ITS033 – Programming & Algorithms

Lecture 04 Brute Force Algorithms

http://www.vcharkarn.com/vlesson/showlesson.php?lessonid=7&pageid=5

Asst. Prof. Dr. Bunyarit Uyyanonvara
IT Program, Image and Vision Computing Lab.
School of Information, Computer and Communication Technology (ICT)
Sirindhorn International Institute of Technology (SIIT)
Thammasat University
http://www.siit.tu.ac.th/bunyarit
bunyarit@siit.tu.ac.th
02 5013505 X 2005

ITS033

Topic 01 - Problems & Algorithmic Problem Solving
Topic 02 – Algorithm Representation & Efficiency Analysis
Topic 03 - State Space of a problem
Topic 04 - Brute Force Algorithm
Topic 05 - Divide and Conquer
Topic 06 - Decrease and Conquer
Topic 07 - Dynamics Programming
Topic 08 - Transform and Conquer
Topic 09 - Graph Algorithms
Topic 10 - Minimum Spanning Tree
Topic 11 - Shortest Path Problem
Topic 12 - Coping with the Limitations of Algorithms Power

http://www.siit.tu.ac.th/bunyarit/its033.php
and http://www.vcharkarn.com/vlesson/showlesson.php?lessonid=7

This Week Overview

- **Brute Force Introduction**
- **Sorting**
- **Sequential search**
- **Exhaustive Search**
- **Problems**

ITS033 – Programming & Algorithms

Lecture 04.1 Brute Force: Introduction

http://www.vcharkarn.com/vlesson/showlesson.php?lessonid=7&pageid=5

Asst. Prof. Dr. Bunyarit Uyyanonvara
IT Program, Image and Vision Computing Lab.
School of Information, Computer and Communication Technology (ICT)
Sirindhorn International Institute of Technology (SIIT)
Thammasat University
http://www.siit.tu.ac.th/bunyarit
bunyarit@siit.tu.ac.th
02 5013505 X 2005

Introduction

- ***Brute force*** is a straightforward approach to problem solving, usually directly based on the problem's statement and definitions of the concepts involved.
- Though rarely a source of clever or efficient algorithms,
- the brute-force approach should not be overlooked as an important algorithm design strategy.

Introduction

- Unlike some of the other strategies, **brute force** is applicable to a very wide variety of problems.
- For some important problems (e.g., sorting, searching, string matching), the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size.

Introduction

- The expense of designing a more efficient algorithm may be unjustifiable **if only a few instances of a problem need to be solved** and a brute-force algorithm can solve those instances with acceptable speed.
- Even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem.
- A brute-force algorithm can serve an important theoretical or educational purpose.

ITS033 – Programming & Algorithms

Lecture 04.2 Brute Force: Sorting Algorithms

http://www.vcharkarn.com/vlesson/showlesson.php?lessonid=7&pageid=5

Asst. Prof. Dr. Bunyarit Uyyanonvara
IT Program, Image and Vision Computing Lab.
School of Information, Computer and Communication Technology (ICT)
Sirindhorn International Institute of Technology (SIIT)
Thammasat University
http://www.siit.tu.ac.th/bunyarit
bunyarit@siit.tu.ac.th
02 5013505 X 2005

Sorting Problem

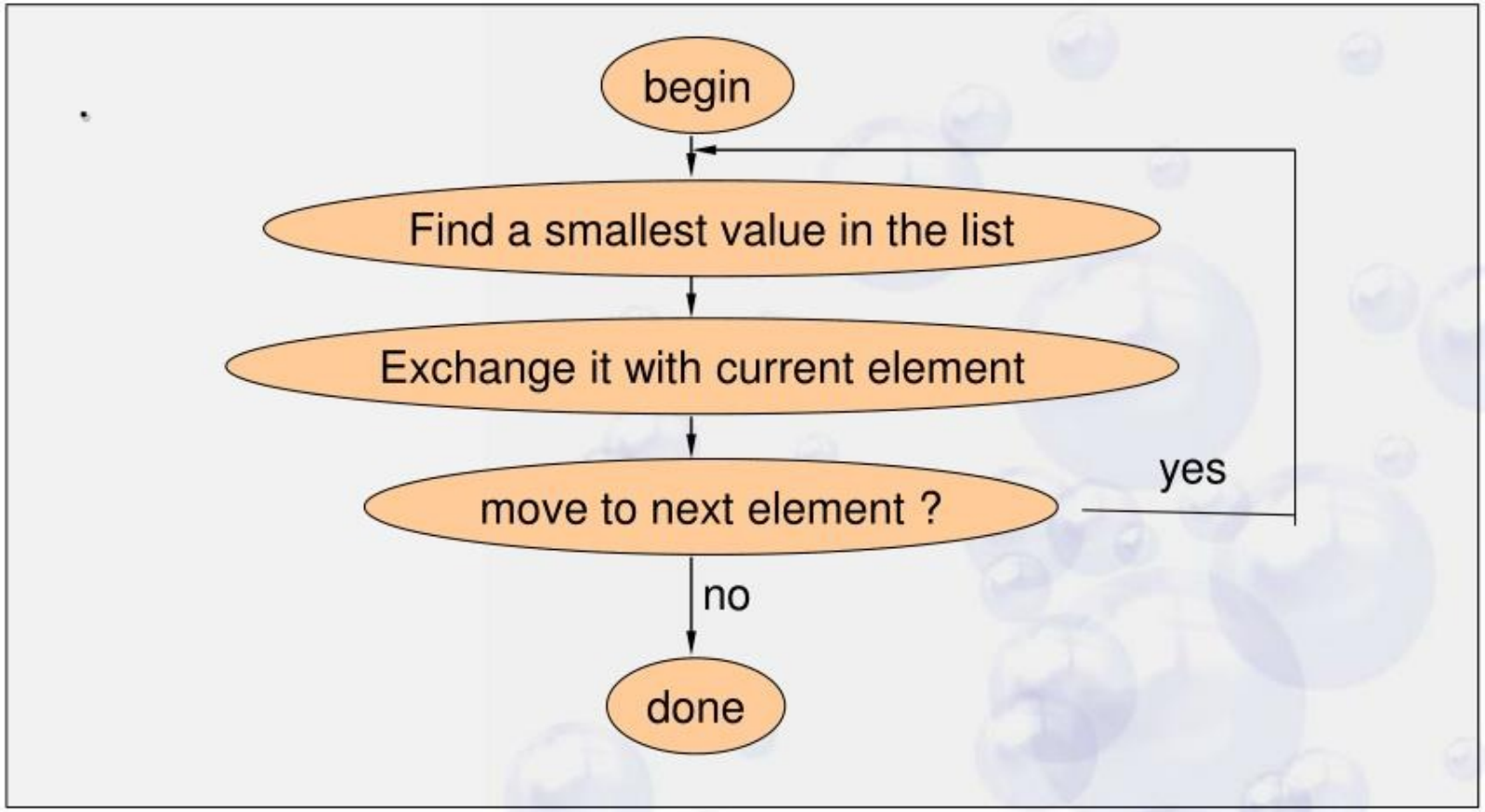
- Brute force approach to sorting:

- Given a list of *n* orderable items (e.g., numbers, characters from some alphabet, character strings), rearrange them in nondecreasing order.

Selection Sort

- We start selection sort by scanning the entire given list
- to find its smallest element
- and exchange it with the first element
- Then we repeat the process

Selection Sort



Selection Sort

ALGORITHM *SelectionSort*(*A*[0..*n* - 1])
//The algorithm sorts a given array by selection sort
//Input: An array *A*[0..*n* - 1] of orderable elements
//Output: Array *A*[0..*n* - 1] sorted in ascending order
    for *i* ← 0 **to** *n* - 2 **do**
        min ← *i*
        for *j* ← *i* + 1 **to** *n* - 1 **do**
            if *A*[*j*]<*A*[*min*] *min* ← *j*
        swap *A*[*i*] and *A*[*min*]

Example

89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

Selection sort's operation on the list 89, 45, 68, 90, 29, 34, 17. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list's tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

Analysis

- The input's size is given by the number of elements n.
- The algorithm's basic operation is the key comparison $A[j] < A[min]$. The number of times it is executed depends only on the array's size and is given by

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

- Thus, selection sort is a $O(n^2)$ algorithm on all inputs.
- Thenumber of key swaps is only $O(n)$ or, more precisely, n-1 (one for each repetition of the i loop).This property distinguishes selection sort positively from many othersorting algorithms.

Bubble Sort

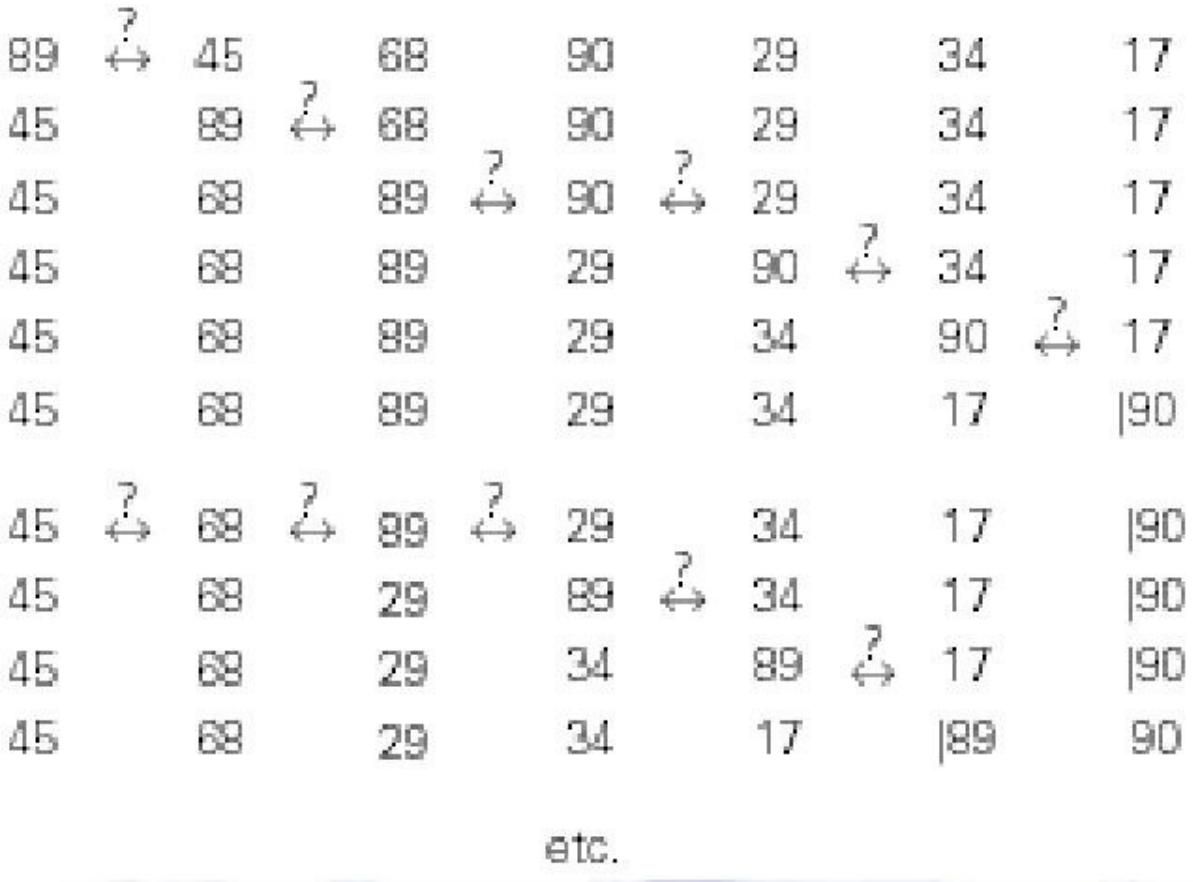
- Compare adjacent elements of the list
- and exchange them if they are out of order
- Then we repeat the process
- By doing it repeatedly, we end up ‘bubbling up’ the largest element to the last position on the list

Bubble Sort

ALGORITHM *BubbleSort(A*[0..*n* - 1]*)*
//The algorithm sorts array *A*[0..*n* - 1] by bubble sort
//Input: An array *A*[0..*n* - 1] of orderable elements
//Output: Array *A*[0..*n* - 1] sorted in ascending order
    for *i* ← 0 **to** *n* - 2 **do**
        for *j* ← 0 **to** *n* - 2 - *i* **do**
            if *A*[*j* + 1]<*A*[*j*] swap *A*[*j*] and *A*[*j* + 1]

Example

- The first 2 passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.



Bubble Sort

the analysis

- Clearly, the outer loop runs ***n*** times.
- The only complexity in this analysis in the inner loop.
- If we think about a single time the inner loop runs, we can get a simple bound by noting that it can never loop more than ***n*** times.
- Since the outer loop will make the inner loop complete *n* times, the comparison can't happen more than $O(n^2)$ times.

Analysis

- The number of key comparisons for the bubble sort version given above is the same for all arrays of size n.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

- The number of key swaps depends on the input. For the worst case of decreasing arrays, it is the same as the number of key comparisons.

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

Analysis

- Observation: if a pass through the list makes no exchanges, the list has been sorted and we can stop the algorithm
- Though the new version runs faster on some inputs, it is still in $O(n^2)$ in the worst and average cases.
- Bubble sort is not very good for big set of input.
- How ever bubble sort is very **simple to code**.

General Lesson From Brute Force Approach

- A first application of the brute-force approach often results in an algorithm that can be improved with a modest amount of effort.

Lecture 04.3

Brute Force: Searching Algorithms

http://www.vcharkarn.com/vlesson/showlesson.php?lessonid=7&pageid=5

Asst. Prof. Dr. Bunyarit Uyyanonvara
IT Program, Image and Vision Computing Lab.
School of Information, Computer and Communication Technology (ICT)
Sirindhorn International Institute of Technology (SIIT)
Thammasat University
http://www.siit.tu.ac.th/bunyarit
bunyarit@siit.tu.ac.th
02 5013505 X 2005

Sequential Search

- Sequential Search:

- Compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search).

Sequential Search

ALGORITHM *SequentialSearch2(A[0..n], K)*
//The algorithm implements sequential search with a search key as a // sentinel
//Input: An array A of n elements and a search key K
//Output: The position of the first element in $A[0..n-1]$ whose value is
// equal to K or -1 if no such element is found

```
A[n] ← K
i ← 0
while A[i] = K do
        i ← i + 1
        if i < n return i
        else return -1
```

Brute-Force String Matching

- Given a string of n characters called the ***text*** and a string of m characters ($m = n$) called the ***pattern***, find a substring of the text that matches the pattern. To put it more precisely, we want to find i—the index of the leftmost character of the first matching substring in the text—such that

$$t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}:$$

$t_0 \ldots t_i \ldots t_{i+j} \ldots t_{i+m-1} \ldots t_{n-1}$ text T

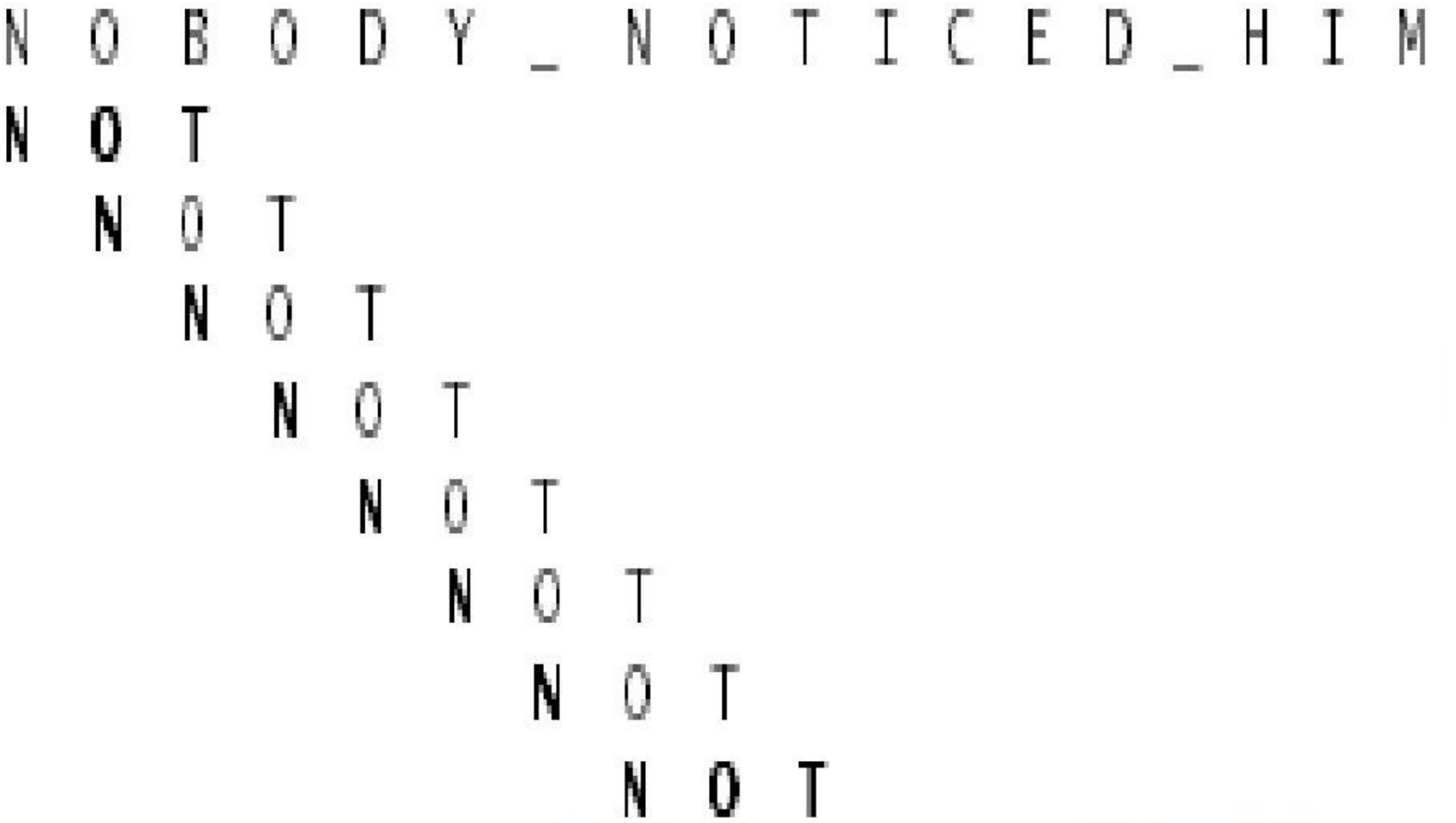
$p_0 \ldots p_j \ldots p_{m-1}$ pattern P

Brute-Force String Matching

ALGORITHM *BruteForceStringMatch(T* [0..*n* - 1], *P*[0..*m* - 1])
//The algorithm implements brute-force string matching.
//Input: An array *T* [0..*n* - 1] of *n* characters representing a text;
// an array *P*[0..*m* - 1] of *m* characters representing a pattern.
//Output: The position of the first character in the text that starts the first
// matching substring if the search is successful and -1 otherwise.

```
for i ← 0 to n - m do
        j ← 0
        while j <m and P[j ]= T [i + j ] do
                j ← j + 1
                if j = m return i
return -1
```

Example



An example of brute-force string matching. (The pattern's characters that are compared with their text counterparts are in bold type.)

Analysis

- The algorithm shifts the pattern almost always after a single character comparison.
- In the worst case, the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the $n - m + 1$ tries.
- Thus, in the worst case, the algorithm is in $\theta(nm)$.

ITS033 – Programming & Algorithms

Lecture 04.4
Brute Force: Exhaustive Search

http://www.vcharkarn.com/vlesson/showlesson.php?lessonid=7&pageid=5

Asst. Prof. Dr. Bunyarit Uyyanonvara
IT Program, Image and Vision Computing Lab.
School of Information, Computer and Communication Technology (ICT)
Sirindhorn International Institute of Technology (SIIT)
Thammasat University
http://www.siit.tu.ac.th/bunyarit
bunyarit@siit.tu.ac.th
02 5013505 X 2005

Exhaustive Search

- A brute-force approach to combinatorial problem.

- It suggests generating each and every element of the problem's domain, selecting those of them that satisfy the problem's constraints, and then finding a desired element

Exhaustive Search

- There are two well – known optimization problem:

1. Traveling Salesman Problem
2. Knapsack Problem

- Both the traveling salesman and knapsack problems, exhaustive search leads to algorithms that are extremely inefficient on every input. In fact, these two problems are the best-known examples of so-called ***NP-hard problems***. No polynomial-time algorithm is known for any *NP*-hard problem.

Traveling Salesman Problem

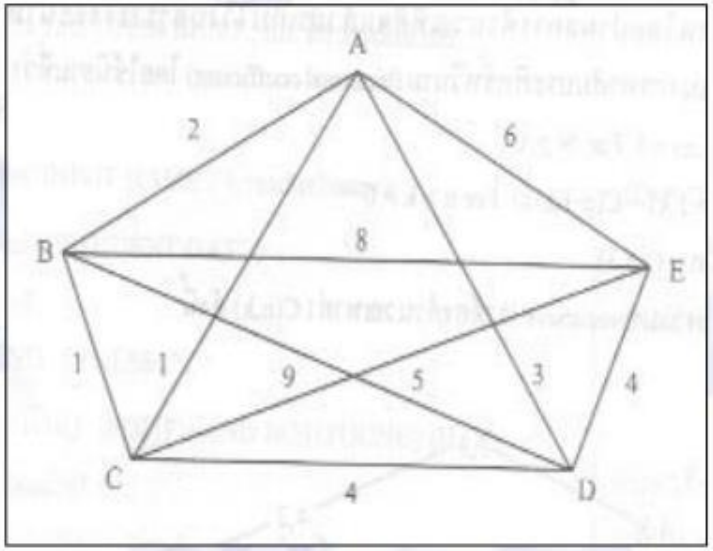
- To find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.
- The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest ***Hamiltonian circuit*** of the graph.
- A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.

9.1 Introduction

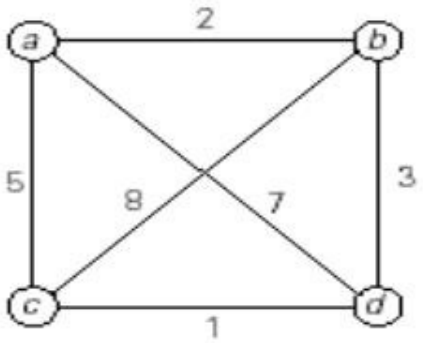
All possible paths are proportional to

(n-1)!

5 cities : 6 possible paths
11 cities: 3.6×10^6 possible paths
21 cities: 2.4×10^{18} possible paths
31 cities: forget it.



Traveling Salesman Problem-solution



Tour	Length	
a --> b --> c --> d --> a	l = 2 + 8 + 1 + 7 = 18	
a --> b --> d --> c --> a	l = 2 + 3 + 1 + 5 = 11	optimal
a --> c --> b --> d --> a	l = 5 + 8 + 3 + 7 = 23	
a --> c --> d --> b --> a	l = 5 + 1 + 3 + 2 = 11	optimal
a --> d --> b --> c --> a	l = 7 + 3 + 8 + 5 = 23	
a --> d --> c --> b --> a	l = 7 + 1 + 8 + 2 = 18	

Knapsack Problem

- Given n items of known weights $w1, \ldots, wn$ and values $v1, \ldots, vn$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

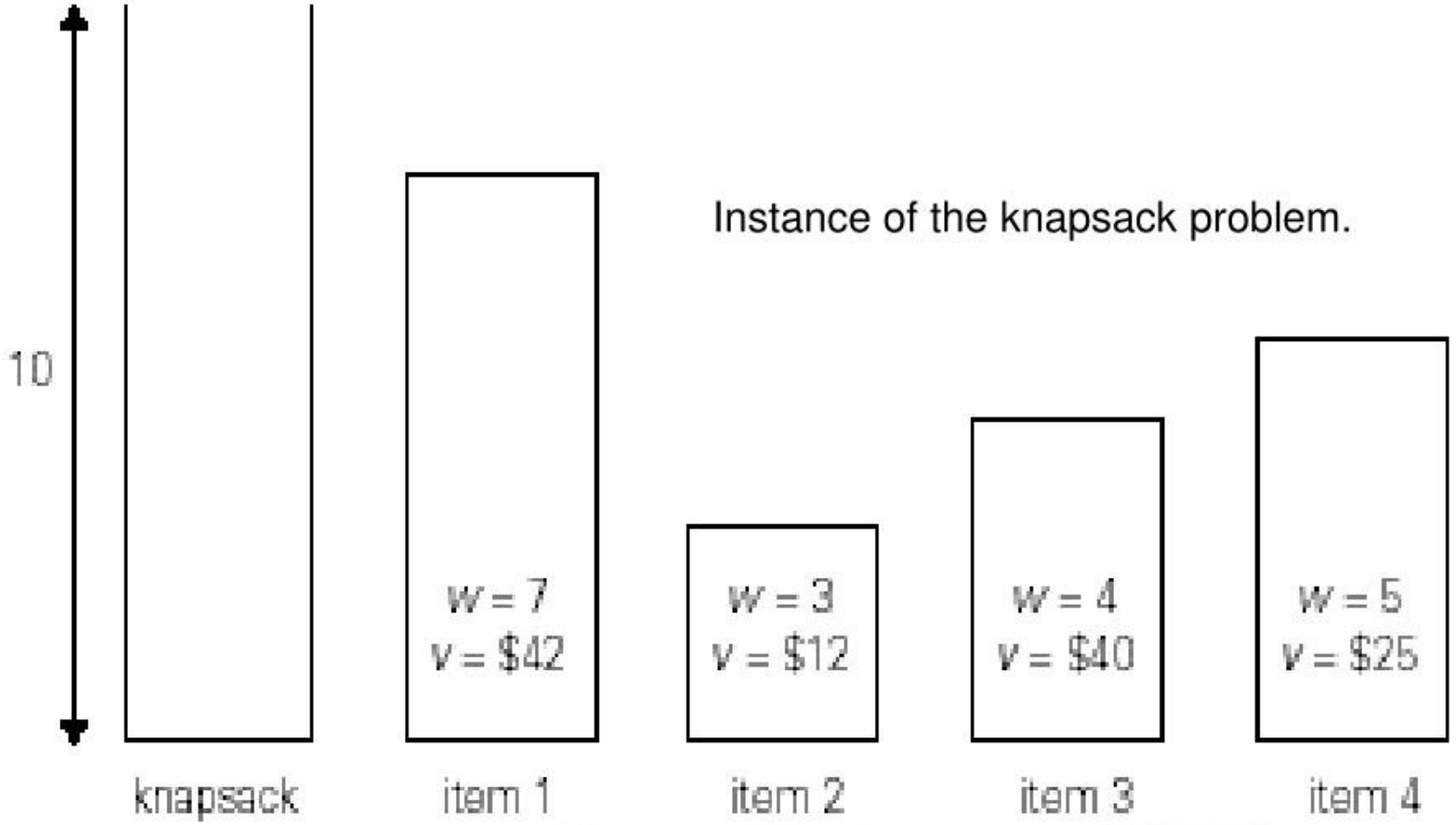
- The 0/1 knapsack problem is a problem that deals with putting the best items out of a set into a backpack without exceeding the backpack's capacity.

- To better explain this, consider packing a backpack for a day . Your backpack has a certain size, its capacity. Now, there are many different items that you could put into your backpack but you want to take items that are appropriate for the trip and leave the not so important things behind.

Knapsack Problem

- The exhaustive search approach to this problem leads to considering all the subsets of the set of n items given, computing the total weight of each subset in order to identify feasible subsets and finding a subset of the largest value among them.

Knapsack Problem - Example



Instance of the knapsack problem.

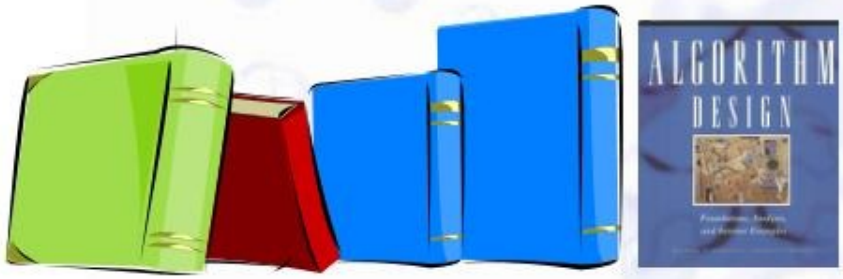
Knapsack Problem - Solution

Subset	Total weight	Total value
∅	0	$0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$36
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

Example

- Given: A set of 5 books, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose books with maximum total benefit but with weight at most 8 kg.

Items:



	1	2	3	4	5
Weight:	4 kg	2 kg	2 kg	6 kg	2 kg
Benefit:	$20	$3	$6	$25	$80

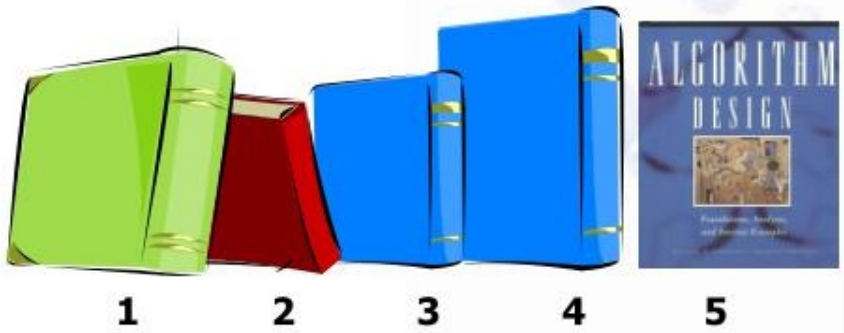
Example

"knapsack"

Solution:

- 5 (2 in)
- 3 (2 in)
- 1 (4 in)

Items:



	1	2	3	4	5
Weight:	4 in	2 in	2 in	6 in	2 in
Benefit:	$20	$3	$6	$25	$80

0/1 Knapsack Problem

- As it is use in Financial problems, it must assign values to each item.
- It must rate items in terms of their profit and cost. Profit would be the importance of the item, while the cost is the amount of space it occupies in the backpack.
- So this becomes a multi-objective problem. We want to maximize our profit while minimizing our cost.

The 0/1 Knapsack Problem

- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W.
- If we are **not** allowed to take fractional amounts, then this is the **0/1 knapsack problem**.
  - In this case, we let T denote the set of items we take
  - Objective: maximize $\sum_{i \in T} b_i$
  - Constraint:

$$\sum_{i \in T} w_i \leq W$$

0/1 Knapsack Problem

- Consider the following example:

Backpack capacity (max cost) = 10

	Cost:	Profit:	
Instant Noodle	3	10	0.3
Chocolate cake	5	12	0.42
Flash light	2	7	0.29
Jacket	4	10	0.4
Pocket knife	1	6	0.17
Basket ball	4	2	2.0

Assignment Problem

- There are n people who need to be assigned to execute n jobs, one person per job. The cost that would accrue if the ith person is assigned to the jth job is a known quantity $C[i, j]$ for each pair $i, j = 1, \ldots, n$. The problem is to find an assignment with the smallest total cost.

- This imply that there is a one-to-one correspondence between feasible assignments and permutations of the first n integers. Therefore the exhaustive approach to the assignment problem would require generating all the permutations of integers $1, 2, \ldots, n$, computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum.

Assignment Problem -Example

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

Assignment Problem -Solution

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

$\langle 1, 2, 3, 4 \rangle$ cost $= 9 + 4 + 1 + 4 = 18$

$\langle 1, 2, 4, 3 \rangle$ cost $= 9 + 4 + 8 + 9 = 30$

$\langle 1, 3, 2, 4 \rangle$ cost $= 9 + 3 + 8 + 4 = 24$

$\langle 1, 3, 4, 2 \rangle$ cost $= 9 + 3 + 8 + 6 = 26$

$\langle 1, 4, 2, 3 \rangle$ cost $= 9 + 7 + 8 + 9 = 33$

$\langle 1, 4, 3, 2 \rangle$ cost $= 9 + 7 + 1 + 6 = 23$

etc.

First few iterations of solving a small instance of the assignment problem by exhaustive search

Assignment Problem

- Since the number of permutations to be considered for the general case of the assignment problem is $n!$, exhaustive search is impractical for all but very small instances of the problem.

- There is an efficient algorithm for this problem, namely, **Hungarian method (chapter 7)**

ITS033

Topic 01 - Problems & Algorithmic Problem Solving
Topic 02 – Algorithm Representation & Efficiency Analysis
Topic 03 - State Space of a problem
Topic 04 - Brute Force Algorithm
Topic 05 - Divide and Conquer
Topic 06 - Decrease and Conquer
Topic 07 - Dynamics Programming
Topic 08 - Transform and Conquer
Topic 09 - Graph Algorithms
Topic 10 - Minimum Spanning Tree
Topic 11 - Shortest Path Problem
Topic 12 - Coping with the Limitations of Algorithms Power

http://www.siit.tu.ac.th/bunyarit/its033.php
and http://www.vcharkarn.com/vlesson/showlesson.php?lessonid=7

Homework

- Famous Alphametic
- A puzzle in which the digits in a correct mathematical expression, such as sum, are replaced by letters is called a **cryptarihm**; if, in addition, the puzzle's words make sense, it is said to be an alphametic. The most well-known alphametic was

```
  S E N D
+ M O R E
= M O N E Y
```

Two conditions are assumed: first, the correspondence between letters and decimal digits is 1-to-1. To solve an alphametic means to find which digit each letter represents.

Your works

A. Write a program for solving any cryptarithms by exhuastive search. (Assume that a given cryptarithm is a sum of two words)

B. Solve the above puzzle.

End of Chapter 4

http://www.vcharkarn.com/vlesson/showlesson.php?lessonid=7&pageid=5

Thank You!