Brute Force

Introduction

- ***Brute force*** is a straight forward 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.

- 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.

- 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.

Sorting Problem

- Brute force approach to sorting:

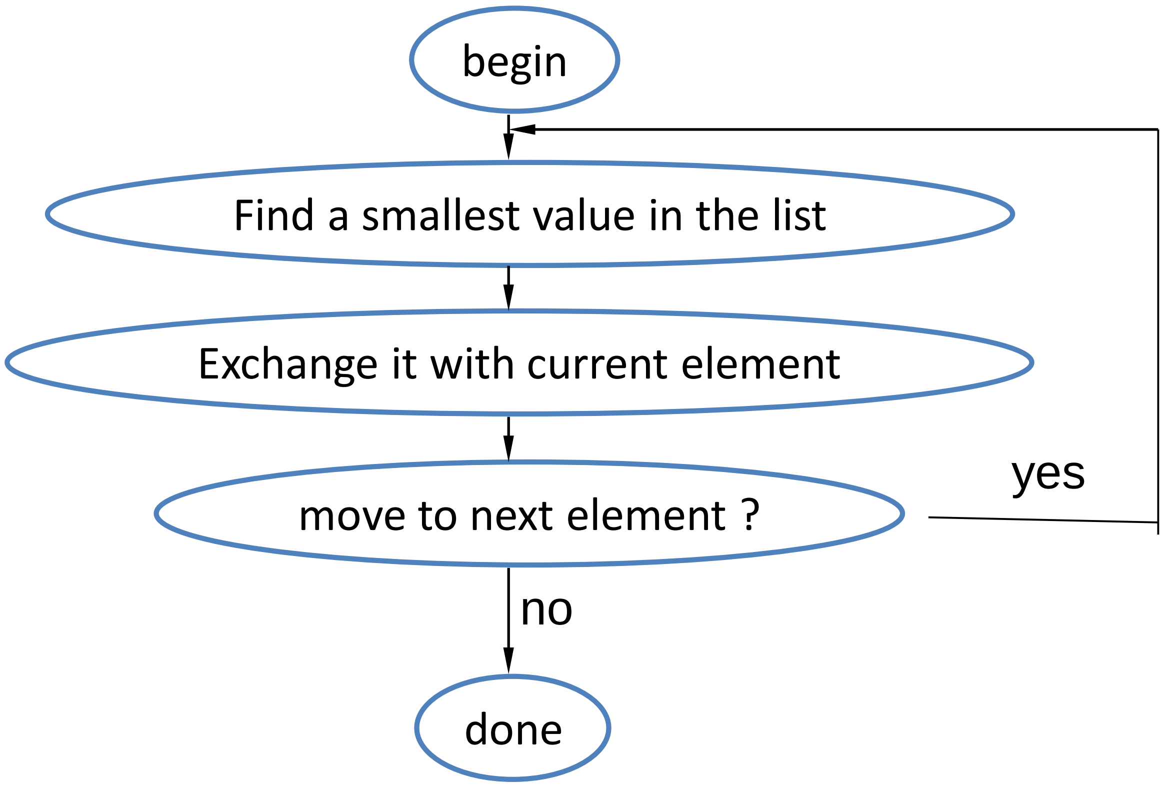
- Given a list of *n* orderable items (e.g., numbers, characters from some alphabet, character strings), rearrange them in non-decreasing order.

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


begin
Find a smallest value in the list
Exchange it with current element
move to next element ?
yes
no
done

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.
- The number 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 other sorting algorithms.

Bubble Sort

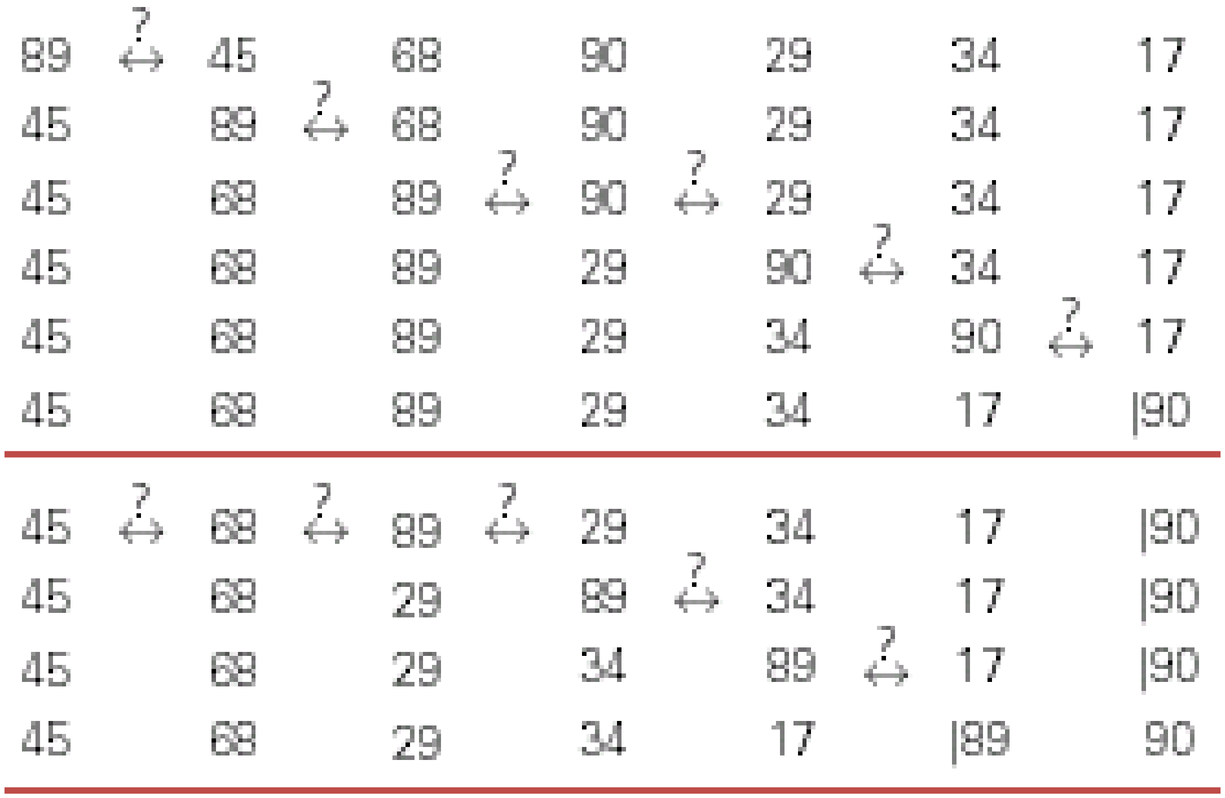
- Compare adjacent elements of the list
- and exchange them if they are out of order
- Then we repeat the process
- By doing it repeatedly, we end up ‘bubbling up’ the largest element to the last position on the list

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.



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.

Analysis

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

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

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

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

- 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.

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.
- However 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.

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

- *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

Examples of Brute-Force String Matching

1. Pattern: 001011

   Text: 10010101101001100101111010

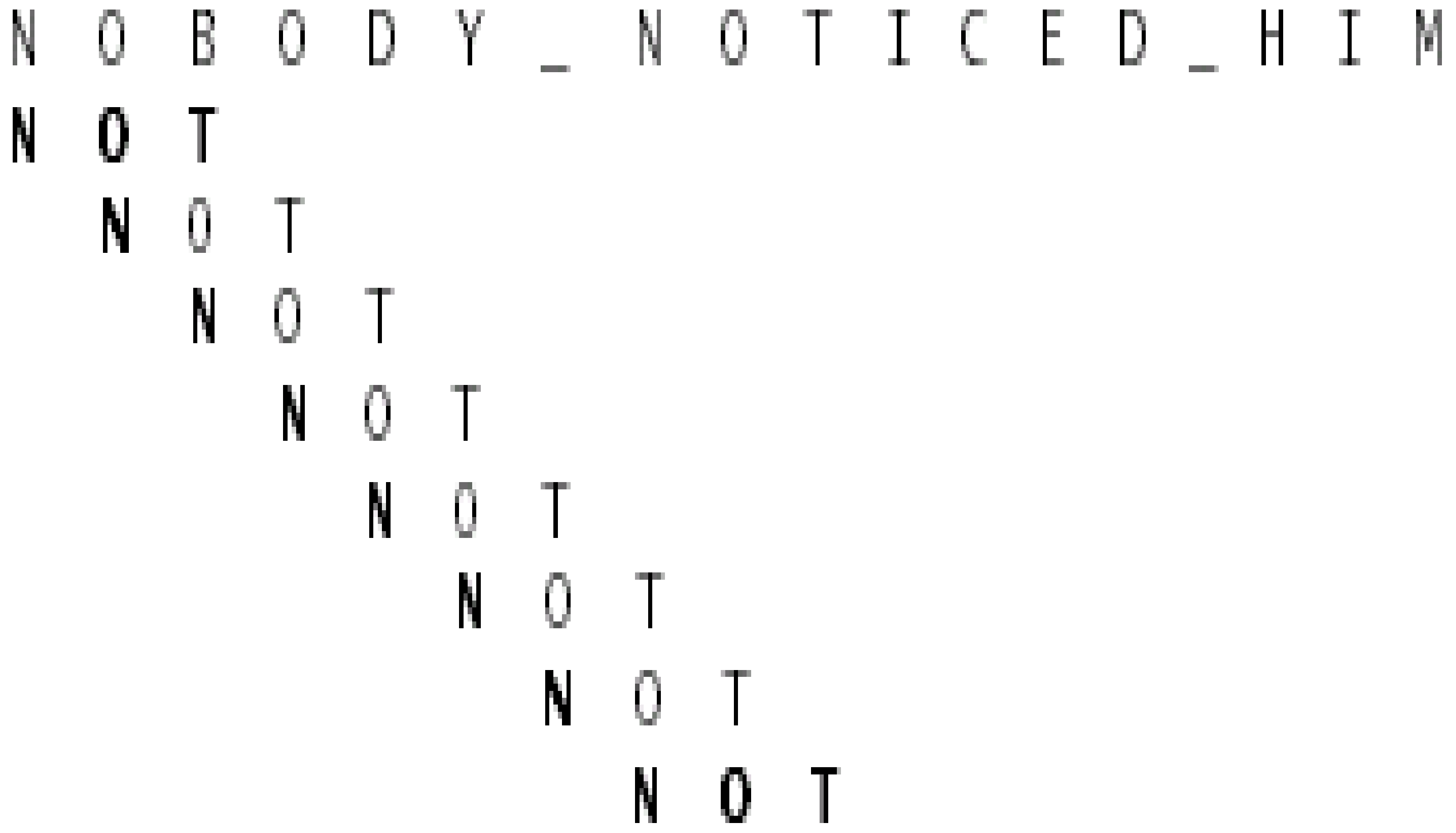
2. Pattern: happy

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

Pseudocode

```
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 $-1$
```

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
```

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 $\theta(nm)$.

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.

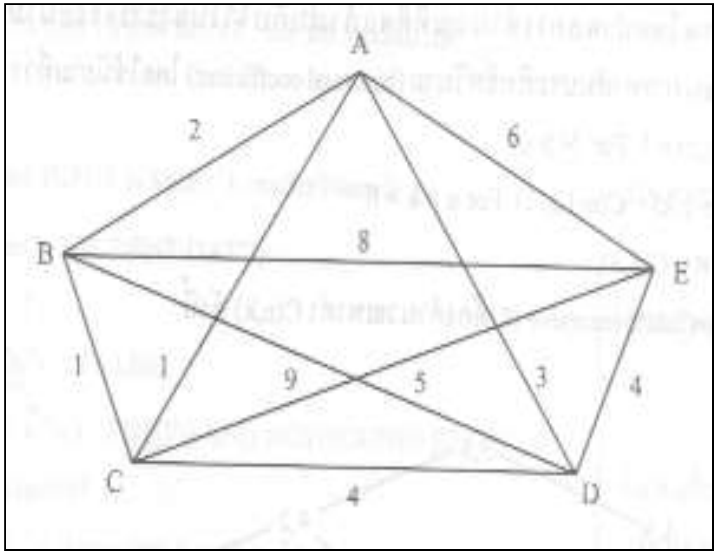
- The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest ***Hamiltonian circuit*** of the graph.

- A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.

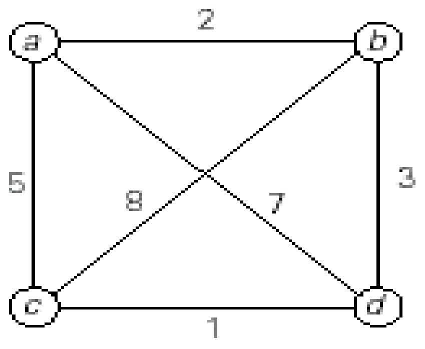
Introduction

All possible paths are proportional to **(n-1)!**

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	