"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. To manage pointers and data structures in MIPS assembly
2. To use function calls and recursion in MIPS assembly

Work that needs to be handed in (via SVN)

First deadline

1. `get_num_carrots.s`: implement the `get_num_carrots` function.
   Run with:
   `QtSpim -file common.s get_num_carrots_main.s get_num_carrots.s`
2. `pick_best_k_baskets.s`: implement the `pick_best_k_baskets` function.
   Run with:
   `QtSpim -file common.s get_num_carrots.s pick_best_k_baskets_main.s pick_best_k_baskets.s`

Second deadline

1. `get_secret_id.s`: implement the `get_secret_id` function.
   Run with:
   `QtSpim -file common.s helper_functions.s get_secret_id_main.s get_secret_id.s`
2. `collect_baskets.s`: implement the `collect_baskets` function.
   Run with:
   `QtSpim -file common.s get_num_carrots.s collect_baskets_main.s collect_baskets.s`
3. `search_carrot.s`: implement the `search_carrot` function.
   Run with:
   `QtSpim -file common.s helper_functions.s get_num_carrots.s pick_best_k_baskets.s get_secret_id.s collect_baskets.s search_carrot_main.s search_carrot.s`

Important!

This is a solo lab. You may not work with anyone else on this lab.

We also cannot stress enough the importance of reading the *entire* lab handout.

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 lab.
- 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.

We're helpful!

We've provided the C++ code we want you to rewrite. It's all in `lab8.cpp` in your SVN for your reference.

We've provided some print functions, in `common.s`. Use it to help you debug!

Structs

What is a Struct?

In this lab, you will utilize several structs in order to traverse a graph of carrots, and write an algorithm to find carrots in this graph. A `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 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.

One useful note about `structs` in C++: how do you access their members? Say we have the following struct and variable declaration:

```
struct Node {
    int num_children;
    Node* children[4];
};
Node mynode;
```

If we want the `num_children` element of `mynode`, we reference `mynode.num_children`. Same thing if we wanted the assignments. But what if all we had was a *pointer* to `mynode`?

```
Node *myptr = &mynode;
```

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

Node & Baskets Structs

We will use the Node struct to represent the graph of carrots, and we will use the Baskets struct to collect Node and Baskets. These structs are declared as follows:

```
struct Node {
    char seen;
    int basket;
    int dirt;
    int id_size;
    int *identity;
    int num_children;
    Node *children[4];
};
struct Baskets {
    int num_found;
    Node *basket[10];
};
```

The Node struct contains seven member variables: a char `seen`, an int `basket`, an int `dirt`, an int `id_size`, an int pointer `identity`, an int `num_children`, and an array of 4 Node pointers `children`.

- The char `seen` acts like a boolean. `seen` is set to 1 (i.e. `true`) when it has been searched using the function `collect_baskets`, and it will be 0 (i.e. `false`) when it has not been searched yet.
- The int `basket`, which represents the basket in the graph that is located at this node.
- The int `dirt`, which represents the dirt in the graph that is located at this node.
- The int `id_size`, is the dimension of the Node's `identity`, where $id\_size \times id\_size$ is the size of the Node's identity pointer (see next item).
- The int pointer `identity`, which points to a $id\_size \times id\_size$ 1D array. The identity of the Node

can be computed by calling the `calculate_identity` function from Lab 7 on this array.

- The int `num_children` is the number of valid children in the `children` array. This can range from 0 to 4.
- The array of Node pointers `children` contains four Node pointers. There can be anywhere from 0 to 4 children, so you will have to use `num_children` to ensure you are accessing the correct children.

The Baskets struct contains two member variables: an integer `num_found`, and an array of 10 Node pointers `basket`.

- The int `num_found` is the number of "found" baskets, that is, it is the number of valid baskets in the `basket` array. This can range from 0 to 10.
- The array of Node pointers `basket` contains 10 Node pointers. There can be anywhere from 0 to 10 baskets in this array, so you will have to use `num_found` to ensure you are accessing the correct baskets.

Part 1 [20 points]

Get Number of Carrots [10 points]

`get_num_carrots()` calculates the number of carrots in the Node by digging in the Node's dirt, and then XOR'ing with the Node's basket to see how many carrots can fit. The following C++ code takes one argument: `spot` (a pointer to a Node struct), and returns an integer, the spot's carrot count. Implement `get_num_carrots` in MIPS assembly in `get_num_carrots.s`.

```
int
get_num_carrots(Node *spot) {
    if (spot == NULL) {
        return 0;
    }
    // Inverts the first and third byte.
    unsigned int dig = spot->dirt ^ 0x00ff00ff;
    // Circular shifts the bytes left one.
    dig = ((dig & 0xffffff) << 8) | ((dig & 0xff000000) >> 24);
    return spot->basket ^ dig;
}
```

We've provided some test cases in `get_num_carrots_main.s` and their C++ versions in `lab8.cpp`. You should read the given test cases, make sure you understand them, and then run your code against them to check its correctness. You're also heavily encouraged to add some more test cases, because we will when autograding. You can do so by adding them to `lab8.cpp` to see what the correct output should be. Then you can add them to `get_num_carrots_main.s` to see if your MIPS does the same.

Pick Best k Baskets [10 points]

`pick_best_k_baskets()` is a function that reorders the `baskets` array such that the k Nodes with the most carrots are the first k elements in the array. The following C++ code takes two arguments: `k` (the number of "best" baskets that we want) and `baskets` (a pointer to the Baskets node). Note that this function does not return anything. It simply reorders the elements in the baskets array. Implement `pick_best_k_baskets` in MIPS assembly in `pick_best_k_baskets.s`.

```
void
pick_best_k_baskets(int k, Baskets *baskets) {
    if (baskets == NULL) {
        return;
    }
    for (int i = 0; i < k; i++) {
        for (int j = baskets->num_found - 1; j > i; j--) {
            if (get_num_carrots(baskets->basket[j - 1]) <
                get_num_carrots(baskets->basket[j])) {
                // Swaps values stored at j-1 and j in place.
                // The address stored in basket array is 32 bits
                // (i.e. You do NOT need to do any casting in MIPS.)
                baskets->basket[j] = (Node *)((intptr_t)baskets->basket[j] ^
                                              (intptr_t)baskets->basket[j - 1]);
                baskets->basket[j - 1] = (Node *)((intptr_t)baskets->basket[j] ^
                                                  (intptr_t)baskets->basket[j - 1]);
                baskets->basket[j] = (Node *)((intptr_t)baskets->basket[j] ^
                                              (intptr_t)baskets->basket[j - 1]);
            }
        }
    }
    return;
}
```

The function applies bubble sort backwards so that the Nodes with the fewest carrots get pushed to the end of the `baskets` array. This will leave the k Nodes with the most carrots at the beginning of the array. Keep in mind that the top k carrots will not have a particular order between one another. Don't worry about how the casting was done in C++, you do NOT need to do any casting in MIPS.

We've provided some test cases in `pick_k_best_baskets_main.s` and their C++ versions in `lab8.cpp`. You should read the given test cases, make sure you understand them, and then run your code against them to check its correctness. You're also heavily encouraged to add some more test cases, because we will when autograding. You can do so by adding them to `lab8.cpp` to see what the correct output should be. Then you can add them to `pick_k_best_baskets_main.s` to see if your MIPS does the same.

Carrot Search

Overview of Implementation

Your carrot search solver will recursively search a graph of nodes for carrots.

Below is an overview of the algorithm used in the solver you will write. It has been provided because understanding an overview of the algorithm can be very helpful with debugging. For the sake of brevity, it is not written below in the same way as the C++ implementation that we want to you translate into MIPS. Therefore, *do **not** write your MIPS code based on this.*

CarrotSearch(*max_baskets***,** *k***,** *root***,** *baskets***):**

1. First, we initialize all of our *baskets* as $NULL$.
2. Collect as many baskets as possible by traversing the graph from its *root* node, using the *collect_baskets* function. This will populate our *baskets* struct.
3. We pick the k baskets with the most carrots from our *baskets* struct using the *pick_best_k_baskets* function. This will reorder the Baskets struct such that the k Nodes with the most carrots will be in the first k elements of the array in the Baskets struct.
4. Finally, we will calculate the secret identity of the k best baskets by using the *get_secret_id* function. This function simply calls *calculate_identity* (the function we wrote for Lab7.2) on each Node, and sums up the different unique identities.

Part 2 [80 points]

Now that you are more familiar with how the CarrotSearch solver will work, you are ready to complete the `get_secret_id`, `collect_baskets`, and `search_carrot` functions, for which you have been building the infrastructure in the last two labs. These three functions make up the bulk of the CarrotSearch solver.

Get Secret Id [20 points]

`get_secret_id()` is a function that computes the secret id of a certain number of baskets using the `calculate_identity` function you wrote in Lab7.2.

The following C++ code takes two arguments: an integer `k` (the number of baskets for which we compute the secret id), and `baskets` (a pointer to a Baskets struct). The function returns an int, `secret_id`, which is the summation of the identities of the first k baskets. Implement `get_secret_id` in MIPS assembly in `get_secret_id.s`.

```
int
get_secret_id(int k, Baskets *baskets) {
    if (baskets == NULL) {
        return 0;
    }
    int secret_id = 0;
    for (int i = 0; i < k; i++) {
        secret_id += calculate_identity(baskets->basket[i]->identity,
                                        baskets->basket[i]->id_size);
    }
    return secret_id;
}
```

The `get_secret_id` function uses helper functions, `calculate_identity` and `accumulate`, which you wrote in Lab 7. After the Lab7.2 late submission period has passed, we will provide the `calculate_identity` and `accumulate` functions for you to use in Lab8. They will be posted at `https://subversion.ews.illinois.edu/svn/sp17-cs233/_shared/Lab7/` on Tuesday night. We recommend copying these solutions into the `helper_functions.s` file so that you can test with the same environment as our autograder. If you choose to get started on this function before Tuesday night, you can paste your own solutions for `calculate_identity accumulate` into `helper_functions.s` until we release our versions.

We've provided some test cases in `get_secret_id_main.s` and their C++ versions in `lab8.cpp`. You should read the given test cases, make sure you understand them, and then run your code against them to check its correctness. You're also heavily encouraged to add some more test cases, because we will when autograding. You can do so by adding them to `lab8.cpp` to see what the correct output should be. Then you can add them to `get_secret_id_main.s` to see if your MIPS does the same.

Collect Baskets [30 points]

`collect_baskets()` is a function that searches the graph for nodes with carrots and stores the first `max_baskets` found in `baskets`.

The following C++ code takes three arguments: an integer `max_baskets` (the max number of baskets we can collect), `spot` (the current Node that we are searching), and `baskets` (a pointer to a Baskets struct). Note that this function does not return anything, rather, it populates the `baskets` struct. Implement `collect_baskets` in MIPS assembly in `collect_baskets.s`.

```
void
collect_baskets(int max_baskets, Node *spot, Baskets *baskets) {
    if (spot == NULL || baskets == NULL || spot->seen == 1) {
        return;
    }
    spot->seen = 1;
    for (int i = 0; i < spot->num_children && baskets->num_found < max_baskets;
         i++) {
        collect_baskets(max_baskets, spot->children[i], baskets);
    }
    if (baskets->num_found < max_baskets && get_num_carrots(spot) > 0) {
        baskets->basket[baskets->num_found] = spot;
        baskets->num_found++;
    }
    return;
}
```

This function works using magic!

Okay, no not really, but it takes some thinking to figure out. Basically, this function searches all of the children Nodes within a Node struct, marking all the seen nodes so that no nodes are accidentally counted more than once. It recursively tries to fill in the `baskets` array until it the array contains `max_baskets` elements.

Note that this function is recursive. This may seem like a tricky thing to deal with, but in actuality, a recursive function call isn't really any different than a normal function call. Just make sure to follow the calling conventions just like with any other function call, otherwise strange things will happen. Also, we recommend learning how to use 'callee saved registers' (the `$s` registers), as you will likely find that they simplify the code. We've provided a number of video examples on using `$s` registers!

This function also uses the `get_num_carrots` function that you wrote for the first part of this lab, so make sure you include all of the necessary files (see the first page of this handout) when you are running your MIPS code.

We've provided four test cases in the file `collect_baskets_main.s` and their C++ versions in `lab8.cpp`. You should read the given test cases, make sure you understand them, and then run your code against them to check its correctness. You're also heavily encouraged to add some more test cases, because we will when autograding. You can do so by adding them to `lab8.cpp` to see what the correct output should be. Then you can add them to the MIPS test cases to see if your MIPS does the same.

Search Carrot [30 points]

This function handles the main logic for your carrot finder algorithm and utilizes everything that you have completed so far for this week's lab.

The following C++ code takes four arguments: an integer `max_baskets` (the max number of baskets we can collect), `k` (the number of "best" baskets), `root` (the root Node of the graph to start the search from), and `baskets` (a pointer to a Baskets struct), and returns an int, which is the secret id of the basket with the most carrots. Implement `search_carrot` in MIPS assembly in `search_carrot.s`.

```
int
search_carrot(int max_baskets, int k, Node *root, Baskets *baskets) {
    if (root == NULL || baskets == NULL) {
        return 0;
    }
    baskets->num_found = 0;
    for (int i = 0; i < max_baskets; i++) {
        baskets->basket[i] = NULL;
    }
    collect_baskets(max_baskets, root, baskets);
    pick_best_k_baskets(k, baskets);
    return get_secret_id(k, baskets);
}
```

This function sets up your `baskets` struct by setting its elements to `NULL`. Then it uses the `collect_baskets` function to populate the `baskets` array up to `max_baskets` elements. Using the `pick_best_k_baskets` function, the `baskets` array is reordered such that the "best" `k` elements occur in the first `k` elements in the array. Finally, the function `get_secret_id` is used to compute the secret id of the best k baskets in your `baskets` array.

It may seem tricky that this function calls many other functions. But as long as you set up each function call correctly, you shouldn't run into any major problems. Just make sure to follow the calling conventions just like with any other function call, otherwise strange things will happen. Again, we recommend learning how to use 'callee saved registers' (the `$s` registers), as you will likely find that they simplify the code. We've provided a number of video examples on using `$s` registers!

This function uses all of the other functions that you have written for this week's lab, as well as the `calculate_identity` function that you wrote in Lab 7. Since there are a quite a few functions that `search_carrot` is dependent on, make sure you include all of the necessary files (see the first page of this handout) when you are running your MIPS code.

We've provided three test cases in the file `search_carrot_main.s` and their C++ versions in `lab8.cpp`. You should read the given test cases, make sure you understand them, and then run your code against them to check its correctness. You're also heavily encouraged to add some more test cases, because we will when autograding. You can do so by adding them to `lab8.cpp` to see what the correct output should be. Then you can add them to the MIPS test cases to see if your MIPS does the same.