CS 137 Part 9

Fibonacci, More on Tail Recursion, Map and Filter

Fibonacci Numbers

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

$$\texttt{fib(n)} = \begin{cases} 0 & \text{if } n == 0 \\ 1 & \text{if } n == 1 \\ \texttt{fib(n-1)} + \texttt{fib(n-2)} & \text{otherwise} \end{cases}$$

Examples in Nature

- Plants, Pinecones, Sunflowers,
- Rabbits, Golden Spiral and Ratio connections
- Tool's song Lateralus
- `https://www.youtube.com/watch?v=wS7CZIJVxFY`

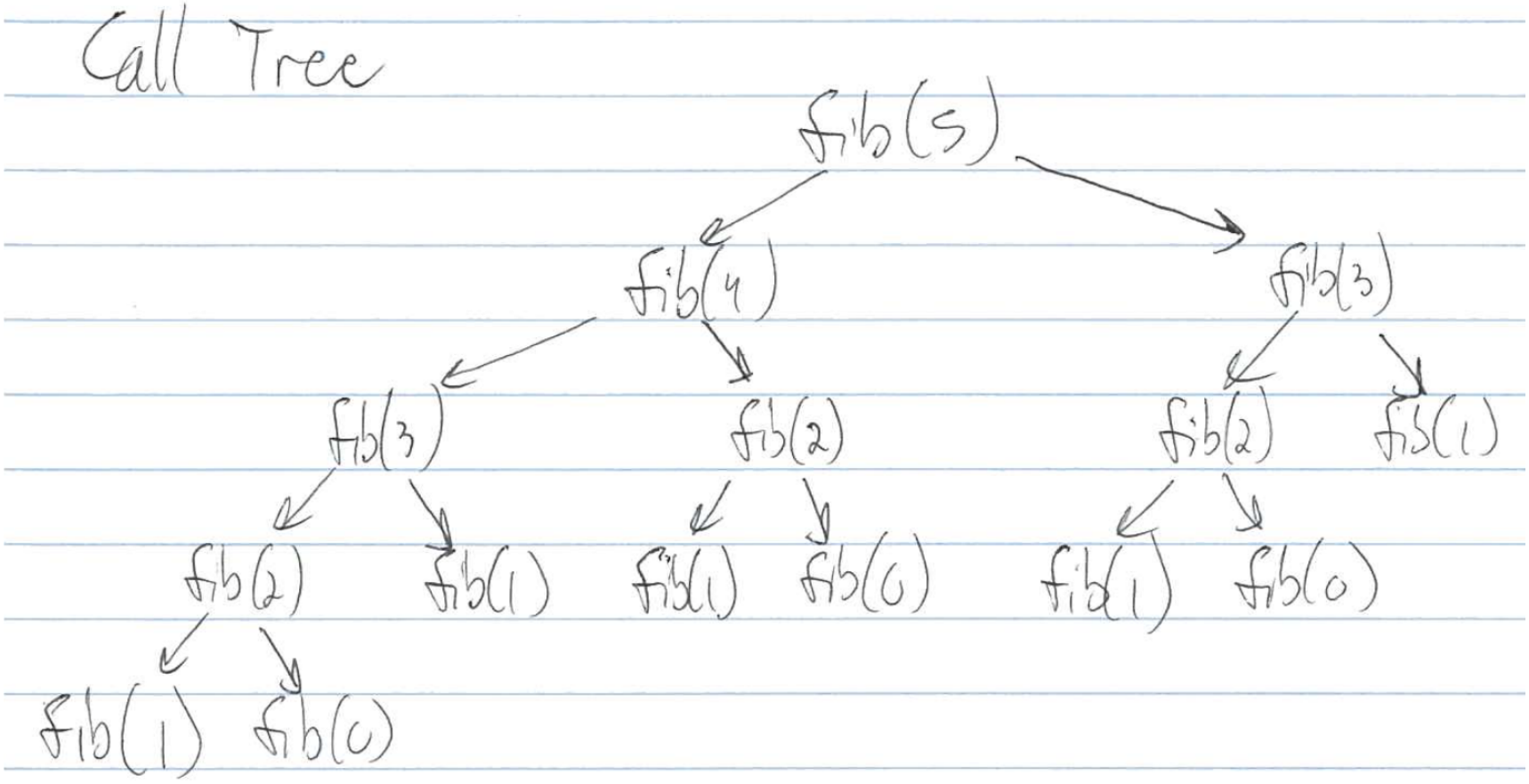
Fibonacci First Attempt

```
#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 ϕ^n where ϕ is the golden ratio (1.618))
- Number of `fib(1)` leaves is `fib(n)`
- Summing these is $O($`fib(n)`$)$
- Thus, the code on the previous slide runs in $O($`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 `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 $O(n)$.

Iterative Fibonacci

```
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;
}
```

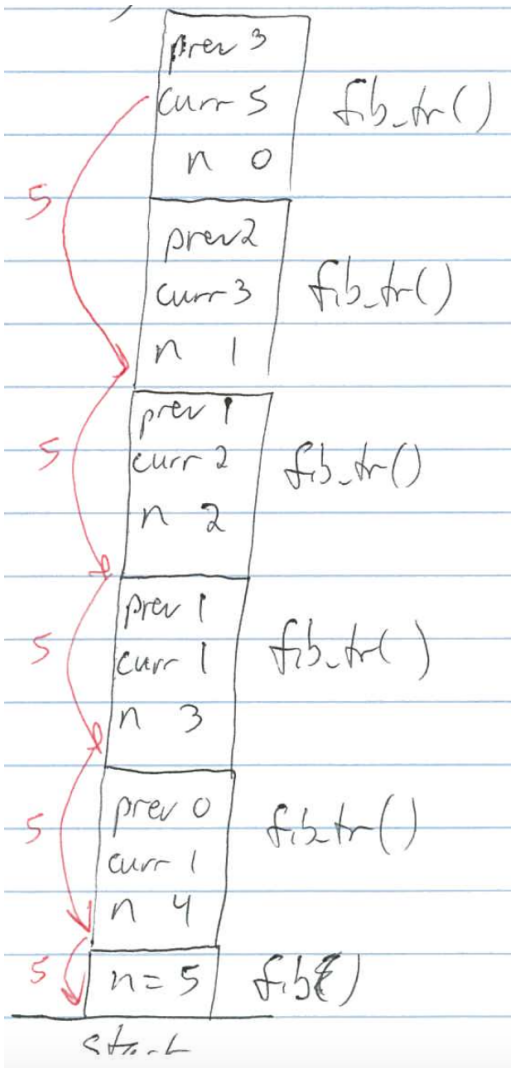
Trace

n	`prev`	`cur`	`next`
1	0	1	
2	1	1	1
3	1	2	2
4	2	3	3
5	3	5	5

Tail Recursive Fibonacci

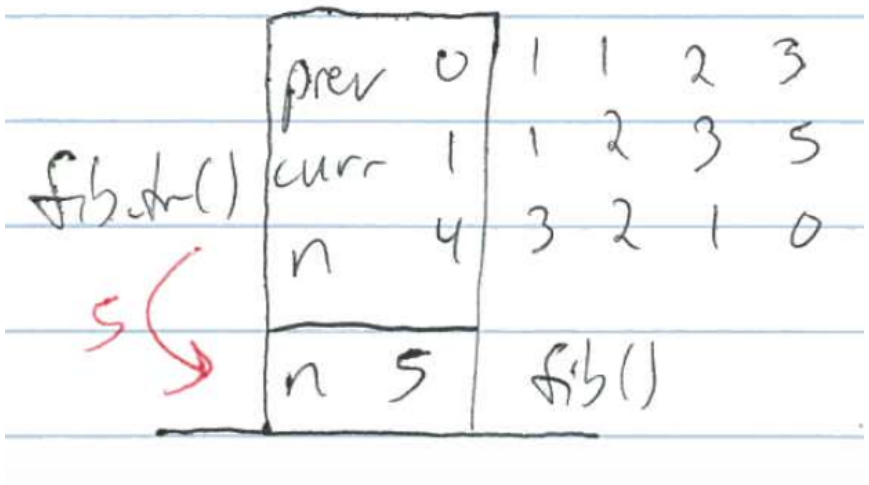
```
int fib_tr(int prev, int cur, int n){
  if(n==0) return cur;
  return fib_tr(cur, prev + cur, n-1);
}
int fib(int n){
 if(n==0) return 0;
 return fib_tr(0,1,n-1);
}
```

Picture


prev 3
curr 5
n 0
fib_tr()
5
prev2
curr 3
n 1
fib_tr()
5
prev 1
curr 2
n 2
fib_tr()
5
prev 1
curr 1
n 3
fib_tr()
5
prev 0
curr 1
n 4
fib_tr()
5
n=5
fib()
stack

Tail Call Elimination

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.

Count Change

```
#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;
}
```

Count Change

```
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);
}
```

Lambda Functions

- There are anonymous functions
- Useful for simple functions needed only in one place
- Unfortunately for us, C does not support them
- However, C++11 (and further) does support this type of function (it's a slightly different language but we will be able to get by)
- Compile with

```
% g++ -std=c++11, name.cpp
```

Examples

```
void (*f)(int) =
  [](int i){printf("%d\n",i);};
```

- The first part says this is a pointer to a function that takes an integer and has no return value
- The [] denotes the start of a lambda function
- Within the braces denotes the function body.
- Return type of the body is inferred from the body (or explicitly using the -> syntax)
- Calling `f(42);` will print out 42.
- Another way to call (type of g is inferred):

```
auto g = [](int i){printf("%d\n",i);};
```

Example with qsort

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  int a[] = {2,-10,14,42,11,-7,0,38};
  const int n = sizeof(a)/sizeof(a[0]);
  qsort(a,n,sizeof(int),
   [](const void *a, const void *b)
   { return *(int*)a-*(int *)b;}); //Change to
  for(int i=0; i<n; i++) printf("%d\n",a[i]);
  return 0;
}
```

Closure

Closure refers to a lambda function that captures variables from its containing scope. The following is only valid in

```
#include <stdio.h>
auto return_fib(){
   int prev = 0, cur =1;
   return [prev, cur]()mutable{
   int next = prev + cur;
   prev = cur;
   cur =next;
   return prev;}
}
```

- `[prev,cur]` captures copies of the variables
- `mutable` makes the captured variables writeable.

```
#include <stdio.h>
int main(void) {
   auto f1 = return_fib();
   auto f2 = return_fib();
   printf("%d %d %d\n", f1(),f1(),f1());
   printf("%d\n",f2());
   return 0;
}
```

Map and Reduce

- Map applies a transformation function to each element in an array, creating a new array.
- Reduce combines all elements into an array with one value.

Example

Below `void *b` is the destination array.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void map(const void *a, size_t n,
  size_t elem_a, void *b, size_t elem_b,
  void (*f)(const void *a, void *b)){
  for(int i=0; i<n; i++){
    f(a,b);
    a = (const char *)a + elem_a;
    b = (char *)b + elem_b;
  }
}
```

Example

Below, `void *b` is the reduction array.

```
void reduce(const void *a, size_t n,
  size_t elem_a, void *b,
  void (*f)(const void *a, void *b)){
  for(int i=0; i<n; i++){
    f(a,b);
    a = (const char *)a + elem_a;
  }
}
```

Main

```
int main(void) {
  char *sentence[] = {"A", "day", "without",
    "sunshine", "is", "like", "night"};
  const int n =
    sizeof(sentence)/sizeof(sentence[0]);
  int lengths[n];
  map(sentence, n, sizeof(char*), lengths,  sizeo
  [](const void *a,void *b) {*(int *)b=strlen(*(c
   int max = -1;
   reduce(lengths,n,sizeof(int), &max,
    [](const void *a,void *b)
   {if(*(int *)a > *(int *)b)
      *(int *)b = *(int *)a;}
  );
  printf("%d\n", max);
  return 0;
}
```