# Introduction to Algorithm Design

Lecture Notes 3

# ROAD MAP

- **Brute Force**
  - **Selection Sort**
  - **Bubble Sort**
  - **Sequential Search**
  - **String Matching**
  - Closest-Pair Problem
  - Convex-Hull Problems
  - Travelling Salesman Problem
    - Exhaustive Search
  - Knapsack Problem
  - Assignment Problem

# Brute Force

- Applicable to wide variety of problems
- Easiest way solving problem
  - Do not think much
  - Just do it!..
- ***Brute force*** is a straightforward approach based on
  - the problem's statement
  - definitions of the concepts

# Brute Force

- Example :

  Compute $a^n$ for a given number $a$ and a nonnegative integer $n$

  By the definition of exponentiation,

$$a^n = \underbrace{a \times \ldots \times a}_{n \quad \text{times}}$$

# Brute Force

- Brute force approach: used for many elementary but important algorithmic tasks

  - compute sum of *n* numbers
  - find largest element in a list
    - yields reasonable algorithms of practical value

  - sorting
  - searching
  - string matching
    - inefficient but can be used to solve small-size instances of a problem

# Sorting Problem

- **Problem definition:**

  Given a list of *n* items, rearrange them in non-decreasing order.

# Selection Sort

- **Approach :**

  1. scan the entire given list to find its smallest element

  2. exchange it with the first element, i.e., put the smallest element in its final position

  3. scan the list starting with second element, find smallest among last n-1 elements

  4. exchange it with second element i.e., put the second smallest element in its final position

# Selection Sort

In general

On the $i^{th}$ pass of the algorithm through the list

search for the smallest item among the last *n-i* elements

swaps it with $A_i$

$$A_0 \leq A_1 \leq \ldots \leq A_{i-1} \quad | \quad A_i , \ldots , A_{min} , \ldots , A_{n-1}$$

*in their final position* *the last n-i elements*

after *n-1* passes, the list is sorted

# Selection Sort

- **Algorithm :**

```
// 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

SelectionSort (A[0 .. n-1])
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]
```

# Selection Sort

- **Example**

<table>
<tr><td>| 89</td><td>45</td><td>68</td><td>90</td><td>29</td><td>34</td><td>17</td></tr>
<tr><td>17 |</td><td>45</td><td>68</td><td>90</td><td>29</td><td>34</td><td>89</td></tr>
<tr><td>17</td><td>29 |</td><td>68</td><td>90</td><td>45</td><td>34</td><td>89</td></tr>
<tr><td>17</td><td>29</td><td>34 |</td><td>90</td><td>45</td><td>68</td><td>89</td></tr>
<tr><td>17</td><td>29</td><td>34</td><td>45 |</td><td>90</td><td>68</td><td>89</td></tr>
<tr><td>17</td><td>29</td><td>34</td><td>45</td><td>68 |</td><td>90</td><td>89</td></tr>
<tr><td>17</td><td>29</td><td>34</td><td>45</td><td>68</td><td>89 |</td><td>90</td></tr>
</table>

Selection sort's operation on the list 89, 45, 68, 90, 29, 34, 17

# Selection Sort

- **Analysis :** The number of comparisons

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

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

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

$$C(n)=\frac{n(n-1)}{2}=\theta(n^2)$$

# Selection Sort

- **Discussion :**
  - The number of key swaps is only θ(n)
    - more precisely, *n-1*
    - one for each repetition of the loop
  - But selection sort is still a $\theta(n^2)$ algorithm

- What about space efficiency?

# Bubble Sort

- **Approach :**

```
1. Compare adjacent elements of the list
2. Exchange them if they are out of order
```

After *i*<sup>th</sup> pass

$$A_0 , \ldots , A_j \overset{?}{\longleftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid A_{n-i} \le \ldots \le A_{n-1}$$

*in their final position*

# Bubble Sort

- **Algorithm :**

```
// The algorithm sorts a given array by buble sort
// Input  : An array A[0 .. n-1] of orderable elements
// Output : Array A[0 .. n-1] sorted in ascending order

BubbleSort ( A[0..n-1])
for i←0 to n-2 do
  for j←0 to n-2-i
    if A[j+1] < A[j] swap A[j] and A[j+1]
```

# Bubble Sort

- **Example**

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 89 | $\overset{?}{\leftrightarrow}$ | 45 | | 68 | | 90 | | 29 | | 34 | | 17 |
| 45 | | 89 | $\overset{?}{\leftrightarrow}$ | 68 | | 90 | | 29 | | 34 | | 17 |
| 45 | | 68 | | 89 | $\overset{?}{\leftrightarrow}$ | 90 | $\overset{?}{\leftrightarrow}$ | 29 | | 34 | | 17 |
| 45 | | 68 | | 89 | | 29 | | 90 | $\overset{?}{\leftrightarrow}$ | 34 | | 17 |
| 45 | | 68 | | 89 | | 29 | | 34 | | 90 | $\overset{?}{\leftrightarrow}$ | 17 |
| 45 | | 68 | | 89 | | 29 | | 34 | | 17 | | \|90 |
| 45 | $\overset{?}{\leftrightarrow}$ | 68 | $\overset{?}{\leftrightarrow}$ | 89 | $\overset{?}{\leftrightarrow}$ | 29 | | 34 | | 17 | | \|90 |
| 45 | | 68 | | 29 | | 89 | $\overset{?}{\leftrightarrow}$ | 34 | | 17 | | \|90 |
| 45 | | 68 | | 29 | | 34 | | 89 | $\overset{?}{\leftrightarrow}$ | 17 | | \|90 |
| 45 | | 68 | | 29 | | 34 | | 17 | | \|89 | | 90 |

etc.

The first two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17

# Bubble Sort

- **Analysis :** The number of comparisons

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

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

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

$$C(n) = \frac{n(n-1)}{2} \in \theta(n^2)$$

# Bubble Sort

An improvement:

- If a pass makes no swaps, the list is already sorted
  - Do not make any more passes
- Does not improve worst case
- Just runs faster for some inputs

# Bubble Sort

- **Discussion :**
  - The number of key swaps depends on the input.
    - for the worst case of decreasing arrays, same as the number of comparisons
  - The improved algorithm works very well on already sorted lists
    - performs just one pass on them

# Search Problem

- **<u>Definition :</u>**

  Search Problem is to find a given item (some search key *K* ) in a list of *n* elements

  - A match with the search key is found in the list
  - The search key is not in the list

# Sequential Search

- **Approach :**

  1. compare successive elements of a given list with a given search key

  2. exit if search key matches the element of the list

  3. exit if search key does not match any elements of the list

# Sequential Search

- **Algorithm :**

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

SequentialSearch(A[0..n], K)
  A[n] ← K
  i ← 0
  while A[i] ≠ K do
    i ← i+1
  if i < n return i
  else return -1
```

# Sequential Search

- **Analysis :**

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

# Sequential Search

- Any idea of improvement?..
- What if the list is sorted?..

# String Matching

- **Problem definition:**

Given a string of **n** characters called ***text***
Given a string of **m** characters called ***pattern***

Find a substring of the text that matches the pattern

# Examples

1. Pattern: 001011
   Text: 10010101101001100101111010

2. Pattern: happy
   Text: It is never too late to have a happy childhood.

# String Matching

- Brute-force Approach?..

# String Matching

- **Approach :**

  1. align the pattern against the first *m* characters of the text

  2. start matching the corresponding pairs of characters from left to right
     - until either all m pairs of the characters match or
     - mismatching pair is encountered

  3. shift the pattern one position to the right

  4. resume character comparisons, starting with the first character of the pattern and its counterpart in the text

# Examples

1. Pattern: 001011
   Text: 10010101101001100101111010

2. Pattern: happy
   Text: It is never too late to have a happy childhood.

# String Matching

- **Algorithm :**

```
// The algorithm implements brute-force string matching
// Input1  : An array T[0..n-1] of n characters
                    represents a text
// Input2  : An array P[0..m-1] of m characters
                    represents a pattern
// Output : The position of the first character in the
            text that starts the first matching substring
            if the search is successfull 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
```

# String Matching

- **Example**

```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

The pattern's characters that are compared with their text counterparts are in **bold** type

# String Matching

- **Analysis??**
  - Time efficiency
  - Space efficiency

# String Matching

- In worst case, the algorithm make all *m* comparisons before shifting the pattern
- There are *n-m+1* tries.
- So in worst case the algorithm is **$\theta$(nm)**
- The average case should be better than worst case
  - All tries cannot make m comparisons!..
- It has been shown that in random texts, algorithm is **$\theta$(n+m) = $\theta$(n)**

# String Matching

- **Discussion :**

  There are several more sophisticated and more efficient algorithms for string matching. We will discuss them later !

# ROAD MAP

- **Brute Force**
  - Selection Sort
  - Bubble Sort
  - Sequential Search
  - String Matching
  - **Closest-Pair Problem**
  - **Convex-Hull Problems**
  - **Travelling Salesman Problem**
    - **Exhaustive Search**
  - **Knapsack Problem**
  - **Assignment Problem**

# Closest - Pair Problem

- **Problem Definition :**

  Find the closest points in a set of *n* points.

  - We consider the two dimensional case of the problem
  - We assume that points in question are specified in a standard fashion by their *(x,y)* Cartesian coordinates
  - We assume that distance between two points $P_i = (x_i, y_i)$ and $P_j = (x_j, y_j)$ is the standard Euclidean distance

$$d(P_i, P_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

# Closest - Pair Problem

- Brute force approach?

# Closest - Pair Problem

- <u>**Approach :**</u>

**1. compute distance between each pair of distinct points**

**2. find a pair with the smallest distance**

- We do not want to compute the distance between same pair of points twice
  - consider only the pairs of points $(P_i, P_j)$ for which $i < j$

# Closest - Pair Problem

- **Algorithm :**

```
// Input  : A list P of (n≥2) points
                P1 = (x1, y1) , … Pn = (xn,yn)
// Output : Indicates index1 and index2 of the
                closest pair of points

dmin ← ∞
for i ← 1 to n-1 do
  for j ← i+1 to n do
    d ← sqrt ( (xi-xj)² + (yi-yj)² )
     if d < dmin
       dmin ← d; index1 ← i ; index2 ← j
return index1, index2
```

# Closest - Pair Problem

- **Analysis :**

What is the basic operation?

# Closest - Pair Problem

- **Discussion :**
  - Basic operation is computing Euclidean distance between two points
    - square root is not a simple operation such as addition or multiplication
  - **So, is it an algorithm?..**
  - We can ignore square root function and compare values $(x_i-x_j)^2 + (y_i-y_j)^2$ themselves
    - the smaller a number of which we take square root, the smaller its square root
    - square root function is strictly increasing

# Closest - Pair Problem

- **Analysis :** The number of basic operations (multiplications)

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

$$C(n) = 2\sum_{i=1}^{n-1} (n-i)$$

$$C(n) = 2\left[(n-1) + (n-2) + \ldots + 1\right]$$

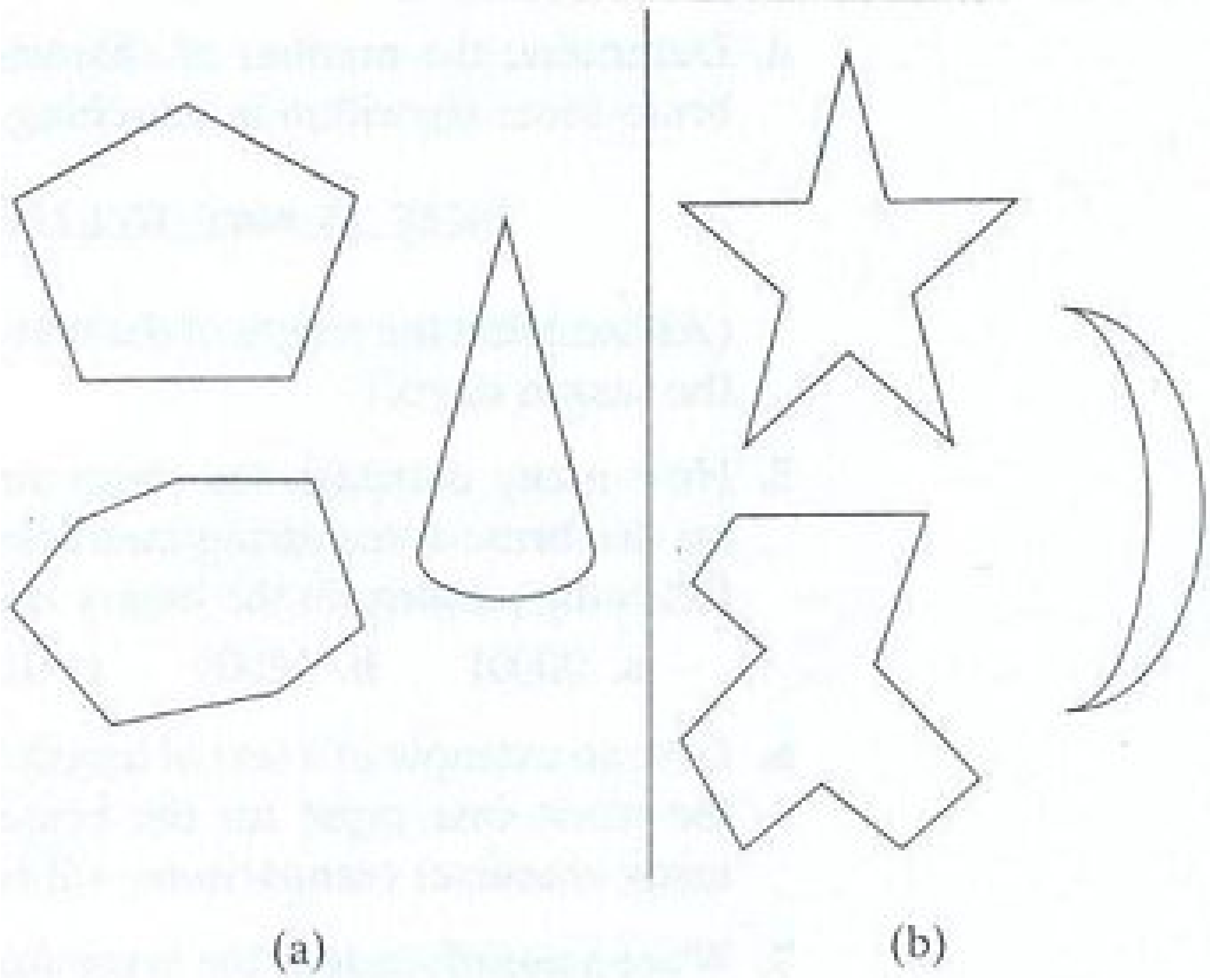
$$C(n) = (n-1)n \in \theta(n^2)$$

# Convex-Hull Problem

- **What is convex ?**

  A set of points (finite or infinite) on the plane is called ***convex*** if for any two points *P* and *Q* in the set, the entire line segment with the end points *P* and *Q* belongs to the set

# Convex-Hull Problem




(a) Convex sets. (b) Sets that are not convex.

# Convex-Hull Problem

- **What is convex hull ?**

The ***convex hull*** of a set *S* of points is the smallest convex set containing *S*.

The *smallest* requirement means that the convex hull of *S* must be a subset of any convex set containing *S*

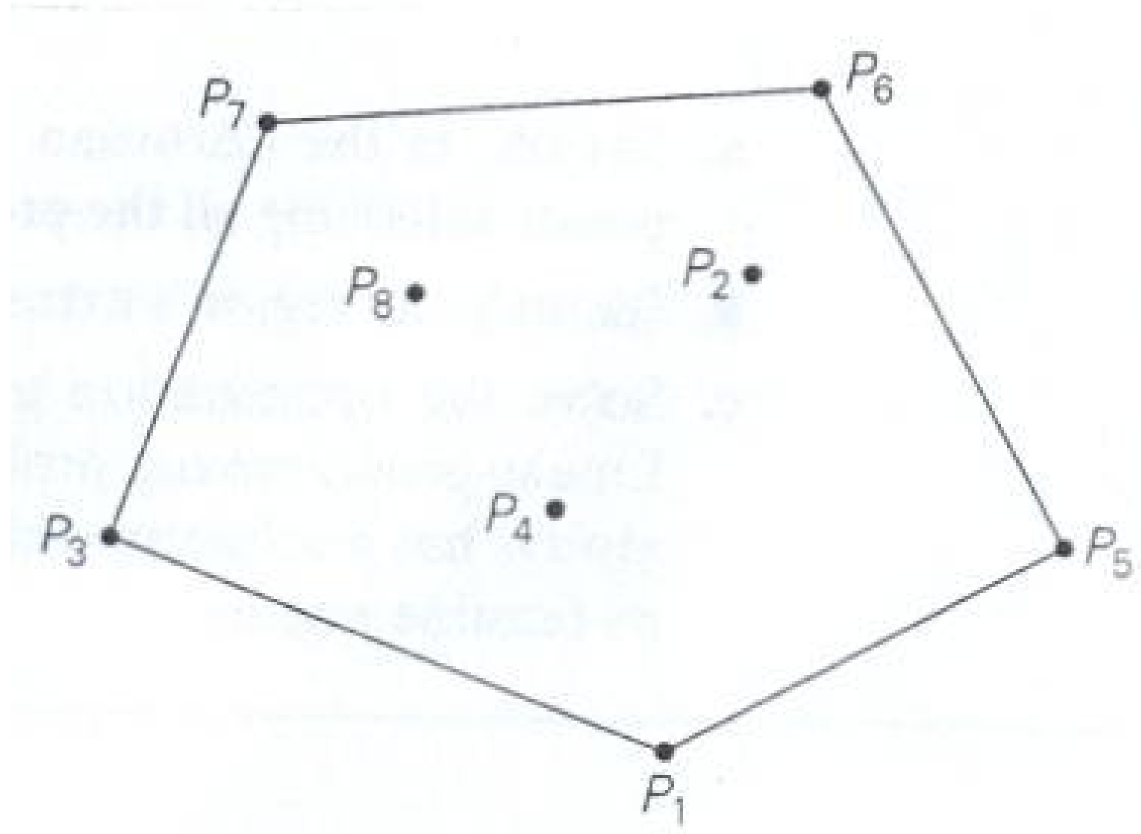
# Convex-Hull Problem

- **What is convex hull ?**
  - If *S* is convex, its convex hull is obviously *S* itself
  - If *S* is a set of two points, its convex hull is the line segment connecting these points
  - If S is a set of three points not on the same line, its convex hull is the triangle with the vertices at the three points given
    - If three points do lie on the same line, the convex hull is the line segment with its end points at the two points that are farthest apart

# Convex-Hull Problem

- **Theorem :**
  - The convex hull of any set *S* of *n>2* points (not on the same line) is a convex polygon with the vertices at some of the points of *S*.
    - If all the points do line on the same line, the polygon degenerates to a line segment but still with the end points at two points of S

# Convex-Hull Problem




**Convex hull for set of 8 points:**
**polygon with its vertices at $P_1$, $P_5$, $P_6$, $P_7$ , $P_3$**

# Convex-Hull Problem

- **Problem Definition :**

*Convex hull problem* is the problem to constructing the *convex hull* for a given set S of *n* points

# Convex-Hull Problem

- Need to find the the vertices of the polygon
  - called "*extreme points*"
  - *extreme point* of a convex set is a point of this set that is not a middle point of any line segment with end points in the set
    - extreme points of a triangle are its three vertices
- Find which pairs of points need to be connected to form the boundary of the convex hull
  - List of extreme points in clockwise order

So, how can we solve convex-hull problem in a brute force manner ?

# Convex-Hull Problem

Problem has no obvious algorithmic solution !

- **Idea :**
  - A line segment connecting two points $P_i$ and $P_j$ of a set of *n* points is a part of its convex hull's boundary iff all the other points of the set lie on the same side of the straight line through these two points
  - Repeating this test for every pair of points yields a list of line segments that make up the convex hull's boundary

# Convex-Hull Problem

How to check whether all points in the set are on the same side of the line connecting two points:

- Straight line through two points $(x_1,y_1)$, $(x_2,y_2)$ in the coordinate plane is

  $$ax+by=c$$

  where $a=y_2-y_1$, $b=x_1-x_2$, $c=x_1y_2-y_1x_2$
- Such a line divides plane into two half-planes
  - The points in one half→ $ax+by>c$
  - The points in other → $ax+by<c$
- To check whether certain points lie on the same side of the line, we can check whether the expression $ax+by-c$ has the same sign at each of these points

# Convex-Hull Problem

- **Analysis :**

  for each of `n(n-1)/2` pairs of distinct points.

  check the sign of `ax+by-c` for each of the other *n-2* points

$$T(n) = O(n^3)$$

There are more efficient algorithms for this problem.
We will discuss them later !

# Convex-Hull Problem

- **Discussion :**
  - *Convex hull problem* has no obvious algorithmic solution.
  - But there exists simple algorithms for this problem

# Traveling Salesman Problem

- **Problem definition :**

  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

# Traveling Salesman Problem

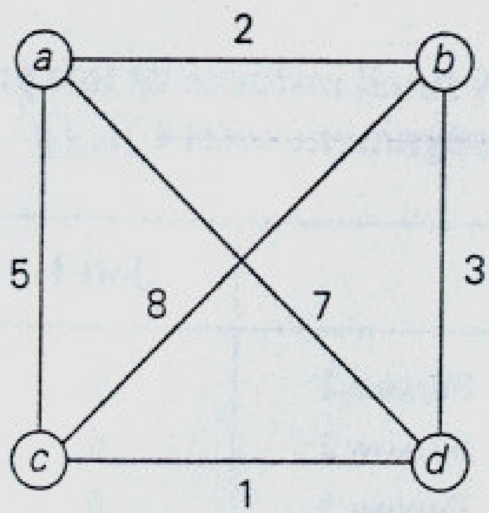
- The problem can be modeled by a weighted graph
- Vertices represents cities
- Edge weights represent the distances

- ***Exhaustive search*** can be used to solve the problem

# Exhaustive Search

- Exhaustive search is a simple brute force approach to combinatorial problems
- Exhaustive search suggests
  - Generating each and every element of the problem's domain
  - Selecting those of them that satisfy the problem constraints
  - Finding a desired element
    - Optimizes some objective function

# Traveling Salesman Problem




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



**A solution to a small instance of the traveling salesman problem by exhaustive search**

# Traveling Salesman Problem

- **Discussion :**
  - This problem has been intriguing for the last 150 years by
    - seemingly simple formulation,
    - important applications and
    - interesting connections to other combinatorial problems

# Knapsack Problem

- **Problem Definition :**

  Given *n* items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity $W$.

  Find the most valuable subset of the items that fit into the knapsack

# Knapsack Problem

- **Exhaustive Search Approach :**
  - Consider all the subsets of the set of *n* items given
  - Compute the total weight of each subset in order to identify feasible subsets
  - Find a subset of the largest value among them

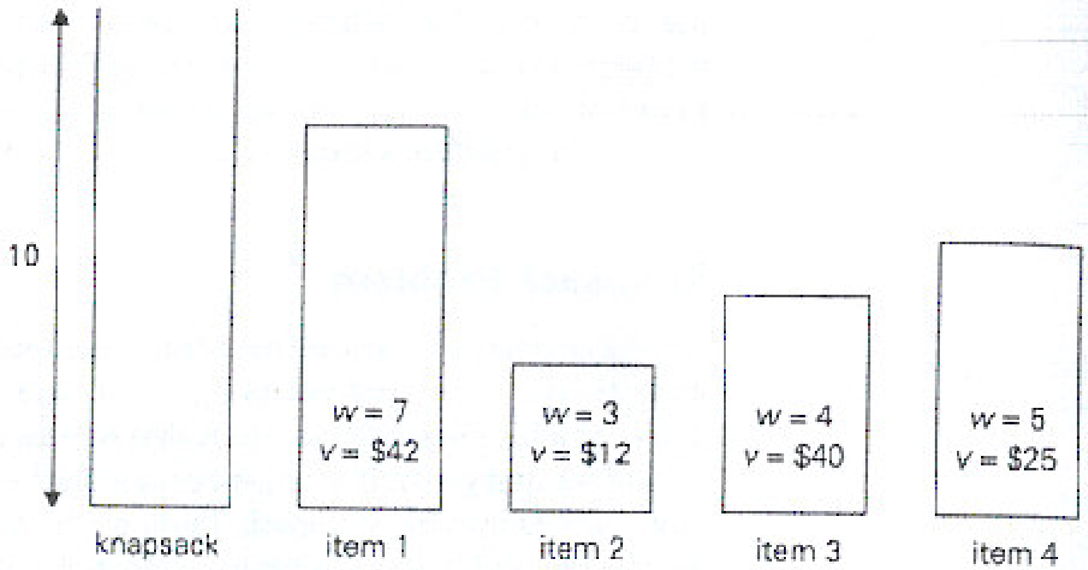
# Knapsack Problem

- **Analysis:**

  \# of subsets of an *n*-element set is $2^n$

  So exhaustive search leads to a $\Omega(2^n)$ algorithm

# Knapsack Problem




| Subset | Total weight | Total value |
|---|---|---|
| Ø | 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 |

# Knapsack Problem

- **Discussion :**
  - Knapsack is a well known problem in algorithmics
  - Exhaustive search leads to algorithms that are extremely inefficient on every input
  - In fact knapsack is one of the examples of *NP-hard* problems
  - No polynomial-time algorithm is known for any NP-hard problem !

# Assignment Problem

- **Problem Definition :**
  - There are *n* people who need to be assigned to execute *n* jobs, one person per job.
  - The cost of assigning the *i*th person to the *j*th job is a known quantity $C[i,j]$ for each pair $i,j=1, \dots, n$.
  - Problem is to find an assignment with smallest total cost !

# Assignment Problem

- **Exhaustive Search Approach :**
  - Assignment problem is completely specified by its cost matrix C [i,j]
  - 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

- **Exhaustive Search Approach :**
  - Describe $n$-tuples $(j_1, \ldots, j_n)$ in which the $i$th component, $i=1, \ldots, n$, indicates the column of the element selected in the $i$th row
  - There is a one to one correspondence between feasible assignments and permutations of the first $n$ integers

```
1. generate all permutations of integers
   1,2,…,n
2. compute the total cost of each assignment
   by summing up the corresponding elements of
   the cost matrix
3. select the one with smallest sum
```

# Assignment Problem

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

| | |
|---|---|
| $<1, 2, 3, 4>$ | cost = 9 + 4 + 1 + 4 = 18 |
| $<1, 2, 4, 3>$ | cost = 9 + 4 + 8 + 9 = 30 |
| $<1, 3, 2, 4>$ | cost = 9 + 3 + 8 + 4 = 24 |
| $<1, 3, 4, 2>$ | cost = 9 + 3 + 8 + 6 = 26 |
| $<1, 4, 2, 3>$ | cost = 9 + 7 + 8 + 9 = 33 |
| $<1, 4, 3, 2>$ | cost = 9 + 7 + 1 + 6 = 23 |

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

# Assignment Problem

- **Discussion :**
  - The fact that the problem's domain grows exponentially (or faster) does not necessarily imply that there can be no efficient algorithm for solving it
  - We will discuss them later !

# Brute-Force Strengths and Weaknesses

- Strengths
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems (e.g., matrix multiplication, sorting, searching, string matching)

- Weaknesses
  - rarely yields efficient algorithms
  - some brute-force algorithms are unacceptably slow
  - not as constructive as some other design techniques