1

a) Shadowing

In a let binding when the more recent bindings (inner bindings) are called in place of identical outer bindings, it is called shadowing.
Example : In this, value a=10 shadows the a=3 value inside the nested let binding. let a=3;; outer scope of global declaration
let a=10 in let b=a in a+b;; nested let binding
int = 20

b)

Consider the following recursion programme,

```
def fact(x):
    if (x==0):
        return 1
    elif (x==1):
        return 1
    else
        return x * fact(x-1)
fact (3)
```

Advantages of Recursion

- Readability and elegance
- Lesser code, so less-prone to error

Disadvantages of Recursion

- Lot of memory is occupied
  Example : In a factorial function **fact(3)**, each stack frame is allocated everytime when the function returns control **i.e,** for **fact(2)**, **fact(1)**, till the base case **fact(1)** is reached. All these stack frames are stored in the call stack. When the base case is not specified, it leads to stack overflow
- May be difficult to debug in a complex programme
- Time complexity is more incase of recursions because the compiler ultimately converts the recursion into its equivalent loop during execution

c)

The given implementation of recursion is not efficient because, every call to max forms a binary tree and a stack frame is allocated which grows exponentially. Efficiency can be improved by,

```
let rec max = function
    [ ] → -1
| h :: t → let max t = (max t) in if h > ( max t ) then h else ( max t )
```

d)

Example to calculate the length of a list using recursion

```
let len_list_tr list =
    let rec helper a list = match list with
        | [ ] → a
        | _ :: t → helper (a + 1) t
    in helper 0 list
```

Tail Recursion:

Tail recursion is a special kind of recursion where the recursive call is the last call in the function. No other calls after this recursive call are made.

Tail call:

The call which corresponds to the tail recursion is called as a tail call.

Tail call optimization:

In programming languages like Ocaml, when the tail call is made, the compiler figures it out and reuses the stack frame allocated, instead of allocating subsequent stack frames for each recursive call. This is useful because it improves memory usage and doesn't let the stack to overflow.

e)

- A1.1 = 10
  Explanation: The value of **a** is globally declared as 10 which is returned to the function f().

- A1.2 = 20
  Explanation: As the value **a** is changed at global scope to 20, the value 20 is returned to function f().
  A2.1 = 10

- **Explanation:** The function gets the value of **a** in the global let binding, and **a** is assigned 10. This is because of static scoping.

- A2.2 = 10
  Explanation: Even though **a** is newly assigned to 20, the outer let binding's value of **a** is displayed. This is because the function f() is not declared as inner let binding to the newly declared let binding of **a**.