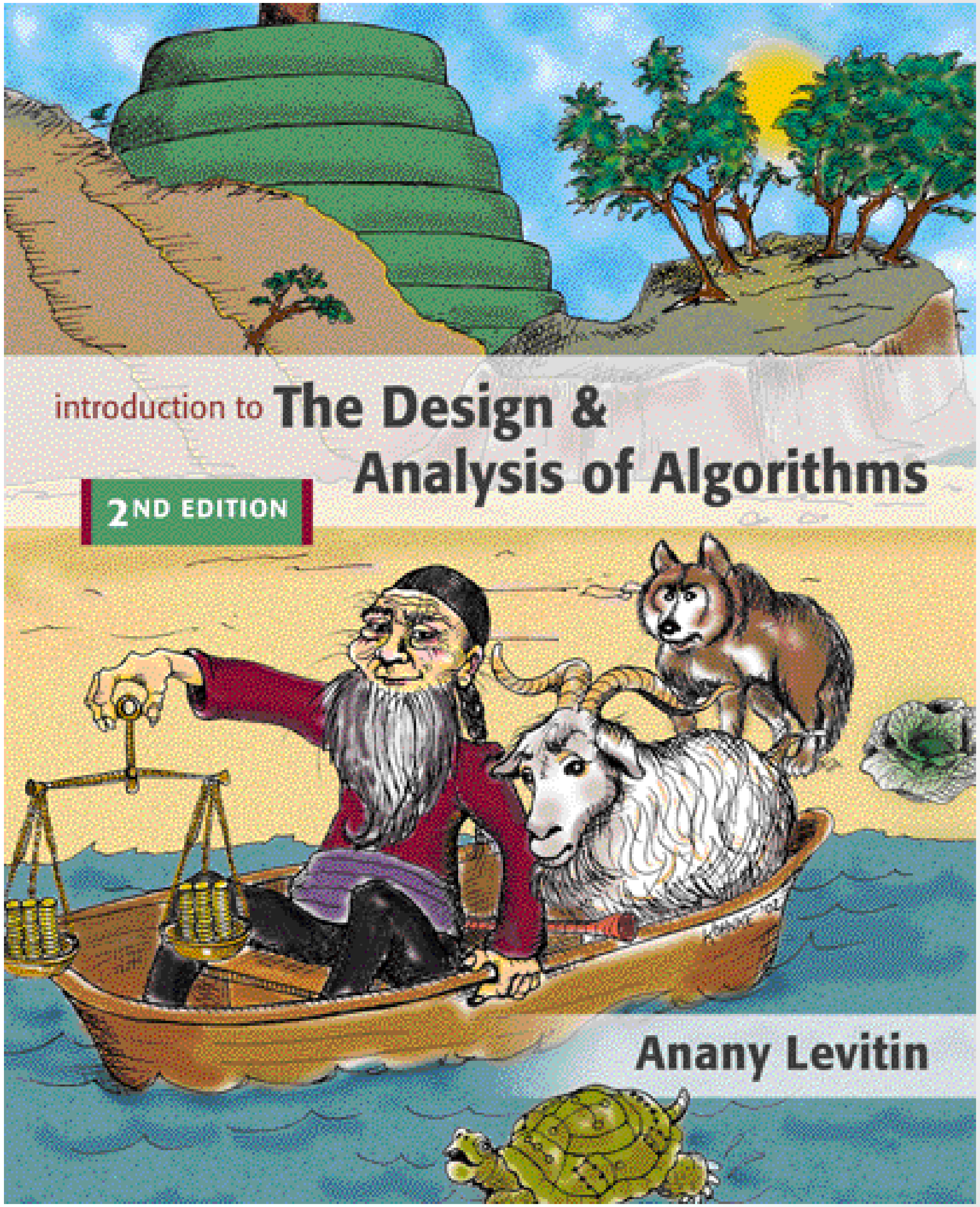
Next Topics

- Brute Force Algorithms
- Divide and Conquer Algorithms
- Decrease and Conquer Algorithms
- Transform and Conquer Algorithms
- Dynamic Programming
- Greedy Technique
- Iterative Improvement – Simplex
- Coping with the limitations

Chapter 3

Brute Force



Brute Force

A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved

Examples:

1. Computing a^n ($a > 0$, n a nonnegative integer)

2. Computing n!

3. Multiplying two matrices

4. Searching for a key of a given value in a list

Brute-Force Sorting Algorithm

Selection Sort   Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second elements. Generally, on pass i ($0 \le i \le n$-2), find the smallest element in $A[i..n-1]$ and swap it with $A[i]$:

$$A[0] \le \; . \; . \; . \; \le A[i-1] \;\; | \;\; A[i], \; . \; . \; . \; , A[min], \; . \; . \; ., A[n-1]$$

in their final positions

Example: 7   3   2   5

Analysis of Selection Sort

ALGORITHM *SelectionSort*$(A[0..n-1])$
    //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 $i \leftarrow 0$ **to** $n-2$ **do**
        $min \leftarrow i$
        for $j \leftarrow i+1$ **to** $n-1$ **do**
            if $A[j] < A[min]$ $min \leftarrow j$
        swap $A[i]$ and $A[min]$

Time efficiency: $\Theta(n^2)$

Space efficiency: $\Theta(n)$

Stability?

Brute-Force String Matching

- *pattern*: a string of *m* characters to search for
- *text*: a (longer) string of *n* characters to search in
- problem: find a substring in the text that matches the pattern

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
- all characters are found to match (successful search); or
- a mismatch is detected

Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

Example of Brute-Force String Matching

Text: 10010101101001100101111010

Pattern: 010

010

010

Pseudocode and Efficiency

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

//Implements brute-force string matching
//Input: An array $T[0..n-1]$ of n characters representing a text and
// an array $P[0..m-1]$ of m characters representing a pattern
//Output: The index of the first character in the text that starts a
// matching substring or -1 if the search is unsuccessful

for $i \leftarrow 0$ **to** $n - m$ **do**
  $j \leftarrow 0$
  while $j < m$ **and** $P[j] = T[i + j]$ **do**
    $j \leftarrow j + 1$
  if $j = m$ **return** i
return -1

Efficiency: worst case = **$\Theta(n.m)$** – when pattern is not found

Average case **= $\Theta(n + m) = \Theta(n)$ (n>>m)**

Brute-Force Polynomial Evaluation

Problem: Find the value of polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0$$

at a point $x = x_0$

<u>Brute-force algorithm</u>

$p \leftarrow 0.0$
for $i \leftarrow n$ **downto** 0 **do**
  $power \leftarrow 1$
    for $j \leftarrow 1$ **to** i **do**   **//compute x^i**
      $power \leftarrow power * x$
    $p \leftarrow p + a[i] * power$

return p

Efficiency: $\Theta(n^2)$

Polynomial Evaluation: Improvement

We can do better by evaluating from right to left:

<u>Better brute-force algorithm</u>

$p \leftarrow a[0]$
$power \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
    $power \leftarrow power * x$
    $p \leftarrow p + a[i] * power$
return p

Efficiency: $\Theta(n)$

Closest-Pair Problem

Find the two closest points in a set of *n* points (in the two-dimensional Cartesian plane).

<u>Brute-force algorithm</u>

Compute the distance between every pair of distinct points

and return the indexes of the points for which the distance is the smallest.

Closest-Pair Brute-Force Algorithm (cont.)

ALGORITHM *BruteForceClosestPoints*(P)

//Input: A list P of n ($n \geq 2$) points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$
//Output: Indices $index1$ and $index2$ of the closest pair of points
$dmin \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow i + 1$ **to** n **do**
    $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ //$sqrt$ is the square root function
    if $d < dmin$
      $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$
return $index1, index2$

Efficiency: $\Theta(n^2)$

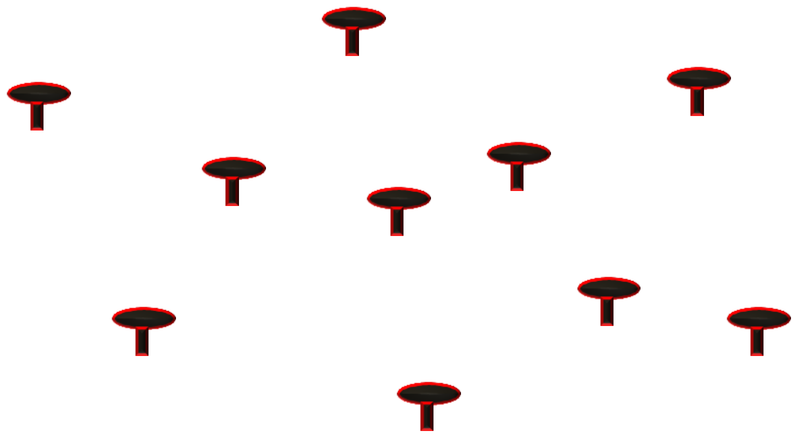
How to make it faster?

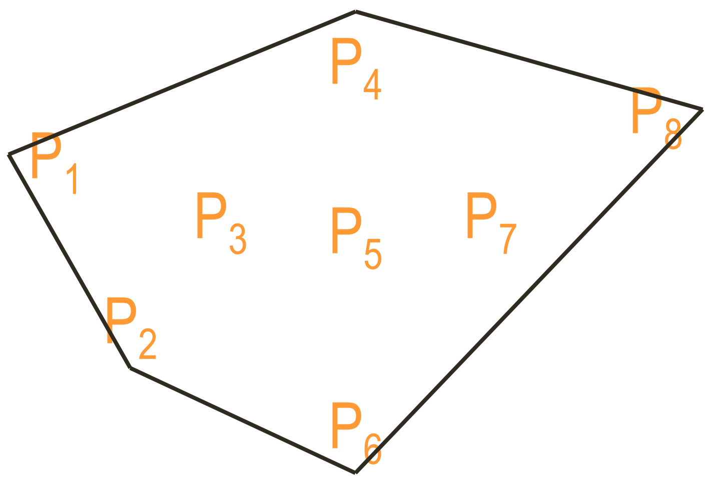
Brute-Force Strengths and Weaknesses

- 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

Brute force Convex hull (envoltória convexa)



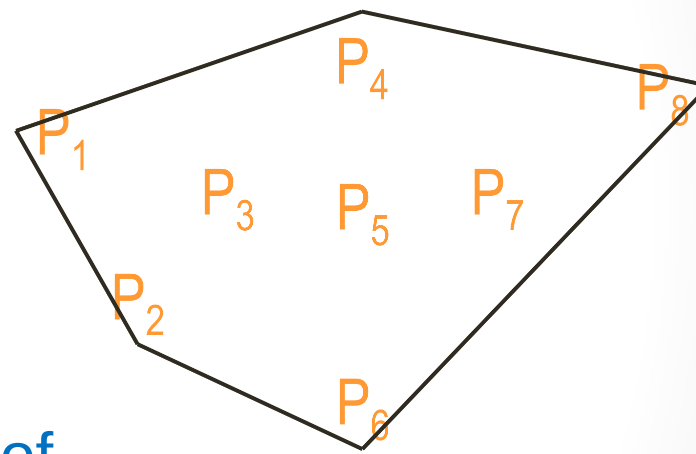


Smallest convex polygon that contains a set of n points in the plane ➔ for any 2 points P and Q, the line segment with the endpoints at P and Q belongs to the set

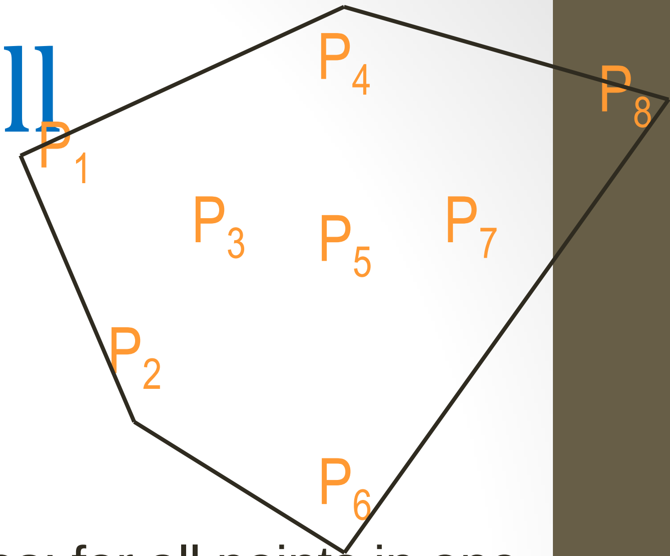
Brute force Convex hull

Simple but inefficient:

- A line segment $\overrightarrow{P_i P_j}$ is a part of its convex hull´s boundary iff all the other points on 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.



Brute force Convex hull



Simple but inefficient:

- Line (x_1, y_1) (x_2, y_2) is given by ax+by=c where $a=y_2-y_1$, $b=x_1-x_2$, $c=x_1x_2-y_1y_2$
- Such a line divides the plane into two half-planes: for all points in one of them, ax+by > c, while in the other, ax+by < c
- To check whether certain points lie on the same side of the line, we check whether the expression ax+by-c has the same sign at each of these points.
- There are n(n-1)/2 pairs of distinct points to consider. For each any other n-2 points, we need to find the sign of ax+by-c
  ➔ n(n-1)/2 * (n-2)
- ∴ **$O(n^3)$**

Exhaustive Search

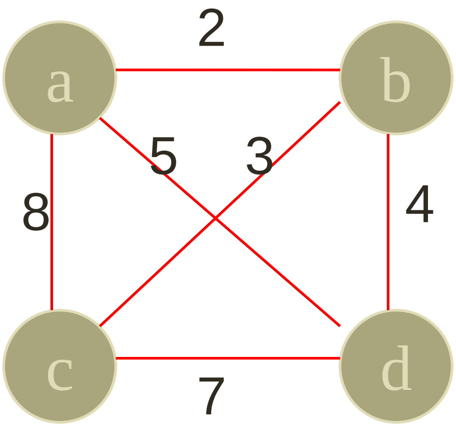
A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.

Method:

- generate a list of all potential solutions to the problem in a systematic manner (see algorithms in Sec. 5.4)
- evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
- when search ends, announce the solution(s) found

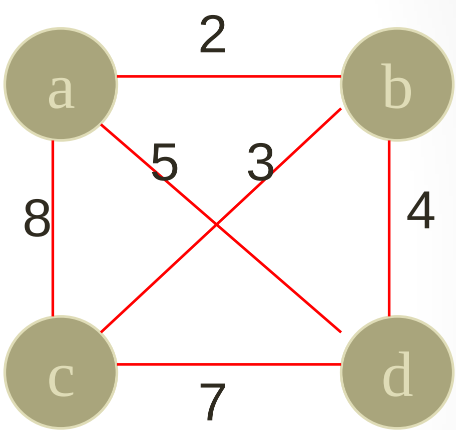
Example 1: Traveling Salesman Problem

- Given *n* cities with known distances between each pair, find the **shortest tour (or minimum path)** that passes through all the cities exactly once before returning to the starting city
- Alternatively: Find shortest *Hamiltonian circuit* in a weighted connected graph
- Example:



TSP by Exhaustive Search

Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17



More tours?

Less tours?

Efficiency: **# permutations = $\Theta(n!)$**

Example 2: Knapsack Problem

Given n items:

- weights: $w_1 \quad w_2 \; \dots \; w_n$
- values: $v_1 \quad v_2 \; \dots \; v_n$
- a knapsack of capacity W

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

Example: Knapsack capacity W=16

item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Knapsack Problem by Exhaustive Search

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	**15**	**$80**
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	not feasible
{1,2,4}	12	$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

Efficiency: $\Theta(2^n)$

Example 3: The Assignment Problem

There are *n* people who need to be assigned to *n* jobs, one person per job. The cost of assigning person *i* to job *j* is C[*i*,*j*]. Find an assignment that minimizes the total cost.

	Job 0	Job 1	Job 2	Job 3
Person 0	9	2	7	8
Person 1	6	4	3	7
Person 2	5	8	1	8
Person 3	7	6	9	4

Algorithmic Plan: (a) Generate all legitimate assignments, (b) compute their costs, and (c) select the cheapest one.

How many assignments are there?

Pose the problem as the one about a COST MATRIX:

Assignment Problem by Exhaustive Search

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

Assignment (col.#s)	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24
1, 3, 4, 2	9+3+8+6=26
1, 4, 2, 3	9+7+8+9=33
1, 4, 3, 2	9+7+1+6=23
	etc.

Efficiency: **# permutations = $\Theta(n!)$**

Final Comments on Exhaustive Search

- Exhaustive-search algorithms run in a realistic amount of time only on very small instances

- In some cases, there are much better alternatives!
  - Euler circuits
  - shortest paths
  - minimum spanning tree
  - assignment problem

- In many cases, exhaustive search or its variation is the only known way to get exact solution