"Beauty is more important in computing than anywhere else in technology because software is so complicated. Beauty is the ultimate defense against complexity. ... The geniuses of the computer field, on the the other hand, are the people with the keenest aesthetic senses, the ones who are capable of creating beauty. Beauty is decisive at every level: the most important interfaces, the most important programming languages, the winning algorithms are the beautiful ones."

D. Gelernter ("Machine Beauty", Basic Books, 1998)

"The road to wisdom? Well, it's plain and simple to express: Err and err and err again, but less and less and less." Piet Hein

"The key to understanding recursion is to begin by understanding recursion. The rest is easy."

Koenig/Moo, Accelerated C++

Learning Objectives

1. Use of bit-wise logical operations in MIPS
2. To manage pointers and data structures in MIPS assembly
3. To use function calls and recursion in MIPS assembly

Work that needs to be handed in (via SVN) this Sunday

1. `problem1.s`: implement the `matrix_column_matches_list_row` function. **Run on EWS with:**
   `~cs232/Linux/bin/QtSpim -file p1_main.s problem1.s`
2. `problem2.s`: implement the `any_matrix_column_matches_list_row` function; *this function uses the function you wrote in* `problem1.s`. **Run on EWS with:**
   `~cs232/Linux/bin/QtSpim -file p2_main.s problem1.s problem2.s`
3. `problem3.s`: implement the `check_remaining_columns` function; *this function uses both of the above functions.* **Run on EWS with:**
   `~cs232/Linux/bin/QtSpim -file p2_main.s problem1.s problem2.s problem3.s`

Work that needs to be handed in (via SVN) by 3/17/13

1. `problem4.s`: implement the `solve` function; *this function uses all of the above functions.* **Run on EWS with:**
   `~cs232/Linux/bin/QtSpim -file p4_main.s problem1.s problem2.s problem3.s problem4.s`

Guidelines

- Guidelines are the same as Lab 7.
- You'll find the assignment *so much easier* if you try to understand the C code you're translating before starting.
- Just as in Lab 7, follow all calling and register-saving conventions you've learned. It's even more important in this MP.
- Don't try to change the algorithms at this point, just write the code in MIPS as closely to the provided C code as possible.

Practice with Pointers

```
int *array[10];

int
sum_array_of_int_ptrs(int **A, int length) {
   int sum = 0;
   for (int i = 0 ; i < length ; i ++) {
      sum += *(A[i]);
   }
   return sum;
}
```

More Practice with Pointers

Below is C code for a function that adds a specified `value` to the data held in each element of a singly-linked list. Write increment as a MIPS function.

```
typedef struct node_s {
   int *data;                   // pointer to an integer
   struct node_s *next;         // pointer to another struct node_s
} node_t;                       // "node_t" is shorthand for "struct node_s"

void
increment(node_t *head, int value) {
   for (node_t *trav = head ; trav != NULL ; trav = trav->next) {
      *(trav->data) += value;
   }
}
```

Linked lists are versatile and consequently ubiquitous data structures. Like arrays, they can be used to store an ordered set of data. Unlike arrays, they don't require an upper bound to their capacity when they are declared, and they permit easy insertion of elements into the middle of the list. However, it's much harder to address elements by index in a linked list than in an array, and linked lists have some performance drawbacks.

You'll remember from earlier classes that linked lists are chains of nodes, each node containing a pointer to the next node in the chain. A *doubly-linked list* sports nodes that point both forward to the next element in the list and backwards to the previous element in the list. In C, a node in a doubly-linked list could be declared:

```
typedef struct node_s {
  int data;
  struct node_s *prev;
  struct node_s *next;
} node_t;
```

This fancy node would hold an integer value as well as a pointer to the previous and next nodes in its list. If a node has no previous node, then its `prev` pointer would be `NULL`[1]; likewise, if a node has no next node, then its `next` pointer would be `NULL`.

A pointer to the head node of a singly linked list is equivalent to a pointer to the entire list. For a doubly-linked list, it's more convenient to have pointers to both ends:

```
typedef struct list_s {
  node_t * head;
  node_t * tail;
} list_t;
```

Don't worry about the `typedef`s in the above declarations – but what's the `struct` for? `struct` is a high-level language tool, a way to reference an aggregation of data with just a single variable or pointer. Physically, a struct is a pointer to a region of memory big enough to hold all the members inside it, like with an array, except that the elements can be all different sizes and types. In the MIPS snippet below (adapted from `p2.s`) `list` holds the start and end elements of a linked list, while the `listitem`s represent individual list nodes, each with an integer data value, a next pointer, and a previous pointer – all 32-bit words.

```
list:      .word   listitem1, listitem3
listitem1: .word   8,         0,         listitem2
listitem2: .word   5,         listitem1, listitem3
listitem3: .word   7,         listitem2, 0
```

Pretty cool! One other note about `struct`s in C: how do you access their members? Say we have the following variable declaration:

```
list_t mylist;
```

If we want the head of `mylist`, we reference `mylist.head`. Same thing if we wanted the tail, or any member of any struct. But what if all we had was a *pointer* to `mylist`?

```
list_t * myptr = &mylist;
```

We could always just dereference the pointer before accessing a member: `(*myptr).head`. But C provides a shorthand for this: `myptr->head`. You'll see a lot of this arrow operator in the upcoming C code.

[1] `NULL` in C is another name for zero

A puzzle:

In this assignment, we'll be translating code that solves a kind of computational puzzle. In this puzzle, you are supposed to reconstruct an $N \times N$ matrix from an array containing the rows and columns. Specifically the puzzle is created as is shown below:

1. a random square matrix is constructed where each element in the matrix is a value between 0 and 15 (so we can print the elements as hexadecimal characters).
2. we extract the rows and columns into an array of arrays of elements (what we call the "list"). The list should contain $2N$ arrays, each of length N.
3. we scramble the order of the arrays on the list.

We then provide you the list and it is your job to reconstruct the matrix (or its transpose... we'll accept either).

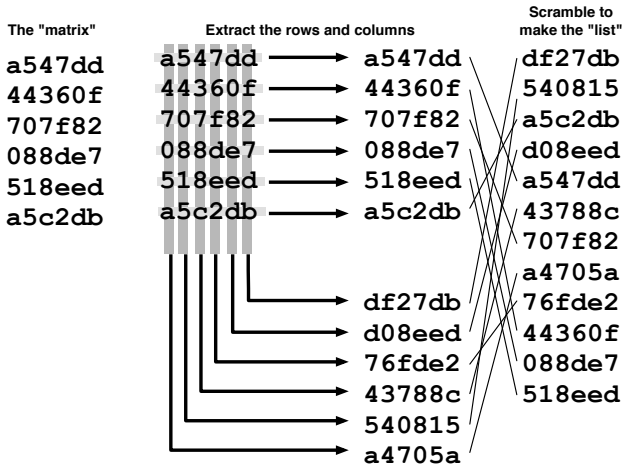


Figure 1. An example puzzle construction.

The code that we've given you to translate into MIPS performs a brute force solution of this puzzle using the following approach:

1. generate all possible solution matrices consisting of N elements from the list as rows.
2. as you generate each possible solution, check if the remaining N elements can serve as the columns. If so, you've found the solution and you are done.

Before we look at this solution in detail, let's first look at the representation for the matrix (which is also the representation for the list) in both C and MIPS.

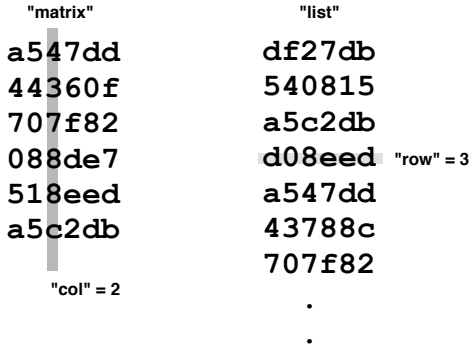
```
typedef struct HexMatrix {
    int size;
    unsigned char** data;
} HexMatrix;
```

```
.data
M2050: .byte 0x4, 0xf
M2070: .byte 0xd, 0x5
M2030: .word M2050, M2070
matrix1: .word 2, M2030
```

The variable data is a `char **`, meaning it is a pointer to an array of char pointers. We're implementing two-dimension array of characters by using an array of pointers, where each pointer points to an array of characters.

Problem 1: matrix_column_matches_list_row

Our solution consists of four functions where A calls B, B calls C, and C calls D. This is function D, the innermost function and the only "leaf" function (a function that doesn't call other functions). This function checks to see if a column in a matrix is the same as a row in a list.



This call to `matrix_column_matches_list_row` would return 0 because the first element of column 2 of the matrix is different than the first element of row 3 of the list.

```
int
matrix_column_matches_list_row(HexMatrix* matrix, int col, HexList* list, int row) {
    for (int k = 0; k < matrix->size; ++ k)    // check each element in row against column
    {
        if (matrix->data[k][col] != list->data[row][k])
        {
            return 0;                       // if an element doesn't match, the row doesn't
        }
    }
    return 1;
}
```

Problem 2: any_matrix_column_matches_list_row

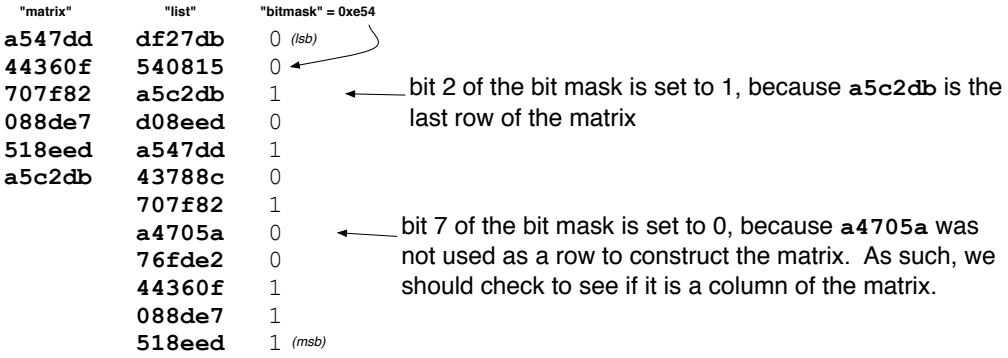
This function uses the previous function. It iterates over all of the columns of "matrix" to see if a given row matches. For the picture above, passing the arguments `matrix, list, 3` would return the value `1`, since row `3` of `list` matches column 4 of `matrix`.

```
int
any_matrix_column_matches_list_row(HexMatrix* matrix, HexList* list, int row) {
    for (int col = 0; col < matrix->size; ++ col)    // check against the col'th column
    {
        if (matrix_column_matches_list_row(matrix, col, list, row))
        {
            return 1;
        }
    }
    return 0;
}
```

Because this function calls another function, a stack frame is required. Check out the videos on using callee saved registers (*e.g.*, $s0) because they can help you on this function since the function is called from inside of a loop.

Problem 3: check_remaining_columns

This function checks to see if a solution has been found. It takes a candidate solution `matrix`, the `list` from which it was made, and a `bitmask` that indicates which elements of the list were used as rows to construct `matrix`. Specifically, there is one bit in `bitmask` for each element in `list`. The least significant bit (lsb) corresponds to the first element of `list`. If a bit is `1`, the corresponding element of `list` was used as a row of matrix; if `0`, then we need to check if that `list` item matches a column of `matrix`. If all of the elements with `0`s in the bitmask are columns of the `matrix`, we've found a valid solution and can return 1. Otherwise, return 0.




```
int
check_remaining_columns(HexMatrix* matrix, HexList* list, unsigned int bitmask)
{
    for (int i = 0; i < list->size; ++ i)
    {
        if ((1L << i) & bitmask) { continue; }   // continue if this was used as a row
        if (!any_matrix_column_matches_list_row(matrix, list, i)) {
            return 0;   // if it doesn't match any columns, this matrix isn't a solution
        }
    }
    return 1;
}
```

Problem 4: solve (due 1 week later)

Once we have a way of checking candidate solutions, we need to generate candidate solutions. We do this by exhaustively trying every `list` element in every row of the matrix in combination with every other `list` element. We do this by using recursion and using a bitmask to keep track of which `list` elements we've already used (since each should be used exactly once).

```
int
solve(HexMatrix* matrix, HexList* list, int depth, unsigned int bitmask)
{
    if (depth == matrix->size)  // check to see if the remaining list items match the columns
    {
        return check_remaining_columns(matrix, list, bitmask);
    }

    for (int i = 0; i < list->size; ++ i) {
        if (((1L << i) & bitmask) == 0)
        {
            for (int j = 0; j < matrix->size; ++ j)
            {
                matrix->data[depth][j] = list->data[i][j];
            }

            if (solve(matrix, list, depth + 1, (1L << i) | bitmask))
            {
                return 1;
            }
        }
    }
    return 0;
}
```

The base case of this recursive function is when we've already allocated as many `list` items as there is room in the `matrix`. Since we'll be allocating 1 `matrix` row per level of the recursion, this occurs when the depth of the recursion (which we keep track of with an function argument) matches the number of rows in the `matrix`. When this happens, we can call the function we already implemented to check the matrix.

When the base case doesn't apply, we need to allocate the next row of the matrix. At a recursion depth of d, we are assigning the dth row of the matrix. Since we're doing exhaustive search, we try every `list` element (that we haven't already used) at that position. We keep track of which `list` elements have already been used using a bitmask (same as problem 3) where there is one bit per `list` element, and we set the bit to 1 when we copy a `list` element to a row of the `matrix`