2017 Section A.

1. a) 1 * 2 * 3 / 4 * 5 - 6

= 6 / 4 * 5 - 6

= 1 * 5 - 6

= 5 - 6

= -1 int.

b) would result in unknown behaviour

- read would read whatever in memory address with offset 10 from a[0].
- write would either write successful to a unknown address or segmentation fault.

c)
```
void swap (int a[], int m, int n) {
    int t = a[m];
    a[m] = a[n];
    a[n] = t;
}
```

d)
```
void swapPairs (int a[], int length) {
    for (int i=0; i < length-1; i++)
        if (a[i] > a[i+1]) swap(a, i, i+1);
}
```

e)
```
void bubbleSort (int a[], int length) {
    for (int i=0; i < length-2; i++)
        swapPairs(a, length);
}
```

2. a)
1. 1.
2. False.
3. 100
4. 1.
5. Error. Num and Char.

b)
```
fac :: Int -> Int
fac x = x * (fac (x-1))
fac 0 = 1.
```

```
sumfac :: Int -> Int
sumfac x = (sumfac (x-1)) + (fac x)
sumfac 0 = fac 0.
```

c)

```
f :: [Int] -> [Int]
f xs = [x | x <- xs, cubed >= 200 && cubed <= 1000]
                                  where cubed = cube x.

cube :: Int -> Int
cube x = x * x * x.
```

d)

```
negSum :: [Int] -> Int
negSum xs = foldr (\x y -> x^2 + y) 0 (filter (<0) xs)
```

Section B

a) malloc allocating heap memory → pointer

free releasing heap memory from pointer / allocation

cohesive function - function that cannot be reduced.

variable scope - the code area where variable is in effect.

Syntax - the layout of words / alphabets a compiler would understand.

b)

```
char* copyStr (char* org) {
    int len = strlen(org);
    char* cpy = malloc(sizeof(char) * (len+1))
    for (int i=0; i<=len; i++) cpy[i] = org[i];
    return cpy;
}.
```

c). struct node {
int key
node * left
node * right
};

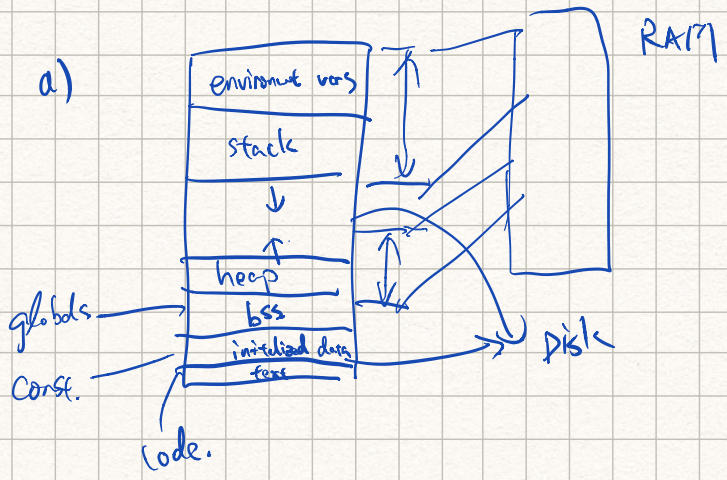
pointers pointing to left/right node

recursively going down from root left/right.

e.g root → key = 5;

root → right → left → key = 3.

a)



4. a) variable life time. : the time period when a variable is accessible.

variable definition : define data type (init value) for a variable.

file scope : global variable in a single C file is accessible with in the file unless use extern keyword.

deference : use pointer to retrieve the value in the mem. addr.

heap memory : a free section of mem that's dynamically allocatable.

b) i) it's printed backwards.

e.g 13 → 1101

but the program print 1011

the remainder is the last digit.

ii). void printAsBinary (int n) {

```
if (n==0) return;
printAsBinary(n/2);
printf("%d", n%2);
```

}

c). Physical memory is not always sufficient.

Programs / process have their own V Mem., which are mapped to phys. mem. / or disk (swap) when not active.

V Mem. provides standalone memory s.t. programs cannot access each other's mem.

d) bool. isSame (char* a, char*b) {.

```
int len_a = strlen(a);
int len_b = strlen(b);
if (len_a != len_b) return FALSE;
for (int i=0; i<len_a; i++) if (a[i] != b[i]) return FALSE;
return TRUE;
```

}.

Section C.

5.

a) higher order function.

a function that takes in a function as a parameter.

operator Section

partially put an operator and its parameter

e.g (+1) + should take 2 param

(+1) means \x -> x+1.

Type class

a set of datatypes with certain functions and characteristics

QuickCheck.

A library in Haskell used for automated random testing of function properties.

b) isPrime :: Int → Bool.

isPrime x = isPrimeRec x-1 x

isPrimeRec :: Int → Int → Bool.
isPrimeRec 1 x = True.
isPrimeRec d x = if x `mod` d == 0 then False
else isPrimeRec d-1 x.

c). iter :: Int → (a → a) → a → a
iter 0 f x = x
iter n f x = iter n-1 f (f x)

quadProd :: [Int] → Int.
quadProd (x:xs) = foldr (\x y → (iter 2 square x) + y) x xs.

d) Base case $2^0 = 1$.

Hypothesis Assume power2 $n = 2^n$. $n \geq 0$

WTS power2 $(n+1) = 2^{n+1}$ $n+1 \geq 1$.

power2 $(n+1) = 2 *$ power2 n ($= 2^n$ IH.)

$n+1 \neq 0$.

$= 2 * 2^n$
$= 2^{n+1}$.

∴ power2 n d gives 2^n. for $n \geq 0$.

6. lazy evaluation
Haskell only evaluates expressions when necessary

tail recursive function.

The recursive call is at the end of program
no other op is done after the return of a recursive dfn.

polymorphic type.

a type that could be two or more actual datatypes

lambda abstraction

an anonymous function. / used to conveniently insert functions.

b). length

```
prop_lengthR :: [a] -> Bool.
prop_lengthR xs = length. xs == length (reverse xs).

prop_lengthDouble :: [a] -> Bool.
prop_lengthDouble xs = 2* (length xs) == length (xs ++ xs).

sumAcc :: [Int] -> Int.
sumAcc xs = sumAccRec xs 0

sumAccRec :: [Int] -> Int -> Int
sumAccRec [] acc = acc
sumAccRec (x:xs) acc = sumAccRec xs acc+x.

prodInts :: IO(Int)
prodInts = do input <- getInt
              if input==0 thenreturn 1.
              else return input * prodInts
```