Question D - Daniyar Nazarbayev [H00204990].

1. Recursion is when a function calls itself. The function must have parameters, and the input parameter should change with each recursion call. By default, it will run endlessly until the memory stack overflow error. Stack is basically a part of the memory where all the temporary variables of the function are stored. Each function call is put in a single stack frame, but once the stack's size gets exceeded, it will give an overflow error. Because of that there will be a conditional statement in the code that will stop the recursion call, also known as the base case.

2. Tail recursion is when the recursion call in the function is the last part of the code. The recursion code is written in a way, so that the stack doesn't have to keep the data of each recursion call. Instead it can use a single stack frame for the whole recursion process. It pretty much creates infinite recursion.

   That is called tail recursion optimization and unfortunately not all programming languages support it. Java for example does not, but most functional languages do.

3. Imperative languages are based on the von Neumann computer architecture. They have variables, and they store these variables in memory addresses. Functional languages on other hand are based on lambda calculus, more specifically on the Church-Rosser theorem.

   Church-Rosser is a mathematical theorem proved by Alonzo Church and Barkley Rosser in 1936. It states that if we start from a term t and perform any two finite reduction sequences, there are always two more reduction sequences that bring the two back to the same term.

   Basically the outcome will be independent of the evaluation order or reduction strategy. It has become a base for every functional language.

4. I would use a low level language for the car chip due to the limited resources provided. First of all, the code doesn't have to be compiled. Obviously it will take up extra computation power, or will increase the time taken for any action, even the boot time. A low level language will not take up as much time, if any at all to convert to binary. Second of all, usually high level languages provide abstraction to the programmer, while taking a hit on the efficiency. That also means that some functionality is taken away from the programmer to make his life easier, like say storing data manually by using memory addresses. All of that is handled by the compiler, but this missing functionality is exactly why Java can't do tail recursion. With the amount of resources a car chip provides, one needs a language that makes 100% use of the hardware.

Languages that are fit for the job are either assembly, C or plain machine code. C is considered a high level language technically, actually any language that provides abstraction can be considered that, but C is far behind the functionality compared to what other high languages offer today.

As for the android app, I would use a high level language, probably object oriented. First of all, there are plenty of android phone manufacturers: Google, HTC, Samsung etc. All with different hardware, probably different CPU architectures. Different CPUs might mean different instruction sets, which might mean that if I were to write the code in low level, I would have to write 2 different pieces of code for each of the phones.

Second of all, OOP provides a very human readable code. It may not be as optimized as the one written purely in machine code, but it's very good for big projects with big teams.

In the case of an android app, the best choice would be Java, obviously since it's the officially supported language on their OS according to Google. It's quite surprising considering that the Android is based on the Linux kernel.

5. It means that there can't be anything the function could interact with, also known as side effects. Global state in programming is known as something that can be accessed by anything in the code. There are also other states known as mutable and immutable – basically meaning whether the variable can be reassigned. Functional languages don't have assignments, actually they shouldn't have. Unlike imperative functions, functions in a functional language always have parameters, and just like a mathematical function they give back an answer. They use recursion instead of iteration or loops.

   Without global state, the function should always provide the same result. That said I personally did face a challenge when working with SML. The problem with recursion is that with each recursive call I lose my original parameter value, and the only way to get it back to my knowledge is have a global variable as a backup.

   By the way, SML provides global state, meaning it's not a pure functional language.

6. I don't know much about either paradigms. I really only used one imperative language – Java, and it was a high level language. As for the functional language, I only used SML for a couple of weeks now – 4 to be exact, and I am not entirely sure of its capabilities and purpose.

In all honesty, when I was coding in java, all optimization was done by the compiler. There was no memory management, and looking at the amount of memory a java application would use in my task manager, I doubt the compiler or I did a good job. The garbage collection was automatic. In all honesty, I fail to see what kind of optimization I could do without going into machine code. The best bet is probably use the right data structure to use less resources and time.

I am not really sure whether you can implement some of the data structures that I done in java. Considering that to get the last element of the list, I have to traverse through the whole list, then the list definitely looks similar to linked list, while the tuple feels like an array. Just having these two will allow me to recreate at least half the stuff I did in java. I am not entirely sure on which paradigm would be more efficient, but I would place my bet on imperative.

That said, functional languages are after all known to be much higher level, even higher than OOP, and the general rule is that the higher the abstraction, the less efficient it is.