# CS 240: HW 6 - 7

Gavi Dhariwal

February 10, 2022

## 1 Part 1

### 1.1 Programming Problem 1: Euclidean Algorithm

Algorithm:

```
def gcd(val1,val2):

    while val2%val1!=0: #as long as val2%val1 does not equal 0

        val1,val2 = val2%val1,val1 #reassign val2 to val 1 and val1 to val2%val1

    return val1 #return val1 before val2%val1 = 0
```

Testing:

- Input: $(9, 11)$
- Output: $1$
- Input: $(12, 18)$
- Output: $6$
- Input: $(26, 91)$
- Output: $13$
- Input: $(792, 1782)$
- Output: $99$

### 1.2 Programming Problem 2: Adding Binary Numbers

Algorithm:

```
def binary_add(n1, n2):

    result = [] #initialize final list that will contain the binary digits

    carry = 0 #initialize carry over variable

    for i in range(1, len(n1)+1): #for loop goes from 1 to length of input binary number

        col_sum = n1[-i] + n2[-i] + carry #negative index starts addition of columns from
        right to left

        carry = col_sum//2 #carry over check by dividing (and flooring) column sum by 2

        result.append(col_sum%2) #mod 2 because carry takes care of the sum if it
        goes beyond 1
```

```
    if (carry!=0):

        result.append(carry) #at the end of adding all digits, if there is a carry over,
        add it to the final binary sum

    return list(reversed(result)) #since .append adds numbers from left to right, we need
    #to reverse the final list to get the correct sum
```

Testing:

- Input: $49 : [1, 1, 0, 0, 0, 1],\ 41 : [1, 0, 1, 0, 0, 1]$
- Output: $90 : [1, 0, 1, 1, 0, 1, 0]$
- Input: $89 : [1, 0, 1, 1, 0, 0, 1],\ 108 : [1, 1, 0, 1, 1, 0, 0]$
- Output: $197 : [1, 1, 0, 0, 0, 1, 0, 1]$

### 1.3 Programming Problem 3: Decimal to Base

Algorithm:

```
def decimal_to_base(number, base):

    #Two if statements to catch a base < 1 or > 10

    if base > 10:
        return "Base is greater than 9"

    if base < 1:
        return "Base cannot be 0 or negative"

    result = [] #initialize final list that will contain the base digits

    mod_after = 0

    while number >= 1: #as long as the number does not reach 1

        mod_after = number%base #number(mod base)

        result.append(mod_after) #append the mod value

        number//=base #reduce the number by dividing (and flooring) it by the base

    return list(reversed(result)) #since .append adds numbers from left to right,
    we need to reverse the final list to get the correct sum
```

### 1.4 Programming Problem 4: Karatsuba Algorithm

Algorithm:

```
import math

global counts
counts = 0

def karatsuba(n1, n2):
    global counts #counter to keep track of number of recursive calls
    length1 = len(str(n1)) #number of digits in n1
    length2 = len(str(n2)) #number of digits in n2
    length = max(length1, length2)
    m = (length//2) #just in case n1 and n2 have different number of digits, we will
    split the numbers up by the largest number of digits (divided by 2 (and then floor))

    if n1<10 or n2<10:
        return (n1*n2) #if we receive single digits (just return the multiplication of
         those digits)

    else:
        #split of n1 into a and b
        a = n1//(10**m)
        b = n1%(10**m)

        #split of n2 into a and b
        c = n2//(10**m)
        d = n2%(10**m)

        ac = karatsuba(a, c) #multiplying a and c using Karatsuba
        counts += 1 #Karatsuba called!  increase counter by 1
        #print(ac)

        bd = karatsuba(b, d) #multiplying b and d using Karatsuba
        counts += 1 #Karatsuba called!  increase counter by 1
        #print(bd)

        ad_bc = karatsuba(a+b, c+d) #using Karatsuba to find (a+b) + (c+d)
        counts += 1 #Karatsuba called!  increase counter by 1
        #print(ad_bc)

        ad_bc_final = ad_bc - ac - bd #ad*bc = (a+b)*(c+d) - ac - bc
        #print(ad_bc_final)
```

```
    return((ac*(10**(2*m))) + ((ad_bc_final)*(10**(m))) + bd) #final return statement
    (formula for Karatsuba) for answer to n1*n2

    #return counts
```

Input:

```
print(karatsuba(12345678910111213141516171819202122232425262728293031323334353636 12,
13579111315171921232527293133353739414345474951535557596163656767 45))
```

Output:

1,676,433,481,817,705,266,129,514,737,418,074,430,686,007,117,601,914,669,622,664,786,590,398,446,623,924,151,007,039,889,745,117,721,236,422,791,304,327,820,391,425,527,602,940

# 2 Part 2

Problems from the Rosen book (sections 4.2 and 4.3 are done on paper; attached).

## 2.1 Problem K

### 2.1.1 a

Consider the following four digit numbers:

$$n1 = 1121, n2 = 3021$$

Every time the karatsuba method is called recursively, that event will be denoted by a Green Alert: K call! (+1)
Steps following the command `print(karatsuba(1121, 3021))`:

$$a = 11, b = 21, c = 30, d = 21$$

| ac = karatsuba(a, c) | bd = karatsuba(b, d) | ad_bc = karatsuba(a+b, c+d) |
|---|---|---|
| K call! (+1) | K call! (+1) | K call! (+1) |
| $a = 1, b = 1, c = 3, d = 0$ | $a = 2, b = 1, c = 2, d = 1$ | $a = 3, b = 2, c = 5, d = 1$ |

| k(1,3) | k(1,0) | k(2,3) | k(2,2) | k(1,1) | k(3,3) | k(3,5) | k(2,1) | k(5,6) |
|---|---|---|---|---|---|---|---|---|
| K call! (+1) | K call! (+1) | K call! (+1) | K call! (+1) | K call! (+1) | K call! (+1) | K call! (+1) | K call! (+1) | K call! (+1) |
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| 1 x 3 | 1 x 0 | 2 x 3 | 2 x 2 | 1 x 1 | 3 x 3 | 3 x 5 | 2 x 1 | 5 x 6 |

We anticipate 12 recursive calls. We'll have to change the code for the method to return the number of Karatsuba calls:

```
    #return((ac*(10**(2*m))) + ((ad_bc_final)*(10**(m))) + bd) # final return statement
    (formula for Karatsuba) for answer to n1*n2

    return counts
```

Input:
```
print(karatsuba(1211, 3021))
```

Output:
```
12
```

So our guess was correct. Let's try this for another pair of four digit numbers:

$$n1 = 6176, n2 = 5981$$

We expect to get 12 Karatsuba calls as an answer since we are multiplying a pair of four digit numbers.
Input:
```
print(karatsuba(6176, 5981))
```

Output:
```
18
```

Unfortunately, conjecture didn't work for these pair of numbers. But it did work for these pairs of numbers:

$$n1 = 1121, n2 = 2112$$

Input:
```
print(karatsuba(1121, 2112))
```

Output:
```
12
```

$$n1 = 4131, n2 = 6231$$

Input:
```
print(karatsuba(4131, 6231))
```

Output:
```
12
```

So what's going on? Let's investigate $n1 = 6176, n2 = 5981$. Steps following the command `print(karatsuba(6176, 5981))`:

$$a = 61, b = 76, c = 59, d = 81$$

ac = karatsuba(a, c)
K call! (+1)
$a = 6, b = 1, c = 5, d = 9$

bd = karatsuba(b, d)
K call! (+1)
$a = 7, b = 6, c = 8, d = 1$

| k(6,5) | k(1,9) | k(7,14) | k(7,8) | k(6,1) | k(13,9) |
|---|---|---|---|---|---|
| K call! (+1) | K call! (+1) | K call! (+1) | K call! (+1) | K call! (+1) | K call! (+1) |
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| 6 x 5 | 1 x 9 | 7 x 14 | 7 x 8 | 6 x 1 | 13 x 9 |

ad_bc = karatsuba(a+b, c+d)

K call! (+1)

$a = 13, b = 7, c = 14, d = 0$

k(13,14)

K call! (+1)

But, this will go into another Karatsuba call!

↓

$a = 1, b = 3, c = 1, d = 4$

k(1, 1) — K call! (+1) ↓ 1 x 1

k(3, 4) — K call! (+1) ↓ 3 x 4

k(4, 5) — K call! (+1) ↓ 4 x 5

k(7,0)

K call! (+1)

↓

7 x 0

k(20,14)

K call! (+1)

But, this will go into another Karatsuba call!

↓

$a = 2, b = 0, c = 1, d = 4$

k(2, 1) — K call! (+1) ↓ 2 x 1

k(0, 4) — K call! (+1) ↓ 0 x 4

k(2, 5) — K call! (+1) ↓ 2 x 5

Result: 18 recursive calls

Hypothesis: On the first Karatsuba call, if $(a + b)$ or $(c + d)$ goes beyond 100, then there will be additional Karatsuba calls. As we can see in the example above, in the very first Karatsuba call, $a + b = 137$ and $b + d = 140$; hence there are 6 additional Karatsuba calls (three each for $(a + b)$ or $(c + d)$ going beyond 100).

Why 100?: For four digit numbers, $a, b, c, d$ are going to be two digit numbers. Now in the first call, if $(a + b)$ or $(c + d)$ go beyond 100, they would need to go through an additional Karatsuba call to bring those digits down below 100. And since this is done recursively, we end up getting six additional calls.

If $(a + b)$ or $(c + d)$ goes beyond $10^{(4/2)}$, then recursive calls increase than expected. $\begin{cases} a = 91 \\ b = 51 \\ c = 87 \\ d = 61 \end{cases}$

Generalization: If we are using Karatsuba to multiply to two $2^n$ digit numbers, and in the first recursive call $(a + b)$ or $(c + d)$ goes beyond $10^{\texttt{floor(math.log10(n1)) + 1}}$, then the recursive calls would increase.

Example:

| Number of Digits | Least Number of Calls | Maximum Number of Calls |
|---|---|---|
| 4 | $(1000, 1000) = 9$ recursive calls | $(9999, 9999) = 27$ recursive calls |
| 8, | $(10000000, 10000000) = 21$ recursive calls | $(99999999, 99999999) = 93$ recursive calls |
| 16 | $(10^{16}, 10^{16}) = 45$ recursive calls | $(9999999999999999, 9999999999999999) = 294$ recursive calls |
| . | . | . |
| . | . | . |
| . | . | . |