PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Brute Force: Selection Sort

Dr. Shylaja S S

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.

The "force" implied by the strategy's definition is that of a computer and not that of one's intellect. "Just do it!" would be another way to describe the prescription of the brute-force approach. And often, the brute-force strategy is indeed the one that is easiest to apply.

As an example, consider the exponentiation problem: compute a^n for a given number **a** and a nonnegative integer **n**. Though this problem might seem trivial, it provides a useful vehicle for illustrating several algorithm design techniques, including the brute-force approach. (Also note that computing a^n mod m for some large integers is a principal component of a leading encryption algorithm.) By the definition of exponentiation, a^n = a*…* a. (n times)

This suggests simply computing a^n by multiplying **1** by **a** n times.

*In computer science, brute-force search or exhaustive search, also known as generate and test, is a very general problem-solving technique and algorithmic paradigm that consists of:

- systematically enumerating all possible candidates for the solution
- checking whether each candidate satisfies the problem's statement

A brute-force algorithm to find the divisors of a natural number n would

- enumerate all integers from 1 to n
- check whether each of them divides n without remainder

A brute-force approach for the eight queens puzzle would

- examine all possible arrangements of 8 pieces on the 64-square chessboard
- check whether each (queen) piece can attack any other, for each arrangement

The brute-force method for finding an item in a table (linear search) checks all entries of the table, sequentially, with the item.

*https://en.wikipedia.org/wiki/Brute-force_search

A brute-force search is simple to implement, and will always find a solution if it exists. But, its cost is proportional to the number of candidate solutions – which in many practical problems tends to grow very quickly as the size of the problem increases (Combinatorial explosion)

Brute-force search is typically used:

- when the problem size is limited
- when there are problem-specific heuristics that can be used to reduce the set of candidate solutions to a manageable size
- when the simplicity of implementation is more important than speed

Selection Sort

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, to find the smallest among the last n - 1 elements and exchange it with the second element, putting the second smallest element in its final position. Generally, on the i^{th} pass through the list, which we number from 0 to n - 2, the algorithm searches for the smallest item among the last n-i elements and swaps it with A_i.

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

in their final positions the last n – i elements

After n - 1 passes, the list is sorted.

Here is a pseudocode of this algorithm, which, for simplicity, assumes that the list is implemented as an array.

```
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]
```

As an example, the action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated in Fig. 1.

| 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

Fig. 1: Example of sorting with selection sort. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list 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.

The analysis of selection sort is straightforward. 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 the following sum:

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

Selection Sort is a $\Theta(n^2)$ algorithm on all inputs.