Supervision 4

1

a. The magic bytes representing the type of binary (in my case, an ELF binary)

b. `.text`, around `0x0000280`

c. `.rodata`, at `0x0000080`

d. `.data`

2

a. `//TODO`

b. They all contain infinite recursive loops with no degenerate case. Since the stack frame for each call is large, the stack overflows very quickly.

c.

i. For `large_01`, `-O1` likely removes the `arr` array since it is never used for anything, which is why the stack overflows more slowly. `-O2` likely performs tail call optimisation, the stack space is constant with respect to the number of recursive calls.

ii. `large_02` is probably treated very similarly to `large_01`, except that `-O1` cannot remove `arr`, as it is used later in the function body. Nonetheless, `-O2` still performs tail call optimisation.

iii. For `large_03`, tail call optimisation will not prevent the crash, as the large array is being passed from one iteration to the next, creating a new one each time. Therefore each has to remain on the stack. `-O3` likely performs some optimisation whereby the argument `p` does not need to be copied from caller to callee as it is never used by the latter.

3

a. Some programming languages require garbage collectors because the programmer is not given direct access to memory. This can be because the language designers want to abstract away the complexities of low-level memory management, or that they want code to be less prone to memory leaks. In either case, the programmer themself does not have the ability to implement garbage collection, and so an automatic garbage collector is required to free up memory which is no longer being used.

b. Garbage can be considered any data stored at a location in memory such that there is no way for the program to access it. For example, a Java object with no references to it. These can be located via reference counting (e.g., incrementing a counter every time a new reference to the data is created, and decrementing it when one is lost), and so any data in memory whose reference count is zero can be considered garbage.

An alternative method is *mark and sweep*, whereby a breadth-first search is conducted from a set of root data, marking all of the data which can be reached by following references therefrom. After the marking phase, all data which has not been marked can be considered garbage.

c. Yes. The programmer needs to make a decision about how the reference counts are stored. While it will almost certainly be unsigned, a larger data type might make overflows less common (as there can be orders of magnitude more references before an overflow occurs) but will also greatly increase the memory overhead required by the garbage collector. On the other hand, a small data type might reduce the memory overhead, but will decrease the maximum number of references a datum can have before an overflow occurs.

d. One benefit of this optimisation is that the number of function calls required to evaluate this expression goes down from 4 to 3. This means less overhead from stack management and register saving/restoring.

However, this is not always correct. This is particularly easy to see if `map` has side effects. For example, printing to the console. Without the optimisation, `map` will print to the console twice, but with the optimisation, `map` will only print to the console once.

4

a. Yes, we do need to define a fixed type for exceptions. If we used an existing type for expressions, then there would be no way to differentiate `e` returning value `v`, from `e` raising value `v`.

b.

```
type exception =
    | ArithmeticException of stacktrace
    | OutOfMemoryException of stacktrace
    | / * Etc. */;;
```

Then, the expression `raise e` would have type `exception`, and `e` would represent one of the above constructors.

This would allow `raise e` to be returned by an expression just like a normal value.

c.

i. Not valid unless `raise` is a monad.

ii. Not valid, as `e1` might have side effects.

iii. Valid, as any side effects `e` has will still be evaluated before `f`.

iv. Valid, as the catching function has no side effects. If `e` raises an exception, then the outcome is the same in both cases, and likewise if `e` does not raise an exception.