1. Time complexity of the module rithm(n) is polynomial. For every iteration carried out by the loop of rithm(n), algo(i) gets executed i times. Therefore, the time complexity is:

$$\frac{n}{2} \times n = \frac{n^2}{2}$$

2.

Exponential	10	100	1000
2^{n+1}	2048	2.535e+30	1.072e+301

Factorial	10	100	1000
n!	3628800	9.333e+157	4.024e+2567

Polynomial	10	100	1000
$(n-1)^3$	729	970299	997002999
n^2	100	10000	1000000
$n^2/(2n+1)$	4.762	49.751	499.75
$n^{1/2}$	3.162	10	31.623

Linear	10	100	1000
n(n+1)/2	55	5050	500500
3n	30	300	3000

Logarithmic	10	100	1000
$n\log_2 n$	33.219	664.386	9965.784
$\log_2 n^2$	6.644	13.288	19.932
$\log_2(\log_2 n)$	1.732	2.732	3.317
$\log_{10} n$	1	2	3

3.

a) The complexity of an algorithm is the algorithm's **running time**, and is usually calculated using the worst-case time complexity of the algorithm. The complexity of a problem refers to the **efficiency** for which a problem can be solved, and is typically calculated based on the lower bound of any algorithm that solves said problem.

b) One of the possible solutions is to always visit the nearest neighboring city. First, choose a random city and then look for the closest unvisited city and go there. When all cities are visited, return to the starting city.

c) The time spent travelling between each city is also another factor affecting the time complexity.

4.

Fermat's theorem on sums of two squares: If any prime number n is of the form 4k+1, then n can be written as x^2+y^2 for some x, y ∈ I. This means if n is of the form 4k+3, then $x^2+y^2=N$ doesn't have any solution. Algorithm: Let n = 4k+1. Find $a^2 \equiv -1 \pmod{n}$. Apply the Euclidean algorithm with n and a. The first two remainders that are less than the square root of p are x and y. Complexity: Polynomial. Currently, the smallest complexity of one of the solutions is $O(n^2)$.

Source: https://doi.org/10.2307/2323912

5.

```python
def SubsetsSum(arr, n, v, sum):
    if (sum == 0):
        for value in v:
            print(value, end=" ")
        print()
        return
    if (n == 0):
        return
    SubsetsSum(arr, n - 1, v, sum)
    v1 = [] + v
    v1.append(arr[n - 1])
    SubsetsSum(arr, n - 1, v1, sum - arr[n - 1])
```

Time complexity: Exponential ($O(2^n)$).