Memory management in C

1.) List all memory resources used by each of the following statements found inside a C function definition.

i. `char *info = "hello";`
ii. `int numbers[4];`
iii. `char *data = (char *) malloc(1024);`
iv. `char **bigdata = (char **) malloc(1024*sizeof(char *));`

i. One character pointer (`info`) resides on the stack; its contents point to a 6-byte string literal (`"hello"` with a null terminator) in the static data section.
ii. This creates a 4-element, 16-byte integer array on the stack; no pointers are created.
iii. Allocates a pointer on the stack; it points to the start of 1024-character block on the heap.
iv. Allocates a pointer on the stack; it points to the start of 1024-pointer block on the heap.

2.) Imagine you are writing an algorithm that uses a binary tree data structure with nodes defined as below. Since space for all the contents of this data structure was allocated on the heap, you need to de-allocate all of the associated memory; write an implementation of `free_tree` that successfully does this. You may assume that the tree is of reasonably shallow depth.

```
struct treenode {
    struct treenode *left_child;
    struct treenode *right_child;
    char *value;
};

void free_tree(struct treenode *root) {
    if (root != NULL) {
        if (root->value != NULL) free(root->value);
        if (root->left_child != NULL) free_tree(root->left_child);
        if (root->right_child != NULL) free_tree(root->right_child);
        free(root);
    }
}
```

3.) **Part I:** Can you find and comment the memory management bugs below? **Part II:** Fix them to the right.

```
#define LEN 64

int *do_things(int *data) {

    int vector[LEN];

    int *tmp = vector;
    // no sizeof(int), no NULL check
    int *values = (int *) malloc(LEN);
    // no sizeof(int), no NULL check
    int *result = (int *) malloc(LEN);

    // do math, populating result
    // NO FREE
    return result;
}
```

```
#define LEN 64

int *do_things(int *data) {
    int vector[LEN];
    int *tmp = vector;
    int *values = (int *) malloc(LEN*sizeof(int));
    if (values == NULL) return NULL;
    int *result = (int *) malloc(LEN*sizeof(int));
    if (result == NULL) {
        free(values);
        return NULL;
    }
    // do math, populating result
    free(values);
    return result;
}
```

Intro to MIPS

Instruction	Syntax	Example
add	`add dst, src0, src1`	`add $s0, $s1, $s2`
add immediate	`addi dst, src0, immediate`	`addi $s0, $s1, 12`
shift left logical	`sll dst, src, shamt`	`sll $t0, $s0, 4`
load word	`lw dst, offset(bAddr)`	`lw $t0, 4($s0)`
store word	`sw src, offset(bAddr)`	`sw $t0, 4($s0)`
branch if not equal	`bne src0, src1, brAddr`	`bne $t0, $t1, notEq`
branch if equal	`beq src0, src1, brAddr`	`beq $t0, $t1, Eq`
jump unconditional	`j jumpAddr`	`j jumpWhenDone`
jump register	`jr reg`	`jr $ra`

Translate each of the following C-code snippets into MIPS assembly. Use up to eight instructions for each segment, but limit the instructions used to those listed in the table above.

1. Assume `a` is held in `$s0`, `b` is held in `$s1`, `c` is held in `$s2`, and `z` is held in `$s3`.

```
int a=4, b=5, c=6, z;
z = a+b+c+10;
```

```
addi  $s0, $0, 4
addi  $s1, $0, 5
addi  $s2, $0, 6
add   $s3, $s0, $s1
add   $s3, $s3, $s2
addi  $s3, $s3, 10
```

2. Assume `$s0` holds `p` after `int *p = (int *) malloc(3*sizeof(int))` and `$s1` holds `a`.

```
p[0] = 0;
int a = 2;
p[1] = a;
p[a] = a;
```

```
sw    $0, 0($s0)
addiu $s1, $0, 2
sw    $s1, 4($s0)
sll   $t0, $s1, 2    # multiply by 4
addu  $t1, $t0, $s0
sw    $s1, 0($t1)
```

3. Assume `$s0` holds `a` and `$s1` holds `b`.

```
int a = 5, b = 10;
if (a + a == b) {
    a = 0;
} else {
    b = a - 1;
}
```

```
      addiu $s0, $0, 5
      addiu $s1, $0, 10
      add   $t0, $s0, $s0
      bne   $t0, $s1, else
      add   $s0, $0, $0
      j     exit
else: addiu $s1, $s0, -1
exit: ...                    # done!
```

Interpret the following MIPS assembly code and provide a written explanation of what it does.

```
addi $s0, $0,  0
      addi $s1, $0,  1
      addi $t0, $0,  30
loop: beq  $s0, $t0, done
      sll  $s1, $s1, 1
      addi $s0, $s0, 1
      j    loop
done: # done!
```

This computes the number 2^{30} by iteratively doubling a value starting at one 30 times.