrence assumes n is an odd multiple of 5 (holds for n in general)
- Time in function selection2 when called from selection 2
 - At most $7n/10 - 3/2$ keys on one side of the pivotpoint (worst case number of keys in call)

Worst-Case Time Complexity (Selection Using the Median)

- Time in selection2 when called from partition2
 - Number of keys is $n/5$ ($n/5$ medians)

Worst-Case Analysis Continued

- Number of comparisons required to find medians:
 - 6 comparisons (see exercises)
 - $n/5$ groups of 5 => total number of comparisons
 $6n/5$
- Number of comparisons required to partition the array => n
- $W(n) = W(7n/10 - 3/2) + W(n/5) + 6n/5 + n$
- $W(n) \approx W(7n/10) + W(n/5) + 11n/5$
- Recurrence does not indicate any obvious solution

Constructive Induction used to obtain Candidate Solution

- Suspect $W(n)$ is linear
- Assume $W(m) \leq cm$ for all $m < n$ and for some constant c
- Recurrence implies
 - $W(n) \approx W(7n/10) + W(n/5) + 11n/5$
 - $\leq c(7n/5) + c(n/5) + 11n/5$
- To conclude $W(n) \leq cn$ solve
 - $\leq c(7n/5) + c(n/5) + 11n/5$
 - $22 \leq c$
- $W(n) \in \Theta(n)$



Space and Time Tradeoffs

A. Levitin "Introduction to the Design & Analysis of Algorithms," 3rd ed., Ch. 10 ©2012 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.



Space-for-time tradeoffs



Two varieties of space-for-time algorithms:

- ➊ *input enhancement* — preprocess the input (or its part) to store some info to be used later in solving the problem
 - counting sorts
 - string searching algorithms
- ➋ *prestructuring* — preprocess the input to make accessing its elements easier
 - hashing
 - indexing schemes (e.g., B-trees)

Review: String searching by brute force



pattern: a string of m characters to search for

text: a (long) string of n characters to search in

Brute force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until either all characters are found to match (successful search) or a mismatch is detected

Step 3 While a mismatch is detected and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

String searching by preprocessing



Several string searching algorithms are based on the input enhancement idea of preprocessing the pattern

- Knuth-Morris-Pratt (KMP) algorithm preprocesses pattern left to right to get useful information for later searching
- Boyer -Moore algorithm preprocesses pattern right to left and store information into two tables
- Horspool's algorithm simplifies the Boyer-Moore algorithm by using just one table

Horspool's Algorithm



A simplified version of Boyer-Moore algorithm:

- preprocesses pattern to generate a shift table that determines how much to shift the pattern when a mismatch occurs
- always makes a shift based on the text's character c aligned with the last character in the pattern according to the shift table's entry for c

How far to shift?



Look at first (rightmost) character in text that was compared:

- The character is not in the pattern

..... **c** (c not in pattern)

BAOBAB

- The character is in the pattern (but not the rightmost)

..... **O** (O occurs once in pattern)

BAOBAB

..... **A** (A occurs twice in pattern)

BAOBAB

- The rightmost characters do match

..... **B**

BAOBAB

Shift table



- Shift sizes can be precomputed by the formula
distance from c 's rightmost occurrence in pattern
among its first $m-1$ characters to its right end

$t(c) =$

pattern's length m , otherwise

by scanning pattern before search begins and stored in a
table called *shift table*

- Shift table is indexed by text and pattern alphabet
Eg, for BAOBAB :

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6

Example of Horspool's alg. application

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	—
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

BARD LOVED BANANAS

BAOBAB

BAOBAB

BAOBAB

BAOBAB (unsuccessful search)

Boyer-Moore algorithm



Based on same two ideas:

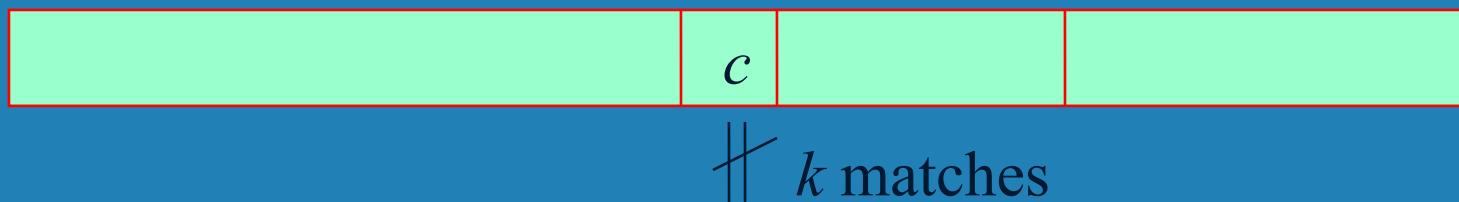
- comparing pattern characters to text from right to left
- precomputing shift sizes in two tables
 - *bad-symbol table* indicates how much to shift based on text's character causing a mismatch
 - *good-suffix table* indicates how much to shift based on matched part (suffix) of the pattern

Bad-symbol shift in Boyer-Moore algorithm



- If the rightmost character of the pattern doesn't match, BM algorithm acts as Horspool's
- If the rightmost character of the pattern does match, BM compares preceding characters right to left until either all pattern's characters match or a mismatch on text's character c is encountered after $k > 0$ matches

text



pattern

bad-symbol shift $d_1 = \max\{t_1(c) - k, 1\}$

Good-suffix shift in Boyer-Moore algorithm



- Good-suffix shift d_2 is applied after $0 < k < m$ last characters were matched
- $d_2(k) =$ the distance between matched suffix of size k and its rightmost occurrence in the pattern that is not preceded by the same character as the suffix

Example: CABABA $d_2(1) = 4$

- If there is no such occurrence, match the longest part of the k -character suffix with corresponding prefix;
if there are no such suffix-prefix matches, $d_2(k) = m$

Example: WOWWOW $d_2(2) = 5, d_2(3) = 3, d_2(4) = 3, d_2(5) = 3$

Good-suffix shift in the Boyer-Moore alg. (cont.)



After matching successfully $0 < k < m$ characters, the algorithm shifts the pattern right by

$$d = \max \{d_1, d_2\}$$

where $d_1 = \max\{t_1(c) - k, 1\}$ is bad-symbol shift

$d_2(k)$ is good-suffix shift

Boyer-Moore Algorithm (cont.)



Step 1 Fill in the bad-symbol shift table

Step 2 Fill in the good-suffix shift table

Step 3 Align the pattern against the beginning of the text

**Step 4 Repeat until a matching substring is found or text ends:
Compare the corresponding characters right to left.**

If no characters match, retrieve entry $t_1(c)$ from the bad-symbol table for the text's character c causing the mismatch and shift the pattern to the right by $t_1(c)$.

If $0 < k < m$ characters are matched, retrieve entry $t_1(c)$ from the bad-symbol table for the text's character c causing the mismatch and entry $d_2(k)$ from the good-suffix table and shift the pattern to the right by

$$d = \max \{d_1, d_2\}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

Example of Boyer-Moore alg. application

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	—
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

B E S S _ K N E W _ A B O U T _ B A O B A B S
 B A O B A B

$$d_1 = t_1(K) = 6 \quad B A O B A B$$

k	pattern	d_2
1	BAOBAB	2
2	BAOBAB	5
3	BAOBAB	5
4	BAOBAB <small>(success)</small>	5
5	BAOBAB	5

$$d_1 = t_1(_) - 2 = 4$$

$$\underline{d_2(2) = 5}$$

B A O B A B

$$\underline{d_1 = t_1(_) - 1 = 5}$$

$$d_2(1) = 2$$

B A O B A B

Boyer-Moore example from their paper



Find pattern AT_THAT in

WHICH_FINALLY_HALTS. ___ AT_THAT

Hashing



- ➊ A very efficient method for implementing a *dictionary*, i.e., a set with the operations:
 - find
 - insert
 - delete
- ➋ Based on representation-change and space-for-time tradeoff ideas
- ➌ Important applications:
 - symbol tables
 - databases (*extendible hashing*)

Hash tables and hash functions



The idea of *hashing* is to map keys of a given file of size n into a table of size m , called the *hash table*, by using a predefined function, called the *hash function*,

$$h: K \rightarrow \text{location (cell) in the hash table}$$

Example: student records, key = SSN. Hash function:
 $h(K) = K \bmod m$ where m is some integer (typically, prime)

Generally, a hash function should:

- be easy to compute
- distribute keys about evenly throughout the hash table

Collisions



If $h(K_1) = h(K_2)$, there is a *collision*

- ➊ Good hash functions result in fewer collisions but some collisions should be expected (*birthday paradox*)
- ➋ Two principal hashing schemes handle collisions differently:
 - *Open hashing*
 - each cell is a header of linked list of all keys hashed to it
 - *Closed hashing*
 - one key per cell
 - in case of collision, finds another cell by
 - *linear probing*: use next free bucket
 - *double hashing*: use second hash function to compute increment

Open hashing (Separate chaining)

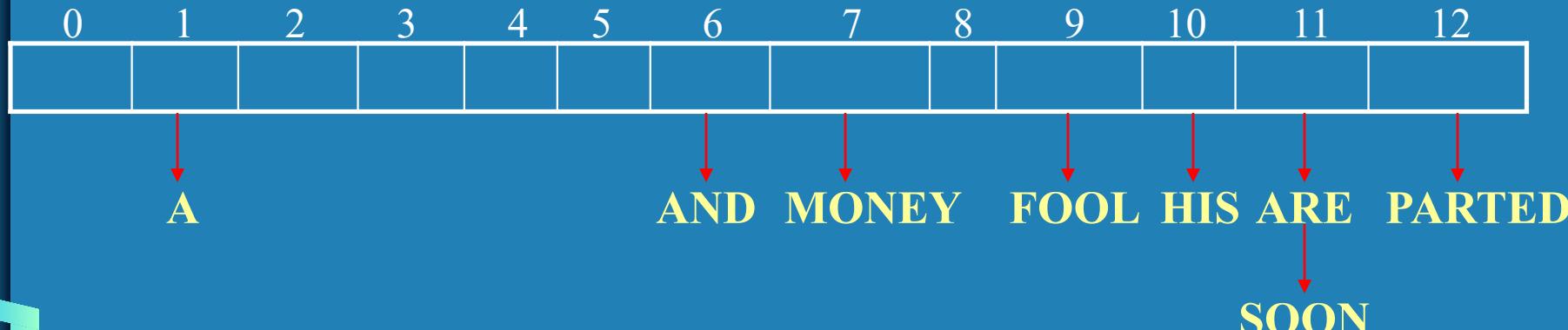


Keys are stored in linked lists outside a hash table whose elements serve as the lists' headers.

Example: A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

$h(K) = \text{sum of } K \text{'s letters' positions in the alphabet MOD 13}$

Key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
$h(K)$	1	9	6	10	7	11	11	12



Open hashing (cont.)



- If hash function distributes keys uniformly, average length of linked list will be $\alpha = n/m$. This ratio is called *load factor*.
- Average number of probes in successful, S , and unsuccessful searches, U :

$$S \approx 1 + \alpha/2, \quad U = \alpha$$

- Load α is typically kept small (ideally, about 1)
- Open hashing still works if $n > m$

Closed hashing (Open addressing)



Keys are stored inside a hash table.

Key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
$h(K)$	1	9	6	10	7	11	11	12

	0	1	2	3	4	5	6	7	8	9	10	11	12
		A											
		A								FOOL			
		A		AND						FOOL			
		A		AND						FOOL	HIS		
		A		AND	MONEY					FOOL	HIS		
		A		AND	MONEY					FOOL	HIS	ARE	
		A		AND	MONEY					FOOL	HIS	ARE	SOON
PARTED	A			AND	MONEY			FOOL	HIS	ARE	SOON		

Closed hashing (cont.)



- Does not work if $n > m$
- Avoids pointers
- Deletions are *not* straightforward
- Number of probes to find/insert/delete a key depends on load factor $\alpha = n/m$ (hash table density) and collision resolution strategy. For linear probing:
 $S = (\frac{1}{2}) (1 + \frac{1}{1-\alpha})$ and $U = (\frac{1}{2}) (1 + \frac{1}{(1-\alpha)^2})$
- As the table gets filled (α approaches 1), number of probes in linear probing increases dramatically:

α	$\frac{1}{2}(1 + \frac{1}{1-\alpha})$	$\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5

Iterative Improvement



Algorithm design technique for solving optimization problems

- Start with a feasible solution
- Repeat the following step until no improvement can be found:
 - change the current feasible solution to a feasible solution with a better value of the objective function
- Return the last feasible solution as optimal

Note: Typically, a change in a current solution is “small” (local search)

Major difficulty: Local optimum vs. global optimum

Important Examples



- ➊ **simplex method**
- ➋ **Ford-Fulkerson algorithm for maximum flow problem**
- ➌ **maximum matching of graph vertices**
- ➍ **Gale-Shapley algorithm for the stable marriage problem**

- ➎ **local search heuristics**

Linear Programming



Linear programming (LP) problem is to optimize a linear function of several variables subject to linear constraints:

maximize (or minimize) $c_1x_1 + \dots + c_nx_n$

subject to $a_{i1}x_1 + \dots + a_{in}x_n \leq (\text{or } \geq \text{ or } =) b_i, i = 1, \dots, m$

$$x_1 \geq 0, \dots, x_n \geq 0$$

**The function $z = c_1x_1 + \dots + c_nx_n$ is called the *objective function*;
constraints $x_1 \geq 0, \dots, x_n \geq 0$ are called *nonnegativity constraints***

Example



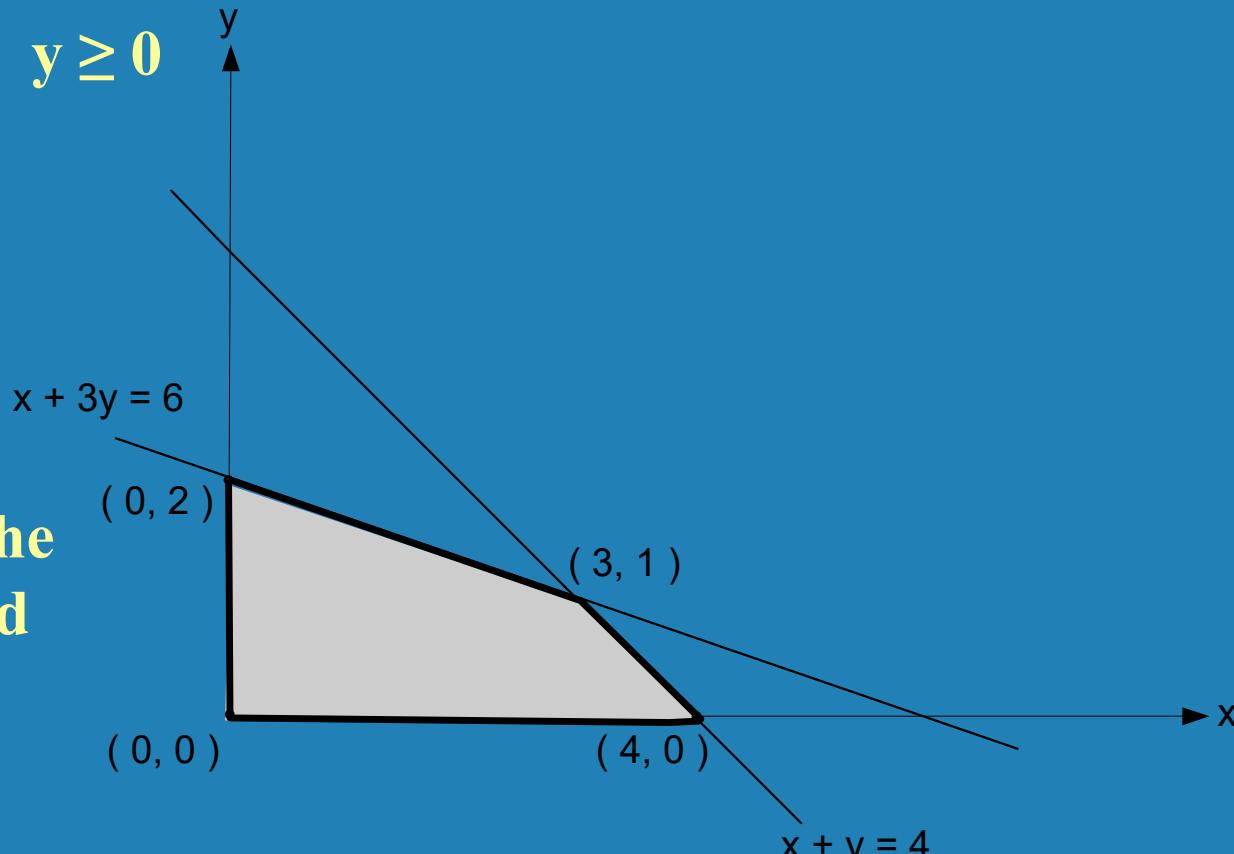
maximize $3x + 5y$

subject to $x + y \leq 4$

$$x + 3y \leq 6$$

$$x \geq 0, y \geq 0$$

Feasible region is the set of points defined by the constraints



Geometric solution



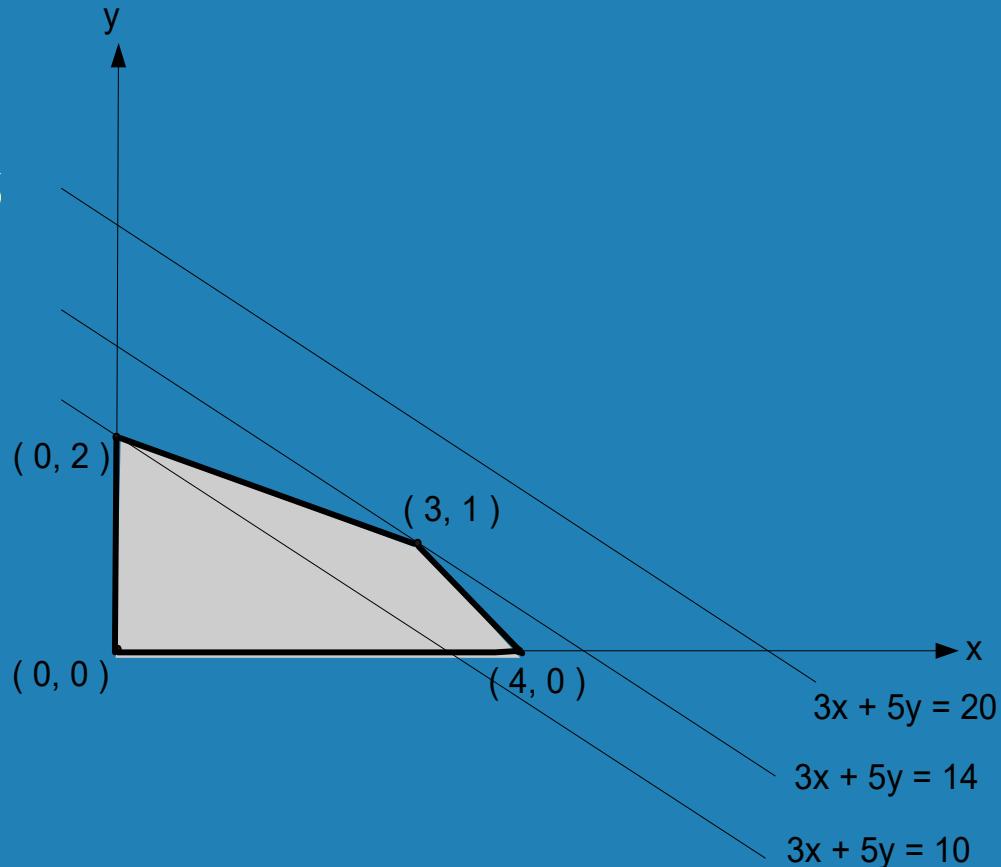
maximize $3x + 5y$

subject to $x + y \leq 4$

$$x + 3y \leq 6$$

$$x \geq 0, y \geq 0$$

Optimal solution: $x = 3, y = 1$



Extreme Point Theorem Any LP problem with a nonempty bounded feasible region has an optimal solution; moreover, an optimal solution can always be found at an *extreme point* of the problem's feasible region.

3 possible outcomes in solving an LP problem



- ➊ has a finite optimal solution, which may no be unique
- ➋ *unbounded*: the objective function of maximization (minimization) LP problem is unbounded from above (below) on its feasible region
- ➌ *infeasible*: there are no points satisfying all the constraints, i.e. the constraints are contradictory

The Simplex Method



- The classic method for solving LP problems;
one of the most important algorithms ever invented
- Invented by George Dantzig in 1947
- Based on the iterative improvement idea:
Generates a sequence of adjacent points of the
problem's feasible region with improving values of the
objective function until no further improvement is
possible

Standard form of LP problem



- must be a maximization problem
- all constraints (except the nonnegativity constraints) must be in the form of linear equations
- all the variables must be required to be nonnegative

Thus, the general linear programming problem in standard form with m constraints and n unknowns ($n \geq m$) is

maximize $c_1x_1 + \dots + c_nx_n$

subject to $a_{i1}x_1 + \dots + a_{in}x_n = b_i, \quad i = 1, \dots, m,$
 $x_1 \geq 0, \dots, x_n \geq 0$

Every LP problem can be represented in such form

Example



$$\text{maximize } 3x + 5y$$

$$\text{subject to } x + y \leq 4$$

$$x + 3y \leq 6 \quad \rightarrow$$

$$= 6$$

$$x \geq 0, \quad y \geq 0$$

$$\text{maximize } 3x + 5y + 0u + 0v$$

$$\text{subject to } x + y + u = 4$$

$$x + 3y + v$$

$$x \geq 0, \quad y \geq 0, \quad u \geq 0, \quad v \geq 0$$

Variables u and v , transforming inequality constraints into equality constraints, are called *slack variables*

Basic feasible solutions



A **basic solution** to a system of m linear equations in n unknowns ($n \geq m$) is obtained by setting $n - m$ variables to 0 and solving the resulting system to get the values of the other m variables. The variables set to 0 are called **nonbasic**; the variables obtained by solving the system are called **basic**.

A basic solution is called **feasible** if all its (basic) variables are nonnegative.

Example

$$\begin{array}{rcl} x + y + u & = 4 \\ x + 3y & + v & = 6 \end{array}$$

(0, 0, 4, 6) is basic feasible solution
(x, y are nonbasic; u, v are basic)

There is a 1-1 correspondence between extreme points of LP's feasible region and its basic feasible solutions.

Simplex Tableau



maximize $z = 3x + 5y + 0u + 0v$

subject to $x + y + u = 4$

$x + 3y + v = 6$

$x \geq 0, y \geq 0, u \geq 0, v \geq 0$

	x	y	u	v		
basic variables	u	1	1	1	0	4
	v	1	3	0	1	6
objective row		-3	-5	0	0	0

basic feasible solution
 $(0, 0, 4, 6)$

value of z at $(0, 0, 4, 6)$

Outline of the Simplex Method



Step 0 [Initialization] Present a given LP problem in standard form and set up initial tableau.

Step 1 [Optimality test] If all entries in the objective row are nonnegative — stop: the tableau represents an optimal solution.

Step 2 [Find entering variable] Select (the most) negative entry in the objective row. Mark its column to indicate the entering variable and the pivot column.

Step 3 [Find departing variable] For each positive entry in the pivot column, calculate the θ -ratio by dividing that row's entry in the rightmost column by its entry in the pivot column. (If there are no positive entries in the pivot column — stop: the problem is unbounded.) Find the row with the smallest θ -ratio, mark this row to indicate the departing variable and the pivot row.

Step 4 [Form the next tableau] Divide all the entries in the pivot row by its entry in the pivot column. Subtract from each of the other rows, including the objective row, the new pivot row multiplied by the entry in the pivot column of the row in question. Replace the label of the pivot row by the variable's name of the pivot column and go back to Step 1.

Example of Simplex Method Application



maximize $z = 3x + 5y + 0u + 0v$

subject to $x + y + u = 4$

$$x + 3y + v = 6$$

$$x \geq 0, y \geq 0, u \geq 0, v \geq 0$$

	x	y	u	v	
u	1	1	1	0	4
v	1	3	0	1	6
	-3	-5	0	0	0

← ←

	x	y	u	v	
u	$\frac{2}{3}$	0	1	$-\frac{1}{3}$	2
y	$\frac{1}{3}$	1	0	$\frac{1}{3}$	2
	$-\frac{4}{3}$	0	0	$\frac{5}{3}$	10

↑ ↑

	x	y	u	v	
x	1	0	$\frac{3}{2}$	$-\frac{1}{2}$	3
y	0	1	$-\frac{1}{2}$	$\frac{1}{2}$	1
	0	0	2	1	14

basic feasible sol.
 $(0, 0, 4, 6)$

$$z = 0$$

basic feasible sol.
 $(0, 2, 2, 0)$

$$z = 10$$

basic feasible sol.
 $(3, 1, 0, 0)$

$$z = 14$$

Notes on the Simplex Method



- Finding an initial basic feasible solution may pose a problem
- Theoretical possibility of cycling
- Typical number of iterations is between m and $3m$, where m is the number of equality constraints in the standard form
- Worse-case efficiency is exponential
- More recent *interior-point algorithms* such as *Karmarkar's algorithm* (1984) have polynomial worst-case efficiency and have performed competitively with the simplex method in empirical tests

Maximum Flow Problem

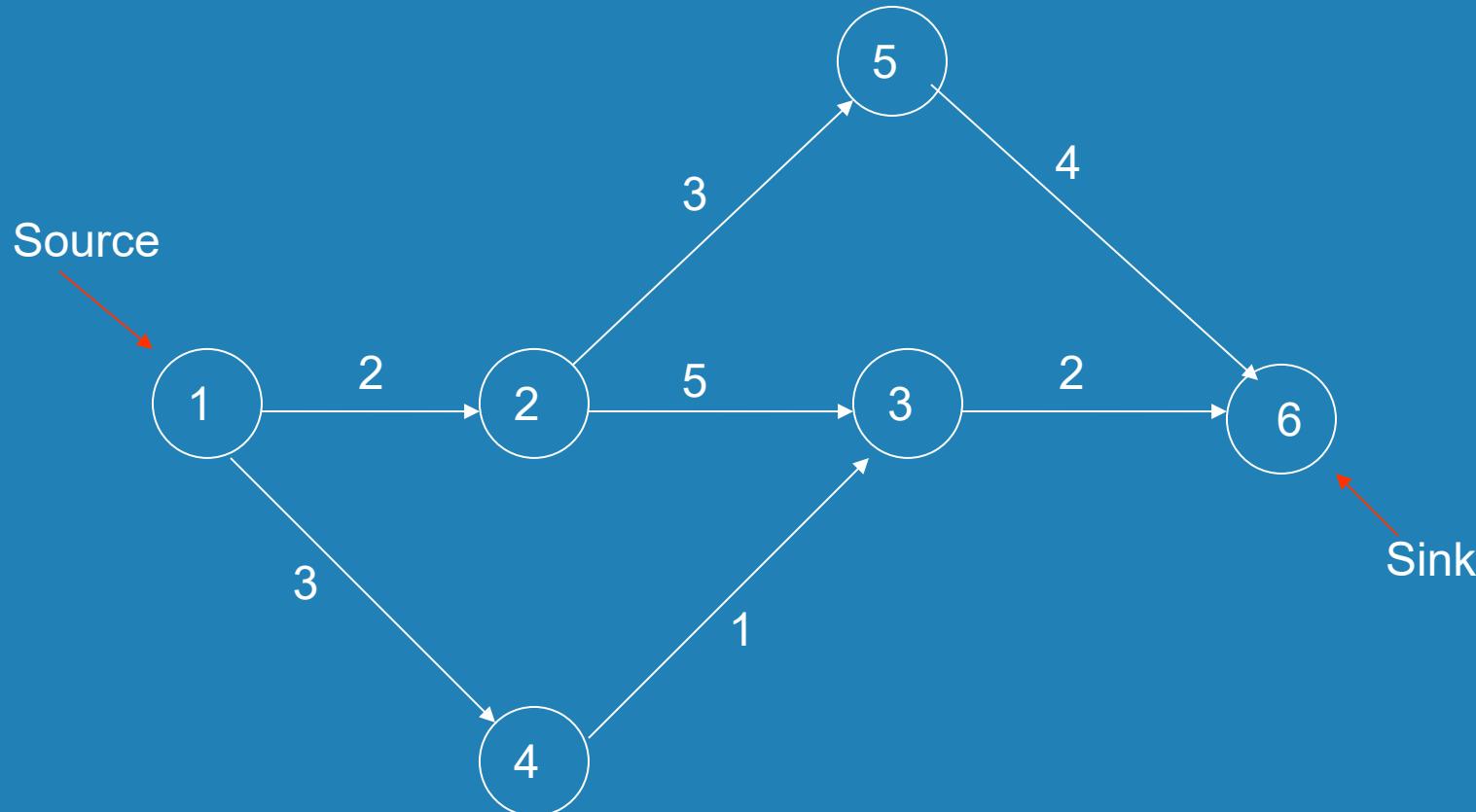


Problem of maximizing the flow of a material through a transportation network (e.g., pipeline system, communications or transportation networks)

Formally represented by a connected weighted digraph with n vertices numbered from 1 to n with the following properties:

- contains exactly one vertex with no entering edges, called the *source* (numbered 1)
- contains exactly one vertex with no leaving edges, called the *sink* (numbered n)
- has positive integer weight u_{ij} on each directed edge (i,j) , called the *edge capacity*, indicating the upper bound on the amount of the material that can be sent from i to j through this edge

Example of Flow Network



Definition of a Flow



A **flow** is an assignment of real numbers x_{ij} to edges (i,j) of a given network that satisfy the following:

- *flow-conservation requirements*

The total amount of material entering an intermediate vertex must be equal to the total amount of the material leaving the vertex

$$\sum_{j: (j,i) \in E} x_{ji} = \sum_{j: (i,j) \in E} x_{ij} \text{ for } i = 2, 3, \dots, n-1$$

- *capacity constraints*

$$0 \leq x_{ij} \leq u_{ij} \text{ for every edge } (i,j) \in E$$

Flow value and Maximum Flow Problem



Since no material can be lost or added to by going through intermediate vertices of the network, the total amount of the material leaving the source must end up at the sink:

$$\sum_{j: (1,j) \in E} x_{1j} = \sum_{j: (j,n) \in E} x_{jn}$$

The *value* of the flow is defined as the total outflow from the source (= the total inflow into the sink).

The *maximum flow problem* is to find a flow of the largest value (maximum flow) for a given network.

Maximum-Flow Problem as LP problem



Maximize $v = \sum_{j: (1,j) \in E} x_{1j}$

subject to

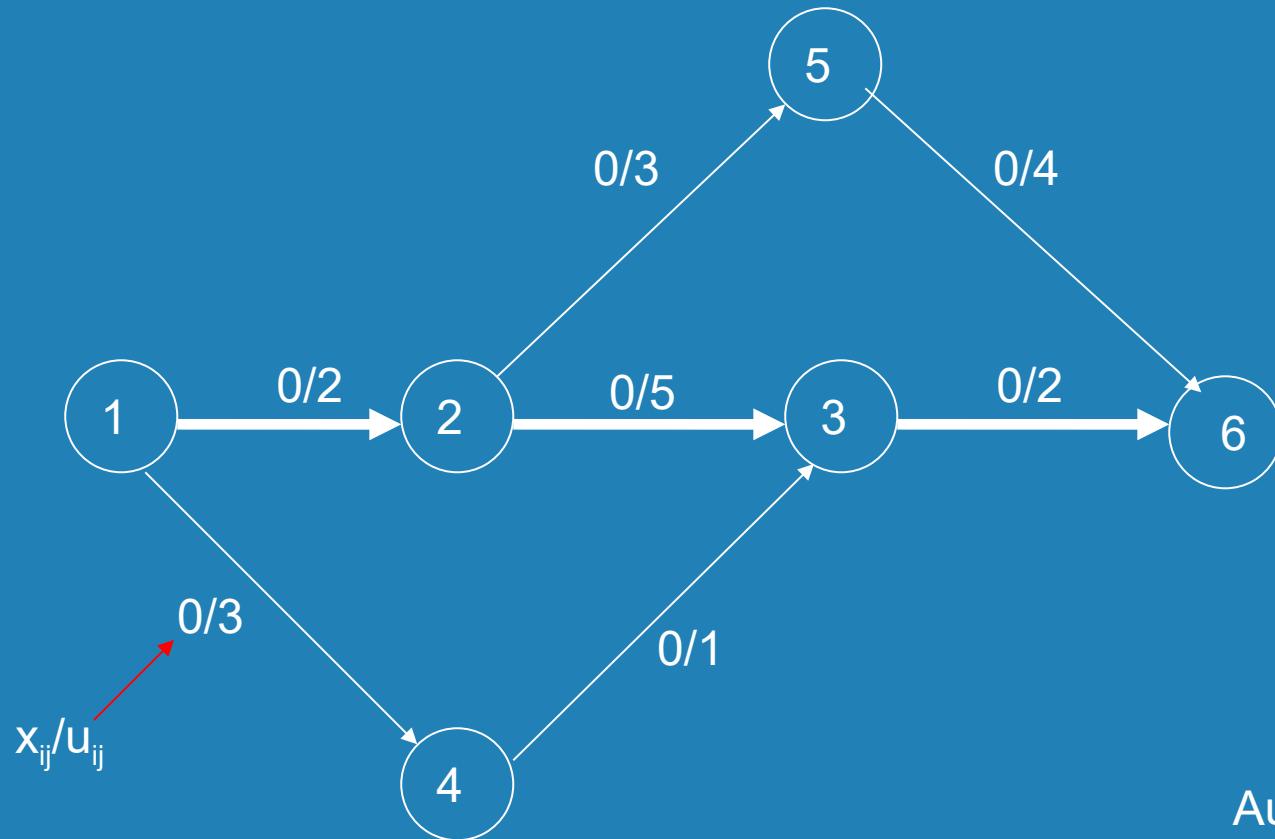
$$\sum_{j: (j,i) \in E} x_{ji} - \sum_{j: (i,j) \in E} x_{ij} = 0 \quad \text{for } i = 2, 3, \dots, n-1$$

$$0 \leq x_{ij} \leq u_{ij} \quad \text{for every edge } (i,j) \in E$$

Augmenting Path (Ford-Fulkerson) Method

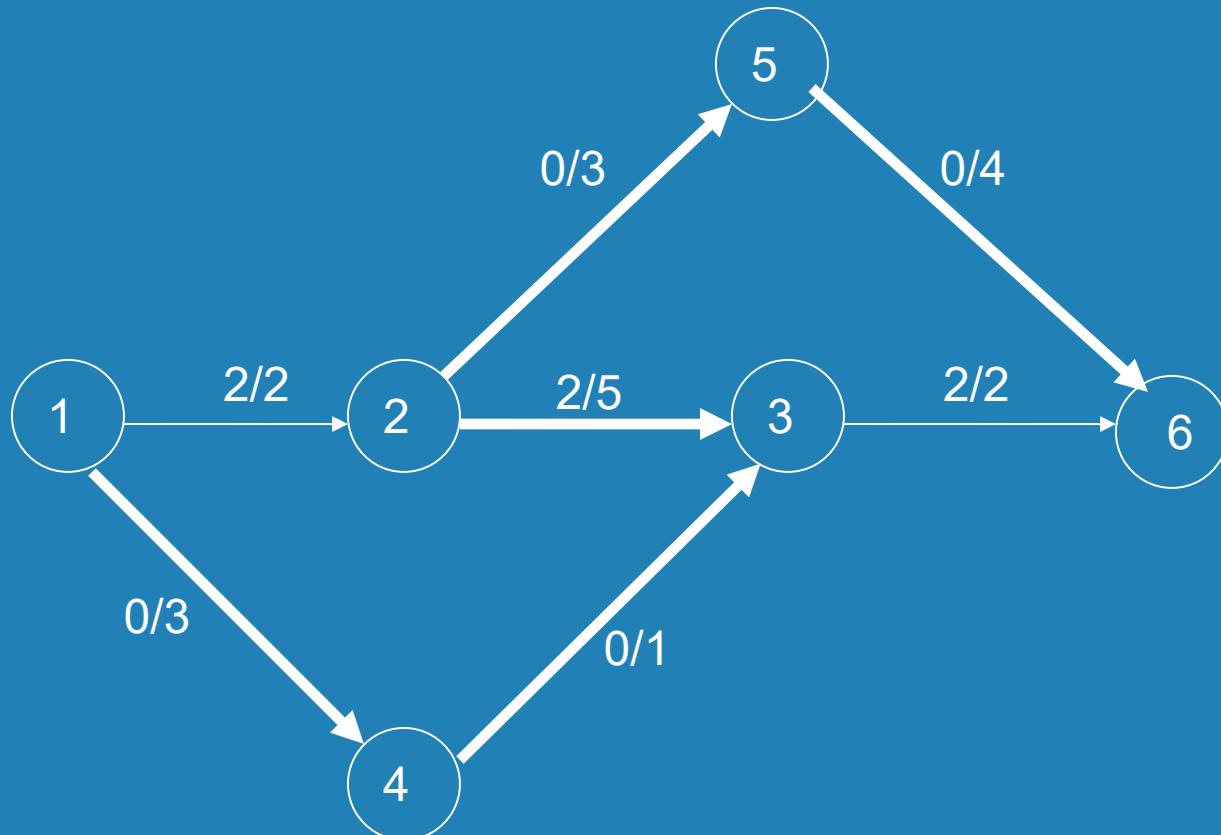
- Start with the zero flow ($x_{ij} = 0$ for every edge)
- On each iteration, try to find a *flow-augmenting path* from source to sink, which a path along which some additional flow can be sent
- If a flow-augmenting path is found, adjust the flow along the edges of this path to get a flow of increased value and try again
- If no flow-augmenting path is found, the current flow is maximum

Example 1



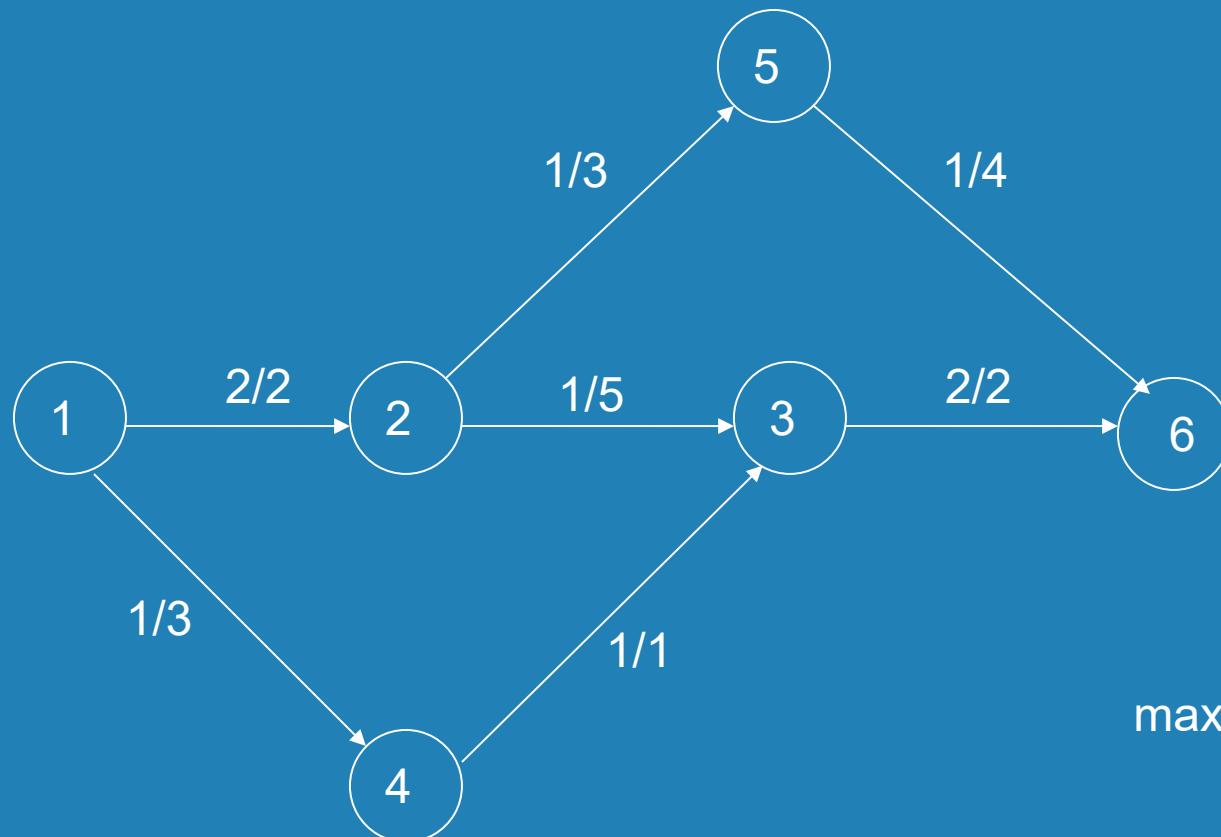
Augmenting path:
1 → 2 → 3 → 6

Example 1 (cont.)



Augmenting path:
1 → 4 → 3 ← 2 ← 5 → 6

Example 1 (maximum flow)



Finding a flow-augmenting path



To find a flow-augmenting path for a flow x , consider paths from source to sink in the underlying undirected graph in which any two consecutive vertices i,j are either:

- connected by a directed edge (i to j) with some positive unused capacity $r_{ij} = u_{ij} - x_{ij}$
 - known as *forward edge* (\rightarrow)
- OR
- connected by a directed edge (j to i) with positive flow x_{ji}
 - known as *backward edge* (\leftarrow)

If a flow-augmenting path is found, the current flow can be increased by r units by increasing x_{ij} by r on each forward edge and decreasing x_{ji} by r on each backward edge, where

$$r = \min \{r_{ij} \text{ on all forward edges, } x_{ji} \text{ on all backward edges}\}$$

Finding a flow-augmenting path (cont.)



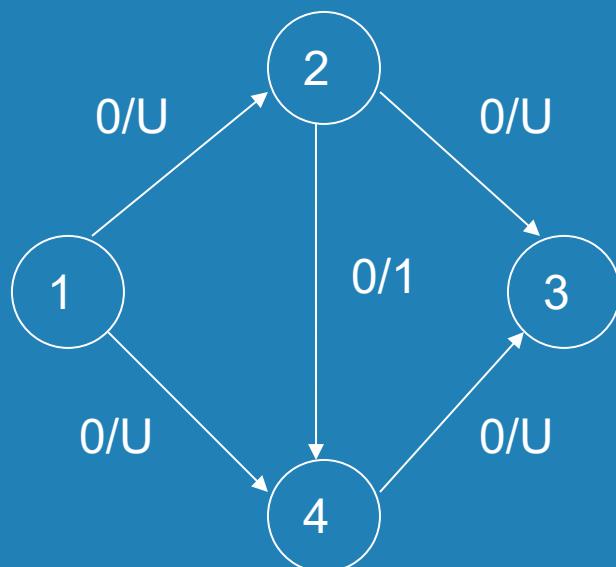
- Assuming the edge capacities are integers, r is a positive integer
- On each iteration, the flow value increases by at least 1
- Maximum value is bounded by the sum of the capacities of the edges leaving the source; hence the augmenting-path method has to stop after a finite number of iterations
- The final flow is always maximum, its value doesn't depend on a sequence of augmenting paths used

Performance degeneration of the method



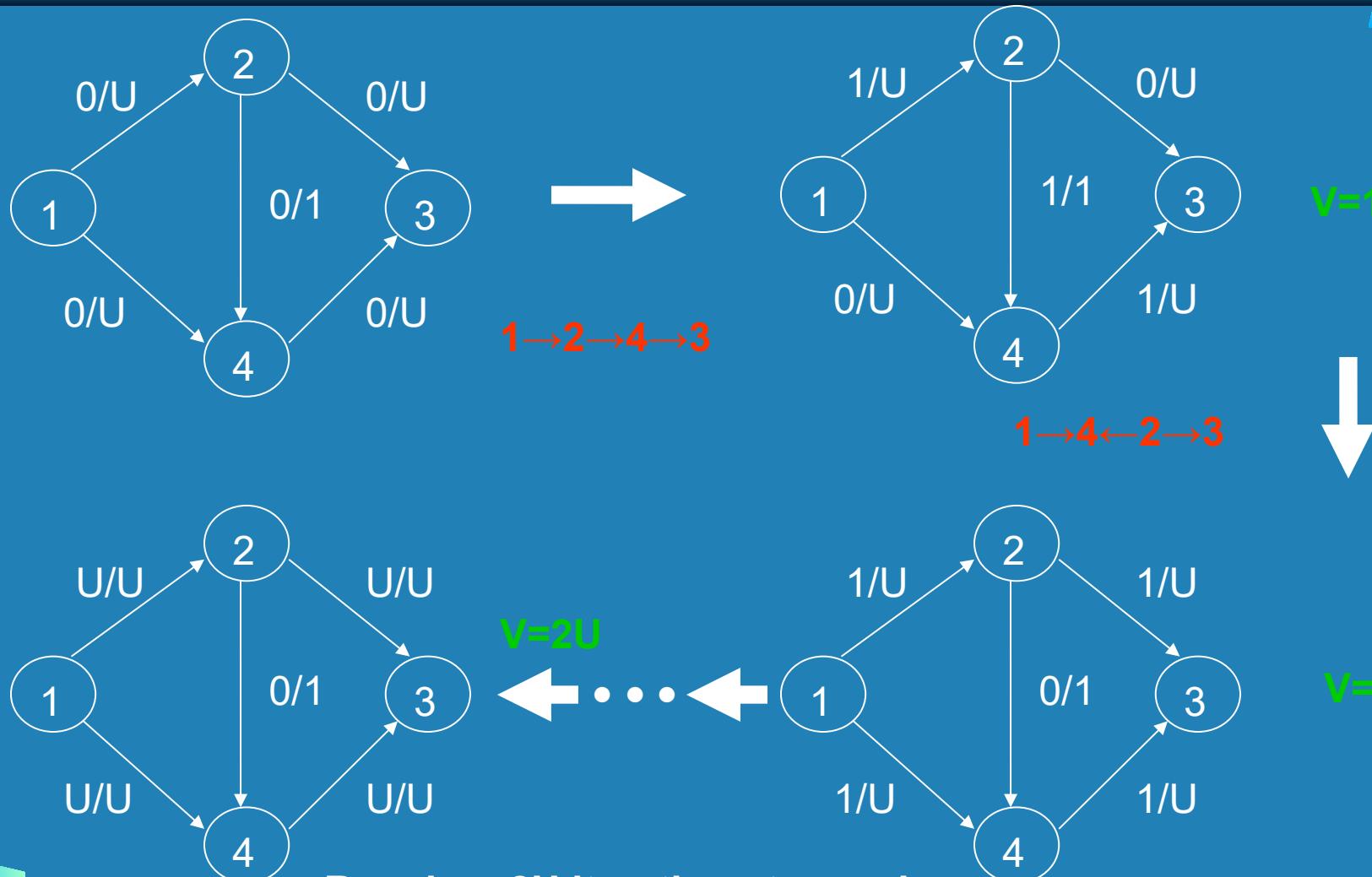
- The augmenting-path method doesn't prescribe a specific way for generating flow-augmenting paths
- Selecting a bad sequence of augmenting paths could impact the method's efficiency

Example 2



$U = \text{large positive integer}$

Example 2 (cont.)



Requires 2U iterations to reach
maximum flow of value 2U

Shortest-Augmenting-Path Algorithm



Generate augmenting path with the least number of edges by BFS as follows.

Starting at the source, perform BFS traversal by marking new (unlabeled) vertices with two labels:

- first label – indicates the amount of additional flow that can be brought from the source to the vertex being labeled
- second label – indicates the vertex from which the vertex being labeled was reached, with “+” or “-” added to the second label to indicate whether the vertex was reached via a forward or backward edge

Vertex labeling



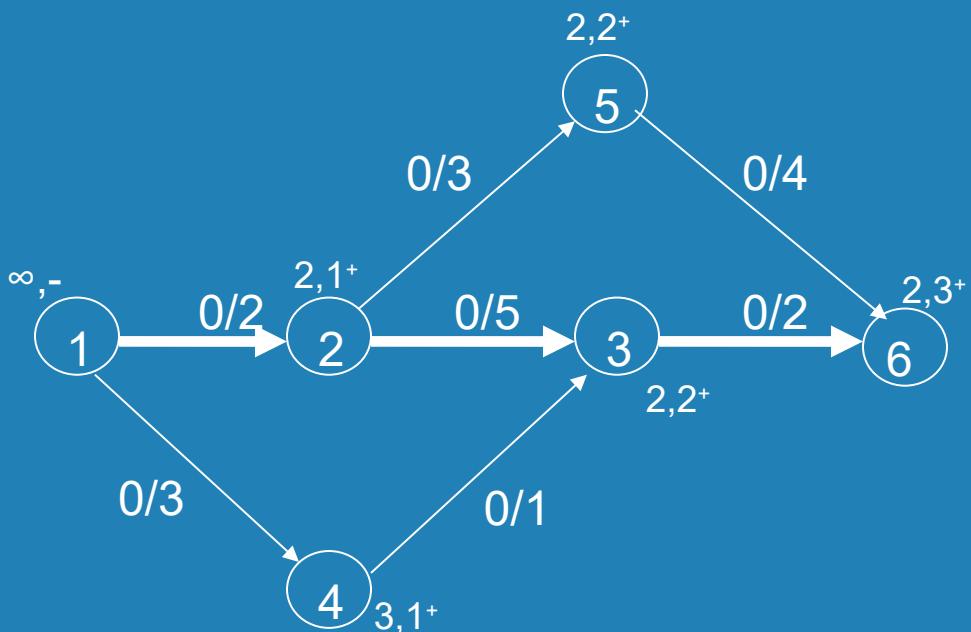
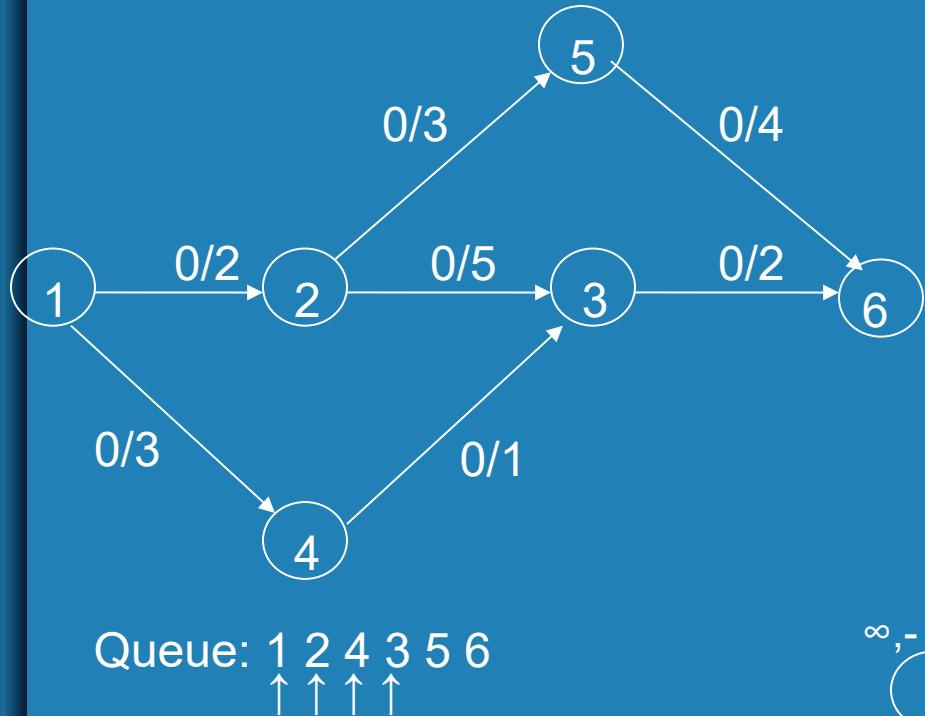
- The source is always labeled with $\infty, -$
- All other vertices are labeled as follows:
 - If unlabeled vertex j is connected to the front vertex i of the traversal queue by a directed edge from i to j with positive unused capacity $r_{ij} = u_{ij} - x_{ij}$ (forward edge), vertex j is labeled with l_j, i^+ , where $l_j = \min\{l_i, r_{ij}\}$
 - If unlabeled vertex j is connected to the front vertex i of the traversal queue by a directed edge from j to i with positive flow x_{ji} (backward edge), vertex j is labeled l_j, i , where $l_j = \min\{l_i, x_{ji}\}$

Vertex labeling (cont.)



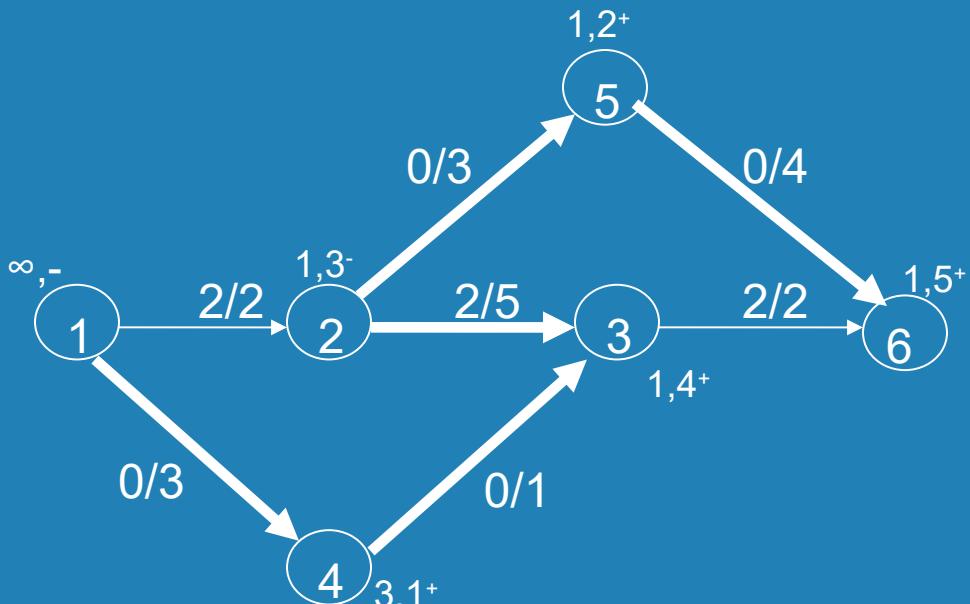
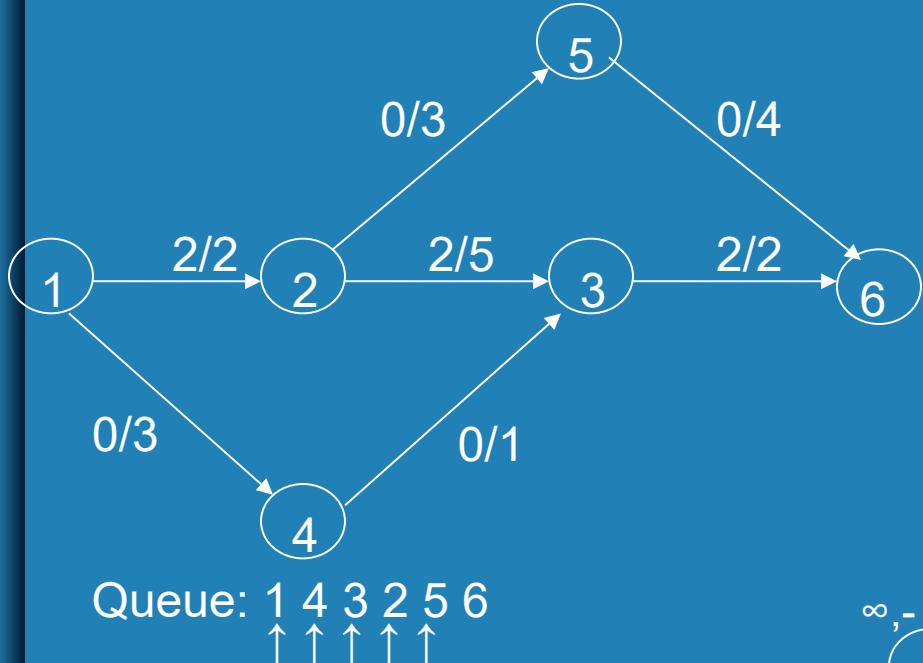
- If the sink ends up being labeled, the current flow can be augmented by the amount indicated by the sink's first label
- The augmentation of the current flow is performed along the augmenting path traced by following the vertex second labels from sink to source; the current flow quantities are increased on the forward edges and decreased on the backward edges of this path
- If the sink remains unlabeled after the traversal queue becomes empty, the algorithm returns the current flow as maximum and stops

Example: Shortest-Augmenting-Path Algorithm



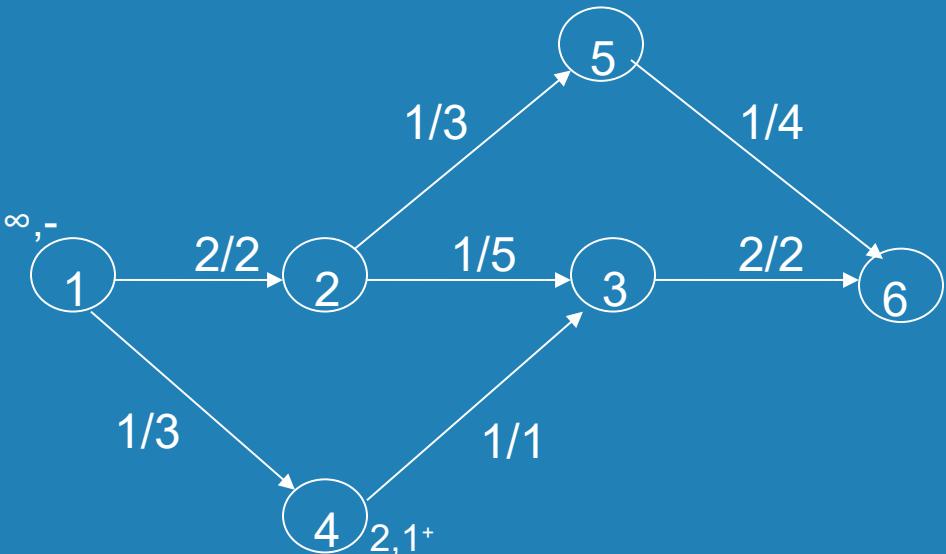
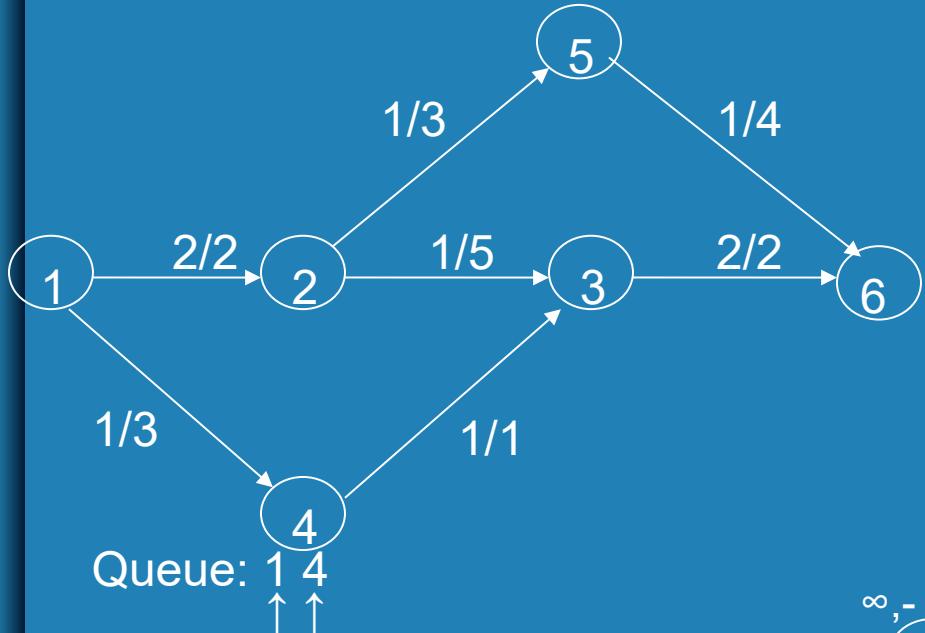
Augment the flow by 2 (the sink's first label) along the path 1 → 2 → 3 → 6

Example (cont.)



Augment the flow by 1 (the sink's first label) along the path $1 \rightarrow 4 \rightarrow 3 \leftarrow 2 \leftarrow 5 \rightarrow 6$

Example (cont.)



No augmenting path (the sink is unlabeled)
the current flow is maximum

Shortest-augmenting-path algorithm

Input: A network with single source 1, single sink n , and positive integer capacities u_{ij} on its edges (i, j)

Output: A maximum flow x

assign $x_{ij} = 0$ to every edge (i, j) in the network

label the source with $\infty, -$ and add the source to the empty queue Q

while not $Empty(Q)$ **do**

$i \leftarrow Front(Q); Dequeue(Q)$

for every edge from i to j **do** //forward edges

if j is unlabeled

$r_{ij} \leftarrow u_{ij} - x_{ij}$

if $r_{ij} > 0$

$l_j \leftarrow \min\{l_i, r_{ij}\};$ label j with l_j, i^+

$Enqueue(Q, j)$

for every edge from j to i **do** //backward edges

if j is unlabeled

if $x_{ji} > 0$

$l_j \leftarrow \min\{l_i, x_{ji}\};$ label j with l_j, i^-

$Enqueue(Q, j)$

if the sink has been labeled

//augment along the augmenting path found

$j \leftarrow n$ //start at the sink and move backwards using second labels

while $j \neq 1$ //the source hasn't been reached

if the second label of vertex j is i^+

$x_{ij} \leftarrow x_{ij} + l_n$

else //the second label of vertex j is i^-

$x_{ji} \leftarrow x_{ji} - l_n$

$j \leftarrow i$

erase all vertex labels except the ones of the source

reinitialize Q with the source

return x //the current flow is maximum

Time Efficiency



- The number of augmenting paths needed by the shortest-augmenting-path algorithm never exceeds $nm/2$, where n and m are the number of vertices and edges, respectively
- Since the time required to find shortest augmenting path by breadth-first search is in $O(n+m)=O(m)$ for networks represented by their adjacency lists, the time efficiency of the shortest-augmenting-path algorithm is in $O(nm^2)$ for this representation
- More efficient algorithms have been found that can run in close to $O(nm)$ time, but these algorithms don't fall into the iterative-improvement paradigm

Definition of a Cut

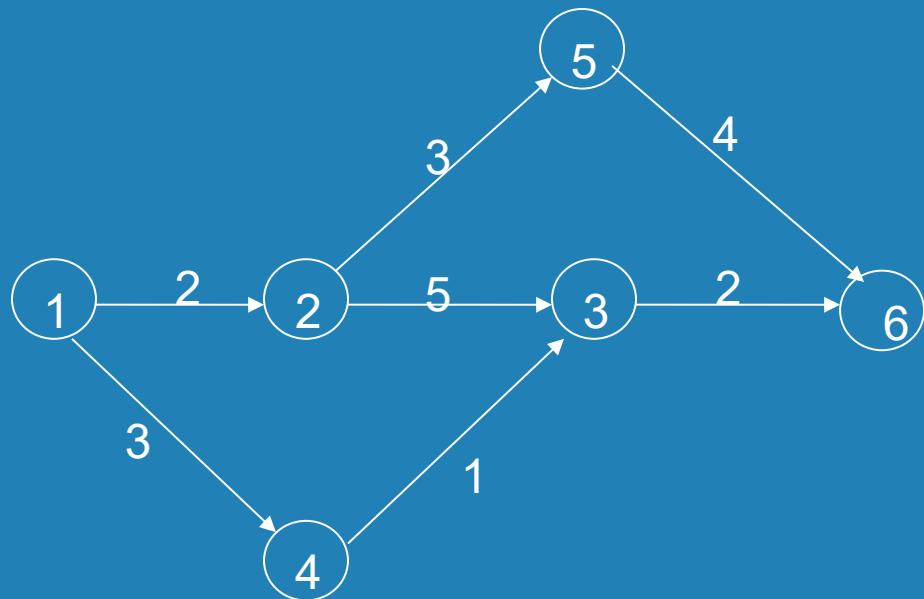


Let X be a set of vertices in a network that includes its source but does not include its sink, and let \bar{X} , the complement of X , be the rest of the vertices including the sink. The *cut* induced by this partition of the vertices is the set of all the edges with a tail in X and a head in \bar{X} .

Capacity of a cut is defined as the sum of capacities of the edges that compose the cut.

- We'll denote a cut and its capacity by $C(X, \bar{X})$ and $c(X, \bar{X})$
- Note that if all the edges of a cut were deleted from the network, there would be no directed path from source to sink
- *Minimum cut* is a cut of the smallest capacity in a given network

Examples of network cuts



If $X = \{1\}$ and $\bar{X} = \{2,3,4,5,6\}$, $C(X, \bar{X}) = \{(1,2), (1,4)\}$, $c = 5$

If $X = \{1,2,3,4,5\}$ and $\bar{X} = \{6\}$, $C(X, \bar{X}) = \{(3,6), (5,6)\}$, $c = 6$

If $X = \{1,2,4\}$ and $\bar{X} = \{3,5,6\}$, $C(X, \bar{X}) = \{(2,3), (2,5), (4,3)\}$, $c = 9$

Max-Flow Min-Cut Theorem



- The value of maximum flow in a network is equal to the capacity of its minimum cut
- The shortest augmenting path algorithm yields both a maximum flow and a minimum cut:
 - maximum flow is the final flow produced by the algorithm
 - minimum cut is formed by all the edges from the labeled vertices to unlabeled vertices on the last iteration of the algorithm
 - all the edges from the labeled to unlabeled vertices are full, i.e., their flow amounts are equal to the edge capacities, while all the edges from the unlabeled to labeled vertices, if any, have zero flow amounts on them

Stable Marriage Problem



There is a set $Y = \{m_1, \dots, m_n\}$ of n men and a set $X = \{w_1, \dots, w_n\}$ of n women. Each man has a ranking list of the women, and each woman has a ranking list of the men (with no ties in these lists).

A *marriage matching* M is a set of n pairs (m_i, w_j) .

A pair (m, w) is said to be a *blocking pair* for matching M if man m and woman w are not matched in M but prefer each other to their mates in M .

A marriage matching M is called *stable* if there is no blocking pair for it; otherwise, it's called *unstable*.

The *stable marriage problem* is to find a stable marriage matching for men's and women's given preferences.

Instance of the Stable Marriage Problem

An instance of the stable marriage problem can be specified either by two sets of preference lists or by a ranking matrix, as in the example below.

men's preferences

1st 2nd 3rd

Bob: Lea Ann Sue

Jim: Lea Sue Ann

Tom: Sue Lea Ann

women's preferences

1st 2nd 3rd

Ann: Jim Tom Bob

Lea: Tom Bob Jim

Sue: Jim Tom Bob

ranking matrix

Ann Lea Sue

Bob 2,3 1,2 3,3

Jim 3,1 1,3 2,1

Tom 3,2 2,1 1,2

{(Bob, Ann) (Jim, Lea) (Tom, Sue)} is unstable

{(Bob, Ann) (Jim, Sue) (Tom, Lea)} is stable

Stable Marriage Algorithm (Gale-Shapley)



Step 0 Start with all the men and women being free

Step 1 While there are free men, arbitrarily select one of them and do the following:

Proposal The selected free man m proposes to w , the next woman on his preference list

Response If w is free, she accepts the proposal to be matched with m . If she is not free, she compares m with her current mate. If she prefers m to him, she accepts m 's proposal, making her former mate free; otherwise, she simply rejects m 's proposal, leaving m free

Step 2 Return the set of n matched pairs

Example



Free men:
Bob, Jim, Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Bob proposed to Lea
Lea accepted

Free men:
Jim, Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Jim proposed to Lea
Lea rejected

Example (cont.)



Free men:
Jim, Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Jim proposed to Sue
Sue accepted

Free men:
Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Tom proposed to Sue
Sue rejected

Example (cont.)



Free men:
Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Tom proposed to Lea
Lea replaced Bob
with Tom

Free men:
Bob

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Bob proposed to Ann
Ann accepted

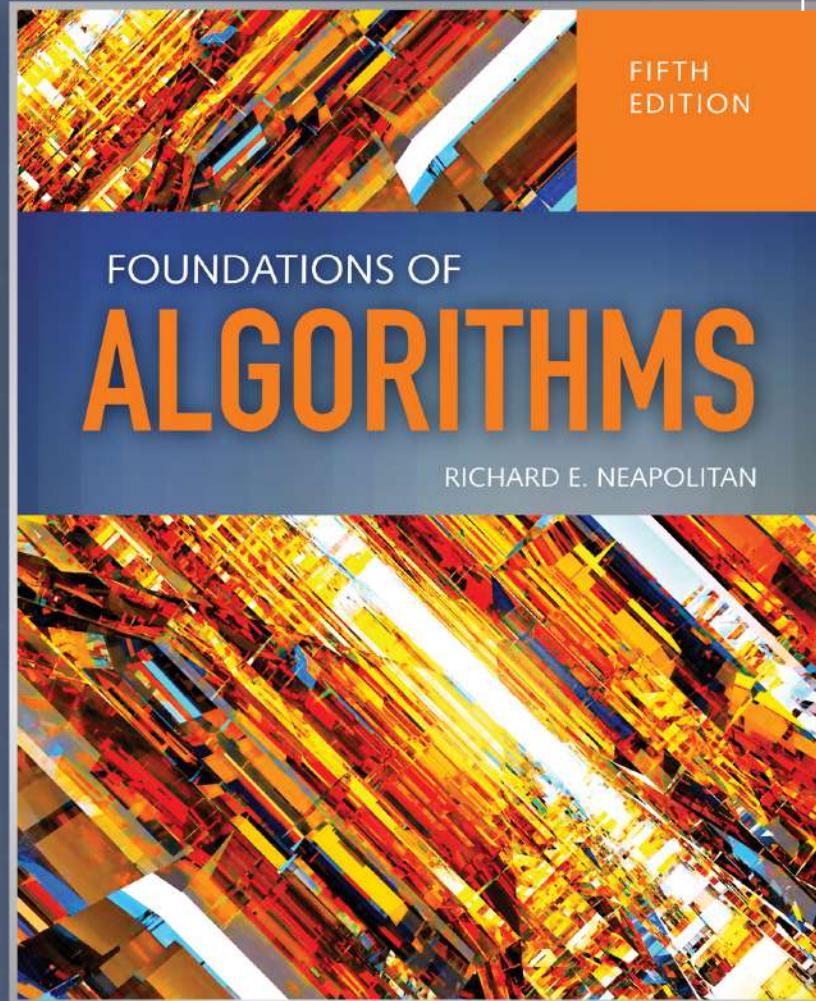
Analysis of the Gale-Shapley Algorithm



- The algorithm terminates after no more than n^2 iterations with a stable marriage output
- The stable matching produced by the algorithm is always *man-optimal*: each man gets the highest rank woman on his list under any stable marriage. One can obtain the *woman-optimal* matching by making women propose to men
- A man (woman) optimal matching is unique for a given set of participant preferences
- The stable marriage problem has practical applications such as matching medical-school graduates with hospitals for residency training

Computational Complexity and Intractability: An Introduction to the Theory of NP

Chapter 9



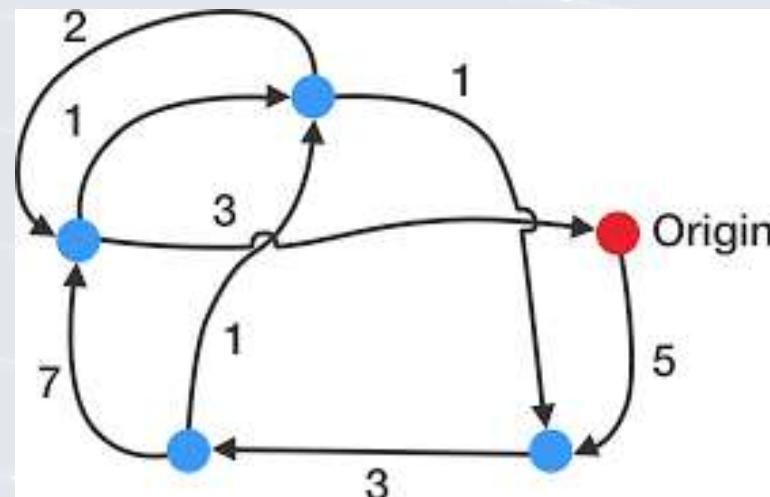


Objectives

- Classify problems as tractable or intractable
- Define decision problems
- **Define the class P**
- Define nondeterministic algorithms
- **Define the class NP**
- Define polynomial transformations
- **Define the class of NP-Complete**

Traveling Salesperson Problem

- No algorithm has ever been developed with a Worst-case time complexity better than exponential
- **It has never been proven that such an algorithm is not possible**
- NP-Complete



Intractability

- Dictionary Definition of intractable: “**difficult to treat or work.**”
- Computer Science: **problem is intractable if a computer has difficulty solving it**

SYNONYM.COM

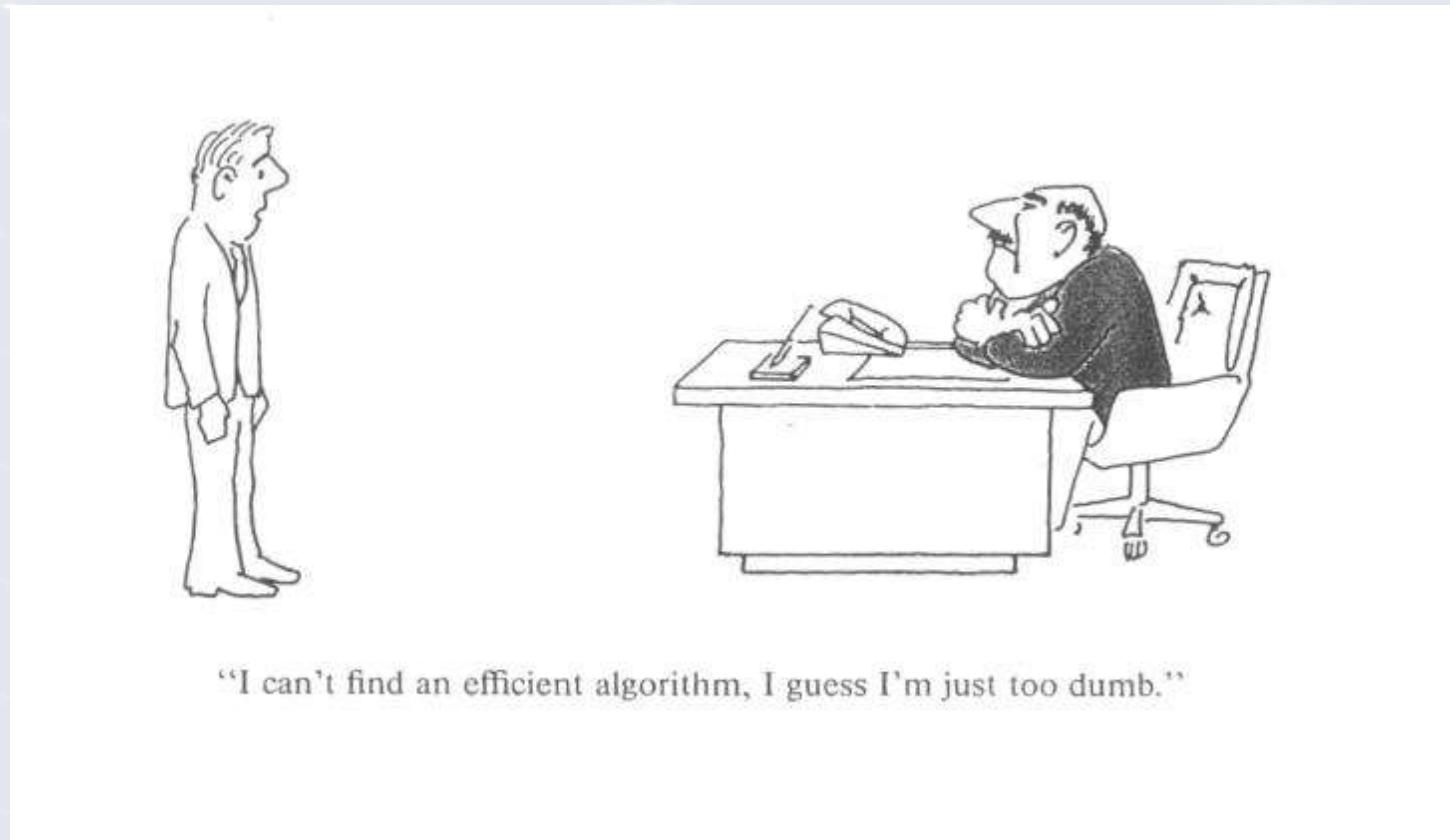
Synonyms for INTRACTABLE

unmalleable obstinate noncompliant balky refractory balking
uncontrollable untamed unregenerate stubborn difficult tractability
disobedient tractleness defiant wild unmanageable





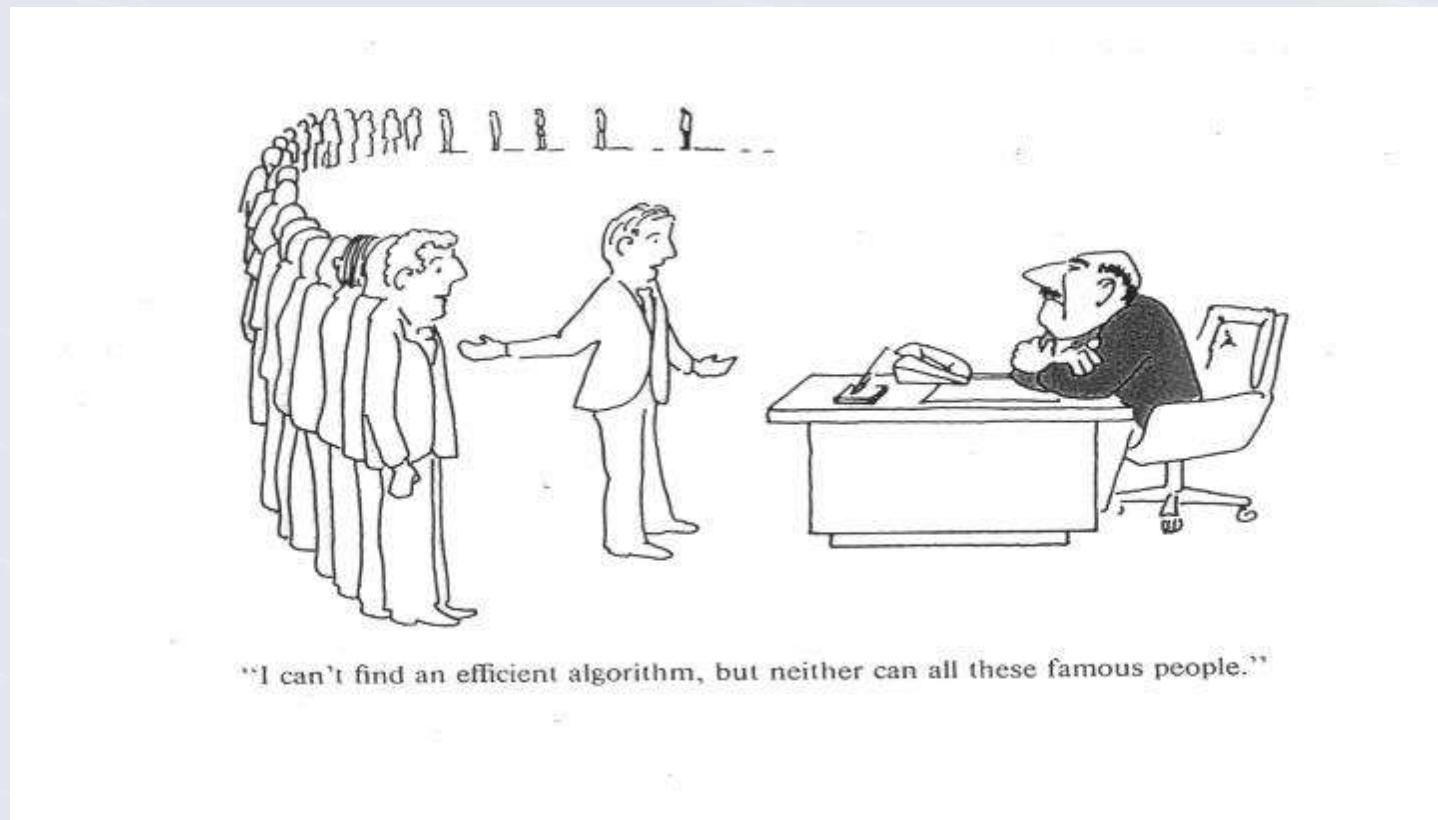
Computational Intractability



"I can't find an efficient algorithm, I guess I'm just too dumb."

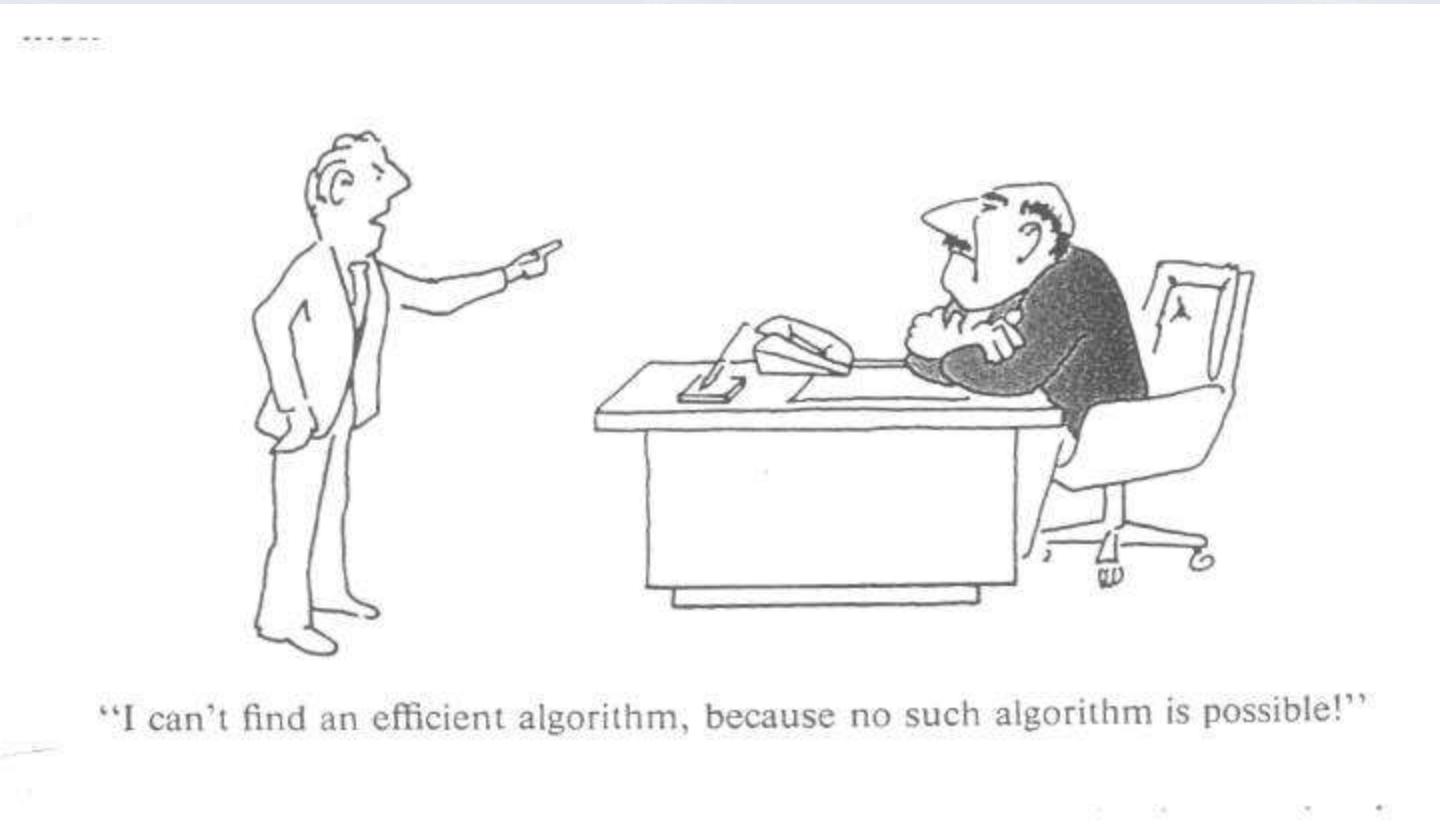


Computational Intractability





Computational Intractability





Tractable

- A problem is **tractable** if there exists a **polynomial-bound algorithm** that solves it.
- Worst-case growth rate can be bounded by a polynomial function of its input size
- $P(n) = a_n n^k + \dots + a_1 n + a_0$ where k is a constant
- $P(n) \in \Theta(n^k)$
- $n \lg n$ not a polynomial
 - $n \lg n < n^2$ bound by a polynomial

Intractable

- “**Difficult to treat or work**”
- A problem in CS is **intractable** if a **computer has difficulty solving it**
- **A problem is intractable if it is not tractable**
- Any algorithm with a growth rate not bounded by a polynomial
- c^n , $c^{.01n}$, $n^{\log n}$, $n!$, etc.
- **Property of the problem not the algorithm**

Three General Categories of Problems

1. Problems for which polynomial-time algorithms have been found
2. Problems that have been proven to be intractable
3. Problems that have not been proven to be intractable, but for which polynomial-time algorithms have never been found



Polynomial-time Algorithms

- $\Theta(n \lg n)$ for sorting
- $\Theta(\lg n)$ for searching
- $\Theta(n^3)$ for chained-matrix multiplication

Polynomial time

- | | | |
|-----------|---|-----------------------|
| n | - | Linear Search |
| $\lg n$ | - | Binary Search |
| n^2 | - | Insertion Sort |
| $n \lg n$ | - | Merge Sort |
| n^3 | - | Matrix Multiplication |



Proven to be Intractable

- **Unrealistic definition of the Problem**
(Hamiltonian Circuits)
- **Un-Decidable problems:** The Halting Problem
(proven un-decidable by Alan Turing).
- **Decidable intractable problems:** researchers
have shown some problems from automata and
mathematical logic intractable

Not proven to be intractable no existing polynomial time algorithm

- Traveling salesperson
- 0-1 Knapsack
- Graph coloring
- Sum of subsets

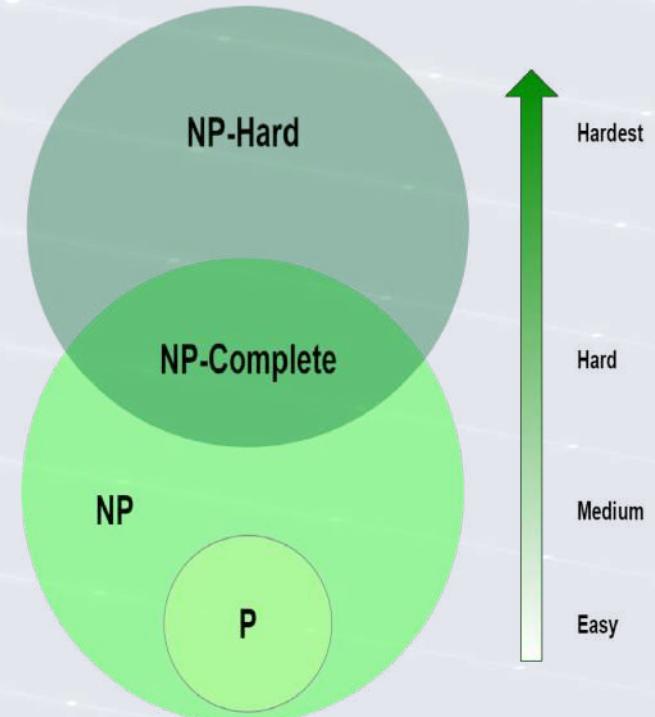
Exponential Time

- | | |
|-------|------------------|
| 2^n | - 0/1 knapsack |
| 2^n | - Travelling SP |
| 2^n | - Sum of Subsets |
| 2^n | - Graph Coloring |
| 2^n | - Hamilton Cycle |



Define

- Decision problems
- **The class P**
- Nondeterministic algorithms
- **The class NP**
- Polynomial transformations
- **The class of NP-Complete**



Decision problem

- **Problem where the output is a simple “yes” or “no”**
- Theory of NP-completeness is developed by restricting problems to decision problems
- **Optimization problems can be transformed into decision problems**
- Optimization problems are at least as hard as the associated decision problem
- If polynomial-time algorithm for the optimization problem is found, we would have a polynomial-time algorithm for the corresponding decision problem



Decision Problems

- **Traveling Salesperson**

- For a given positive number d , is there a tour having length $\leq d$?

- **0-1 Knapsack**

- For a given profit P , is it possible to load the knapsack such that total weight $\leq W$?



Class P

- **The set of all decision problems that can be solved by polynomial-time algorithms**
- Decision versions of searching, shortest path, spanning tree, etc. belong to P
- Do problems such as traveling salesperson and 0-1 Knapsack (no polynomial-time algorithm has been found), etc., belong to P?
 - No one knows
 - **To know a decision problem is not in P, it must be proven it is not possible to develop a polynomial-time algorithm to solve it**

Nondeterministic Algorithms – consist of 2 phases

1. Nondeterministic phase –

Guessing Phase: given an instance of a problem, a solution is guessed

2. Deterministic phase –

Verification Phase

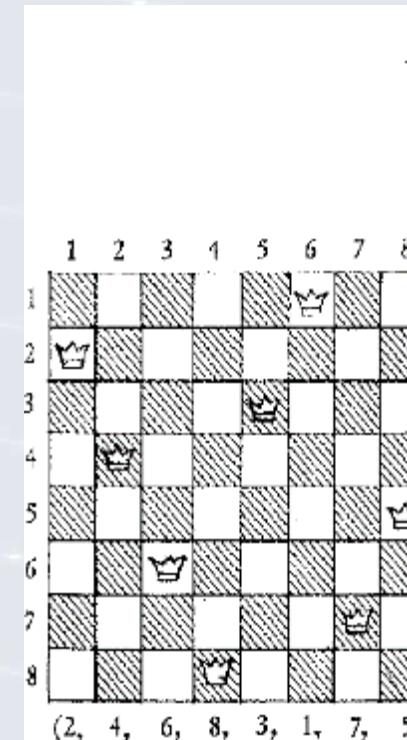


FIG. 1

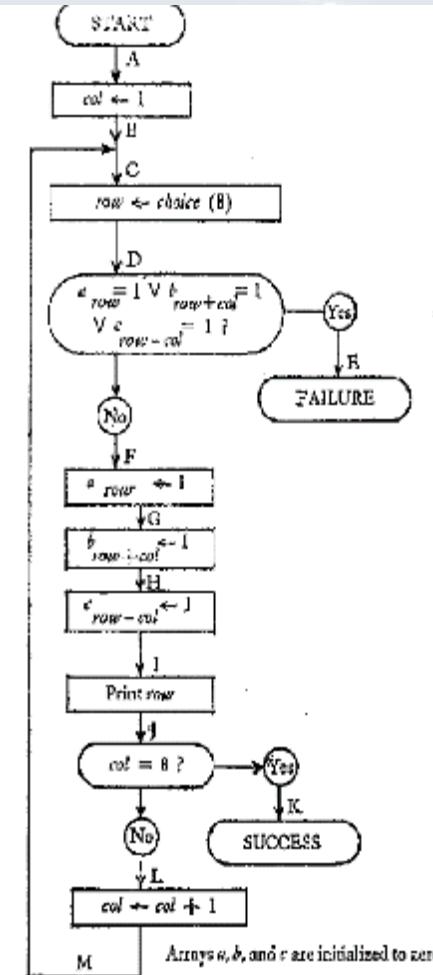


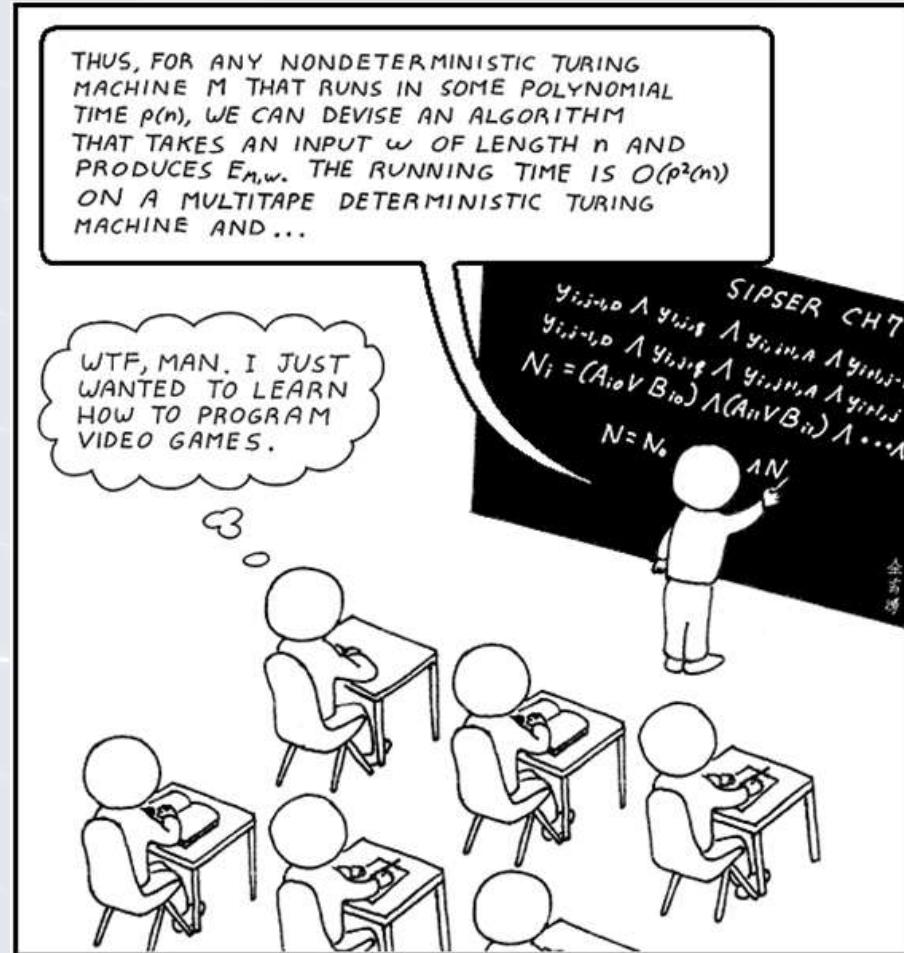
FIG. 2. Nondeterministic algorithm for the eight queens problem

Deterministic Phase – Verification Phase

- Input
 - Instance of the problem
 - String s: **the guess**
- Phase proceeds in an ordinary, deterministic manner:
 - Eventually halts with an answer yes – the guess, s, **is a solution to the problem**
 - Eventually halts with an answer no – the guess, s, **is not a solution to the problem**
 - Continues executing for ever

Polynomial-time Nondeterministic Algorithm (NDA)

- A nondeterministic algorithm whose **verification stage** is a polynomial-time algorithm





Class NP

- **The set of all decision problems that can be solved by polynomial-time nondeterministic algorithms**
- Nondeterministic polynomial
- For a problem to be in NP, there must be an algorithm that does the verification in polynomial time
- **Traveling salesperson decision problem belongs to NP**
 - Show a guess, s , length polynomial bounded
 - Yes answer verified in a polynomial number of steps

Suppose answer yes for a given instance of traveling sales person

- You may guess all $(n-1)!$ tours before guessing the tour with the yes answer
- **Polynomial-time verifiability, not solvability**
- The guess for that tour is done in polynomial time
- The verification of the guess is done in polynomial time
- **Branch and bound probably produces a better solution**
- Purpose: problem classification

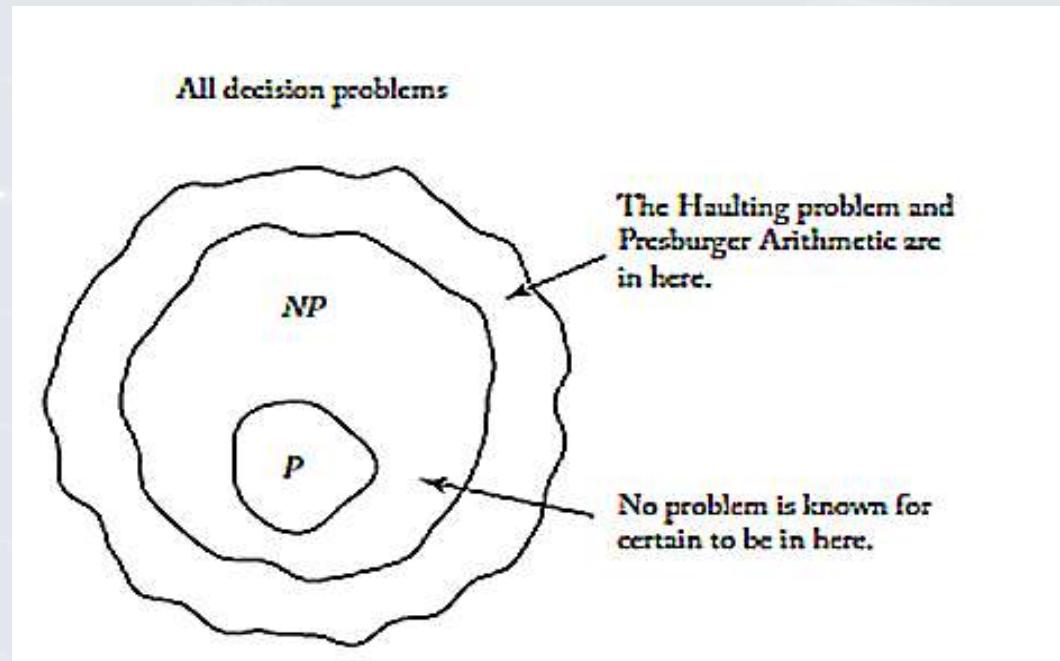


Is $P \subseteq NP$?

- It has not been proven that there is a problem in NP that is not in P
- $NP - P$ may be empty
- **P=NP? One of the most important questions in CS**
- To show $P \neq NP$, find a problem in NP that is not in P
- To show $P = NP$, find polynomial-time algorithm for each problem in NP



Figure 9.3





CNF

- **CNF – Conjunctive Normal Form**
- Logical (Boolean) variable: can have one of two values: TRUE or FALSE
- **Literal:** logical variable or the negation of a logical variable
 - x is a logical variable
 - $\neg x$ is the negation of a logical variable

CNF

- A **clause** is a disjunction of literals (e.g. $p \vee q \vee s$)
- A logical expression is in **Conjunctive Normal Form** if it is a conjunction of clauses
 - $(p \vee q \vee s) \wedge (\neg q \vee r) (\neg p \vee r) \wedge (\neg r \vee s) \wedge (\neg p \vee \neg s \vee \neg q)$

CNF - Satisfiability Decision Problem

- Is there a truth assignment for the variables of a CNF expression which evaluates to true?
- For the answer to be yes, each clause must evaluate to TRUE
- Assume n variables, 2^n rows in the truth table
- **Easy to write a polynomial-time algorithm takes as input a logical CNF expression and a set of truth assignments to the variables and verifies whether it evaluates to TRUE**

CNF Satisfiability Decision Problem

- Problem is in NP
- No polynomial-time algorithm for it has been found
- Cook proved that if CNF-Satisfiability is in P, P = NP

CNF Satisfiability

CNF Boolean Formula

- Variables, $\{x, y, z\}$ in ex.
- Literals, e.g. x, \bar{y}, \dots
- Clauses, disjunctions of literals
- CNF formula, a conjunction of clauses.

A formula is satisfiable if \exists a truth assignment such that the formula evaluates to true.

E.g. $(x \vee y) \wedge (\bar{y} \vee z)$

OR
↓
 $(x \vee y)$
↓ AND
 \wedge
 $(\bar{y} \vee z)$
↓ OR
literals
clauses
CNF formula



Polynomial-time Reducibility

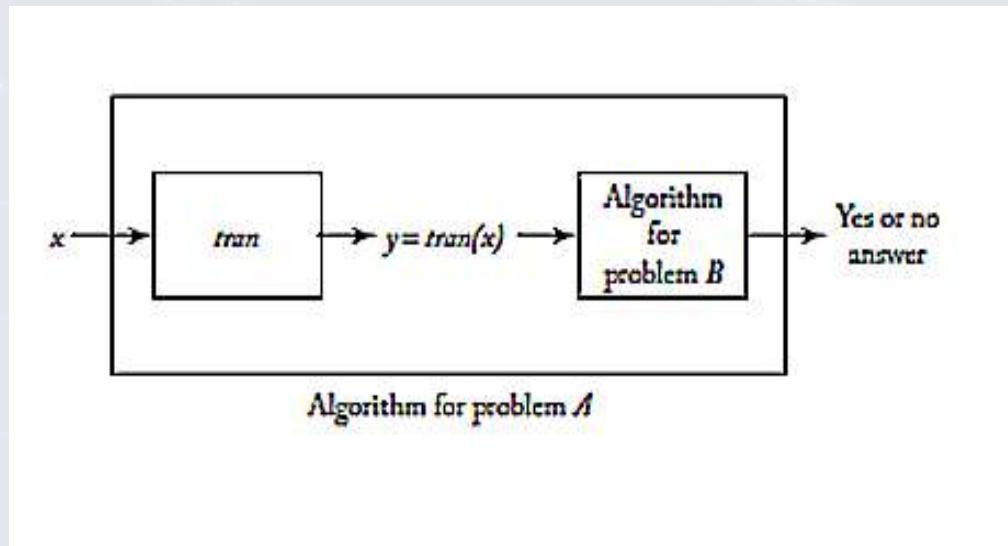
- Want to solve decision problem A
- Have an algorithm to solve decision problem B
- **Can write an algorithm that creates instance y of problem B from every instance x of problem A such that:**
 - Algorithm for B answers yes for y if the answer to problem A is yes for x



Polynomial-time Reducibility

- **Transformation algorithm**
 - Function that maps every instance of problem A to an instance of problem B
 - $y = \text{trans}(x)$
- Transformation algorithm + algorithm for problem B yields an algorithm for problem A

Figure 9.4



Polynomial-time many-one reducible

- If there exists a polynomial-time transformation algorithm from decision problem A to decision problem B, problem A is polynomial-time many-one reducible to problem B
- $A \leq^{\infty} B$
- **Many-one: transformation algorithm is a function that may map many instances of problem A to one instance of problem B**
- If the transformation algorithm is polynomial-time and the algorithm for problem B is polynomial, The algorithm for A must be polynomial



Theorem 9.1

- If decision problem B is in P and $A \propto B$, then decision problem A is in P

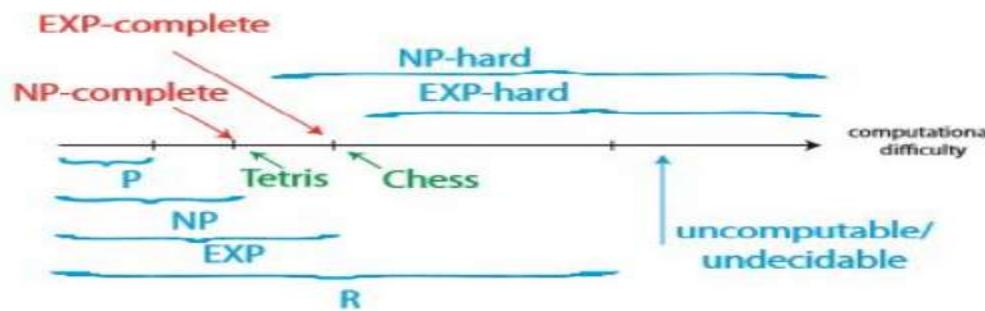
Proof

- Let p be the polynomial bound on the computation of the transformation algorithm T
- Let q be the polynomial bound on the algorithm M for B
- The size of $T(x)$ is at most $p(n)$ for input x of size n
- Algorithm M with input $T(x)$ does at most $q(p(n))$ steps
- The total work to transform x to $T(x)$ and then apply M to get the correct answer:
 - $p(n) + q(p(n))$, which is polynomial in n

NP-Complete

- A problem B is called NP-complete if both the following are true:
 1. B is in NP
 2. For every other problem A in NP, $A \leq^{\sim} B$

If $P \neq NP$



Definitions:

- $P = \{ \text{problems solvable in polynomial (nc) time} \}$ (what this class is all about)
- $EXP = \{ \text{problems solvable in exponential (2nc) time} \}$
- $R = \{ \text{problems solvable in finite time} \}$ "recursive" [Turing 1936; Church 1941]
- $NP = \{ \text{Decision problems solvable in polynomial time via a "lucky" algorithm} \}$.

In other words, $NP = \{ \text{decision problems with solutions that can be "checked" in polynomial time} \}$.

- $\text{NP-hard} = \text{"as hard as" every problem } \in \text{NP}$. In fact $\text{NP-complete} = \text{NP} \cap \text{NPhard}$.



Theorem 9.2 – Cook's Theorem

- CNF-Satisfiability is NP-complete.
- Proof found in Cook (1971) and in Gary and Johnson (1979)

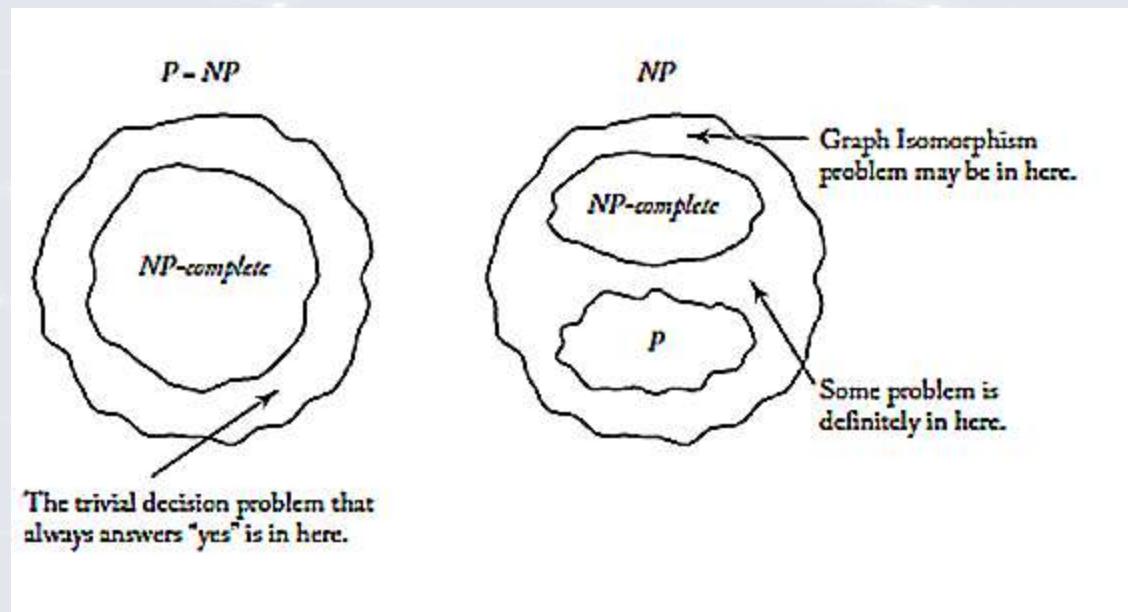


Theorem 9.3

- A problem C is NP-complete if both of the following are true:
 1. C is in NP
 2. For some other NP-complete problem B, $B \leq C$

Proof is based on the transitivity of reducibility ...

Figure 9.7



Analýza a Zložitosť Algoritmov

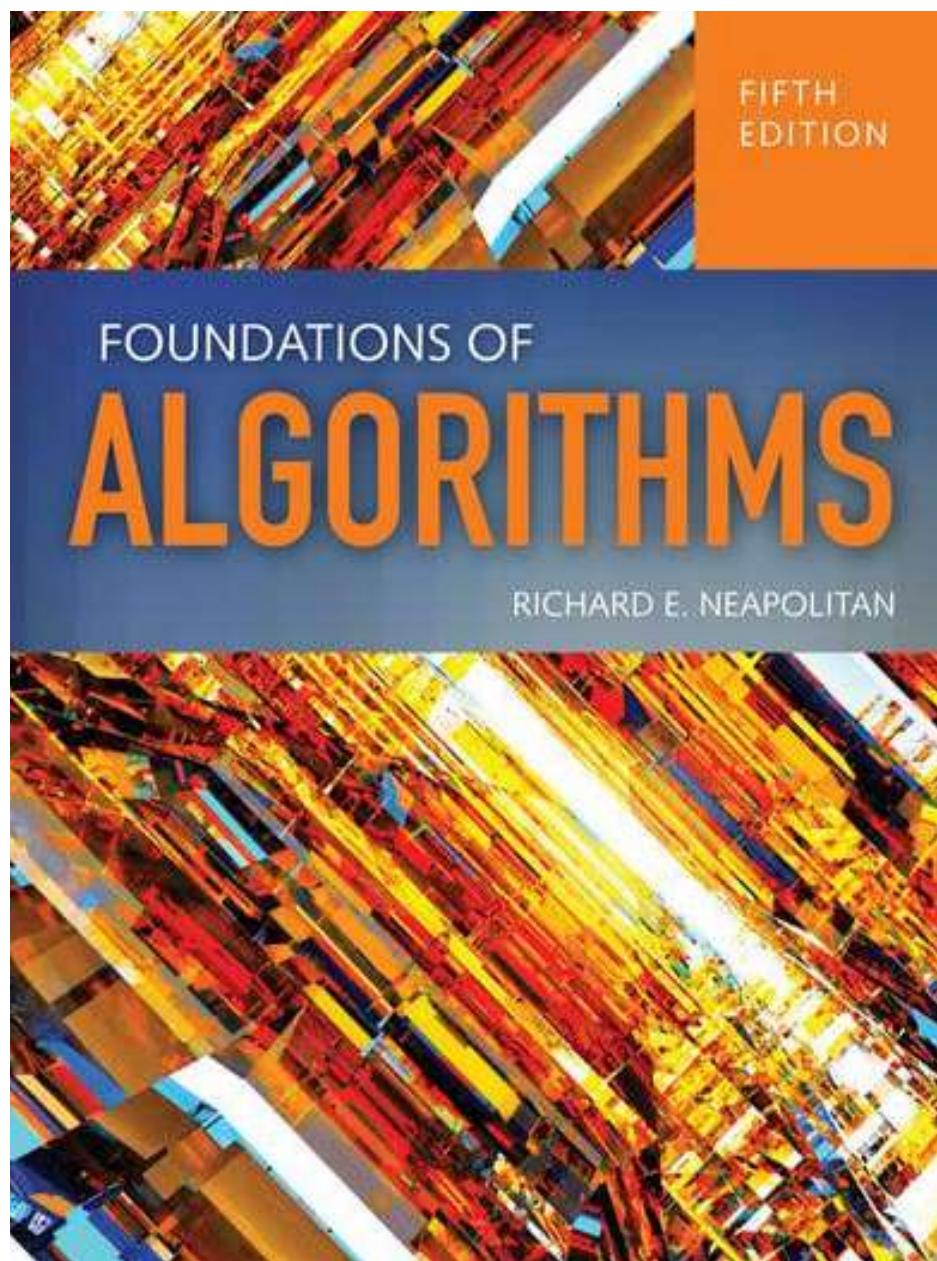
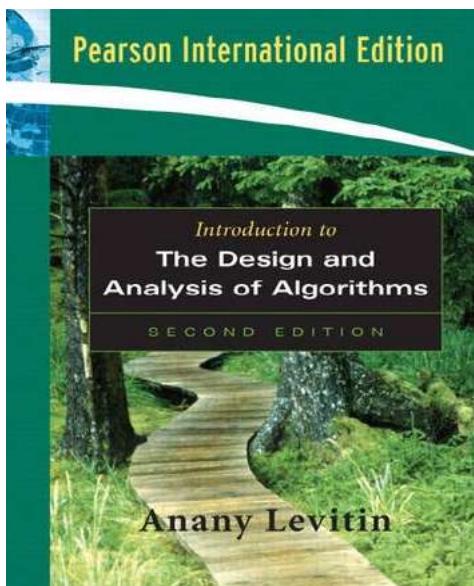
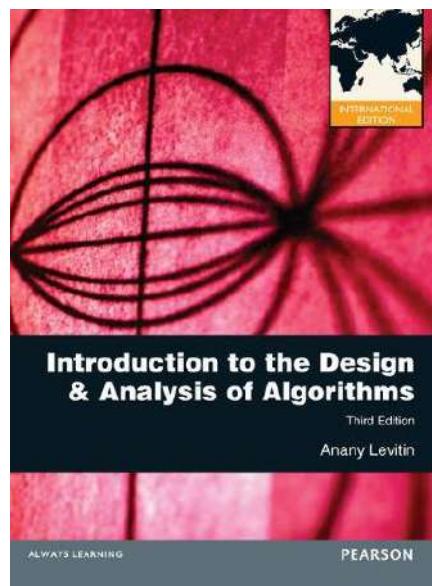
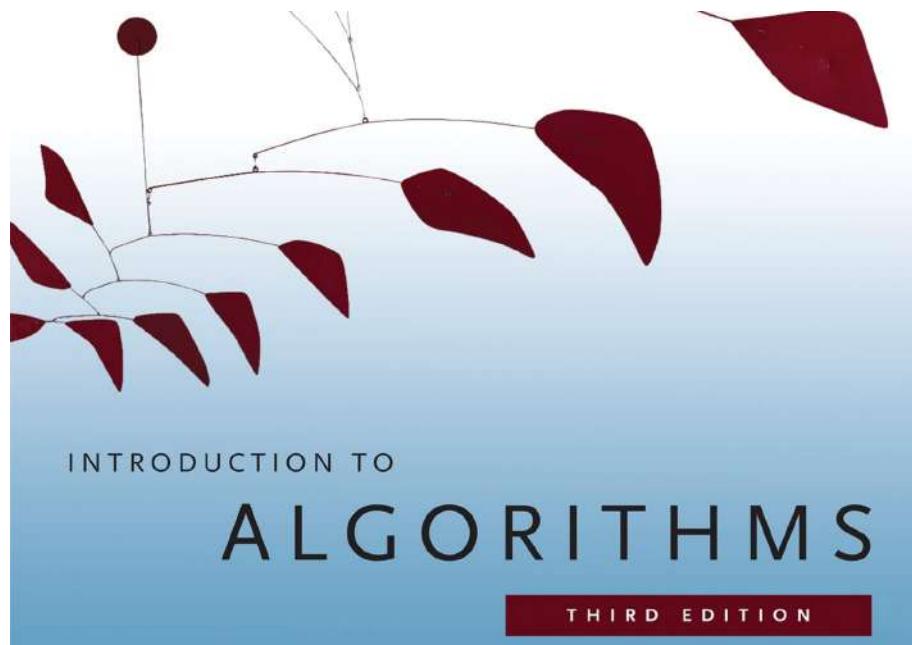
AZA

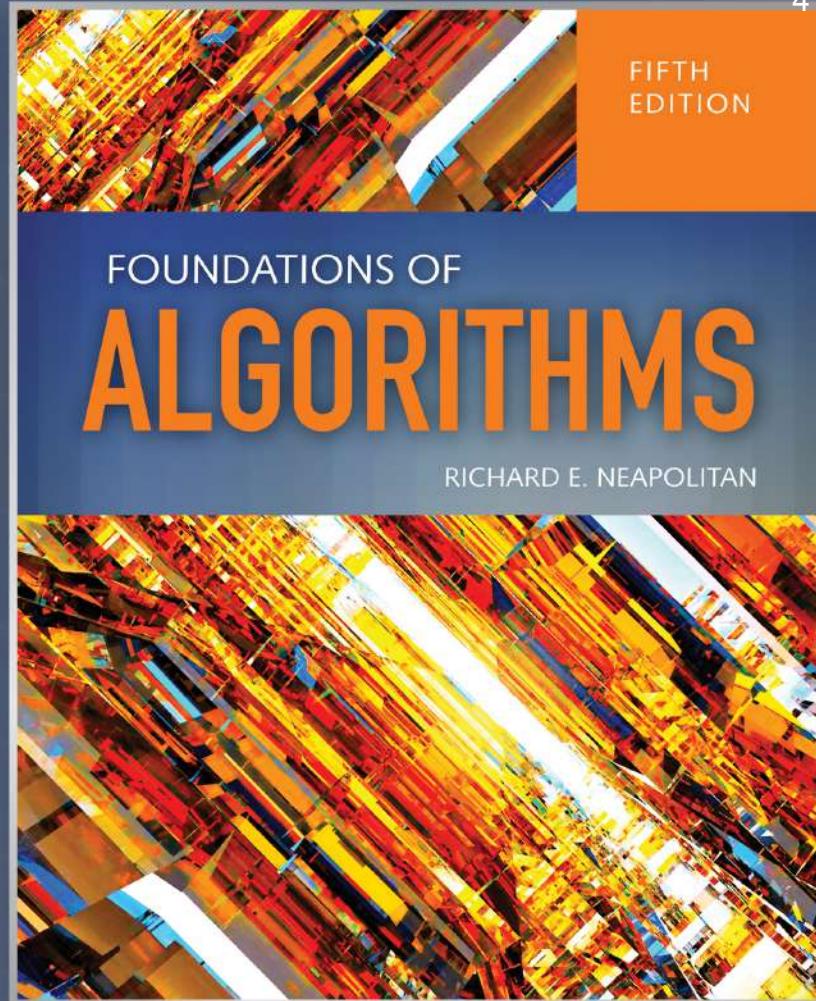
Doc. RNDr. Silvester Czanner, PhD
silvester.czanner@stuba.sk (subject AZA)

Semester Schedule II.

- 16/9
 - Session 1: Intro, Logistics, Algorithms: Efficiency, Analysis, and Order
 - Session 2: Divide-and-Conquer
- 30/09
 - Session 1: Dynamic programming,
 - Session 2: The Greedy approach
- 14/10
 - Session 1: Backtracking
 - Session 2: Branch-and-Bound
- **21/10**
 - Session 1: written test (1h) – material from first 5 labs
 - Session 2: written test (1h) – material from first 5 labs
- 04/11
 - Session 1: Computational Complexity: The Sorting Problem
 - Session 2: Computational Complexity: The Searching Problem
- 11/11
 - Session 1: An introduction to the theory of NP
 - Session 2: Limitations of Algorithm Power
- 18/11
 - Session 1: Wrap up, Q&A session
- Practical work submission deadline **06/12 5PM**

- 9/12
 - Session 1 – nahradná zápočtová písomka
 - Session 2 – Wrap – up, Q&A session





Algorithms: Efficiency, Analysis, and Order Chapter 1



Objectives

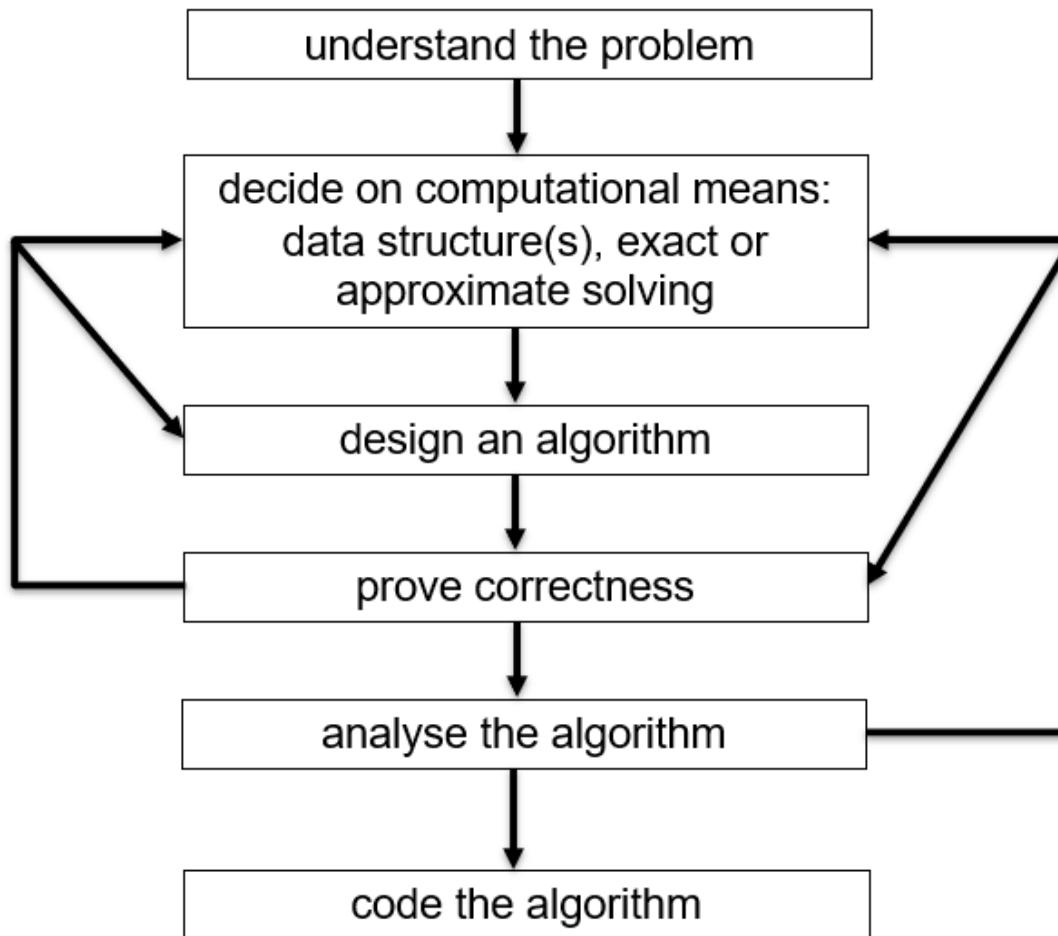
- Analyze techniques for solving problems
- Define an algorithm
- Define growth rate of an algorithm as a function of input size
- Define Worst case, average case, and best case complexity analysis of algorithms
- Classify functions based on growth rate
- Define growth rates: Big O, Theta, and Omega



Fibonacci: Iterative vs Recursive

- $\text{Fib}_0 = 0$
- $\text{Fib}_1 = 1$
- $\text{Fib}_n = \text{Fib}_{n-1} + \text{Fib}_{n-2}$
- Calculate the nth Fibonacci Term:
 - Recursive calculates $2^{n/2}$ terms
 - Iterative calculates $n+1$ terms

Algorithm Design and Analysis

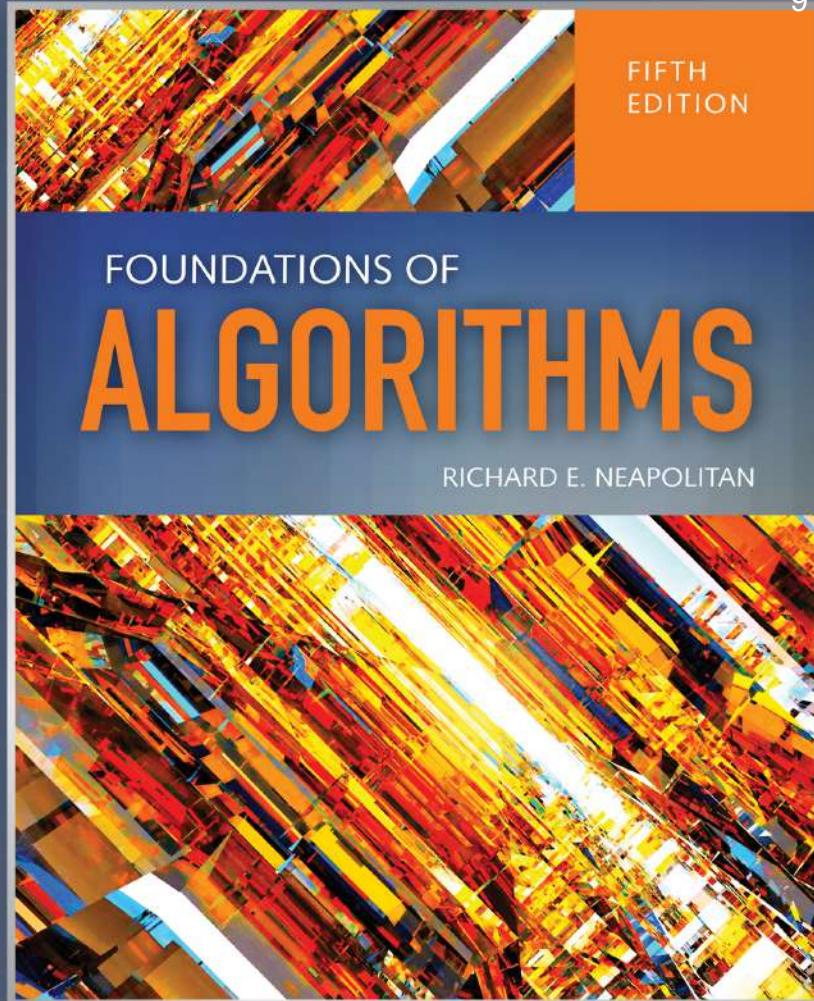


Recap

- Algorithms have different types of voracity for consuming time and memory.
- Assess algorithms with time and space complexity analysis.
- Calculate time complexity by finding the exact $T(n)$ function, the number of operations performed by an algorithm.
- Express time complexity using the Big-O notation.
- Perform simple time complexity analysis of algorithms using this notation.
- Calculating $T(n)$ is not necessary for inferring the Big-O complexity of an algorithm.
- The cost of running exponential algorithms explode and not runnable for big inputs.
- If an algorithm is too slow, would optimizing the algorithm or using a supercomputer help?

Divide and Conquer

Chapter 2





Objectives

- Describe the divide-and-conquer approach to solving problems
- Apply the divide-and-conquer approach to solve a problem
- Determine when the divide-and-conquer approach is an appropriate solution approach
- Determine complexity analysis of divide and conquer algorithms
- Contrast worst-case and average-case complexity analysis

Examples

- **Binary Search** is a searching algorithm. In each step, the algorithm compares the input element (x) with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if x is less than the middle element, then the algorithm recurs to the left side of the middle element, else it recurs to the right side of the middle element.
- **Quicksort** is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to the left side of the pivot, and all greater elements move to the right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.
- **Merge Sort** is a sorting algorithm. The algorithm divides the array into two halves, recursively sorts them, and finally merges the two sorted halves. The time complexity of this algorithm is $O(n\log n)$, be it best case, average case or worst case. Its time complexity can be easily understood from the recurrence equates to: $T(n) = 2T(n/2) + n$.
- **Strassen's Algorithm** is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops and is $O(n^3)$. Strassen's algorithm multiplies two matrices in $O(n^{2.8974})$ time.
- **Divide and Conquer should be used when same subproblems are not evaluated many times.**

Binary Search (Recursive)

Problem: Determine whether x is in the sorted array S of size n .

Inputs: positive integer n , sorted (nondecreasing order) array of keys S indexed from 1 to n , a key x .

Outputs: *location*, the location of x in S (0 if x is not in S).

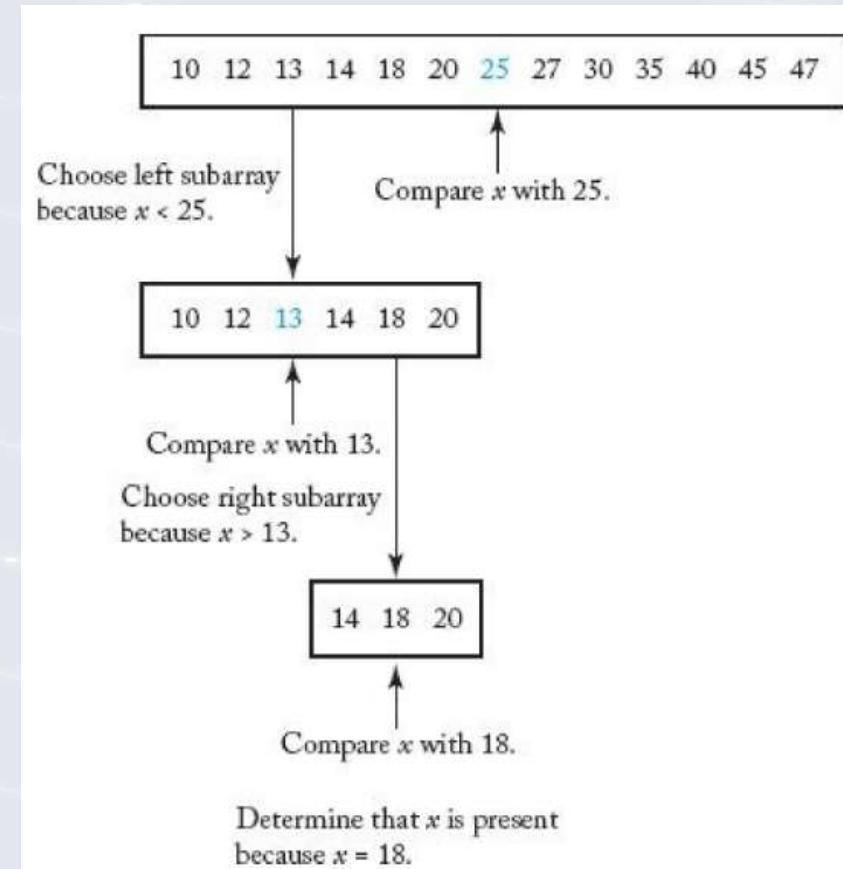
12

$x = 18$

```
index location (index low, index high)
{
    index mid;

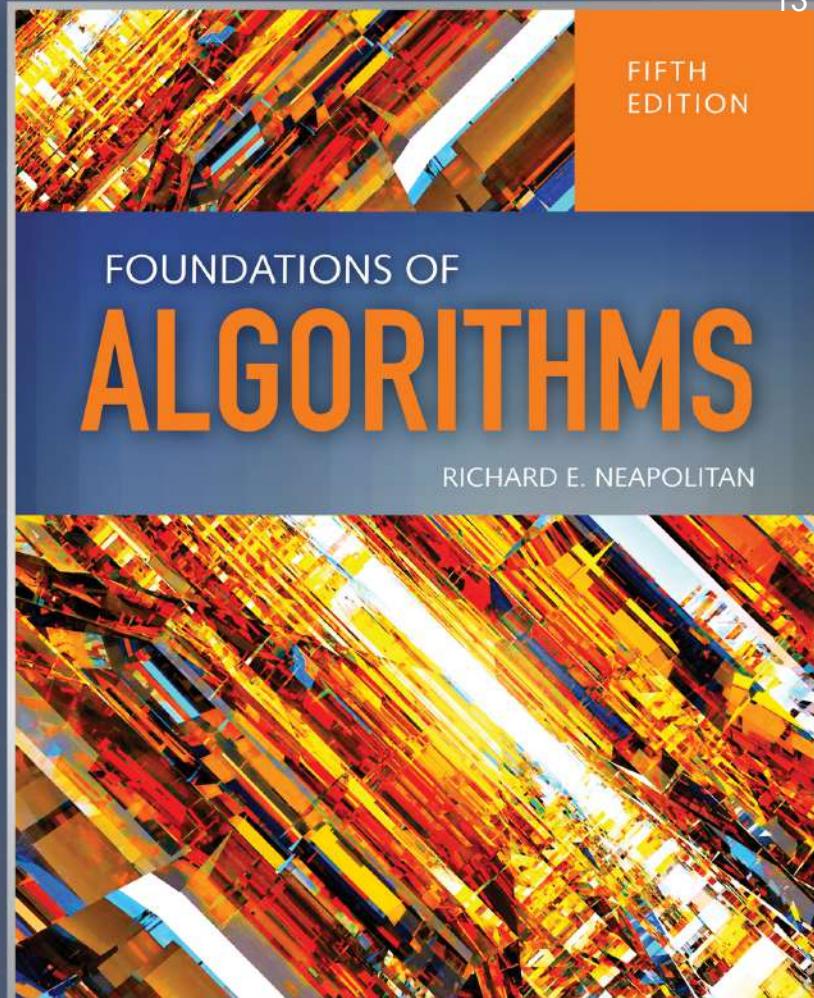
    if (low > high)
        return 0;
    else {
        mid = ⌊(low + high)/2⌋;
        if (x == S[mid])
            return mid
        else if (x < S[mid])
            return location(low, mid - 1);
        else
            return location(mid + 1, high);
    }
}
```

locationout = *location* (1 , n);



Dynamic Programming

Chapter 3





Objectives

- Describe the Dynamic Programming Technique
- Contrast the Divide and Conquer and Dynamic Programming approaches to solving problems
- Identify when dynamic programming should be used to solve a problem
- Define the Principle of Optimality
- Apply the Principle of Optimality to solve Optimization Problems

Steps to develop a dynamic programming algorithm

1. Establish a recursive property that gives the solution to an instance of the problem
2. Compute the value of an optimal solution in a bottom-up fashion by solving smaller instances first

Divide-and-Conquer

- It partitions the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
- A divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems.

Dynamic Programming

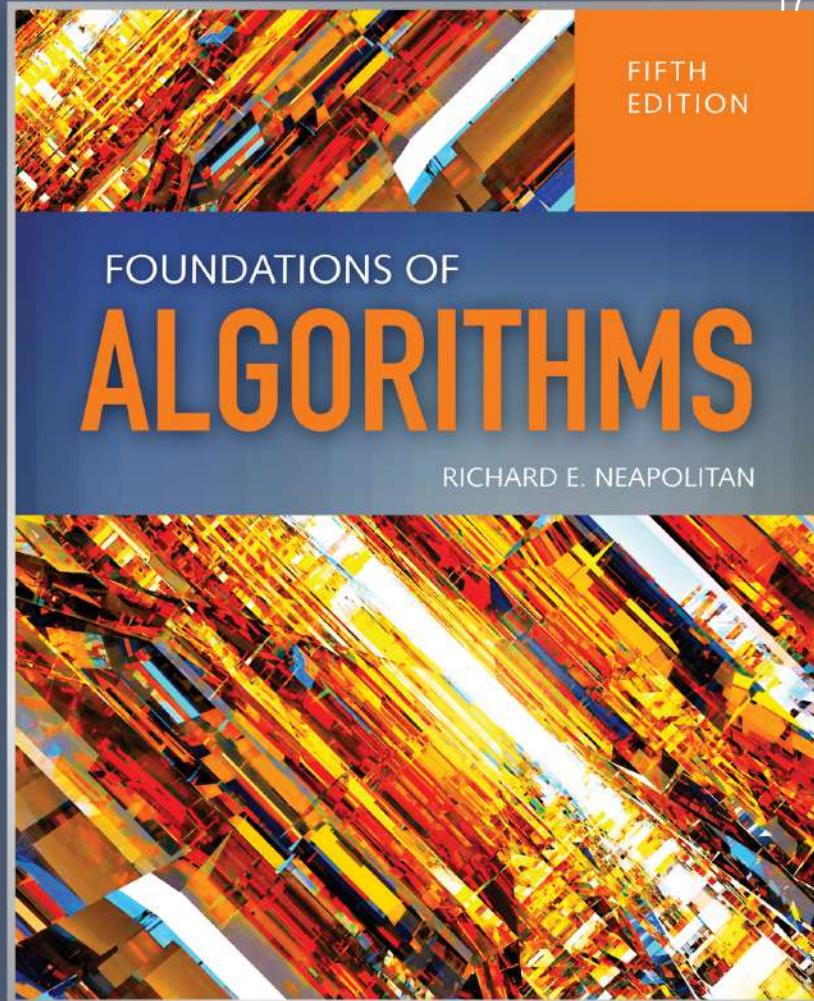
- Dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems.
- It solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

Optimization Problem

- Multiple candidate solutions
- Candidate solution has a value associated with it
- Solution to the instance is a candidate solution with an optimal value
- Minimum/Maximum

The Greedy Approach

Chapter 4



Objectives

- Describe the Greedy Programming Technique
- Contrast the Greedy and Dynamic Programming approaches to solving problems
- Identify when greedy programming should be used to solve a problem
- Prove/disprove greedy algorithm produces optimal solution
- Solve optimization problems using the greedy approach

Greedy Technique



Constructs a solution to an *optimization problem* piece by piece through a sequence of choices that are:

- *feasible*
- *locally optimal*
- *irrevocable*

For some problems, yields an optimal solution for every instance.
For most, does not but can be useful for fast approximations.

Applications of the Greedy Strategy



- ➊ Optimal solutions:

- change making for “normal” coin denominations
- minimum spanning tree (MST)
- single-source shortest paths
- simple scheduling problems
- Huffman codes

- ➋ Approximations:

- traveling salesman problem (TSP)
- knapsack problem
- other combinatorial optimization problems

Spanning Tree

- Assume: Connected, weighted, undirected graph G
- **Spanning tree** of a connected graph G: a connected acyclic subgraph of G that includes all of G's vertices
- **Minimum spanning tree (MST)** of a weighted, connected graph G: a spanning tree of G of minimum total weight

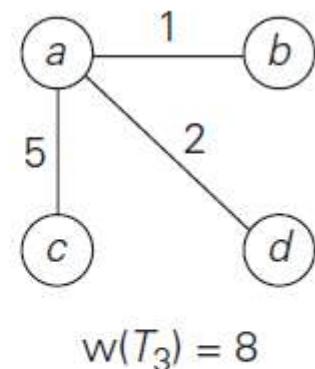
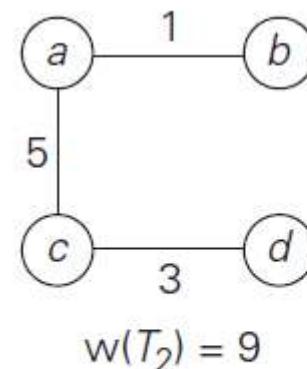
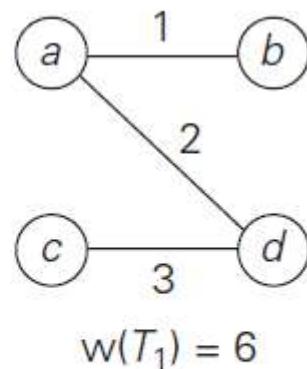
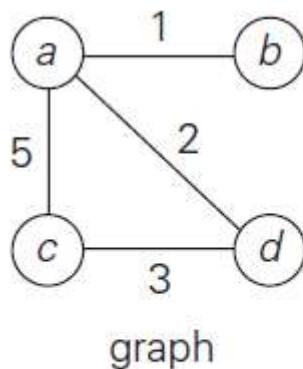
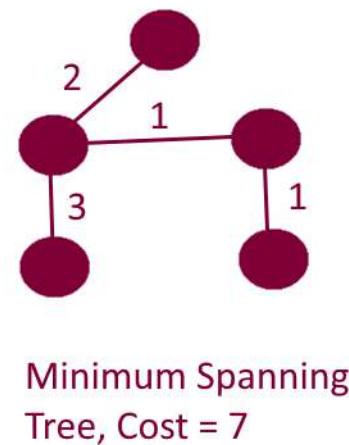
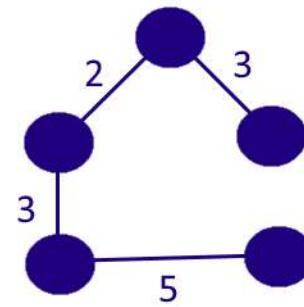
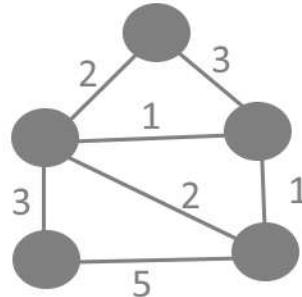


FIGURE 9.2 Graph and its spanning trees, with T_1 being the minimum spanning tree.

Minimum Spanning Tree for G

- Let $G = (V, E)$
- Let T be a spanning tree for G : $T = (V, F)$ where $F \subseteq E$
- Find T such that the sum of the weights of the edges in F is minimal



PRIMS ALGORITHM VERSUS KRUSHAL ALGORITHM

PRIMS ALGORITHM

A greedy algorithm that finds a minimum spanning tree for a weighted undirected graph

Generates the minimum spanning tree starting from the root vertex

Selects the root vertex

Selects the shortest edge connected to the root vertex

KRUSHAL ALGORITHM

A minimum spanning tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest

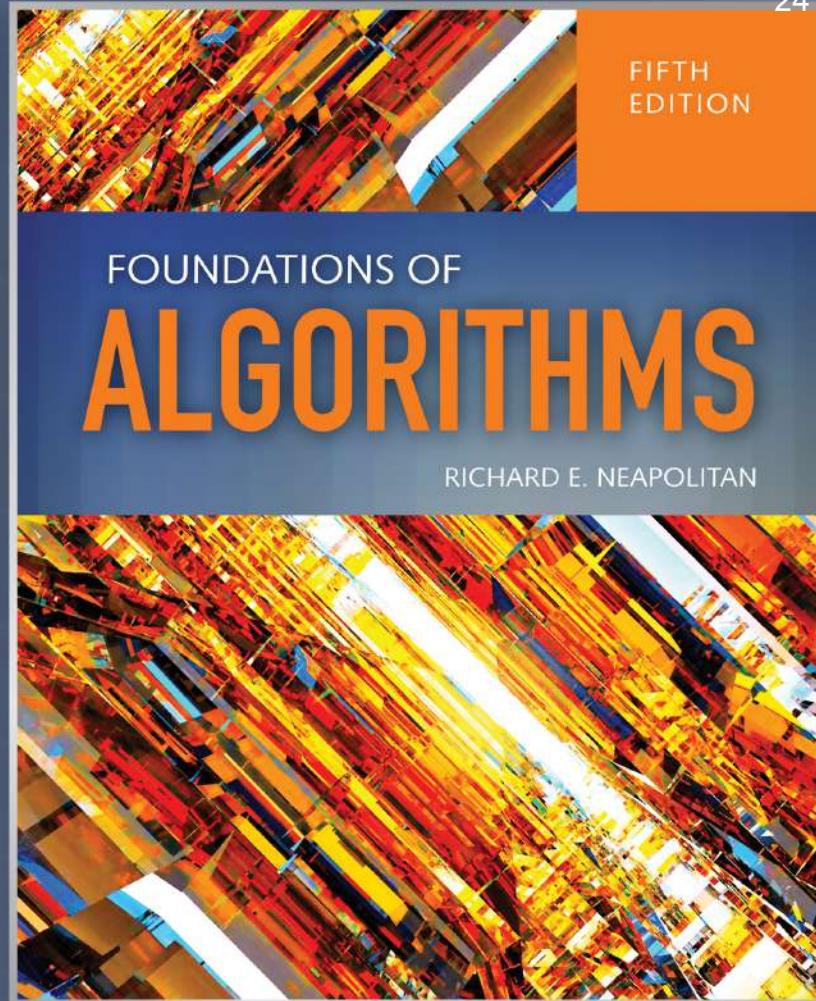
Generates the minimum spanning tree starting from the least weighted edge

Selects the shortest edge

Selects the next shortest edge

Backtracking

Chapter 5



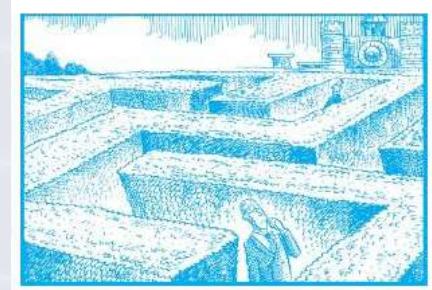


Objectives

- Describe the backtrack programming technique
- Determine when the backtracking technique is an appropriate approach to solving a problem
- Define a **state space tree** for a given problem
- Define when a node in a state space tree for a given problem is **promising/non-promising**
- Create an algorithm to **prune** a state space tree
- Create an algorithm to apply the backtracking technique to solve a given problem

Finding Your Way Thru a Maze

- Follow a path until a dead end is reached
- Go back until reaching a fork
- Pursue another path
- Suppose there were signs indicating path leads to dead end?
- Sign positioned near beginning of path – time savings enormous
- Sign positioned near end of path – very little time saved





N-Queen Problem

- Goal: **position n queens on a n x n board such that no two queens threaten each other**
 - No two queens may be in the same row, column, or diagonal
- **Sequence:** n positions where queens are placed
- **Set:** n^2 positions on the board
- **Criterion:** no two queens threaten each other



Sum-of-Subsets Problem

- Let $S = \{s_1, s_2, \dots, s_n\}$
- Let W be a positive integer
- Find every $S' \subseteq S$ such that

$$\sum_{s \in S'} s = W$$

$s \in S'$

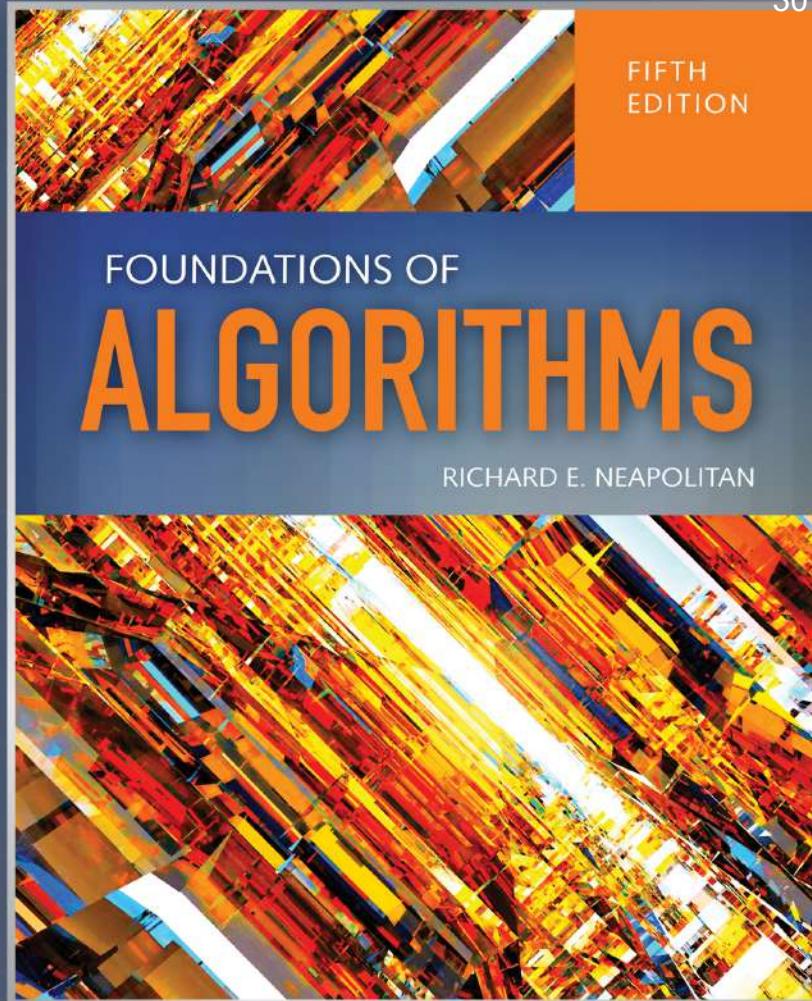
Example

- $S = \{w_1 = 5, w_2 = 6, w_3 = 10, w_4 = 11, w_5 = 16\}$ and $W=21$
- Solutions:
 - $\{w_1, w_2, w_3\} : 5 + 6 + 10 = 21$
 - $\{w_1, w_5\} : 5 + 16 = 21$
 - $\{w_3, w_4\} : 10 + 11 = 21$

the solutions are $\{w_1, w_2, w_3\}$, $\{w_1, w_5\}$, and $\{w_3, w_4\}$.

Branch-and-Bound

Chapter 6

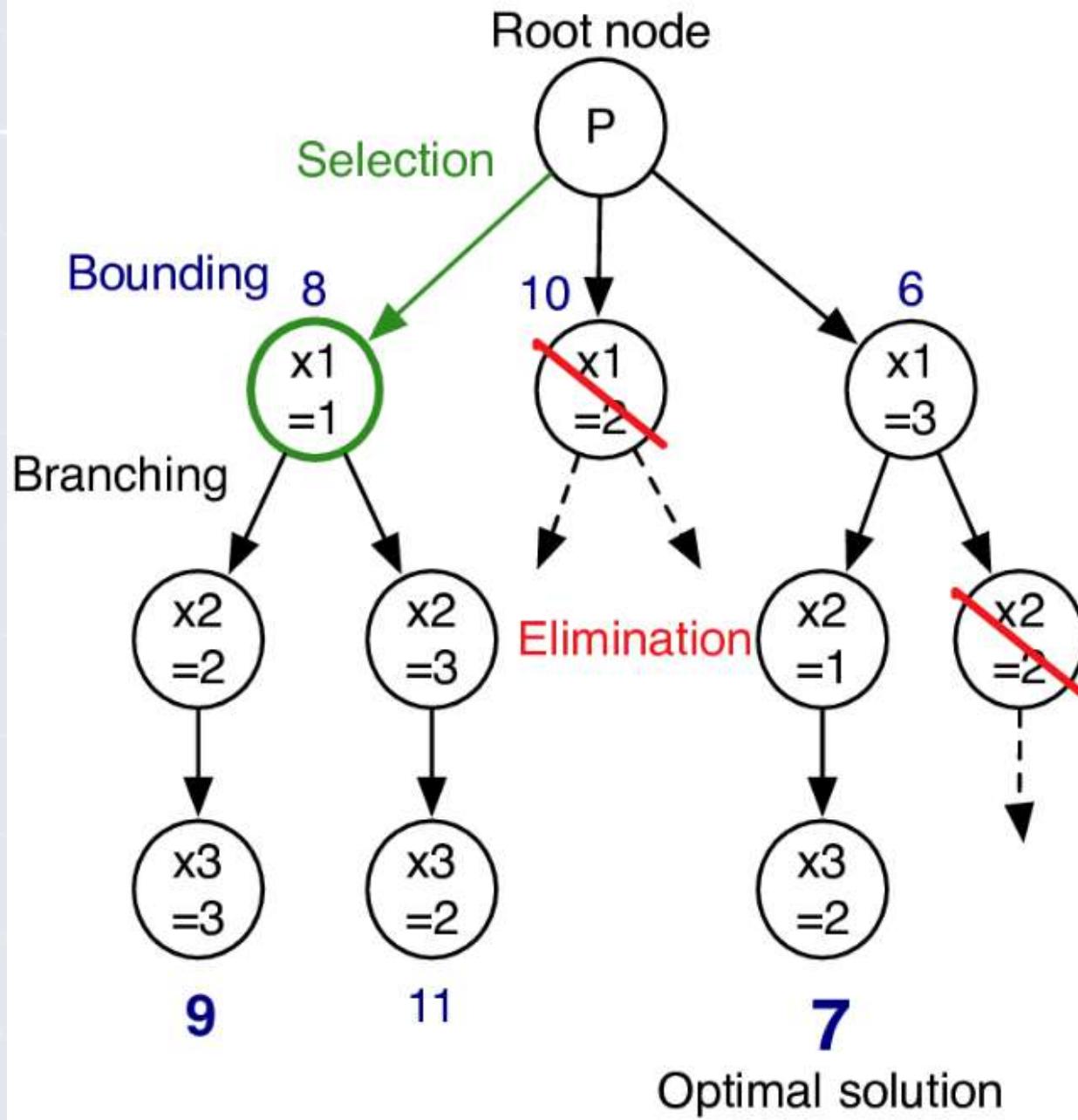


Objectives

- Describe the branch-and-bound technique for solving optimization problems
- Contrast the branch-and-bound technique for solving problems with the backtracking technique
- Identify when it is appropriate to use the branch-and-bound technique to solve a particular problem.
- Apply the branch-and bound technique to solve the N-Queen Problem

Branch-and-Bound Design Strategy

- Branch-and-bound design strategy is **similar to backtracking**
- **State space tree** used to solve problem
- Difference between branch-and-bound and backtracking:
 - 1. branch-and-bound is **not limited to a particular tree traversal**
 - 2. branch-and-bound is used **only for optimization problems**



Lower Bounds



Lower bound: an estimate on a minimum amount of work needed to solve a given problem

Examples:

- number of comparisons needed to find the largest element in a set of n numbers
- number of comparisons needed to sort an array of size n
- number of comparisons necessary for searching in a sorted array
- number of multiplications needed to multiply two n -by- n matrices

Lower Bounds (cont.)



- Lower bound can be
 - an exact count
 - an efficiency class (Ω)
- Tight lower bound: there exists an algorithm with the same efficiency as the lower bound

Problem	Lower bound	Tightness
sorting	$\Omega(n \log n)$	yes
searching in a sorted array	$\Omega(\log n)$	yes
element uniqueness	$\Omega(n \log n)$	yes
n -digit integer multiplication	$\Omega(n)$	unknown
multiplication of n -by- n matrices	$\Omega(n^2)$	unknown

Methods for Establishing Lower Bounds



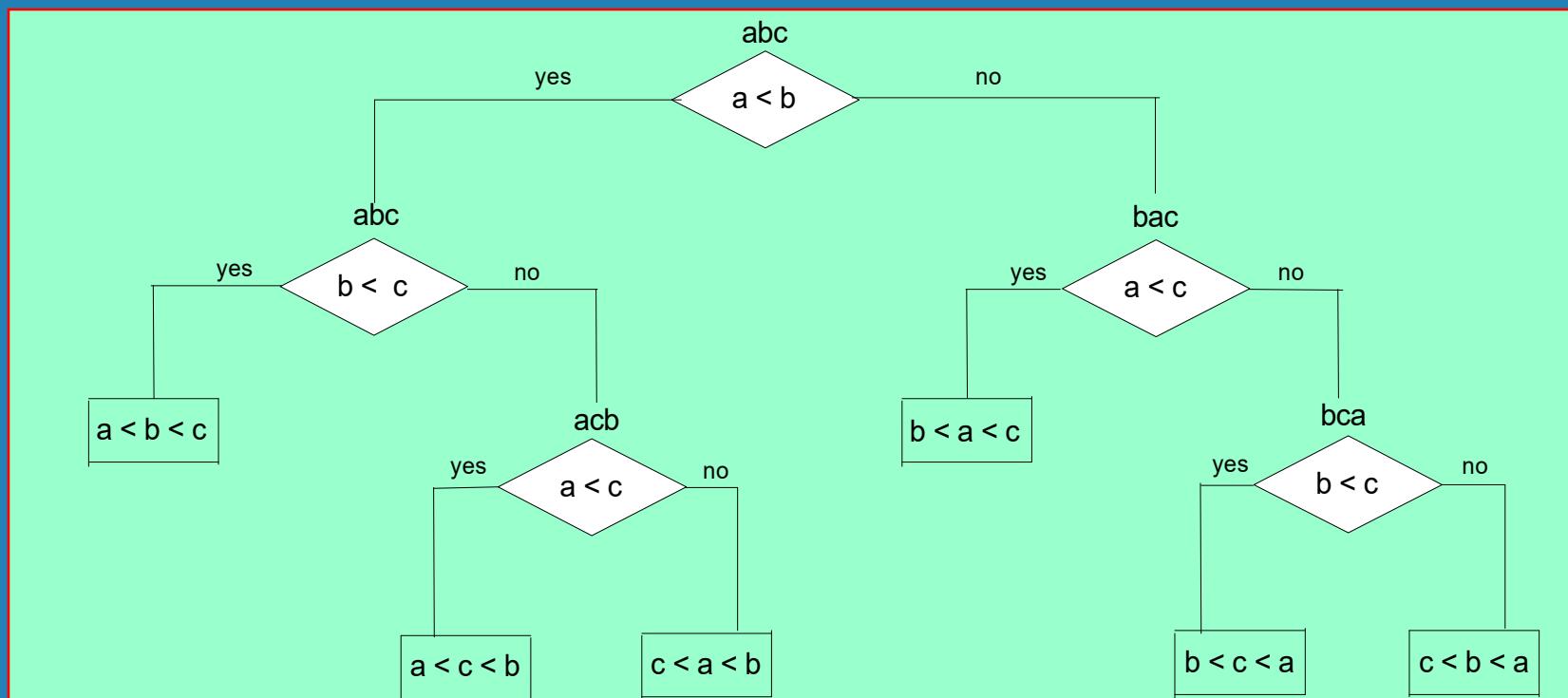
- ➊ trivial lower bounds
- ➋ information-theoretic arguments (decision trees)
- ➌ adversary arguments
- ➍ problem reduction

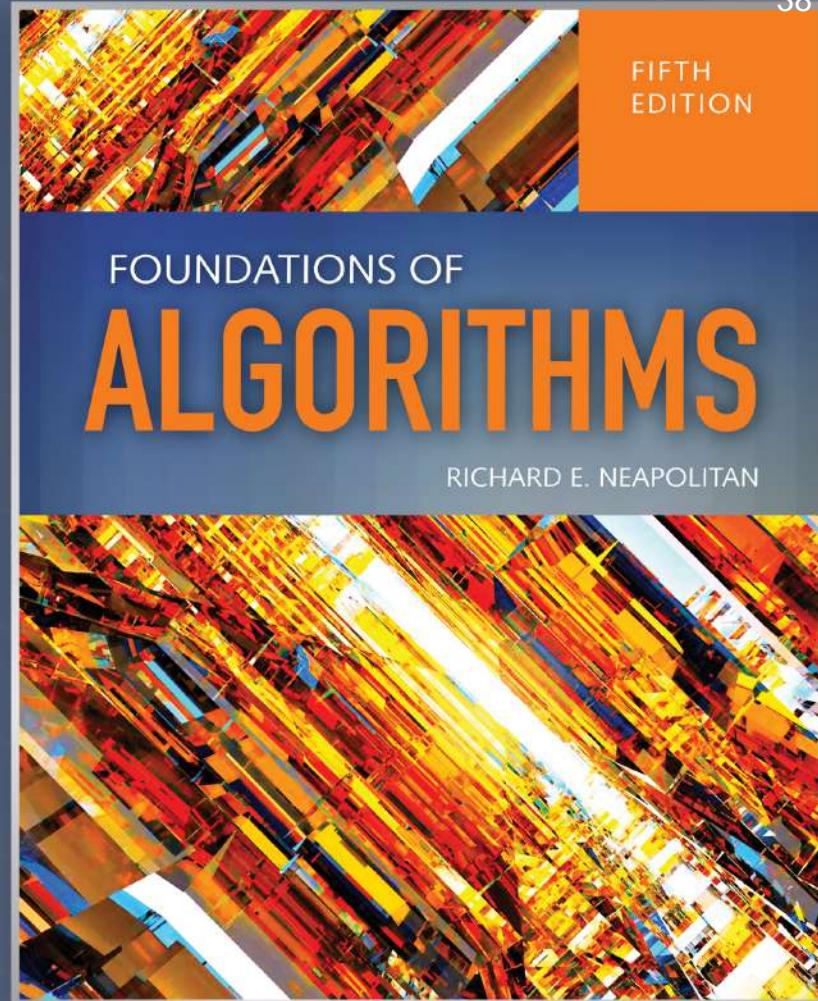
Decision Trees

Decision tree — a convenient model of algorithms involving comparisons in which:

- internal nodes represent comparisons
- leaves represent outcomes

Decision tree for 3-element insertion sort





Introduction to Computational Complexity: The Sorting Problem

Chapter 7

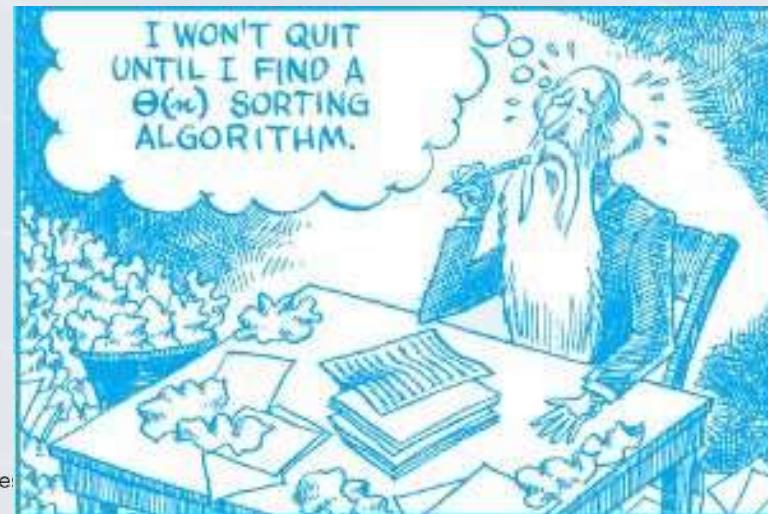
Objectives

- Use computational complexity analysis to determine a lower bounds on sorting algorithms
- Analyze algorithms that sort only by comparison of keys
- Use computational complexity analysis to determine a lower bounds of quadratic time on a the class of sorting algorithms that remove one inversion per comparison
- Analyze the class of $\Theta(n \lg n)$ sorting algorithms

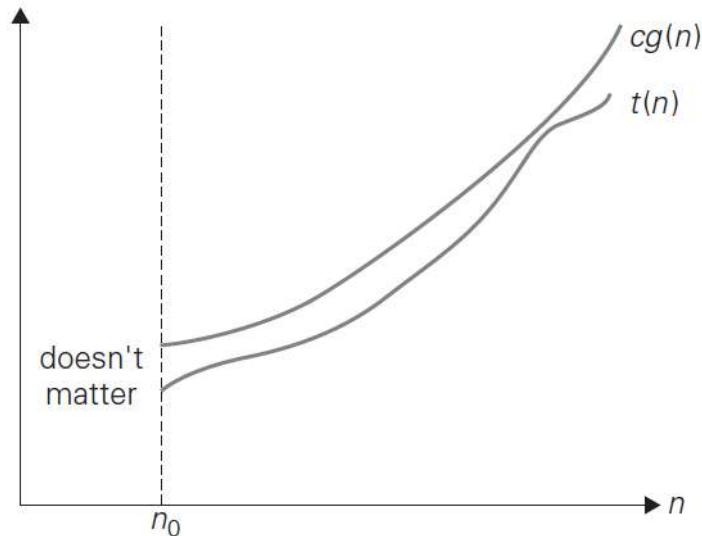


Computational Complexity

- Study of all possible algorithms that solve a given problem
- Determine a lower bound on efficiency of all algorithms for a given problem
- Problem analysis as opposed to algorithm analysis



O-notation



DEFINITION A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n , i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

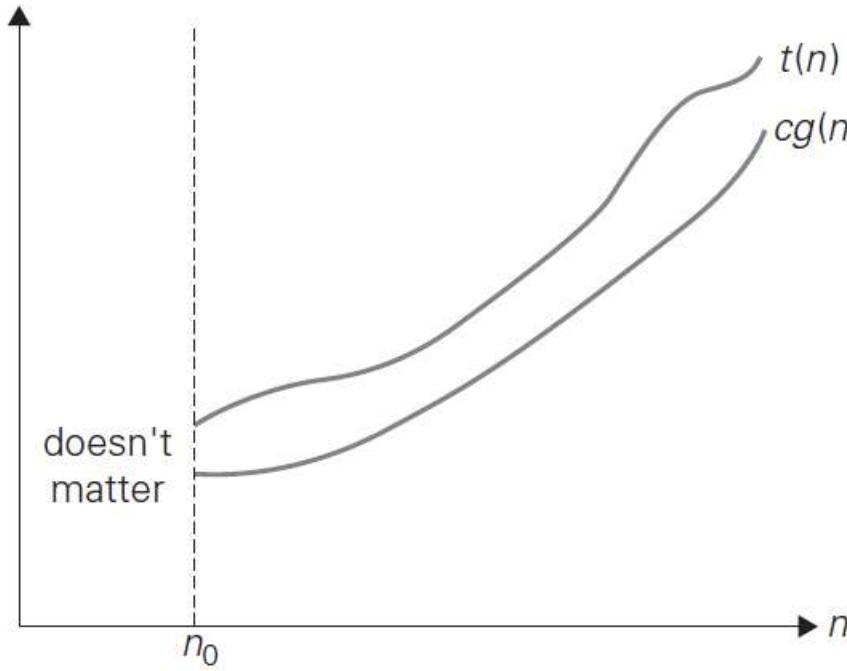
$$t(n) \leq cg(n) \quad \text{for all } n \geq n_0.$$

As an example, let us formally prove one of the assertions made in the introduction: $100n + 5 \in O(n^2)$. Indeed,

$$100n + 5 \leq 100n + n \text{ (for all } n \geq 5\text{)} = 101n \leq 101n^2.$$

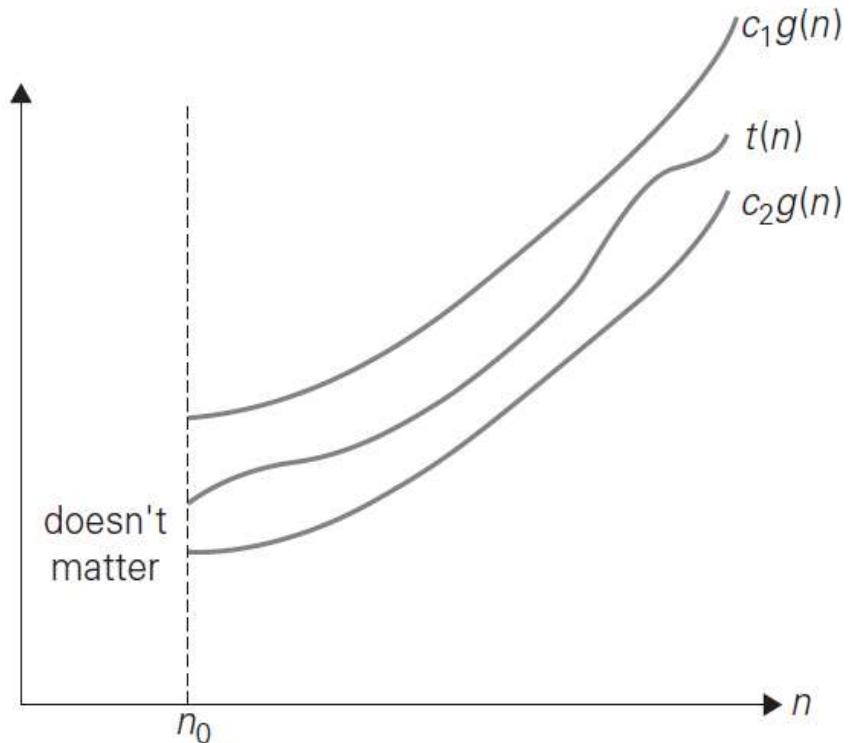
Thus, as values of the constants c and n_0 required by the definition, we can take 101 and 5, respectively.

Ω and Θ notations



$$t(n) \in \Omega(g(n)).$$

$$t(n) \geq cg(n) \quad \text{for all } n \geq n_0.$$



$$t(n) \in \Theta(g(n)).$$

$$c_2g(n) \leq t(n) \leq c_1g(n) \quad \text{for all } n \geq n_0.$$



Sort

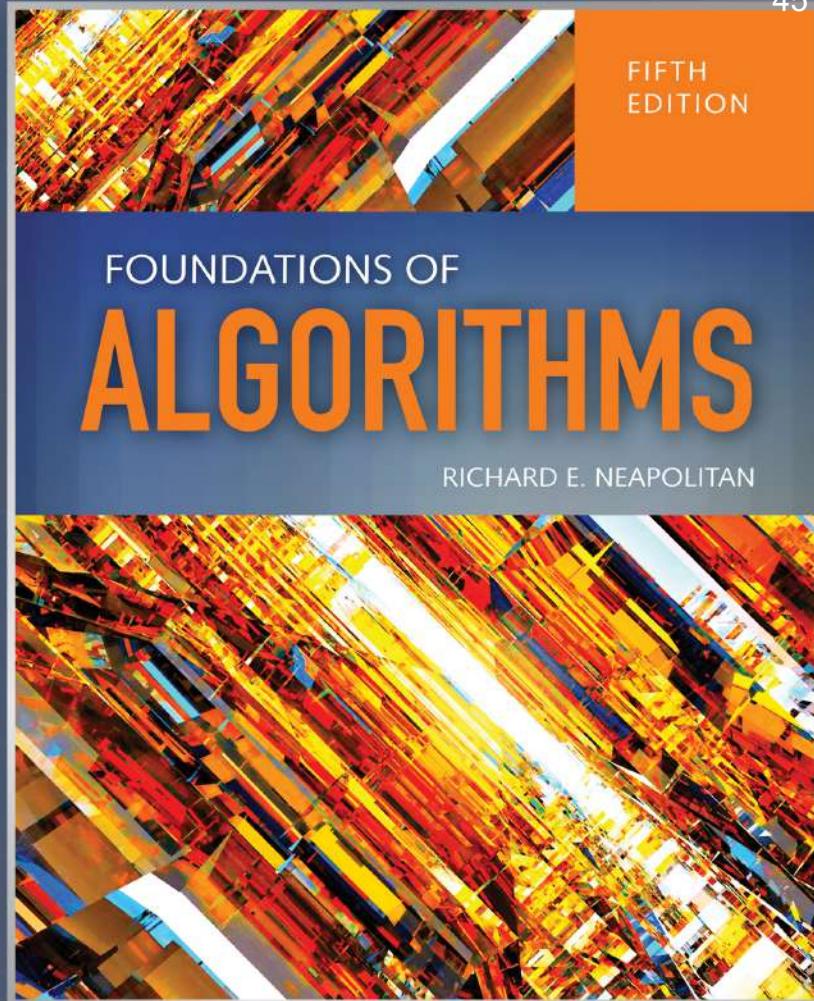
- Re-arrange records according to a key field
- Algorithms that sort by comparison of keys can **compare 2 keys to determine which is larger** and can copy keys
- **Cannot do other operations on them**

Examples

- **Selection sort:** compare elements to place the minimum elements to the front positions
- **Insertion sort:** compare elements to decide the position of an element in the partially sorted array
- **Merge Sort:** compare elements of two sorted elements to merge them into the final sorted array
- **Quicksort:** compare elements of partition the unsorted array two different halves around the pivot value.
- **Heapsort:** compare elements during the heapify process to place the minimum elements to the front of the array (If we are using the min-heap).

More Computational Complexity: The Searching Problem

Chapter 8





Objectives

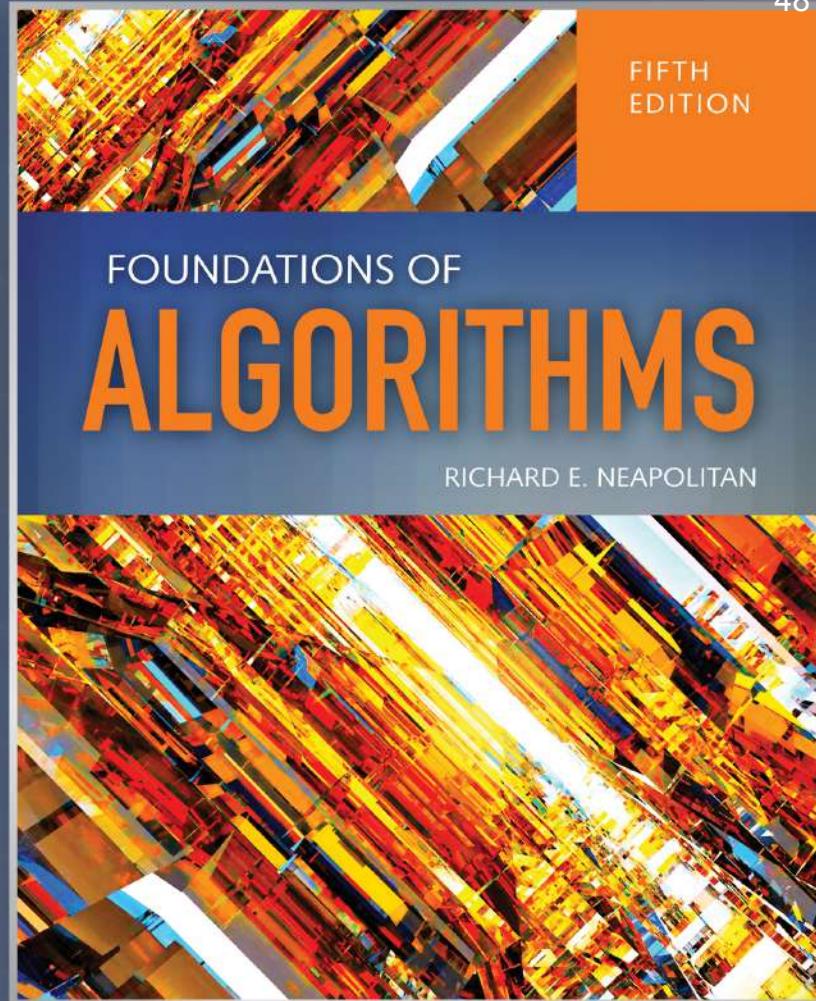
- Establish a lower bound on searching by comparison of keys of n distinct keys for key x .
- Prove binary search is optimal
- Apply interpolation search to evenly distributed data
- Differentiate between static and dynamic searching
- Establish the average binary search tree search time

Examples

- **Binary Search** is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to $O(\log n)$.
- **Interpolation Search** is an improvement over Binary Search for instances, where the values in a sorted array are uniformly distributed. Interpolation constructs new data points within the range of a discrete set of known data points. Binary Search always goes to the middle element to check. On the other hand, interpolation search may go to different locations according to the value of the key being searched.
- **Search by hashing** (hash table, slot, hash function, collision), avoid unnecessary comparisons.

Computational Complexity and Intractability: An Introduction to the Theory of NP

Chapter 9





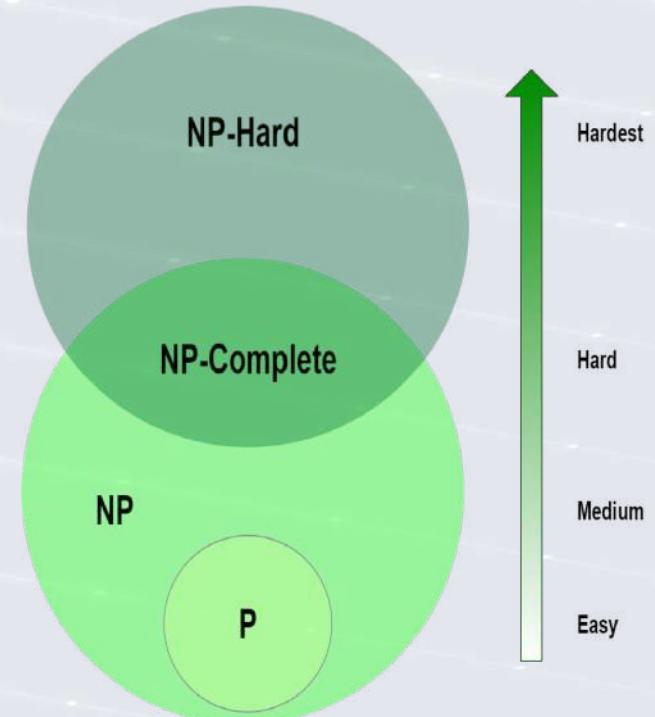
Objectives

- Classify problems as tractable or intractable
- Define decision problems
- **Define the class P**
- Define nondeterministic algorithms
- **Define the class NP**
- Define polynomial transformations
- **Define the class of NP-Complete**



Define

- Decision problems
- **The class P**
- Nondeterministic algorithms
- **The class NP**
- Polynomial transformations
- **The class of NP-Complete**





Class P

- **The set of all decision problems that can be solved by polynomial-time algorithms**
- Decision versions of searching, shortest path, spanning tree, etc. belong to P
- Do problems such as traveling salesperson and 0-1 Knapsack (no polynomial-time algorithm has been found), etc., belong to P?
 - No one knows
 - To know a decision problem is not in P, it must be proven it is not possible to develop a polynomial-time algorithm to solve it



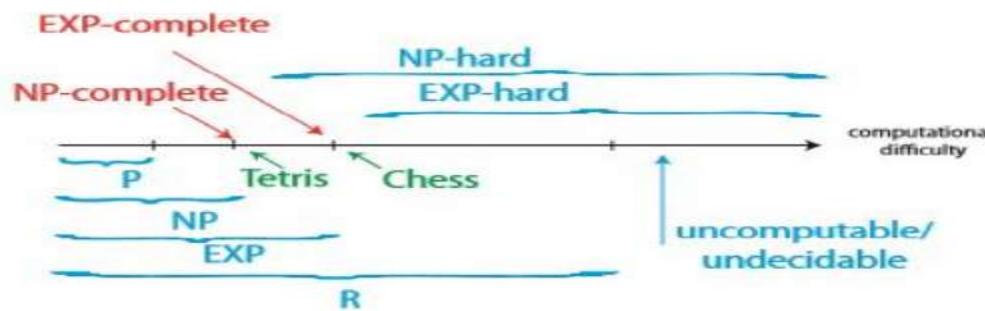
Class NP

- **The set of all decision problems that can be solved by polynomial-time nondeterministic algorithms**
- Nondeterministic polynomial
- For a problem to be in NP, there must be an algorithm that does the verification in polynomial time
- **Traveling salesperson decision problem belongs to NP**
 - Show a guess, s , length polynomial bounded
 - Yes answer verified in a polynomial number of steps

NP-Complete

- A problem B is called NP-complete if both the following are true:
 1. B is in NP
 2. For every other problem A in NP, $A \leq^{\sim} B$

If $P \neq NP$



Definitions:

- $P = \{\text{problems solvable in polynomial (nc) time}\}$ (what this class is all about)
- $EXP = \{\text{problems solvable in exponential (2nc) time}\}$
- $R = \{\text{problems solvable in finite time}\}$ “recursive” [Turing 1936; Church 1941]
- $NP = \{\text{Decision problems solvable in polynomial time via a ‘lucky’ algorithm}\}$.

In other words, $NP = \{\text{decision problems with solutions that can be ‘checked’ in polynomial time}\}$.

- $\text{NP-hard} = \text{“as hard as” every problem } \in \text{NP}$. In fact $\text{NP-complete} = \text{NP} \cap \text{NPhard}$.



Space and Time Tradeoffs

A. Levitin "Introduction to the Design & Analysis of Algorithms," 3rd ed., Ch. 10 ©2012 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.



Space-for-time tradeoffs



Two varieties of space-for-time algorithms:

- ➊ *input enhancement* — preprocess the input (or its part) to store some info to be used later in solving the problem
 - counting sorts
 - string searching algorithms
- ➋ *prestructuring* — preprocess the input to make accessing its elements easier
 - hashing
 - indexing schemes (e.g., B-trees)

Review: String searching by brute force



pattern: a string of m characters to search for

text: a (long) string of n characters to search in

Brute force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until either all characters are found to match (successful search) or a mismatch is detected

Step 3 While a mismatch is detected and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

String searching by preprocessing



Several string searching algorithms are based on the input enhancement idea of preprocessing the pattern

- Knuth-Morris-Pratt (KMP) algorithm preprocesses pattern left to right to get useful information for later searching
- Boyer -Moore algorithm preprocesses pattern right to left and store information into two tables
- Horspool's algorithm simplifies the Boyer-Moore algorithm by using just one table

Iterative Improvement



Algorithm design technique for solving optimization problems

- Start with a feasible solution
- Repeat the following step until no improvement can be found:
 - change the current feasible solution to a feasible solution with a better value of the objective function
- Return the last feasible solution as optimal

Note: Typically, a change in a current solution is “small” (local search)

Major difficulty: Local optimum vs. global optimum

Important Examples



- ➊ **simplex method**
- ➋ **Ford-Fulkerson algorithm for maximum flow problem**
- ➌ **maximum matching of graph vertices**
- ➍ **Gale-Shapley algorithm for the stable marriage problem**

- ➎ **local search heuristics**

Skúška - Exam

Pondelok 13 Januar 2025 – Veľká Aula

prvý beh – 8:00 -10:00

druhý beh – 10:30 – 12:30

Náhradná /opravná skúška

- Menej ako 22 bodov zo skúšky
- Ospravedlnenie od lekára

Pondelok 3 Február 2025 - Veľká aula

prvý beh – 8:00 – 10:00

Dovolený „ťahák“ = A4 popísaná perom z oboch strán (nie vytlačené)