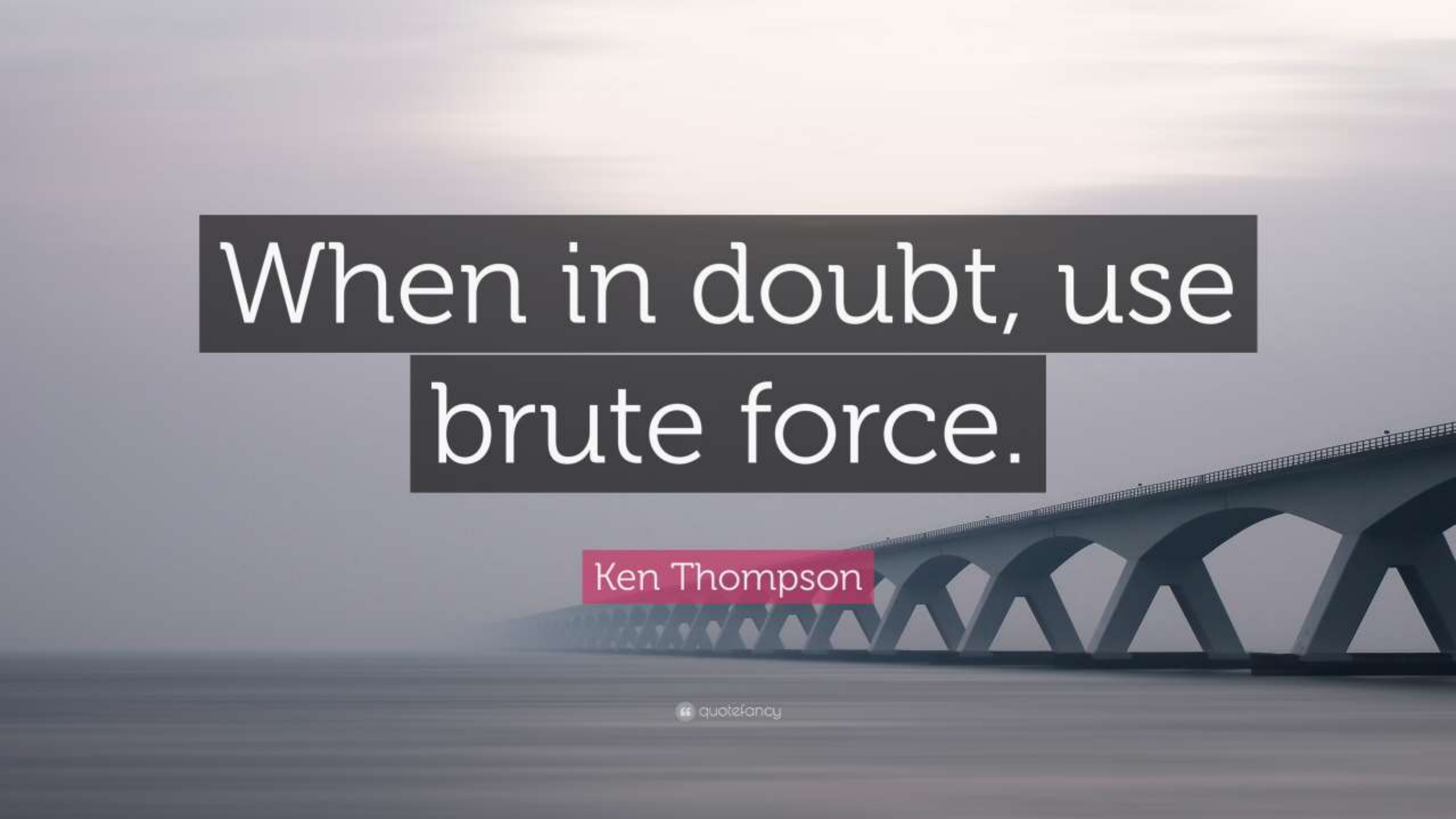
Brute Force

UNIT 2


When in doubt, use brute force.
Ken Thompson
quotefancy

What is Brute Force?

Brute Force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

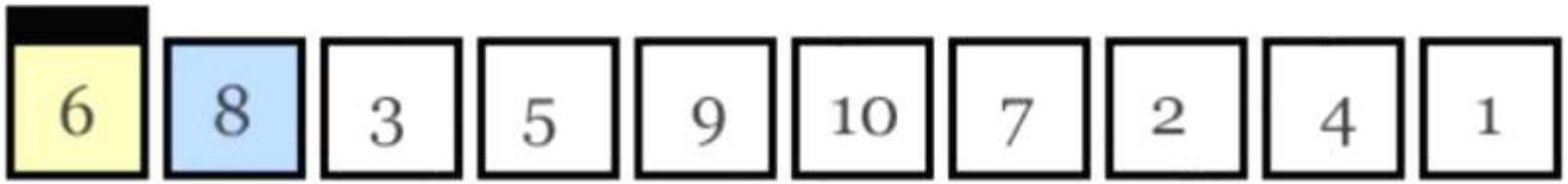
BRUTE FORCE SORTING ALGORITHMS

Selection Sort – Idea

- 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 element.
- Generally, on pass i ($0 \le i \le n-2$), find the smallest element in $A[i..n-1]$ and swap it with $A[i]$

$A[0] \le A[1] \le A[2] \ldots \le A[i-1] \mid A[i+1], \ldots, A[min], \ldots, A[n-1]$

in their final positions — the last n - i elements



Yellow is smallest number found

Blue is current item

Green is sorted list

Selection Sort - Algorithm

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 ← 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 - Analysis

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

Selection Sort is a $\Theta(n^2)$ algorithm

Bubble Sort – Idea

- Compare adjacent elements of the list and exchange them if they are out of order.
- By doing it repeatedly, we end up bubbling the largest element to the last position on the list.
- The next pass bubbles up the second largest element and so on and after n - 1 passes, the list is sorted.
- Pass i ($0 \leq i \leq n - 2$) can be represented as follows:

$$A[0], A[1], A[2], \ldots, A[j] \longleftrightarrow A[j+1], \ldots, A[n-i-1] \mid A[n-i] \leq \ldots. \leq A[n-1]$$

5 2 4 6 1 3

Bubble Sort - Algorithm

ALGORITHM $BubbleSort(A[0..n-1])$

//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in ascending order
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow 0$ **to** $n-2-i$ **do**
        if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

Bubble Sort - Analysis

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

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

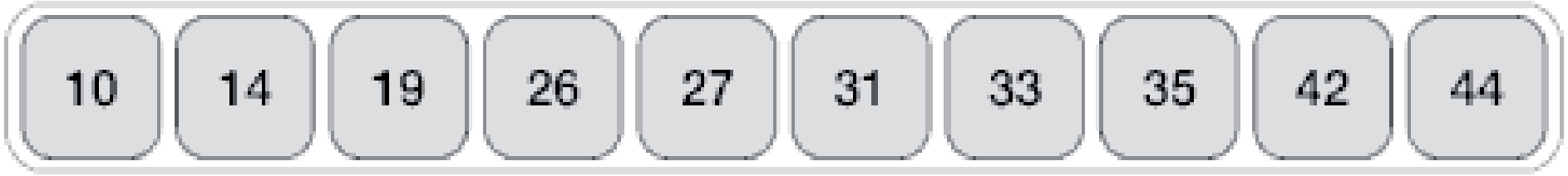
Bubble Sort is a $\Theta(n^2)$ algorithm

SEQUENTIAL SEARCH

Sequential Search – Idea

- Compares successive elements of a given list with a given search key until:
  - A match is encountered (Successful Search)
  - List is exhausted without finding a match (Unsuccessful Search)
- An improvisation to the algorithm is to append the key to the end of the list.
- This means the search has to be successful always and we can eliminate the end of list check.

Linear Search



=
33

Sequential Search - Algorithm

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

```
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index 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
```

Sequential Search - Analysis

Sequential Search is a Θ(n) algorithm

BRUTE – FORCE STRING MATCHING

String Matching – Terms

- *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

String Matching – Idea

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

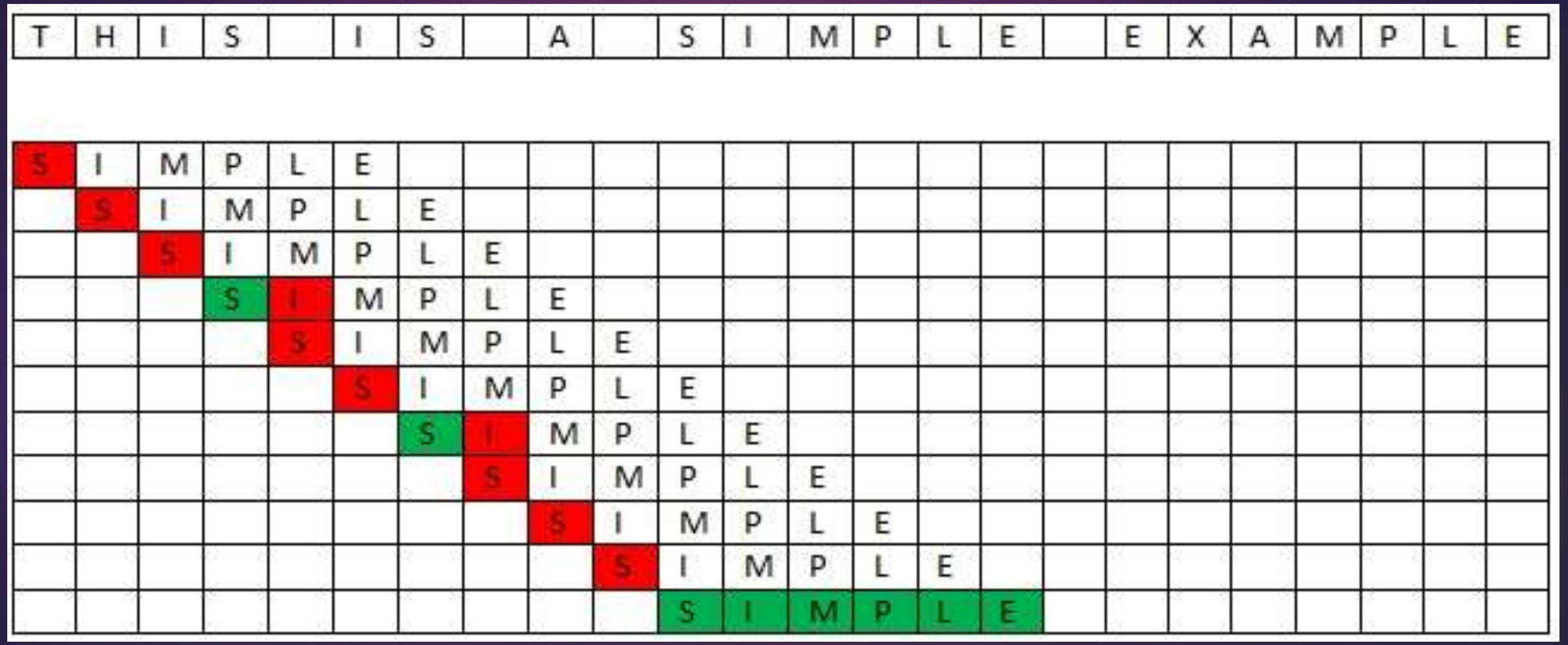
Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

String Matching - Algorithm

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 ← 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
```


T H I S I S A S I M P L E E X A M P L E
S I M P L E
S I M P L E
S I M P L E
S I M P L E
S I M P L E
S I M P L E
S I M P L E
S I M P L E
S I M P L E
S I M P L E
S I M P L E

String Matching - Analysis

Worst Case:

The algorithm might have to make all the ‘m’ comparisons for each of the (n-m+1) tries.

Therefore, the algorithm makes m(n-m+1) comparisons.

Brute Force String Matching is a O(nm) algorithm

EXHAUSTIVE SEARCH

What is Exhaustive Search?

- Exhaustive Search is a brute – force problem solving technique
- It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints and then finding a desired element.
- The desired element might be one which minimizes or maximizes a certain characteristic.
- Typically the problem domain involves combinatorial objects such as permutations, combinations and subsets of a given set.

Exhaustive Search - Method

- Generate a list of all potential solutions to the problem in a systematic manner
- 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

TRAVELING SALESMAN PROBLEM

The Problem

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?

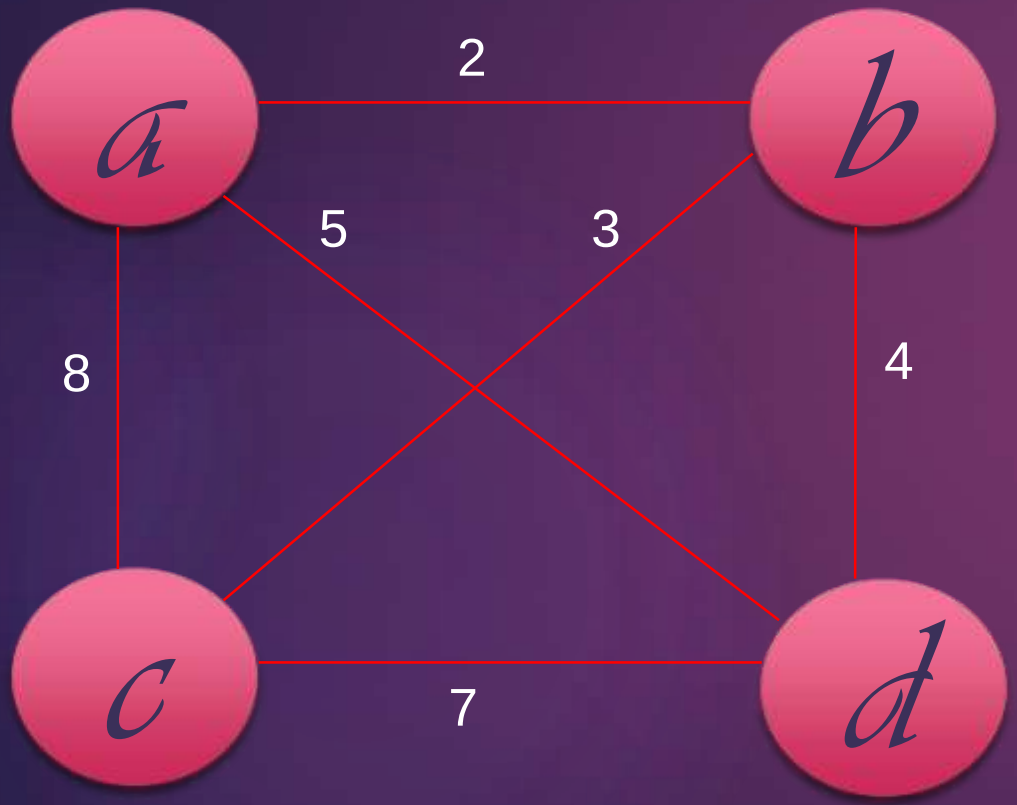
Alternative way to state the problem:

Find the shortest Hamiltonian Circuit in a weighted connected graph

History and Relevance

- The Travelling Salesman Problem was mathematically formulated by Irish Mathematician *Sir William Rowan Hamilton*.
- It is one of the most intensively studied problems in optimization
- It has applications in logistics and planning.

Example



Tour	Length
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

Efficiency

1. The Exhaustive Search solution to the Travelling Salesman problem can be obtained by keeping the origin city constant and generating permutations of all the other n – 1 cities.
2. Thus, the total number of permutations needed will be (n – 1)!

KNAPSACK PROBLEM

The Problem

Given n items:

- weights: w_1 w_2 … w_n
- values: v_1 v_2 … v_n
- a knapsack of capacity W

Find the most valuable subset of 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

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

Knapsack Problem by Exhaustive Search

Efficiency

1. The Exhaustive Search solution to the Knapsack Problem is obtained by generating all subsets of the set of n items given and computing the total weight of each subset in order to identify the feasible subsets,
2. The number of subsets for a set of n elements is 2^n.
3. The Exhaustive Search solution to the Knapsack Problem belongs to $\Omega(2^n)$.

The Exhaustive Search solution to the Knapsack Problem belongs to $\Omega(2^n)$.

THE ASSIGNMENT PROBLEM

The 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.

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

Algorithmic Plan

1. Generate all legitimate assignments
2. Compute their costs
3. Select the cheapest one

Assignment	Cost
1, 2, 3, 4	9 + 4 + 1 + 4 = 18
1, 3, 4, 2	9 + 8 + 9 + 7 = 33
1, 4, 3, 2	9 + 6 + 1 + 7 = 23
1, 4, 2, 3	9 + 6 + 3 + 8 = 26
1, 3, 2, 4	9 + 8 + 3 + 4 = 24
1, 2, 4, 3	9 + 4 + 9 + 8 = 30

Etc.

The Assignment Problem by Exhaustive Search

Efficiency

1. The Assignment Problem is solved by generating all permutations of n.
2. The number of permutations for a given number n is n!
3. Therefore, the exhaustive search is impractical for all but very small instances of the problem.