Sample Problems for Quiz # 3

1. **Implementing XOR with AND gates**

   Recall that the Exclusive-OR (XOR) function is 1 if an odd number of the inputs are 1, and 0 otherwise. Suppose that we have to build an circuit that computes XOR with AND gates, OR gates, and inverters.

   (a) Draw a circuit to compute the XOR of 13 variables with the fewest possible number of AND and OR gates, and as many inverters as you want. (Solutions has 30 AND and OR gates.)

   (b) Draw a circuit to compute the XOR of 15 variables with the fewest possible number of AND and OR gates, and as many inverters as you want. (Solutions has 35 AND and OR gates.)

   (c) Draw a circuit to compute the XOR of 23 variables with the fewest possible number of AND and OR gates, and as many inverters as you want. (Solutions has 58 AND and OR gates.)

   (d) Draw a circuit to compute the XOR of 27 variables with the fewest possible number of AND and OR gates, and as many inverters as you want. (Solutions has 65 AND and OR gates.)
2. Batcher Sorting Network

A comparator is a device which sorts two numbers $x$ and $y$, as shown in Figure 1.

![Comparator diagram](image)

Figure 1: a comparator.

The Batcher sorting network was presented in class. It is constructed with a block called the Merge network. Given two sorted input sequences $x_0, x_1, \ldots, x_{n/2-1}$ and $x'_0, x'_1, \ldots, x'_{n/2-1}$, the Merge[$n$] network produces a sorted output sequence $y_0, y_1, \ldots, y_{n-1}$. The recursive construction of the Merge[8] network is shown in Figure 2. The recursive construction of the Batcher[8] network, based on the Merge[8] network, is shown in Figure 3. The Merge[2] and Batcher[2] networks both consist of a single balancer.

![Merge[8] network diagram](image)

Figure 2: the Merge[8] network.

(a) What is the depth of the Merge[$n$] network, $n \geq 2$?

(b) What is the depth of the Batcher[$n$] network, $n \geq 2$?
(c) Note that the Batcher[4] network consists of six comparators, as shown in
Figure [4]. Find a sorting network with four inputs consisting of only five com-
parators. (Hint: Try two comparators in the first stage, two in the second stage
and one in the third stage.) Prove that your network sorts all possible input
sequences.

Figure 3: the Batcher[8] network.

Figure 4: the Batcher[4] network.
3. A Bit of Horticulture – Traversing Trees

Consider the following data structure:

```c
struct node {
    int x;
    struct node *left;
    struct node *right;
};
```

The tedious code to setup a tree is shown at the end. There is also a sketch of the corresponding tree.

(a) What does the following function print out?

```c
void dfs(struct node *p) {
    printf("%d ", p->x);
    if (p->left != NULL ) {
        dfs(p->left);
    }
    if (p->right != NULL ) {
        dfs(p->right);
    }
}
```

```c
int main(int argc, char **argv) {
    struct node *p = setup_tree();
    dfs(p);
}
```

Solution

```c
1
2
4
5
8
9
3
6
7
10
12
11
```
(b) What does the following function print out?

```c
void dfs(struct node *p) {
    if (p->left != NULL) {
        dfs(p->left);
    }
    if (p->right != NULL) {
        dfs(p->right);
    }
    printf("%d ", p->x);
}
```

```c
int main(int argc, char **argv) {
    struct node *p = setup_tree();
    dfs(p);
}
```

Solution

4
8
9
5
2
6
12
10
11
7
3
1
(c) **Breadth-First Search**

Now consider the following program. It creates a tree and then calls the function `bfs`. The function `bfs` performs a “breadth-first” traversal of the tree. It does so by maintaining a queue of nodes to visit, in the form of a linked list. In each iteration of the `while` loop, it visits the node at the front of the linked list, and then frees this node. For each node that it visits, it adds the node’s children to the end of linked list.

```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int x;
    struct node *left;
    struct node *right;
};

struct list {
    struct node *item;
    struct list *next;
};

void bfs(struct node *p) {
    struct node *q;
    struct list *l, *m, *r, *t;
    l = malloc(sizeof(struct list));
    l->item = p;
    l->next = NULL;
    r = l;
    while(l != NULL) {
        q = l->item;
        if (q->left != NULL) {
            r->next = (struct list *) malloc(sizeof(struct list));
            r->next->item = q->left;
            r->next->next = NULL;
            r = r->next;
        }
        if (q->right != NULL) {
            r->next = (struct list *) malloc(sizeof(struct list));
            r->next->item = q->right;
            r->next->next = NULL;
            r = r->next;
        }
    }
}
```


```c
    t = l;
    l = l->next;
    free(t);

    m = l;
    while (m != NULL) {
        printf("%d ", m->item->x);
        m = m->next;
    }
    printf("\n");
}

struct node *setup_tree(void);
int main(int argc, char **argv) {
    struct node *p = setup_tree();
    bfs(p);
}
```

What will the program print out?
Here is the (tedious) code to create the tree.

```c
struct node *setup_tree(void) {
    // create tree
    struct node *p = (struct node *)malloc(sizeof(struct node));
    p->left = (struct node *)malloc(sizeof(struct node));
    p->right = (struct node *)malloc(sizeof(struct node));
    p->left->left = (struct node *)malloc(sizeof(struct node));
    p->left->right = (struct node *)malloc(sizeof(struct node));
    p->right->left = (struct node *)malloc(sizeof(struct node));
    p->right->right = (struct node *)malloc(sizeof(struct node));
    p->right->right->left = (struct node *)malloc(sizeof(struct node));
    p->right->right->right = (struct node *)malloc(sizeof(struct node));
    p->left->right->left->left = (struct node *)malloc(sizeof(struct node));
    p->left->right->left->right = (struct node *)malloc(sizeof(struct node));
    p->left->right->right->left = (struct node *)malloc(sizeof(struct node));
    p->left->right->right->right = (struct node *)malloc(sizeof(struct node));
    p->right->right->left->left->left = (struct node *)malloc(sizeof(struct node));
    p->right->right->left->right = (struct node *)malloc(sizeof(struct node));
    p->right->right->right->left = (struct node *)malloc(sizeof(struct node));
    p->right->right->right->right = (struct node *)malloc(sizeof(struct node));
    p->right->right->left->left->x = 12;
    p->right->right->left->left->left = NULL;
    p->right->right->left->left->right = NULL;
    p->right->right->right->left->x = 11;
    p->right->right->right->right = NULL;
    return p;
}
```
Figure 5: Tree
4. Pointers

(a) What does the following program print out?

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {

    int x = 1, y = 2, z[3];
    int *ip;

    z[0] = 3;
    z[1] = 4;
    z[2] = 5;

    ip = &x;
    y = *ip;
    *ip = 6;
    ip = &z[y];

    printf("%d %d %d\n", x, y, *ip);

}
```

Solution

6 1 4

(b) What does the following program print out?

```c
#include <stdio.h>

int main(int argc, char **argv) {

    int x = 1;
    int y = 2;

    int *p;
    int **q;
    printf("%d %d\n", x, y);

    p = &x;
    q = &p;
```
y = *p;
x = 3;
printf("%d %d\n", x, y);

x = 4;
y = 5;
*q = p;
*p = y;
printf("%d %d\n", x, y);
}

Solution
1 2
3 1
5 5

(c) What does the following program print out?

#include <stdio.h>

int main(int argc, char **argv) {

int i;
int x = 1;
int y = 2;
int z[10] = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};

int *p;
int **q;
int ***r;

int *s, *t;

p = &x;
q = &p;
r = &q;

*p = 3;
**q = 4;
***r = 5;
printf("%d %d %d %d %d\n", x, y, *p, **q, ***r);
```c
x = 6;
y = 7;

s = &x;
t = &y;

p = s;
s = t;
t = p;
printf("%d %d\n", *s, *t);

s = &z[y];
t = s + (100 * x / 243);

*s = 8;
*t = 9;

for (i = 0; i < 10; i++) {
    printf("%d ", z[i]);
}
printf("\n");
```

Solution

```
5 2 5 5 5
7 6
10 11 12 13 14 15 16 8 18 9
```
5. Memory Allocation

(a) Add the requisite lines to the following code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv) {

    int i, r;

    struct funky {
        int i;
        float f;
    };

    struct funky **tut;

    /* Allocate an array of 10 pointers to pointers to structures funky */
    tut = (struct funky **)malloc(10*sizeof(struct funky *));

    for (i = 0; i < 10; i++) {
        if (i % 2 == 0) {
            /* If i is even, allocate an array of length 2. */
            /* WRITE THIS LINE */
            r = rand() % 10;
            tut[i][0].i = r;
            tut[i][0].f = sin(3.14159265 * r);
            r = rand() % 10;
            tut[i][1].i = r;
            tut[i][1].f = cos(3.14159265 * r);
        } else {
            /* Else i is odd; allocate an array of length 1. */
```
/* WRITE THIS LINE */

r = rand() % 13;
(*tut[i]).i = r;
(*tut[i]).f = tan(3.14159265 * r);
Solution

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv) {

    int i, r;

    struct funky {
        int i;
        float f;
    };

    struct funky **tut;

    /* Allocate an array of 10 pointers to pointers to structures funky */
    tut = (struct funky **)malloc(10*sizeof(struct funky *));

    for (i = 0; i < 10; i++) {

        if (i % 2 == 0) {

            // If i is even, allocate an array of 2 structures funky;
            tut[i] = (struct funky *)malloc(2*sizeof(struct funky));

            r = rand() % 10;
            tut[i][0].i = r;
            tut[i][0].f = sin(3.14159265 * r);

            r = rand() % 10;
            tut[i][1].i = r;
            tut[i][1].f = cos(3.14159265 * r);
        } else {

            // If i is odd, allocate a single structure funky;
            tut[i] = (struct funky *)malloc(sizeof(struct funky));

            r = rand() % 13;
        }
    }
}
(*tut[i]).i = r;
(*tut[i]).f = tan(3.14159265 * r);
}
}
}
(b) The following code should allocate and then free a three-dimensional array of integers. Fill in the missing code. (Use the malloc function).

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int x = atoi(argv[1]);
    int y = atoi(argv[2]);
    int z = atoi(argv[3]);
    int ***array;
    int i, j, k;
    // allocate array
    // FILL IN: array = ...
    for (i = 0; i < x; i++) {
        // FILL IN: array[i] = ...
        for (j = 0; j < y; j++) {
            // FILL IN: array[i][j] = ...
            for (k = 0; k < z; k++) {
                array[i][j][k] = i + j + k;
            }
        }
    }
    // print
    for (i = 0; i < x; i++) {
        for (j = 0; j < y; j++) {
            for (k = 0; k < z; k++) {
                printf("%d %d %d: %d
", i, j, k, array[i][j][k]);
            }
        }
    }
    // free array
    for (i = 0; i < x; i++) {
        for (j = 0; j < y; j++) {
            // FILL IN: free(...);
        }
        // FILL IN: free(...);
    }
    // FILL IN: free(...);
}
```
Solution

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int x = atoi(argv[1]);
    int y = atoi(argv[2]);
    int z = atoi(argv[3]);
    int ***array;
    int i, j, k;

    // allocate array
    array = (int ***)malloc(x*sizeof(int **));
    for (i = 0; i < x; i++) {
        array[i] = (int **)malloc(y*sizeof(int *));
        for (j = 0; j < y; j++) {
            array[i][j] = (int *)malloc(z*sizeof(int));
            for (k = 0; k < z; k++) {
                array[i][j][k] = i + j + k;
            }
        }
    }

    // print
    for (i = 0; i < x; i++) {
        for (j = 0; j < y; j++) {
            for (k = 0; k < z; k++) {
                printf("%d %d %d: %d\n", i, j, k, array[i][j][k]);
            }
        }
    }

    // free array
    for (i = 0; i < x; i++) {
        for (j = 0; j < y; j++) {
            free(array[i][j]);
        }
        free(array[i]);
    }
    free(array);
}
```
6. Error Correcting Codes

Suppose that Alice wants to send Bob 4 bits of information at a time, \( x_0, x_1, x_2, x_3 \), over a noisy Wi-fi connection that occasionally flips bits. She decides to encode her information by adding three extra bits \( x_5, x_6, x_7 \) computed as follows (here + represents exclusive OR):

\[
\begin{align*}
    x_4 &= x_1 + x_2 + x_3 \\
    x_5 &= x_0 + x_2 + x_3 \\
    x_6 &= x_0 + x_1 + x_3
\end{align*}
\]

She sends the 7 bits \( x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7 \) to Bob.

Consider the following matrix, called a parity check matrix:

\[
H = \begin{bmatrix}
    0 & 1 & 1 & 1 & 1 & 0 & 0 \\
    1 & 0 & 1 & 1 & 0 & 1 & 0 \\
    1 & 1 & 0 & 1 & 0 & 0 & 1
\end{bmatrix}.
\]

The vector of the seven 7 bits that Alice sends, \( X = [x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7] \), satisfies

\[
HX^T = \begin{bmatrix}
    0 \\
    0 \\
    0
\end{bmatrix}
\]

Problems

(a) Suppose that Alice wants to send the bits 1, 0, 01. What seven bits will she transmit?

**Solution**

1, 0, 0, 1, 1, 0, 0

(b) Suppose that Bob receives the seven bits 0, 0, 1, 0, 1, 0, 1. Which bit was flipped?

What will he conclude were Alice’s original four bits?

**Solution**

Second bit. Her original bits were 0, 1, 1, 0.