CS32 Discussion Section 1B Week 5

TA: Zhou Ren

Recursion

- Function-writing technique where the function refers to itself.

- Recall the following function:

```
int factorial(int n)
{
    if (n <= 1)
        return 1;

    return n * factorial(n – 1);
}
```

- Let us talk about how to come up with such a function.

Decomposition of the problem

- You're all used to the following technique.

```
int factorial(int n)
{
    int temp = 1;
    for (int i = 1; i <= n; i++)
        temp *= i;
    return temp;
}
```

- $n! = 1 * 2 * 3 * \ldots * (n-1) * n$

Decomposition of the problem

- You're all used to the following technique.

```
int factorial(int n)
{
    int temp = 1;
    for (int i = 1; i <= n; i++)
        temp *= i;
    return temp;
}
```

- n! = 1 * 2 * 3 * … * (n-1) * n
  = factorial(n-1)!

Decomposition of the problem

- You're all used to the following technique.

```
int factorial(int n)
{
    int temp = 1;
    for (int i = 1; i <= n; i++)
        temp *= i;
    return temp;
}
```

- n! = 1 * 2 * 3 * … * (n-1) * n
- n! = factorial(n-1) * n

Decomposition of the problem

```
int factorial(int n)
{
    int temp = factorial(n - 1) * n;


    return temp;
}
```

- n! = $\boxed{1 * 2 * 3 * \ldots * (n-1)}$ * n
- n! = factorial(n-1) * n

Power of Belief

- **BELIEVE factorial(n - 1) will do the right thing.**

```
int factorial(int n)
{
    int temp = factorial(n - 1) * n;


    return temp;
}
```

- factorial(n) will believe that factorial(n-1) will return the right value.
- factorial(n-1) will believe that factorial(n-2) will return the right value.
- …
- factorial(2) will believe that factorial(1) will return the right value.

Power of Belief

- <u>**BELIEVE factorial(n - 1) will do the right thing.**</u>

```
int factorial(int n)
{
    int temp = factorial(n - 1) * n;


    return temp;
}
```

- factorial(n) will believe that factorial(n-1) will return the right value.
- factorial(n-1) will believe that factorial(n-2) will return the right value.
- …
- factorial(2) will believe that factorial(1) will return the right value.
- **AND MAKE factorial(1) return the right value!**

Base Case

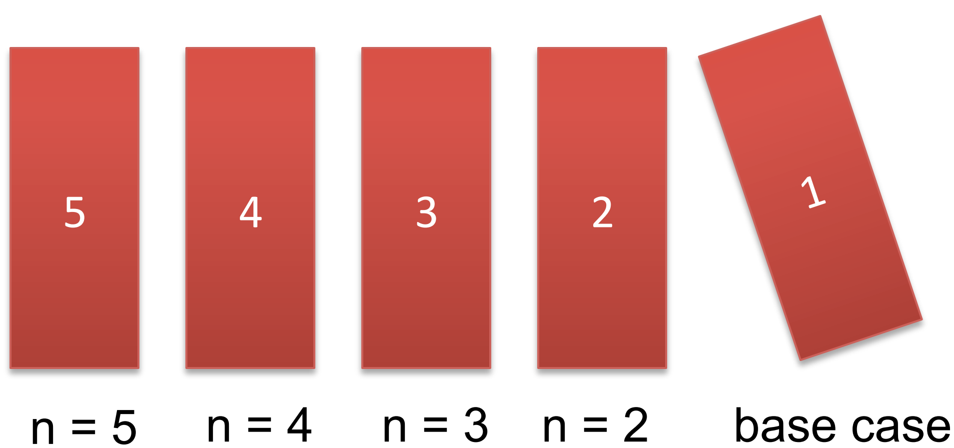
- **<u>BELIEVE factorial(n - 1) will do the right thing.</u>**

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

- factorial(n) will believe that factorial(n-1) will return the right value.
- factorial(n-1) will believe that factorial(n-2) will return the right value.
- …
- factorial(2) will believe that factorial(1) will return the right value.
- **AND MAKE factorial(1) return the right value!**

Base Case

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



Pattern

- How to Write a Recursive Function for Dummies
  1. Find the base case(s).
     - What are the trivial cases? e.g. empty string, empty array, etc.
     - When should the recursion stop?
  2. Decompose the problem.
     - Example: Tail recursion
       - Take the first (or last) of the n items of information.
       - Make a recursive call to the rest of n-1 items, believing the recursive call will give you the correct result.
       - Given this result and the information you have on the first (or last) item, conclude about the n items.
  3. Just solve this subproblem!

Pattern

- Write your function this way:

```
recursive_function(set of data)
        1. Take care of all base cases
        2. x = current data item
        3. result = recursive_function(set of data – {x})
        4. combine x and result
```

- Lastly, look back and make sure every call to this function hits some base case eventually.
- There are variations:
  – You may need to make multiple recursive calls.
  – You may make a different recursive call based on x.

Practice 1: Average

```
double average(const double arr[], int n)
{
      // assume n > 0









}
```

Practice 1: Average

- What is the base case?

- What is the relationship between (n)-step and (n-1)-step? That is, how do I get the average of n items, knowing the average of n-1 of them?

Practice 1: Average

- What is the base case?
- n = 1, where the average is just the value of the only item.

- What is the relationship between (n)-step and (n-1)-step? That is, how do I get the average of n items, knowing the average of n-1 of them?
- average(arr, n) = total(all n items) / n

  = {total(first n-1 items) + n-th item} / n

  = $\dfrac{\textbf{average(arr, n-1)} * (n-1) + \text{n-th item}}{n}$

Practice 1: Average

```
double average(const double arr[], int n)
{
    if (n == 1)
      return arr[0];

    double prevAvg = average(arr, n – 1);
    return ((n – 1) * prevAvg + arr[n – 1]) / n;
}
```

Practice 2: Summing Digits

```
int sumOfDigits(const int n)
{
    // assume n >= 0






}
```

Practice 2: Summing Digits

- What is the base case?

- What is the relationship between (n)-step and (n-1)-step? That is, how do I get the sum of n digits, knowing the sum of digits of (n-1) digits?

Practice 2: Summing Digits

- What is the base case?
- n < 10 (i.e. n is a single digit number), where the sum of digits is simply n

- What is the relationship between (n)-step and (n-1)-step? That is, how do I get the sum of n digits, knowing the sum of digits of (n-1) digits?
- Just add the last digit to the sum!

Practice 2: Summing Digits

```
int sumOfDigits(const int n)
{
   if (n < 10)
      return n;

   return n % 10 + sumOfDigits(n / 10);
}
```

Practice 3: Deleting characters

```
string deleteChar(const string &s, const char c)
{

}
```

Practice 3: Deleting characters

- What is the base case?
- What is the relationship between (n)-step and (n-1)-step?

Practice 3: Deleting characters

- What is the base case?
- s == “”: There is no character to delete! Just return “”.

- What is the relationship between (n)-step and (n-1)-step?
- Suppose the string is of length n, and you make a recursive call on s.substr(1). (e.g. If the string is “hello”, the recursive call will be made on “ello”.)
- What’s returned by deleteChar must not contain any c. You only need to append s[0] to it if s[0] != c. If s[0] == c, just return it.

Practice 3: Deleting characters

```
string deleteChar(const string &s, const char c)
{
   if (s.empty())
      return s;

   if (s[0] == c)
      return deleteChar(s.substr(1), c);
   else
      return s[0] + deleteChar(s.substr(1), c);
}
```

Practice 4: Fibonacci numbers

- Fibonacci numbers refer to the sequence of numbers of the following form:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

  - $F(0) = 0$
  - $F(1) = 1$
  - $F(n) = F(n-1) + F(n-2), n >= 2$

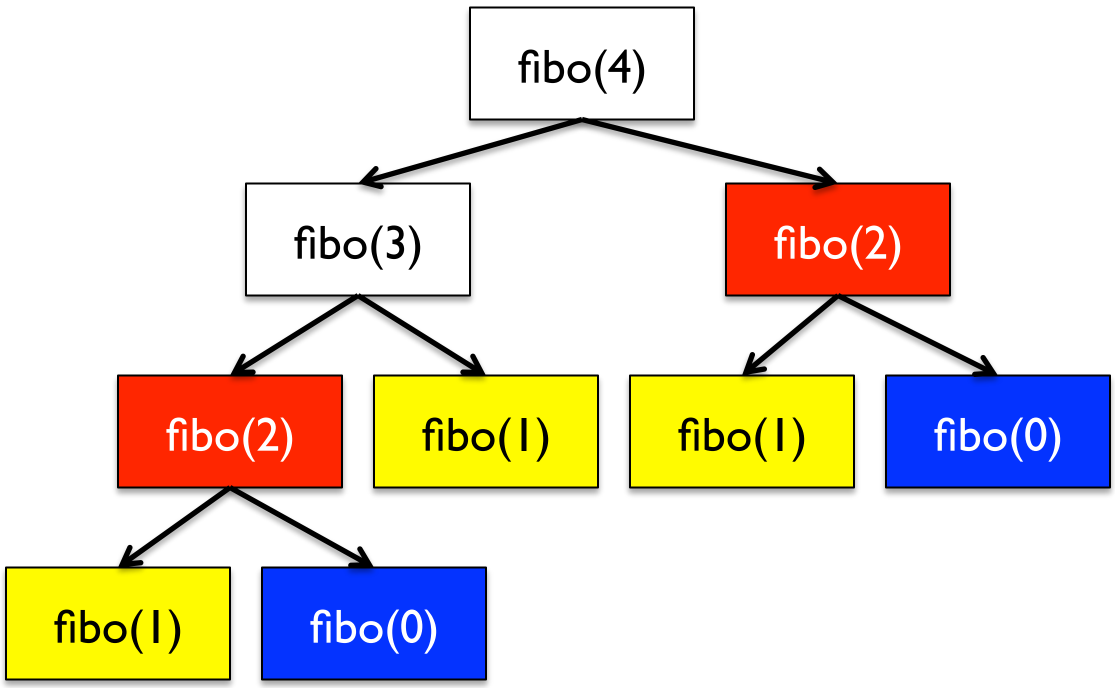
Practice 4: Fibonacci numbers

```
// A little too obvious now, isn't it?
int fibo(const int n)
{
   if (n == 0)
      return 0;
   if (n == 1)
      return 1;

   return fibo(n – 1) + fibo(n – 2);
}
```

Practice 4: Fibonacci numbers

- Note that fibo() makes two recursive calls.



- Look at all this redundancy!!!!

Trick: Memoization

```
int fibMem[100];  // global
for (int i = 2; i < 100; ++i)
   fibMem[i] = -1;
fibMem[0] = 0;
fibMem[1] = 1;

int fibo(const int n)
{
   int fib1, fib2;

   if (fibMem[n-1] != -1)
      fib1 = fibMem[n-1];
   else
      fib1 = fibo(n-1);

   if (fibMem[n-2] != -1)
     fib2 = fibMem[n-2];
   else
     fib2 = fibo(n-2);

   fibMem[n] = fib1 + fib2;

   return fib1 + fib2;
}
```

Memoization is an optimization technique that helps avoid computing the same value over and over by remembering it.

We like this because memory is cheap, but computing time is not!

Practice 5: Palindrome

- Examples: “eye”, “racecar”, “deed”

```
bool palindrome(const string &s)
{




}
```

- Base case?
- General case?

Practice 5: Palindrome

- Examples: “eye”, “racecar”, “deed”

```
bool palindrome(const string &s)
{


}
```

- Cases: (1) size = 0, (2) size = 1, (3) the first char differs from the last one, (4) first char = last char

Practice 5: Palindrome

- Examples: “eye”, “racecar”, “deed”

```
bool palindrome(const string &s)
{
  if (s.size() <= 1)
    return true;
  if (s[0] != s[s.size() - 1])
    return false;
  return palindrome(s.substr(1, s.size() - 2));
}
```

- Cases: (1) size = 0, (2) size = 1, (3) the first char differs from the last one, (4) first char = last char

More Practice

- Write reverse(), which recursively reverses a string and return the reversed version.

```
string reverse(const string &s)
{



}
```

More and More Practice

- Generate mnemonic phone numbers (e.g. 1-800-UCLA-CSD).
- Assume the following function is given:

```
string digitToLetters(char digit)
{
    switch (digit)
    {
        case '0': return "0";
        case '1': return "1";
        case '2': return "ABC";
        case '3': return "DEF";
        ...
        case '9': return "WXYZ";
        default: cout << "ERROR" << endl;  abort();
    }
}
```

More and More Practice

- Given `digitToLetters()`, write a function mnemonics, which prints all possible mnemonic numbers given the prefix and digits. (No restrictions here – you can use a loop.)

```
void mnemonics(const string &prefix, const string &digits);
```

`mnemonics("", "723");`	mnemonics("1-800-", "723")
PAD	1-800-PAD
PAE	1-800-PAE
PAF	1-800-PAF
PBD	1-800-PBD
PBE	1-800-PBE
PBF	1-800-PBF
…	…

Practice helps

- Recursion is somewhat counter-intuitive when confronted for the first time.
- Just do a lot of practice and you will see some patterns.
- Try finding more examples by googling.

- Again, the key to recursion is to “believe”! Do not try to track the call stack down and see what happens until you really have to.

Advanced Practice

- Given a set of integers, represented by an array s, write a function called exactSum that checks if the elements of some subset of s sums up to exactly target.

e.g.If s[] = {1, 2, 3}, then exactSum(s, 3, 6) returns true.
If s[] = {-3, 5, 2}, then exactSum(s, 3, 2) returns true.
If s[] = {-3, 5, 2}, then exactSum(s, 3, 8) returns false.
If s[] = {}, then exactSum(s, 0, 10) returns false.
If s[] = {-3, 5, 2, -10}, then exactSum(s, 3, 2) returns true.

```
bool exactSum(int a[], int n, int targetSum) {
}
```

Advanced Practice

- Write a isPrimeNumber function to test if a number is prime number or not.

How to do it in an un-iterative way? What's the algorithm?

```
Bool isPrimeNumber(int n){

}
```