Part 1

- Let PS(x) be the list of all sublists of natural list x, with each sublist folded over the sum operation, such that, given some natural n, PS(x)[n] is the nth element of PS(x), well ordered as if the nth element of x was the nth power of 2 before each sublist was folded over the sum operation
- **NOTE: To clarify what "folded over the sum operation" means, here is the list [1, 2, 3] folded over the sum operation in pseudocode: "[1, 2, 3].fold(sum) = 1 + 2 + 3 = 6"**
- **NOTE: To clarify, PS(x) is the list of all sublist sums of x, well ordered as if each element of x was a unique power of 2**
- **NOTE: To clarify, "well ordered" means smaller naturals are always before larger naturals. This does not well order PS(x), unless each element of x was well ordered and much larger than the previous element. However, in this proof, x is always unordered, therefore PS(x) is always unordered**
- Let a "valid power key" be a natural such that, for some list x, for all natural n, PS(x)[n ⊕ (the valid power key of PS(x))] is the nth largest element of PS(x)
- **NOTE: ⊕ is the Boolean exclusive or operation. If you apply ⊕ against some natural x to every natural from 0 (inclusive) to 2^n (exclusive), those naturals are reordered such that every unique x gives a unique order**
- **NOTE: Deciding the valid power key that works for all elements of PS(x) is the same as well ordering PS(x). This is because PS(x)[n] is the nth element of PS(x), unordered, and PS(x)[n ⊕ (the valid power key of PS(x))] is the nth element of PS(x), well ordered, so having the valid power key that works for all elements of PS(x) means you effectively have a well ordered PS(x)**
- **NOTE: If all elements of PS(x) are unique, there is only 1 valid power key for PS(x). Again, 1 valid power key works for all elements of PS(x)**
- Let A be an unordered natural list, given as input
- Let KEY be a natural, given as input
- Let the decision problem be "Given unordered list A as input and natural KEY as input, is KEY not the valid power key of A?"
- A deterministic polynomial time verifier can verify a YES solution to the decision problem if list A, natural KEY, natural x, and natural y are given, such that (x < y) ≠ (PS(A)[x ⊕ KEY] < PS(A)[y ⊕ KEY])
- If a deterministic polynomial time verifier exists for a YES solution to a decision problem such that all deterministic Turing machines deciding it must run in superpolynomial time, P ≠ NP
  - If the decision problem can't be solved in polynomial time, P ≠ NP
  - If the decision problem can be solved in polynomial time, see part 2

Part 2

- It's implied that algorithm ALGORITHM exists such that ALGORITHM can determine if a power key is invalid or not in polynomial time
- **NOTE: If ALGORITHM is polynomial time for a YES solution to a decision problem, ALGORITHM polynomial time for a NO solution to a decision problem, and vice versa**
- If ALGORITHM exists, deterministic polynomial time verifier V exists such that V can verify if a power key is valid for any set of subsets and also determine if that power key is even (YES) or odd (NO)
- Let M be some deterministic time Turing machine such that M, given only A, decides the power key of A, then determines if it's even (YES) or odd (NO)
  - Any such deterministic Turing machine runs in superpolynomial time. Otherwise, it could sort a set of subsets without looking at every subset, which is a logical contradiction
- It is implied that V can verify M's superpolynomial decision problem in polynomial time, given A and the power key of A, using ALGORITHM, therefore, P ≠ NP