CPS 188

Computer Programming Fundamentals

Prof. Alex Ufkes

Topic 9.2: Branching recursion

Toronto Metropolitan University

Notice!

Obligatory copyright notice in the age of digital delivery and online classrooms:

Previously: Linear Recursion

Iterative Solution:

```
int factorial (int n)
{
   int i, fact = 1;
   for (i = n; i > 1; i--)
      fact = fact * i;
   return fact;
}
```

Recursive Solution:

```
int factorial (int n)
{
   if (n < 2)
      return 1;
   else
      return n*factorial(n-1);
}
```

```
int factorial (int n) {
  if (n < 2)
    return 1;
  else return
    n*factorial(n-1);
}
```
factorial(6)

```
int factorial (int n) {
  if (n < 2)
    return 1;
  else return
    n*factorial(n-1);
}
```
factorial(4)

```
int factorial (int n) {
  if (n < 2)
    return 1;
  else return
    n*factorial(n-1);
}
```
factorial(2)

```
int factorial (int n) {
  if (n < 2)
    return 1;
  else return
    n*factorial(n-1);
}
```
factorial(5)

```
int factorial (int n) {
  if (n < 2)
    return 1;
  else return
    n*factorial(n-1);
}
```
factorial(3)

```
int factorial (int n) {
  if (n < 2)
    return 1;
  else return
    n*factorial(n-1);
}
```
factorial(1)

- We have six *unique* function instances! Each has its own stack frame.
- They are all **different** *instances* of the **same** *function*.

Previously: Linear Recursion

```
int mult (int a, int b)
{
  int product = 0, i;
  for (i = 0; i < b; i++)
  {
    product += a;
  }
  return product;
}
```

```
int mult (int a, int b)
{
  if (b == 1)
    return(a);
  else
    return(a + mult(a, b-1));
}
```

Previously: Linear Recursion

```c
int sum_digits(int n)
{
    int sum = 0;
    while (n != 0) {
        sum += n % 10;
        n /= 10;
    }
    return sum ;
}
```

```c
int sum_digits(int n)
{
  if (n == 0)
    return 0;
  return n%10 + sum_digits(n/10);
}
```

Today

More Recursion Examples,
Branching Recursion,
Recursion Challenges

Example #4

Count Occurrences

Count Occurrences

Count the number of occurrences of a specific value in an array

```
int count_chars (const char* str, char key)
{
   if (str[0] == '\0’)
      return 0;
   else if (str[0] == key)
      return 1 + count_chars(str + 1, key);
   else
      return count_chars(str + 1, key);
}
```

Count Occurrences

Clever alternative?

```
int count_chars (const char* str, char key)
{
   if (str[0] == '\0’)
      return 0;
   return (str[0] == key) + count_chars(str+1, key);
}
```

Evaluates to 0 or 1

recursive_count.c

```c
#include <stdio.h>

int count_chars (const char* str, char key)
{
    if (str[0] == '\0')
        return 0;
    return (str[0] == key) + count_chars(str + 1, key);
}

int main (void)
{
    printf("%d\n", count_chars("Hello world!", 'l'));
    printf("%d\n", count_chars("abracadabra", 'a'));
    printf("%d\n", count_chars("abracadabra", 'r'));
    printf("%d\n", count_chars("", ' '));
    printf("%d\n", count_chars("Toronto Metropolitan University", 'o'));

    return (0);
}
```

C:\WINDOWS\SYSTEM32\cmd.exe

```
3
5
2
0
5

-----------------
(program exited

Press any key to
```

Example #5

Greatest Common Divisor

Greatest Common Divisor (GCD)

Given two integer inputs, what is the largest value that divides them both?

Use Euclid's algorithm to find `gcd(a,b)`:

1. If **b** is 0, **gcd(a, b)** is **a**
2. else, **gcd(a, b)** is **gcd(b, a % b)**

https://en.wikipedia.org/wiki/Euclidean_algorithm

Greatest Common Divisor (GCD)

Given two integer inputs, what is the largest value that divides them both?

Use Euclid's algorithm:

1. If **b** is 0, **gcd(a, b)** is **a**
2. else, **gcd(a, b)** is **gcd(b, a % b)**

```
int gcd (int a, int b)
{
    if (b == 0)
        return a;
    return gcd(b, a % b);
}
```

recursive_gcd.c

```c
#include <stdio.h>

int gcd (int a, int b)
{
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

int main (void)
{
    printf("%d\n", gcd(10, 4));
    printf("%d\n", gcd(63, 21));
    printf("%d\n", gcd(21, 49));
    printf("%d\n", gcd(7, 13));
    printf("%d\n", gcd(270, 192));

    return (0);
}
```

C:\WINDOWS\SYSTEM32\cmd.exe

```
2
21
7
1
6

--------------------

(program exited with code: 0)
```

Example #6

Recursive Selection Sort

```c
void sel_sort(int arr[], int size)
{
    for (int i = 0; i < size-1; i++)
    {
        int min_idx = i;
        for (int j = i + 1; j < size; j++)
        {
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }
        swap(&arr[i], &arr[min_idx]);
    }
}
```

Iterate through every element except the last

Variable to store index of smallest element

Find the index of smallest element. Note relationship between **i** and **j**!

Once found, swap smallest element with front of *unsorted* region

Selection Sort

Repeatedly find smallest value, move it to the front of the array:

We'll use familiar helper functions min_id() and swap():

```c
int min_id(const int *a, int len)
{
    int min_id = 0;
    for (int i = 1; i < len; i++)
        if (a[i] < a[min_id])
            min_id = i;
    return min_id;
}
```

```c
void swap (int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Recursive Selection Sort

```
void sel_sort(int arr[], int len)
{
    if (len == 1)
        return; // Do nothing, just return
    swap (&arr[0], &arr[ min_id(arr, len) ]);
    sel_sort(arr + 1, len - 1);
}
```

recursive_selsort.c

```c

void sel_sort(int arr[], int len)
{
    if (len == 1)
        return;
    swap (&arr[0], &arr[min_id(arr, len)]);
    sel_sort(arr + 1, len - 1);
}

int main (void)
{
    int nums[] = {17, 1, 4, 0, 2, 13, 9, 2};
    int n = sizeof(nums)/sizeof(int);

    print_arr(nums, n);
    sel_sort(nums, n);
    print_arr(nums, n);

    return (0);
}

```

C:\WINDOWS\SYSTEM32\cmd.exe

```
17 1 4 0 2 13 9 2
0 1 2 2 4 9 13 17



--------------------
(program exited with code: 0)
```

We’ve seen many recursive functions:

- Factorial
- Additive multiplication
- Digit summation
- Count occurrences
- Greatest common divisor
- Selection sort

Recursion or Iteration?

- Entirely problem dependent.
- Some tasks are far easier to solve recursively.
- Others are far easier to solve iteratively.
- One (or few) base cases? Obvious recursive case? Try recursion.
- Otherwise try iteration.
- Recursive functions can be very elegant, but don't force it

```
int factorial (int n)
{
  if (n < 2)
    return 1;
  return n*factorial(n-1);
}
```

```
int mult (int a, int b)
{
  if (b == 1)
    return(a);
  return(a + mult(a, b - 1));
}
```

Linear Recursion

- In each of the previous examples, we made a single recursive call at each step.
- We call this linear recursion. Execution happens in a straight line, no *branching*.
- Any linear recursion can straightforwardly be converted to iteration.
- Aside from slightly smaller or simpler code, we don't ***gain*** anything by using recursion.

Recursive Branching

The true power of recursion:

- The true power of recursion lies in the ability to branch in two or more directions.
- Contrast this with iteration - loops simply execute in a linear fashion until they are finished.
- Nested loops still execute in a single direction.
- In 2D, we move across the “rows” one at a time.
- To visualize, just imagine concatenating all the rows of a matrix end to end.

Recursive Branching

When would we branch in two or more directions?

Staying grounded:

- Solving `factorial(5)` requires solving `factorial(4)`
- Solving `mult(4, 3)` requires solving `mult(4, 2)`
- Solving `sum_digits(123)` requires solving `sum_digits(12)`

In each case above, solving the larger problem requires first solving a ***single*** self-similar sub-problem

Who can think of a problem that requires solving ***more than one*** self-similar sub-problem?

Fibonacci Sequence

- Each Fibonacci number is the sum of the **previous two**.
- We cannot compute the **20th** Fibonacci number without first computing the **18th** and **19th**
- Solving the larger sub-problem requires solving **TWO** self-similar sub-problems!

Recursive Fibonacci

```
int fib (int n)
{
    if (n <= 2)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

Base Case

***Two* Recursive Calls!**

fibonacci.c

```c
#include <stdio.h>

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

int main (void)
{
    printf("%d\n", fib(3));
    printf("%d\n", fib(4));
    printf("%d\n", fib(5));
    printf("%d\n", fib(6));
    printf("%d\n", fib(7));

    return (0);
}
```

- This is a very simple, elegant function.
- It has one teeny tiny problem…

C:\WINDOWS\SYSTEM32\cmd.exe

```
2
3
5
8
13


--------------------
(program exited with code: 0)
```

Try it Yourself

fibonacci.c

```c
#include <stdio.h>

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

int main (void)
{
    printf("%d\n", fib(500));

    return (0);
}
```

How long will it take to find the 500th Fibonacci number?

Recursive Fibonacci

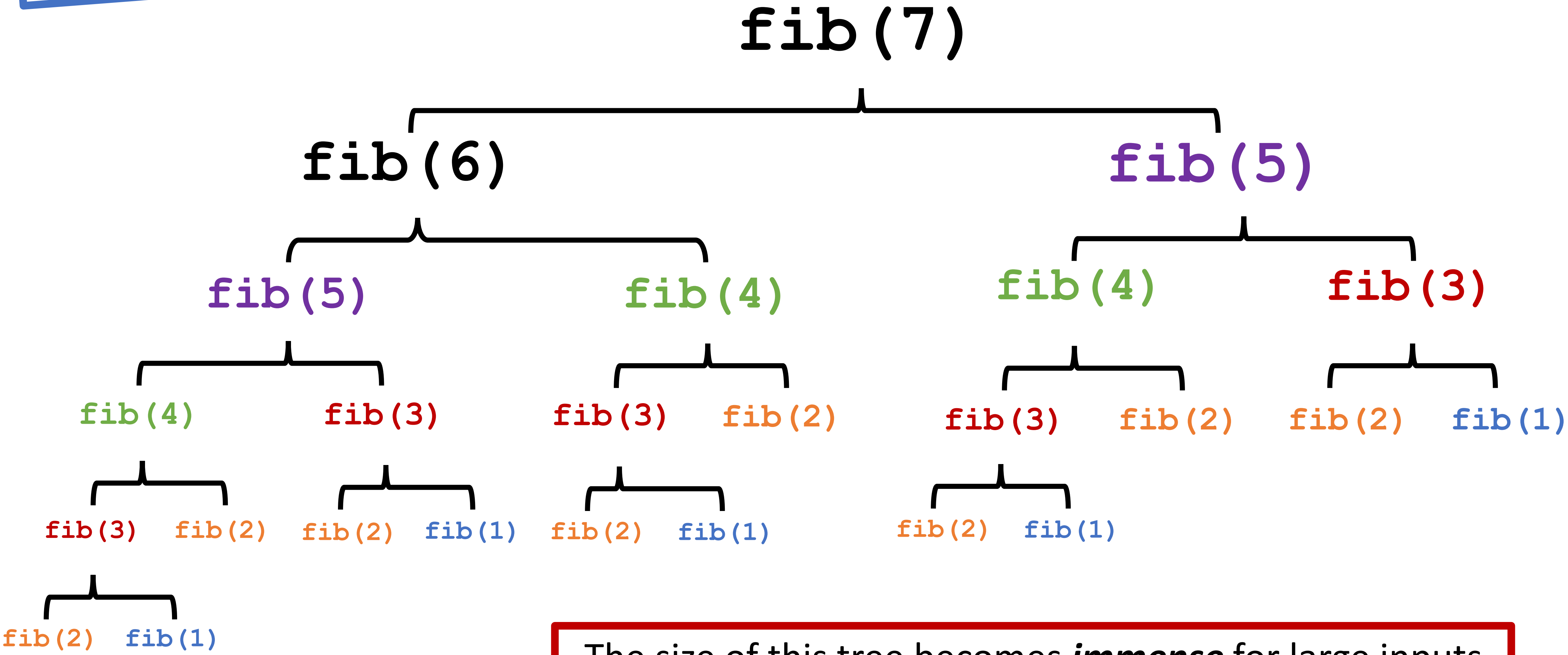
```
fib(n) = fib(n-1) + fib(n-2)
```

```
fib(10) = fib(9) + fib(8)
 fib(9) = fib(8) + fib(7)
 fib(8) = fib(7) + fib(6)
 fib(7) = fib(6) + fib(5)
 fib(6) = fib(5) + fib(4)
 fib(5) = fib(4) + fib(3)
 fib(4) = fib(3) + fib(2)
 fib(3) = fib(2) + fib(1)
 fib(2) = 1
 fib(1) = 1
```

Who sees the issue?

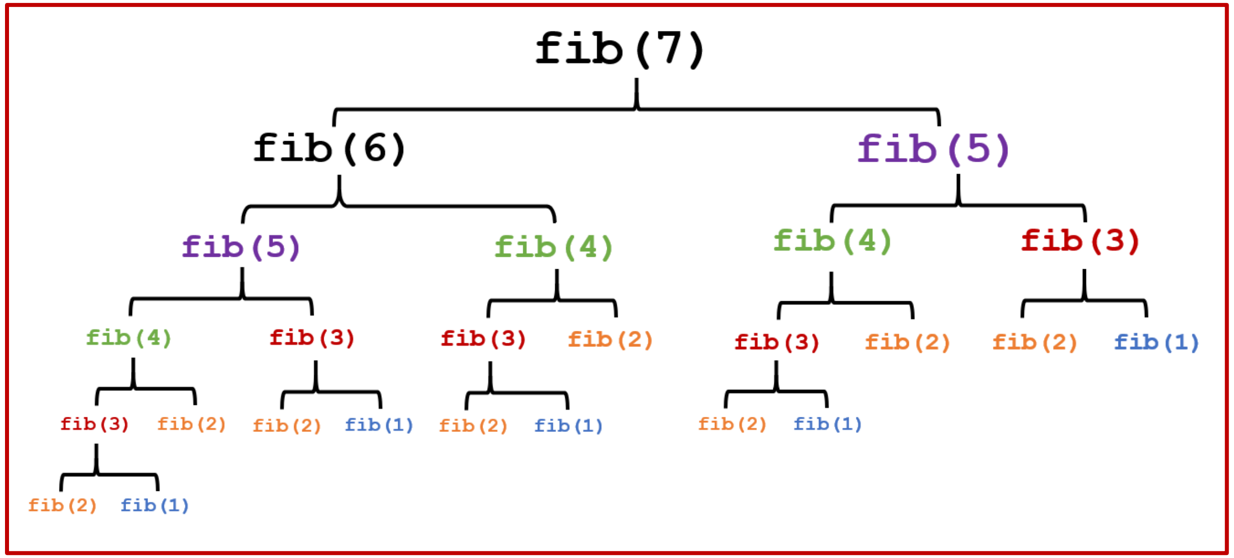
- Finding the 10th makes two recursive calls, to find 9th and 8th.
- Finding 9th also makes two recursive calls, to find 8th and 7th
- We're finding the 8th number twice here!
- 10th needs 8th, as does 9th
- These are called ***overlapping subproblems***

Overlapping Subproblems?



The size of this tree becomes ***immense*** for large inputs

Exponential Time Complexity



- Every layer of this tree has roughly twice as many calls as the layer above
- How many calls for fib(n)?
- Roughly, $\mathbf{2^n}$
- What is $\mathbf{2^{100}}$? $\mathbf{2^{500}}$?

$$2^{128} = 3.403^{38}$$

- In nanoseconds, this is around 10,000,000,000,000,000,000,000 years.
- Stars and galaxies won’t exist anymore. Good luck with that.

Don't Believe Me?

`fib(45)` took around 20 seconds to calculate

recursive_fibonacci.c

```c
int call_count = 0;

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

int main (void)
{
    for (int i = 5; i <
    {
        int f = fib(i);
        printf("F(%d) =
        call_count = 0;
    }

    return (0);
}
```

C:\WINDOWS\SYSTEM32\cmd.exe

```
F(5) = 5 -> Made 15 recursive calls
F(10) = 55 -> Made 177 recursive calls
F(15) = 610 -> Made 1973 recursive calls
F(20) = 6765 -> Made 21891 recursive calls
F(25) = 75025 -> Made 242785 recursive calls
F(30) = 832040 -> Made 2692537 recursive calls
F(35) = 9227465 -> Made 29860703 recursive calls
F(40) = 102334155 -> Made 331160281 recursive calls
F(45) = 1134903170 -> Made -622343491 recursive calls
```

Overflow!

VS Iterative Fibonacci?

```
int fib (int n)
{
    int a = 1, b = 1;
    for (int i = 3; i <= n; i++) {
        int tmp = b;
        b = a + b;
        a = tmp;
    }
    return b;
}
```

fibonacci.c

```c

int fib (int n)
{
    int a = 1, b = 1;
    for (int i = 3; i <= n; i++) {
        int tmp = b;
        b = a + b;
        a = tmp;
    }
    return b;
}

int main (void)
{
    printf("%d\n", fib(20));
    printf("%d\n", fib(21));
    printf("%d\n", fib(22));
    printf("%d\n", fib(23));
    printf("%d\n", fib(24));

    return (0);
}
```

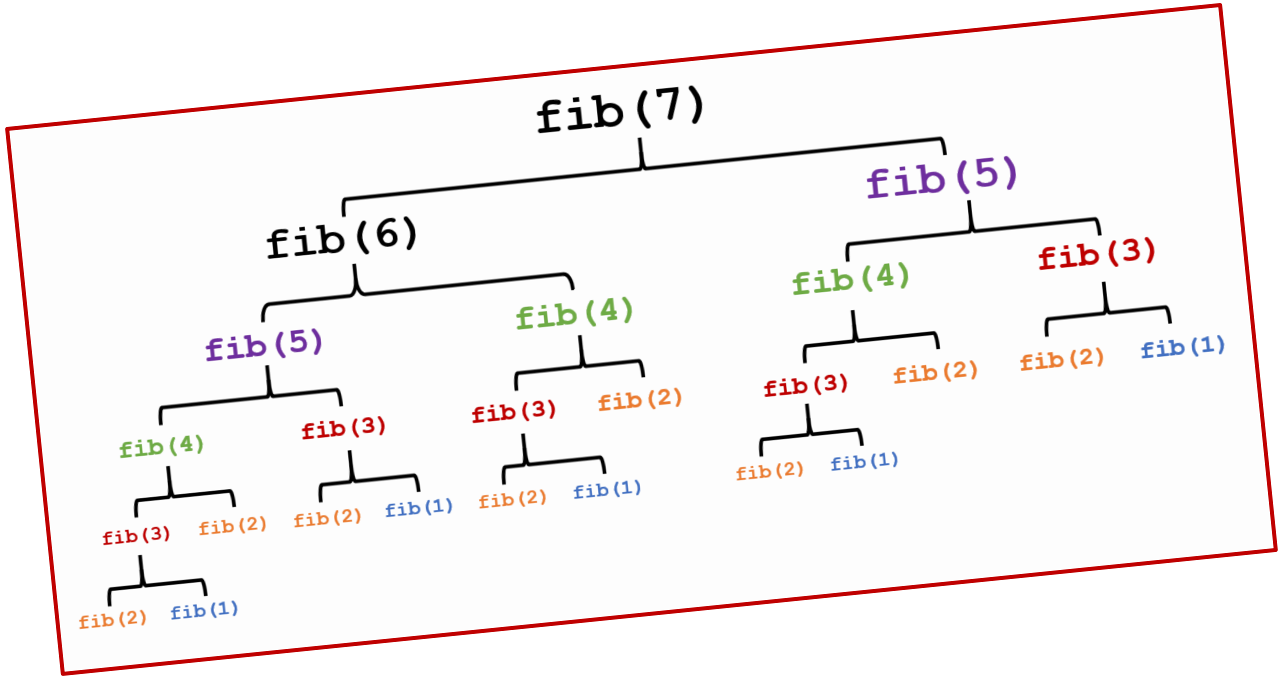
C:\WINDOWS\SYSTEM32\cmd.exe

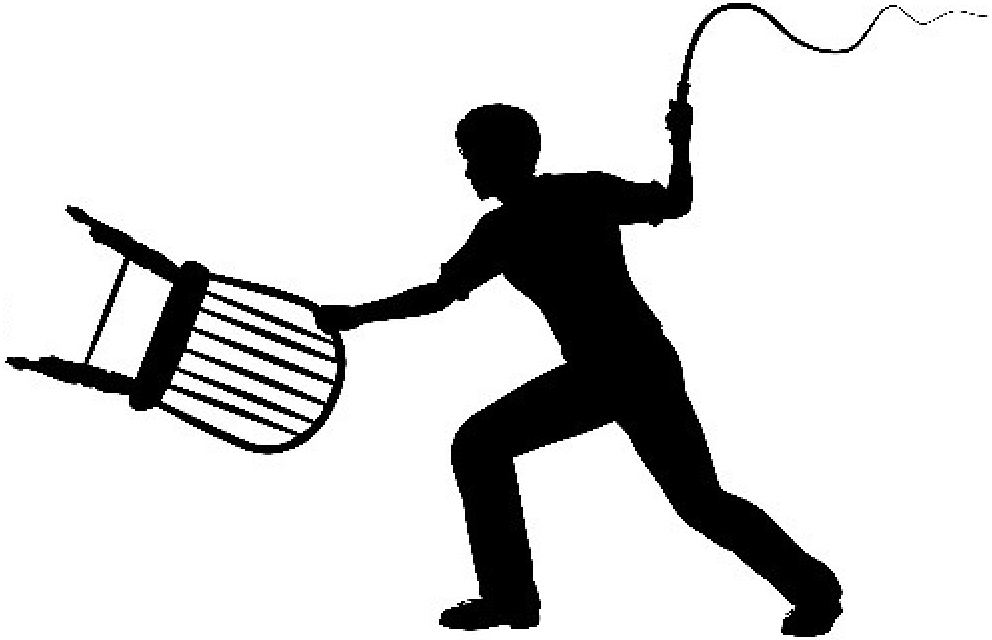
```
6765
10946
17711
28657
46368



------------------
(program exited with code: 0)

Press any key to continue . . .
```

Recursive Fibonacci: Overlapping Subproblems

- Avoiding overlapping subproblems for Fibonacci is easy enough, just iterate instead.
- If we insist on recursion, we can also *memoize*. Coming up.
- Not so simple for other more complex problems.
- Additionally, branching recursion ***in no way*** promises there will be overlapping subproblems (merge sort).
- This is a large topic in future courses CCPS305 and CCPS616

Taming Exponential Recursion





Memoization

- ***Memoization*** of a branching recursive function is one way to avoid recomputing overlapping subproblems.
- It’s a fancy made-up word for storing the solution to every previously computed subproblem.
- A very handy use for an array!

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

Bad! Do not want!

```c
int fib_table[500];

int fib (int n)
{
    if (n <= 1)
        fib_table[n] = n;
    else if (fib_table[n] == -1)
        fib_table[n] = fib(n-1) + fib(n-2);
    return fib_table[n];
}

int main (void)
{
    for (int i = 0; i < 500; i++)
        fib_table[i] = -1;
    …
}
```

Declare helper array in global scope

- Check for base cases, add them to array.
- Alternatively? Add base cases down in **main()** when we initialize **fib_table**.

- If we don't have fib(n) in the table…
- Calculate it and put it in the table!

Initialize helper array with dummy values in main()

recursive_fibonacci.c

```c

int fib_mem (int n)
{
    call_count++;
    if (n <= 1)
        fib_table[n] = n;
    else if (fib_table[n] == -1)
        fib_table[n] = fib_mem(n-1) + fib_mem(n-2);
    return fib_table[n];
}

int main (void)
{
    for (int i = 5; i <= 45; i += 5)
    {
        for (int i = 0; i < 500; i++) /
            fib_table[i] = -1;
        int f = fib_mem(i);
        printf("F(%d) = %d -> Made %d r
        call_count = 0;
    }

    return (0);
}

```

C:\WINDOWS\SYSTEM32\cmd.exe

```
F(5) = 5 -> Made 9 recursive calls
F(10) = 55 -> Made 19 recursive calls
F(15) = 610 -> Made 29 recursive calls
F(20) = 6765 -> Made 39 recursive calls
F(25) = 75025 -> Made 49 recursive calls
F(30) = 832040 -> Made 59 recursive calls
F(35) = 9227465 -> Made 69 recursive calls
F(40) = 102334155 -> Made 79 recursive calls
F(45) = 1134903170 -> Made 89 recursive calls
```

C:\WINDOWS\SYSTEM32\cmd.exe

```
F(5) = 5 -> Made 9 recursive calls
F(10) = 55 -> Made 19 recursive calls
F(15) = 610 -> Made 29 recursive calls
F(20) = 6765 -> Made 39 recursive calls
F(25) = 75025 -> Made 49 recursive calls
F(30) = 832040 -> Made 59 recursive calls
F(35) = 9227465 -> Made 69 recursive calls
F(40) = 102334155 -> Made 79 recursive calls
F(45) = 1134903170 -> Made 89 recursive calls
```

C:\WINDOWS\SYSTEM32\cmd.exe

```
F(5) = 5 -> Made 15 recursive calls
F(10) = 55 -> Made 177 recursive calls
F(15) = 610 -> Made 1973 recursive calls
F(20) = 6765 -> Made 21891 recursive calls
F(25) = 75025 -> Made 242785 recursive calls
F(30) = 832040 -> Made 2692537 recursive calls
F(35) = 9227465 -> Made 29860703 recursive calls
F(40) = 102334155 -> Made 331160281 recursive calls
F(45) = 1134903170 -> Made -622343491 recursive calls
```

Summary

Recursion:

- Base case, recursive case
- Linear VS branching recursion
- The good, the bad, the ugly
- Overlapping subproblems
- Memoization

Questions?