CS 137 Part 9
Fibonacci, More on Tail Recursion, Map and Reduce
Fibonacci Numbers

- An ubiquitous sequence named after Leonardo de Pisa (circa 1200) defined by

\[
\text{fib}(n) = \begin{cases} 
0 & \text{if } n == 0 \\
1 & \text{if } n == 1 \\
\text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise}
\end{cases}
\]
Examples in Nature

- Plants, Pinecones, Sunflowers,
- Rabbits, Golden Spiral and Ratio connections
- Tool’s song Lateralus
- https://www.youtube.com/watch?v=wS7CZIJVxFY
#include <stdio.h>

int fib (int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-1)+fib(n-2);
}

int main () {
    printf("%d\n",fib(3));
    printf("%d\n",fib(10));
    // f_45 is largest that fits in integer.
    printf("%d\n",fib(45));
    return 0;
}

Fibonacci Call Tree
Fibonacci Call Tree

- The tree is really large, containing $O(2^n)$ many nodes (Actually grows with $\phi^n$ where $\phi$ is the golden ratio (1.618))
- Number of $\text{fib}(1)$ leaves is $\text{fib}(n)$
- Summing these is $O(\text{fib}(n))$
- Thus, the code on the previous slide runs in $O(\text{fib}(n))$ which is exponential!
Improvements

- This implementation of Fibonacci shouldn’t take this long - After all, by hand you could certainly compute more than $\text{fib}(45)$.
- We could change the code so that we’re no longer calling the stack each time, rather we’re using iterative structures.
- This would reduce the runtime to $O(n)$. 
int fib(int n){
    if(n==0) return 0;
    int prev = 0, cur = 1;
    for(int i=2; i<n; i++){
        int next = prev + cur;
        prev = cur;
        cur = next;
    }
    return cur;
}
<table>
<thead>
<tr>
<th>$n$</th>
<th>prev</th>
<th>cur</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Tail Recursive Fibonacci

```c
int fib_tr(int prev, int cur, int n){
    if(n==0) return cur;
    return fib_tr(cur, prev + cur, n-1);
}
int fib(int n){
    if(n==0) return 0;
    return fib_tr(0,1,n-1);
}
```
Tail call elimination can reuse the activation record for each instance of fib_tr().
Counting Change

• Given an unlimited number of coins of specified denominations (say 1,5,10,25,100,200), count the number of unique ways to make change.
• For example, 5,5,10 and 5,10,5 are the same.
• Related to https://projecteuler.net/problem=31
Key Idea

• Take the number of ways to do this using all the coins up to coin $i$ and then count all ways to do this without using coin $i$ and decrease.
#include <stdio.h>

int count_change(int coin[], int n, int amount);

int main(void) {
    int coin[] = {1,5,10,25,100,200};
    const int n = sizeof(coin)/sizeof(coin[0]);
    printf("%d\n", count_change(coin,n,20));
    printf("%d\n", count_change(coin,n,200));
    return 0;
}
int count_change(int coin[], int n, int amount) { 
    if (amount == 0) return 1;
    if (amount < 0) return 0;
    if (n == 0) return 0;
    return count_change(coin, n, amount - coin[n-1]) + count_change(coin, n-1, amount);
}
Higher Order Functions

Tail recursion is an important part of functional programming. C is an imperative language. We can still take common tools/problems solutions of functional languages though and use them to our advantage as programmers!

A big part of functional programming is Higher Order Functions which are simply functions that operate on functions (take functions as at least one of their parameters). We’ve already seen this with `qsort` and functions we’ve written that have taken function pointers.
Higher Order Functions

Here are some examples of common higher order functions:

- **filter** - Based on a boolean function produce a new list with only certain elements.
- **map** - Create a new list by applying a function to each of the elements of an existing list.
- **reduce** - Apply a binary function over a list to produce a new final answer. (e.g. multiply all integers in an array)
Let's first write `map` for only collections of integers. What should our signature look like?

```c
// Going to map function f onto src, placing the result in dest. Assume dest has enough space!
void map(int *src, size_t n, int *dest,
        int (*f)(int, int))
```
Integer Map

Let's first write `map` for only collections of integers. What should our signature look like?

```c
// Going to map function f onto src, placing the
// result in dest. Assume dest has enough space!
void map(int *src size_t n, int *dest,
         int (*f)(int, int)) {
    if (n == 0) return;
    *dest = f(*src);
    map(++src, --n, ++dest, f);
}
```
```c
int printInt(int n) {
    printf("\%d ", n);
    return n;
}

int multByTwo(int n) { return n*2; }

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    map(arr, 5, arr, printInt);    // 1 2 3 4 5
    map(arr, 5, arr, multByTwo);
    map(arr, 5, arr, printInt);    // 2 4 6 8 10
}
```
The function we just wrote is great, but it only allows us to operate on integers. But since we’re taking a function as our parameter, shouldn’t we be able to take any unary function? And map it onto a list of the correct corresponding type of elements?

Ideally yes. If we want behaviour like this in C though we have to work for it.

**Observation:** a `void *` can point at anything. This is how generalizations are often done in C, with parameters of type `void *`. 
Generalized Map

If we try to be as true to what map is as possible, we want to take a unary function. We also need our src and our dest - but now they're void *s. Which means we don’t know what they store, so in order to make our updates we need additional information. Specifically the size of the types of each of our src and dest types (they don’t have to match! The function we take can have a parameter of type T and return a different type S)
Generalized Map - first attempt

- src is our source array of \( n \) elements of size \( src\_bytes \) each
- dest is our destination array of elements of size \( dest\_bytes \)
- \( f \) is our function that takes a pointer to a src type and returns a pointer to our dest type... but they must be declared void here only the caller of the map actually knows those types!

Herein lies our problem - \( f \) must return a pointer. What does that mean about the pointer it must return?

\[
\text{void map(} \text{void } *\text{src, size_t } n, \text{size_t } src\_bytes, \\
\text{void } *\text{dest, size_t } dest\_bytes, \\
\text{void } *(*f)(\text{void } *)\text{);} 
\]
Generalized Map - first attempt

Two main issues:

- Since in order for \( f \) to return arbitrary types it must return a `void *`, it also means the function needs to make sure those pointers are valid (not allocated on its own stack). So we’re going to have to assume \( f \) is allocating its return with malloc and we’re responsible for freeing it. That’s a lot of unnecessary allocations!

- Since we don’t know what the return type of \( f \) actually is how can we copy it into \( dest \)? We at least know the return type is `dest_bytes` large, so too is the data in \( dest \). So we can copy it byte by byte, but that is a pain!

```c
void map(void *src, size_t n, size_t src_bytes,
         void *dest, size_t dest_bytes,
         void *(*f)(void *));
```
void map(void *src, size_t n, size_t src_bytes,
       void *dest, size_t dest_bytes,
       void *(*(f)(void *))
)
if (n == 0) return;
void *ret = f(src);
memcpy(dest, ret, dest_bytes);
free(ret);
map(src+src_bytes, --n, src_bytes,
     dest+dest_bytes, dest_bytes, f);
Generalized Map - first attempt

Now this can be used as so:

```c
void *float_to_int(void *f) {
    float d = *(float*)f;
    int *i = malloc(sizeof(int));
    *i = (int)d;
    return i;
}

int main() {
    float f[] = {7.2, 4.5, 2.0};
    int a[3] = {0};
    map(f, 3, sizeof(float), a,
         sizeof(float), float_to_int);
    // a = {7, 4, 2}
}
```
Generalized Map - Second attempt

The previous map works, but the two issues discussed make it undesirable, particularly the number of allocations made. We can fix this, and make map itself less complicated if the function it takes works slightly different. Instead of returning a value, it should take a pointer to where the result should be stored and write the result there itself!

```c
void map2(void *src, size_t n, size_t src_bytes,
          void *dest, size_t dest_bytes,
          void (*f)(void *, void *));
```
Now we assume the writer of function $f$ knows what they are doing, and they know the types they expect. So our new generalized map can look much like our original int one!

```c
void map2(void *src, size_t n, size_t src_bytes,
          void *dest, size_t dest_bytes,
          void (*f)(void *, void *)) {
    if (n == 0) return;
    f(src, dest); // f writes to dest itself.
    map2(src+src_bytes, --n, src_bytes,
          dest+dest_bytes, dest_bytes, f);
}
```
Now users of this function just need to write their functions to make sure to account for this:

```c
void float_to_int2(void *src, void *dest) {
    *(int *) dest = (int)*(float *) src;
}

int main() {
    float f[] = {7.2, 4.5, 2.0};
    int a[3] = {0};
    map2(f, 3, sizeof(float), a,
         sizeof(float), float_to_int2);
    // a = {7, 4, 2}
}
```
Reduce

Now let’s apply the lessons we’ve learned from map to create the function reduce. Our reduce will take a src array and a dest value and compute the following:

\[ src = [a_0, a_1, a_2, \ldots, a_n] \]
\[ dest = f(a_0, f(a_1, f(a_2, \ldots(f(a_n, ?)))))) \]

What does the question mark at the end represent? We want to be able to still work on arrays of one element. As such we need a base case argument. In our implementation the user must store the base in dest, so we don’t have to take an extra parameter.
Reduce

```c
void reduce(void *src, size_t n, size_t src_bytes, void *dest, void (*f)(void*, void*)) {
    if (n == 1) {
        f(src, dest);
        return;
    }
    reduce(src + src_bytes, n - 1, src_bytes, dest, f);
    f(src, dest);
}
```
This does what we want, we can use it like so:

```c
void mult(void *lhs, void *rhs) {
    *(int*)rhs = *(int*)lhs **(int*)rhs;
}

int main() {
    int a[] = {5, 4, 3, 2, 1};
    int answer = 1;
    reduce(a, 5, sizeof(int), &answer, mult);
    // answer = 120.
}
```
Functional Tools

Learning to use functional tools in useful ways can help you to improve as a programmer. While C may not have the best support for such methods of programming, you’ll soon be learning C++ which has much better support - including built in functions such as these we’ve written!