1. Develop an efficient algorithm to evaluate $x^{2^k}$ given the value of $x$ and $k$. State its time complexity.

First method: we will multiply $x$ with itself $2^k - 1$ times.

Pseudo code:

```python
float evaluate_x_2k (float x, int pow2k, {
    // pow2k = $2^k$; given to us
    prod = 1;
    for (i = 0; i < pow2k; i++)
        prod = prod * x;
    return prod;
}
```

Time Complexity: $\Theta(2^k)$ as the for loop (over i) runs $2^k$ times.

Second method: We can make this more efficient by observing that while $x^4 = x * x * x * x$, $x^4$ can also be
written as $x^4 = (x^2)^2$. Similarly $x^8 = x^{2^3} = (x^2)^8 = (x^4)^2 = ((x^2)^2)^2$.

$x^{16} = (x^2)^4 = x^8$.

$x^{16} = (x^8)^2 = ((x^4)^2)^2 = (((x^2)^2)^2)^2$.

To compute $x^{2k}$, you successively compute

$k$ squares.

Pseudo-code

```python
float evaluate_x_2k(float x, float pow2k)
{
    float k = log2(pow2k);
    float prod = x;
    for (i = 1; i <= k; i++)
        prod = prod * prod;
    return prod;
}
```

Eg: for $x^{25}$, prod = $x$.

$i = 1 \rightarrow$ prod = $x^2$

$i = 2 \rightarrow$ prod = $(x^2)^2 = x^4$
Time Complexity: $\Theta(k)$ — much faster than the first method.

2) Develop an algorithm to locate the maximum element of a "unimodal array". A unimodal array is an array whose elements are in increasing order until they reach a maximum value. Thereafter, the value of the elements is in strict decreasing order. For simplicity, we will assume that all elements in the array are unique.

Example of a unimodal array:

$10, 20, 30, 40, 50, 49, 27, 18, 9, 4, 0.$

$5, 17, 16, 11, 9, 6, 2, 0, -5, -18$

The following arrays are not unimodal:

$5, 15, 20, 18, 35, 40, 41$
Algorithm: The naive method to locate the maximum will require $\Theta(n)$ time given an array with $n$ elements. But we can do better — in fact $\Theta(\log n)$! How? We observe that if the maximum value of the array occurs at index $i$, then $\text{arr}[i-1] < \text{arr}[i]$ and $\text{arr}[i+1] < \text{arr}[i]$. On the other hand, for any index $j$ which does not contain the maximum value, one of the following two conditions must hold true:

1. if $j < i$, then $\text{arr}[j-1] < \text{arr}[j] < \text{arr}[j+1]$
2. if $j > i$, then $\text{arr}[j-1] > \text{arr}[j] > \text{arr}[j+1]$

The reason for 1 is that the elements are in strict increasing order from index 1 until index $i$ (which contains the maximum value). The reason for 2 is that the elements are in strict decreasing order from index $i$ till the last index of the array.
You can make use of this information within a "binary search" type of algorithm.

Compute the midpoint of the array. Let the midpoint index be "k". If \( \text{arr}[k-1] < \text{arr}[k] \) and \( \text{arr}[k] > \text{arr}[k+1] \), we know we have located the maximum. If \( \text{arr}[k-1] < \text{arr}[k] < \text{arr}[k+1] \), then we are in the "left" (or the "increasing" portion) of the array. So we know that the maximum must lie after index \( k \). If \( \text{arr}[k-1] > \text{arr}[k] > \text{arr}[k+1] \), then we are in the "decreasing portion" (or the "right" portion) of the array. So we know that the maximum must lie before the index \( k \).
int max_unimodal (int * arr, int n) 
{
    int left = 0, right = n-1;
    int mid = floor(left + right) / 2;

    while (left <= right)
    {
        if (arr[mid-1] < arr[mid] &&
            arr[mid+1] < arr[mid])
        {
            return mid;
        } else if (arr[mid-1] < arr[mid] &&
                    arr[mid] < arr[mid+1])
        {
            left = mid + 1;
        } else // if arr[mid-1] > arr[mid] &&
                // arr[mid] > arr[mid+1]
        {
            right = mid - 1;
        }
    }
    return SOMETHING_IS_WRONG;
} // maybe array was not unimodal
Quick sort

Quick-sort is a very efficient sorting algorithm. It is much faster than bubble sort / insertion sort / selection sort. The quick-sort algorithm proceeds as follows: From the array, pick any one element called as the "pivot." Rearrange the array such that all elements less than or equal to the pivot are placed before the pivot, and all elements larger than the pivot are placed after the pivot. For example, consider the array:

76 8 65 91 20 2 41 57

With "57" as the pivot. After rearranging, the array looks like:

8 20 2 41 57 76 65 91

Note that in this re-arranged array, the pivot "57" is in the correct place. That is, in a sorted version of this
array, 51 would have to be placed at index 5, just as it is right now after the re-arrangement. Thus we now have two sub-arrays 8, 20, 241 and 76 65 91 which can be independently sorted. So we run the same afore-mentioned procedure on these two sub-arrays. Two different pivots can be chosen for these sub-arrays. For example, we will choose 20 as a pivot for 8, 20, 2, 41 and 76 as a pivot for 76, 65, 91. This will yield after re-arrangement:

8 2 20 41 57 65 76 91

Sub-arrays to be sorted.

Sub-arrays of length 1 need not be "sorted". The sub-array 8, 2 can be sorted using the afore-mentioned procedure recursively to yield 2 8 20 41 57 65 76 91.
The main example: By default in this example, the first element of the subarray will be taken as pivot.

76 54 98 7 102 67 19 200

54 7 67 19 76 98 102 200

7 19 54 67 76 98 102 200.

done.

Any number can be chosen as pivot. The accuracy of the algorithm is not affected. But the speed of the algorithm is affected. If the pivot always divides the array into two parts of equal size, the time complexity is \(O(\log n)\). If the pivot divides the array into one very small + one much larger part (skewed division), the complexity will be \(O(n^2)\).