Q1-

1- I have designed a recursive algorithm to find three primes which add up to a given number n > 5, or return (0, 0, 0) if none exist.

It goes as follows:

I have initially defined a helper function: { findPrimes: int → bool} to check whether a number is prime or not.

It begins by defining another helper function: {smallestDivisor: int*int → int} which checks if we have any divisor less than the square root of the given number.

If we do not find any divisor of the given number less than its square root other than 1, we can conclude that the given number is a prime number.

After defining this, I have defined the recursive algorithm to find three primes which add up to a given number n > 5, or which returns (0, 0, 0) if none exist.

For this I have defined a function {findPrimes (int→ int*int*int)} which finds the three primes if possible.

It goes as follows:

It finds the three numbers by fixing any one number(a, init= 2) and then increases the value of another number(b, init =2) by 1 and subsequently changes value of (c= n-a-b) and also by the same time it checks if all these numbers are primes by the above defined isPrime function. If we get possible values then it returns those values(a,b,c) as answer else it returns(0,0,0).

Let's prove the correctness

Base Case:

n = 5

findPrimes(6) =

a=2, b=2, c = 4 -> false

a=2, b=3, c = 3 -> true

Hence base case n = 6 can be represented as a sum of three primes.

Induction Hypothesis:

For 5 < k < n, (for any n, k ∈ N) let's assume that k = a + b + c where a, b, c are any three prime numbers.

Induction Step:

For any n ∈ N, we have to show that n = a + b + c where a, b, c are prime numbers.

We have suppose; (n-a) as another number where a is any prime number.

Then, since n >5 ⇔ n-a >3 and this is equivalent to saying b + c > 3.

Since we know that b and c are natural numbers, therefore b + c >=4.

b and c have to be primes. We will again use induction to prove this also:

Base case:

b + c = 4 ⇔ b= c =2; (only primes possible which sum up to 4 = 2 + 2)

Hence, base case is true.

Induction Hypothesis:

b + c = k for: 4 < k < n; we have b and c are primes.

Induction Step:

For b + c = n (for any n, k ∈ N), we have suppose c is a prime number, then we can surely find another prime number satisfying the above equality.

Hence any b + c >=4, can be expressed as in a way such that both b and c are primes.

Therefore, we can say that b + c = n-a >=4 can be expressed as in a way such that both b and c are primes.

Hence, proved.

2- Analyzing the space and time complexity of our algorithm:

Time:

The function (isPrime) I've defined is finding if a number is a prime or not by making up to square root (n) calls.

T(n) = T(n-1)+1 …….. up to ($\sqrt{n}$): therefore O ($\sqrt{n}$)

So each time we use isPrime we are using O ($\sqrt{n}$) time.

Hence for 3 times we take O ($n^{1.5}$) time.

For the function findPrimes we are calling the functions in total n times or

T(n) = T(n-1)+1 …….. up to (n): therefore O (n)

we are taking O(n) time for computing the value of findprimes . This is because in total we compute a + b + c = n and for this we have to call 'a' suppose k times and call 'b' suppose l times and 'c' suppose m times and we are calling all these operations simultaneously, so time is maximum of [l, m, n] which is less than O(n).

Hence in total we have O ($n^{2.5}$) time taken.

Space:

The function isPrime is taking a O(1) space and our function findPrimes is taking O (n) space. Let's prove this:

isPrime is an iterative function that takes O(1) space because we constantly update the value of isPrime and also in smallestDivisor function we are using iteration to find the smallest divisor.

Coming to the function findPrimes , this function takes another O(1) space because we define a tail recursive function f(a,b,c) which calls itself again by updating the value of a, b, c till we get the value equal to n.
Hence the overall space complexity of the problem is O(1).

3- The algorithm has been successfully implemented in sml.

Q2-

1- For the problem, I've designed an algorithm that maximizes the total value, without exceeding the total weight W. To do so, I have initially declared a function max (int* int-> int) which computes the maximum of any two integers given as input by using the > operator. After this, I have designed another function maximumValue [int * int*(int->int)*(int->int)-> int] that has four variables namely n, W, v, w and here W is the weight limit, n is the number of items.

Base case:

W = 0 for this the algorithm is correct as it gives the correct value 0 that can be carried by taking none of the items in the dikki.

Induction Hypothesis:

Let a given weight k < W is supposed to be carried and we assume we can get the maximum value for total weight <=k.

Induction Step:

For a weight limit W we have:
As the computation takes place, let's call i' as the item that breaks the weight limit i.e. W-w(i') <0 then for this case our algorithm rejects this value and adds a 0 to the answer and it stops. For all i <i' we have W-w(i) >=0 and now we will check if our algorithm gives the maximum value that can be taking. Since we are recursively calling our algorithm to find max of (a, b), each time we have two options either to choose a value of not. We compare the value without choosing an item p to the case with choosing p. Whichever case gives us the greater value, we add that to our answer and decrease the weight permitted by w (p). Hence we only maximize our value by choosing max value weights. So our algorithm is working correctly for any weight limit W.

2- Time Complexity

The time complexity of algorithm is clearly O (2^n) because for each value of n we have two options either choose it or not. Formally:

T (n) = 2*T (n-1)

…….. T (n) = 2^n*T(0) if T(0) = c then

T (n) = c*2^n => **O(2^n)**

Space Complexity

Our algorithm recursively makes calls till n = 0 and in each call it occupies a unit space in the system. Therefore, the space complexity of our algorithm is **O(n).**

3- The code is implemented in SML successfully.

Q3-

1- I have designed a recursive algorithm to convert an integer n of the smallest unit to a human readable string using the name and factor functions. Firstly I've defined a function toString (int->string) which converts any number to a string. For doing that I've defined a string value for every number from 0-9. Then I have used div and mod operators to divide and concatenate the further digits. Hence, any number is changed to string by this function. After this I've defined another couple of function namely convertUnitsRec and convertUnitsIter. These functions are used to convert a raw measurement expressed in one unit into a human-readable form using a combination of large and small units. The first one is based on recursion whereas the second one uses iteration to perform the conversion.

The function convertUnitsRec has three variables namely n, name, factor. n is the input value name is the first operation and value factor is the second operation that is supposed to be specified by the user to convert the measurement expressed in one unit into a human-readable form. In each recursive call I change the value of n to n div factor(j) and concatenate the answer with the value of name factor at each call. I have also called the function convert to string to the value: n mod factor(j) and also concatenated the whole answer with name(j). Hence this recursive algorithm gives the desired output.

Now coming to iterative algorithm. I have designed it in a way that it again has three variables namely n, name, factor. This function has another local function iter that has three parameters n, j, ans. Each time the function is called the algorithm updates the allotted space to answer by concatenating the previous answer. Rest algorithm is similar to convertUnitsRec algorithm.

2- Time Complexity:

The above two algorithm make a call and then the value of n is changed to n div factor(j). So the time complexity is given as:
Using the fact that each computation of name and factor takes O(1), we have:
T(n) = 2*O(1) + T(n div factor(j)) = ….. =2*O (log (n) = O (log n)
Now, according to question the size of concatenation (^) is given as O (n + m) where n and m are sizes of strings to be concatenated. For each computation we concatenate three times. Therefore the total concatenation time + computation time = 3*O(log n) which is same as writing time complexity of the algorithm is **O(log n).**

3- The algorithm has been implemented in SML successfully.

Q4-

1- I've designed a fast iterative algorithm to compute the square root of a given integer. It works as follows. I have firstly defined a power function to compute the value of 4^p.This takes constant time as we have specified an upper bound to the value of p which is the largest power of 4 that divides the n in such a way that n div 4^p >1.
To find that largest power I have made a function g(n, k) which finds the largest power of k that divides the n. It uses the div operator to compute.
After this I have defined a function intSqrt2(n) that I have used to compute the square root of n iteratively. To do so, I have defined a local helper function f(n, p, i) which checks if $(2i + 1)^2$ > n div 4^p ; if this is true the a call is made for f(n, p-1, 2i). If this is false it iteratively call for another set of values of f(n, p-1, 2i-1). What is happening is that we are constantly updating the value/space allocated to i to 2i or 2i + 1; depending on the case mentioned above. Initially the value of i = 0 and the value of p is the largest power of 4 that divides the n in such a way that n div 4^p >1. Then we successively change the value of n to (n div 4^p) and then iteration is used to update the value of i to nearest possible square root.

Base Case:

n = 5 then largest power of 4 that divides 5 is 1 i.e. p = 1;

Then we see $(2 * 0 + 1)^2$ >1 is false

so we move to the else definition which updates value of i to 1 and change value of p = 0 and value of n to 5 div 4 = 1

Now the value of $(2 * 1 + 1)^2$ > 1 is true so the value of i is now changed to 2 and p to -1 which is <0 so the first statement is used to execute the answer to 2.

Hence the nearest integer square root of 5 is 2.

Hence base case is true.

Induction Hypothesis:

For any k<n we assume this algorithm works correctly to give the right answer.

Induction Step:

Let's see the case for n

Suppose for (n-1) the value of p is l

Then we have two possibilities:

For this, when we compute n div 4^p we get a value of suppose q. Now q < n for all n and n div 4^p is also a possible value of k and the algorithm is true for all k. Hence proved.

2- Time complexity

The time complexity can be calculated as follows.

For the computation of 4^p the time complexity is

T(n) = 1 + T(n div 4) = O(log n)

Now for computing the time taken to calculate the answer in intSqrt2 we divide n by 4^p so for that we have already calculated the time which is O (log n). Hence total time taken is **O (log n).**

Space Complexity

Since, we can see clearly that this is an iterative process, so once we have assigned some space to a function we overwrite that space with the updated value of the argument taken into consideration. In this case we overwrite the space occupied by n with n div 4^p and p with p-1. Hence as a total we take a fixed space which is O(1).

3- The algorithm is implemented successfully in SML and it takes very less time to compute the value of big numbers like 400,000,000.