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:

Space efficiency:

Stability:

- 89, 45, 68, 90, 29, 34, 17

| 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

Bubble Sort

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 nondecreasing 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 $A[j+1]$

a. What is the time efficiency of the brute-force algorithm for computing a^n as a function of n? As a function of the number of bits in the binary representation of n?

b. If you are to compute $a^n \bmod m$ where $a > 1$ and n is a large positive integer, how would you circumvent the problem of a very large magnitude of a^n?

ALGORITHM *Mystery*(n)

```
//Input: A nonnegative integer n
S ← 0
for i ← 1 to n do
    S ← S + i * i
return S
```

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

```
//Input: An array A[0..n − 1] of n real numbers
minval ← A[0]; maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] < minval
        minval ← A[i]
    if A[i] > maxval
        maxval ← A[i]
return maxval − minval
```

ALGORITHM *Enigma*($A[0..n-1, 0..n-1]$)

```
//Input: A matrix A[0..n − 1, 0..n − 1] of real numbers
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i, j] ≠ A[j, i]
            return false
return true
```

Polynomial evaluation

Algorithm *BruteForcePolynomialEvaluation*($P[0..n]$, x)
//The algorithm computes the value of polynomial P at a given point x
//by the "highest-to-lowest term" brute-force algorithm
//Input: Array $P[0..n]$ of the coefficients of a polynomial of degree n,
// stored from the lowest to the highest and a number x
//Output: The value of the polynomial at the point x
$p \leftarrow 0.0$
for $i \leftarrow n$ **downto** 0 **do**
  $power \leftarrow 1$
  for $j \leftarrow 1$ **to** i **do**
    $power \leftarrow power * x$
  $p \leftarrow p + P[i] * power$
return p