Question 1

Polynomials may generally be represented in the form:

$$P(x) = \sum_{i}^{n} p_i x^i$$

A sparse polynomial is one in which the coefficient p_i is zero for many of the terms in the polynomial. The coefficient representation of such polynomial is inefficient as many values of the list will be zero. We will utilise a "sparse representation" S as an array of exponent-coefficient pairs (i, p_i), in an attempt to reduce wasted space.

For example, the polynomial:

$$P(x) = 4x^2 + x^7 + 15x^{11} + 6x^{28} + 2x^{31}$$

May be represented by the matrix:

$$\begin{bmatrix} 2, & 7, & 11, & 28, & 31 \\ 4, & 1, & 15, & 6, & 2 \end{bmatrix}$$

You are given an integer k and a sparse representation $S[n][2]$ for a sparse polynomial $P(x)$ that has degree $\leq 2^n$. Note: The values of S may not necessarily be in order.

1.1 Design a simple algorithm that evaluates $P(k)$ in $O(n2^n)$ and describe it in English.

For each element of the array, multiply k by itself i times to get k^i and add up all the $p_i k^i$ terms. Calculating k^i takes $O(2^n)$ and we do this for each element so the time complexity of the algorithm is therefore $O(n2^n)$

- **[1 mark]** Correctly works out how each element in the polynomial is summed in the loop.
- **[1 mark]** Provides justification for time complexity.

1.2 Describe an improved version of your algorithm from **1.1** that is able to compute $P(k)$ in $O(2^n)$.

How can you re-use values of k^i that you have already computed?

We sort the pairs (i, p_i) in the matrix according to the value of i. We initialise a pointer to the first value of i in our matrix and then create a loop to calculate powers of k from k^0 to k^{2^n}. At each power of k we check if the power corresponds to the value of i we are pointing at. If so, we may multiply our current power of k by p_i, add this to our total, and increment our pointer.

We can sort the matrix in $O(n \log n)$, we calculate all the powers of k in $O(2^n)$, and we calculate the sum of all relevant $p_i k^i$ values in $O(n)$. The time complexity is therefore $O(n \log n + 2^n + n) = O(2^n)$.

- **[1 mark]** Sorts the array in reasonable time complexity.
- **[1 mark]** Correctly uses previous exponent values to calculate current exponent value.
- **[1 mark]** Provides justification for time complexity.

1.3 Design a *divide-and-conquer* algorithm that is able to solve this problem in $O(n^2)$.

Can you find a faster way to calculate the value of exponents?

We will use a *divide-and-conquer* technique referred to as "binary exponentiation" to reduce the time complexity for calculating a given power of k from $O(2^n)$ to $O(n)$. It is important to notice the following idea:

$$x^n = \left\{ \begin{array}{ll} 1 & \text{if } n == 0 \\ x^{(n/2)^2} & \text{if } n > 0 \text{ and n is even} \\ x * x^{((n-1)/2)^2} & \text{if } n > 0 \text{ and n is odd} \end{array} \right\}$$

Calculating the value of x^n takes $O(logn)$ multiplications. Since the powers can be up to 2^n, it will take $O(log2^n)$ for each entry in the matrix. Thus, the time complexity is $O(n * log2^n)$ $= O(n^2)$.

- **[2 marks]** Uses the recursive approach for binary exponentiation to calculate powers.
- **[1 mark]** Provides justification for time complexity.
- Faster solutions to calculate powers (2^k-ary method, sliding-window method, etc.) are also accepted.

1.4 Inspired by our time saved in **1.2** we again try to reduce the time complexity by storing the powers of k so that we do not have to recompute them. Will storing the values of powers improve the time complexity of our $O(n^2)$ algorithm?

One approach is to calculate the values of x^{2^i} to x^{2^n}, where $i = 0..n.$ in $O(n)$ and store it in a hash table or similar. Looking up the values will take $O(1)$ for the corresponding exponent to be calculated. Each exponent calculation will again take $log(2^n)$ multiplication, so this is still $O(n^2)$ overall, and this approach does not improve the time complexity. A faster way to calculate the exponents is a sure-fire way to improve the overall time complexity.

- **[1 mark]** Explains how to store the powers and pre-compute values still doesn't improve the time complexity.
- **[1 mark]** Provides justification for time complexity, including lookup time.
- Any faster improvements.