2023-04-21

Upcoming Schedule

Monday	Wednesday	Friday
04/17/2023 (90) • Review: Unit 5, Unit 9 **• AP CS Question 2: Classes**	**04/19/2023 (90)** • Review: Units 6-7 **• AP CS Question 3: Array/ArrayList**	**04/21/2023 (45)** • Review: Unit 10 • More recursion exercises like we did on Apr-7
04/24/2023 (90) • Review: Unit 8 **• AP CS Question 4: 2D Array**	**04/26/2023 (90)** **• AP CS Multiple Choice Game**	**04/28/2023 (45)** • Review: Unit 7, Unit 10 • Algorithms: Iterative/recursive binary search, selection sort, insertion sort, merge sort
05/01/2023 **• FINAL**	**05/03/2023** **• AP EXAM**	

Unit 10
Recursion

Recursive Terminology

Recursion	Recursive Programming
The definition of an operation in terms of itself. Solving a problem using recursion depends on solving smaller occurrences of the same problem.	Writing methods that call themselves to solve problems recursively. ● An equally powerful alternative to iteration (for, while loops, etc.) ● Particularly well-suited for solving certain types of problems.

Recursive Terminology

Base Case	Recursive Case
The **base case** of a recursive method is the case where it does not recursively call itself, that is, the method terminates. The base case is a problem that is so simple, we already know the answer to it!	The **recursive case**, or **general case**, is the case where the method calls itself. It's called the general case because it's the case that usually happens when a recursive algorithm is executing. For the algorithm to work, the recursive case must diminish the problem so that it eventually approaches the base case.

Some recursive algorithms have more than one base or recursive case, but all have at least one of each. A crucial part of recursive programming is identifying these cases.

Tracing Recursive Methods

On the the AP Exam, you may be asked to trace through "mystery" recursive methods and determine what is returned or output.

This isn't as simple as tracing through loops, where you can build a trace table, because a recursive method may call itself and then do additional manipulation on the result.

What you can do is draw a box every time you encounter a recursive call, and inside that box, determine the result of that recursive call.

(This may require drawing more boxes inside the boxes, recursively...)

Recursive Tracing

Consider the following recursive method.

```
public static int mystery(int n) {
  if (n < 10) {
    return (10 * n) + n;
  } else {
    int a = mystery(n / 10);
    int b = mystery(n % 10);
    return (100 * a) + b;
  }
}
```

What is the result of `mystery(348)`?

Recursive Trace

```
public static int mystery(int n) {
  if (n < 10) {
    return (10 * n) + n;
  } else {
    int a = mystery(n / 10);
    int b = mystery(n % 10);
    return (100 * a) + b;
  }
}
```

```
mystery(348):

- int a = mystery(34);
    - int a = mystery(3);
        - return (10 * 3) + 3;      // 33
    - int b = mystery(4);
        - return (10 * 4) + 4;      // 44
    - return (100 * 33) + 44;       // 3344
- int b = mystery(8);
    - return (10 * 8) + 8;          // 88
- return (100 * 3344) + 88;         // 334488
```

Calculating modulo (%)

```
500 % 16 = ?

      31
    +----
16 | 500
     48
     ---
      20
      16
      --
       4
```

```
31 % 16 = ?

      1
    +----
16 | 31
     16
     --
     15
```

```
 0 % 16 == 0
 1 % 16 == 1
 2 % 16 == 2
 3 % 16 == 3
 4 % 16 == 4
 5 % 16 == 5
 6 % 16 == 6
 7 % 16 == 7
 8 % 16 == 8
 9 % 16 == 9
10 % 16 == 10
11 % 16 == 11
12 % 16 == 12
13 % 16 == 13
14 % 16 == 14
15 % 16 == 15
16 % 16 == 0
17 % 16 == 1
18 % 16 == 2
```

Recursive Trace Exercise (10 minutes)

```java
// Precondition: i >= 0
public String i2x(int i) {
  int j = i % 16;
  String c = "0123456789abcdef".substring(j, j+1);
  if (i >= 16) {
    return i2x(i / 16) + c;
  } else {
    return c;
  }
}
```

What is printed?

```java
System.out.println(i2x(15));
System.out.println(i2x(500));
```

Recursive Trace Exercise (10 minutes) – ANSWERS

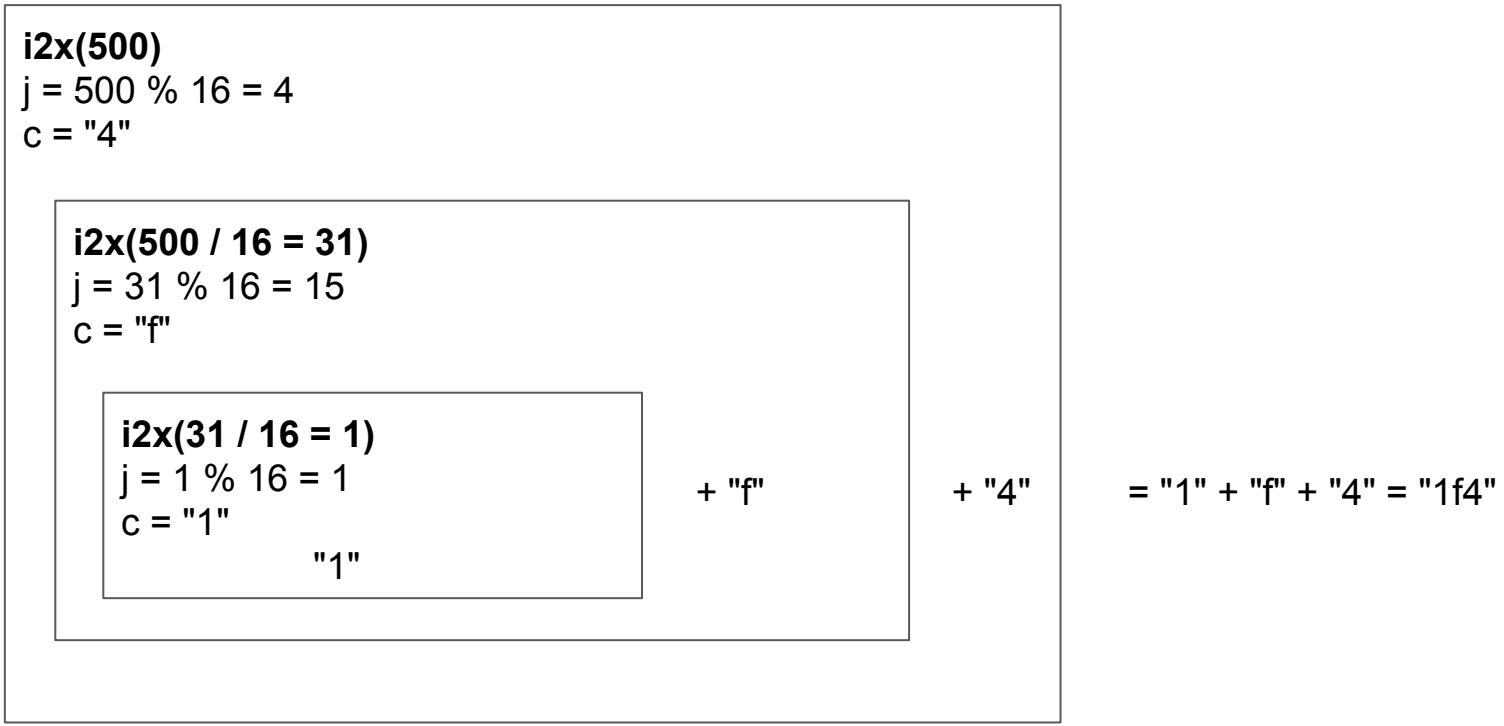
```java
// Precondition: i >= 0
public String i2x(int i) {
  int j = i % 16;
  String c = "0123456789abcdef".substring(j, j+1);
  if (i >= 16) {
    return i2x(i / 16) + c;
  } else {
    return c;
  }
}
```

What is printed?

```
System.out.println(i2x(15));
> f

System.out.println(i2x(500));
> 1f4
```

i2x(500) = i2x(500 / 16) + "4" = i2x(31) + "4"
= (i2x(31 / 16) + "f") + "4" = i2x(1) + "f" + "4" = "1" + "f" + "4" = "1f4"


i2x(500)
j = 500 % 16 = 4
c = "4"
i2x(500 / 16 = 31)
j = 31 % 16 = 15
c = "f"
i2x(31 / 16 = 1)
j = 1 % 16 = 1
c = "1"
"1"
+ "f"
+ "4"
= "1" + "f" + "4" = "1f4"

Recursive Trace Exercise (10 minutes)

```java
// Precondition: s != null && s.length() > 0
public int o2i(String s) {
  int len = s.length();
  String t = s.substring(len - 1);
  int i = "01234567".indexOf(t);
  if (i == -1) {
    throw new IllegalArgumentException();
  }
  if (len > 1) {
    return 8 * o2i(s.substring(0, len - 1)) + i;
  } else {
    return i;
  }
}
```

What is printed?

```java
System.out.println(o2i("4"));
System.out.println(o2i("3747"));
```

Recursive Trace Exercise (10 minutes) – ANSWERS

```java
// Precondition: s != null && s.length() > 0
public int o2i(String s) {
  int len = s.length();
  String t = s.substring(len - 1);
  int i = "01234567".indexOf(t);
  if (i == -1) {
    throw new IllegalArgumentException();
  }
  if (len > 1) {
    return 8 * o2i(s.substring(0, len - 1)) + i;
  } else {
    return i;
  }
}
```

What is printed?

```
System.out.println(o2i("4"));
> 4

System.out.println(o2i("3747"));
> 2023
```

o2i("3747") = 8 * o2i("374") + 7 = 8 * (8 * o2i("37") + 4) + 7
= 8 * (8 * (8 * o2i("3") + 7) + 4) + 7 = 8*(8*(8*3+7)+4)+7 = 2023

What was this method doing? Hexadecimal (Base 16)

```
// Precondition: i >= 0
public String i2x(int i) {
  int j = i % 16;
  String c = "0123456789abcdef".substring(j, j+1);
  if (i >= 16) {
    return i2x(i / 16) + c;
  } else {
    return c;
  }
}
```

What is printed?

```
System.out.println(i2x(500));
> 1f4

System.out.println(Integer.toString(500, 16));
> 1f4
```

Hexadecimal is commonly used enough that converting an `int`/`long` to a hex string is built-in to Java.

What was this method doing? Octal (Base 8)

```java
// Precondition: s != null && s.length() > 0
public int o2i(String s) {
  int len = s.length();
  String t = s.substring(len - 1);
  int i = "01234567".indexOf(t);
  if (i == -1) {
    throw new IllegalArgumentException();
  }
  if (len > 1) {
    return 8 * o2i(s.substring(0, len - 1)) + i;
  } else {
    return i;
  }
}
```

What is printed?

```
System.out.println(o2i("3747"));
> 2023
System.out.println(Integer.parseInt("3747", 8));
> 2023
```

Converting from `String` -> `int`, `long` in bases other than 10 is also directly supported by Java.

Hexadecimal

Computers understand binary, 1's and 0's, but binary is hard for humans to read.

1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	1
9				**4**				**E**				**1**			

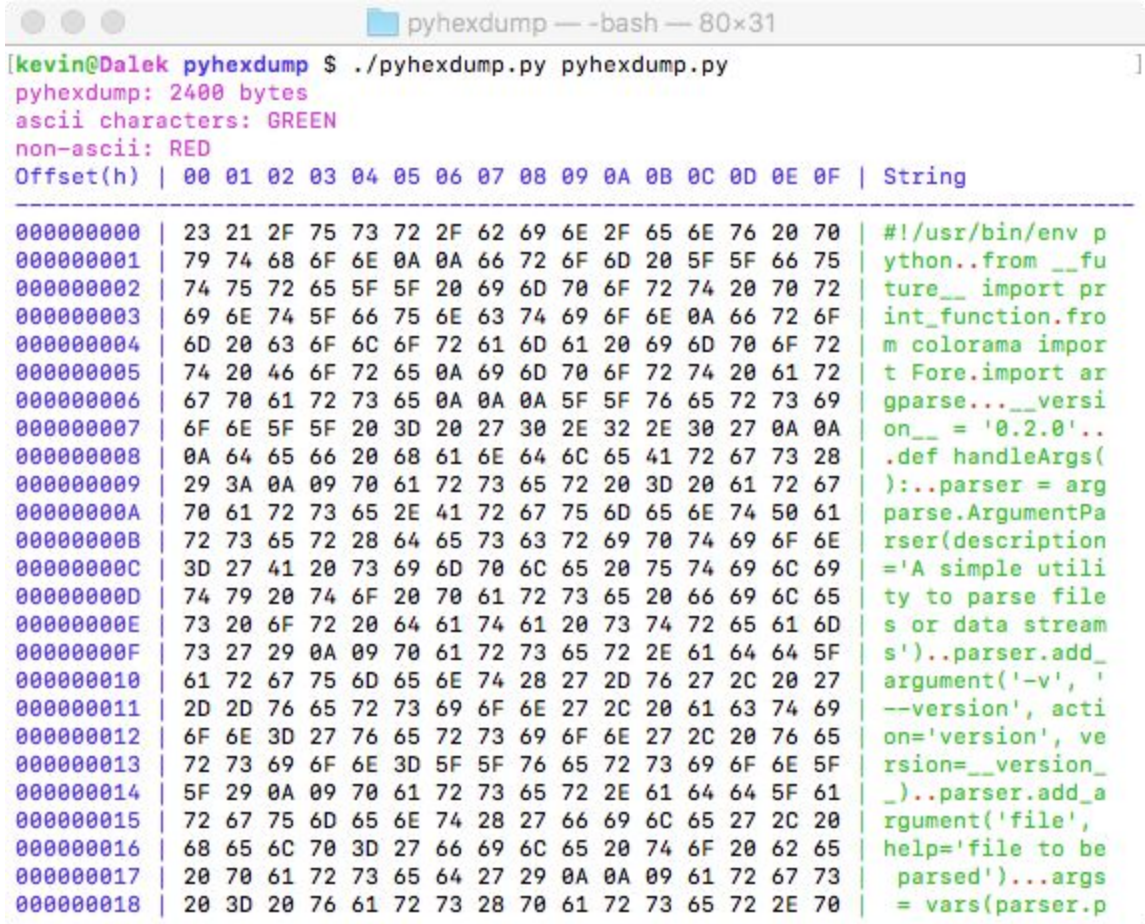
When humans work with binary data, it is common to use hexadecimal (base 16).
Every 4 bits (a *nibble*) maps to a standard digit (0-9) or one of six extra digits (A-F).

0000	0
0001	1
0010	2
0011	3

0100	4
0101	5
0110	6
0111	7

1000	8
1001	9
1010	A (10)
1011	B (11)

1100	C (12)
1101	D (13)
1110	E (14)
1111	F (15)


pyhexdump — -bash — 80×31
[kevin@Dalek pyhexdump $ ./pyhexdump.py pyhexdump.py
pyhexdump: 2400 bytes
ascii characters: GREEN
non-ascii: RED
Offset(h) | 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F | String
000000000 | 23 21 2F 75 73 72 2F 62 69 6E 2F 65 6E 76 20 70 | #!/usr/bin/env p
000000001 | 79 74 68 6F 6E 0A 0A 66 72 6F 6D 20 5F 5F 66 75 | ython..from __fu
000000002 | 74 75 72 65 5F 5F 20 69 6D 70 6F 72 74 20 70 72 | ture__ import pr
000000003 | 69 6E 74 5F 66 75 6E 63 74 69 6F 6E 0A 66 72 6F | int_function.fro
000000004 | 6D 20 63 6F 6C 6F 72 61 6D 61 20 69 6D 70 6F 72 | m colorama impor
000000005 | 74 20 46 6F 72 65 0A 69 6D 70 6F 72 74 20 61 72 | t Fore.import ar
000000006 | 67 70 61 72 73 65 0A 0A 0A 5F 5F 76 65 72 73 69 | gparse..__versi
000000007 | 6F 6E 5F 5F 20 3D 20 27 30 2E 32 2E 30 27 0A 0A | on__ = '0.2.0'..
000000008 | 0A 64 65 66 20 68 61 6E 64 6C 65 41 72 67 73 28 | .def handleArgs(
000000009 | 29 3A 0A 09 70 61 72 73 65 72 20 3D 20 61 72 67 |):..parser = arg
00000000A | 70 61 72 73 65 2E 41 72 67 75 6D 65 6E 74 50 61 | parse.ArgumentPa
00000000B | 72 73 65 72 28 64 65 73 63 72 69 70 74 69 6F 6E | rser(description
00000000C | 3D 27 41 20 73 69 6D 70 6C 65 20 75 74 69 6C 69 | ='A simple utili
00000000D | 74 79 20 74 6F 20 70 61 72 73 65 20 66 69 6C 65 | ty to parse file
00000000E | 73 20 6F 72 20 64 61 74 61 20 73 74 72 65 61 6D | s or data stream
00000000F | 73 27 29 0A 09 70 61 72 73 65 72 2E 61 64 64 5F | s')..parser.add_
000000010 | 61 72 67 75 6D 65 6E 74 28 27 2D 76 27 2C 20 27 | argument('-v', '
000000011 | 2D 2D 76 65 72 73 69 6F 6E 27 2C 20 61 63 74 69 | --version', acti
000000012 | 6F 6E 3D 27 76 65 72 73 69 6F 6E 27 2C 20 76 65 | on='version', ve
000000013 | 72 73 69 6F 6E 3D 5F 5F 76 65 72 73 69 6F 6E 5F | rsion=__version_
000000014 | 5F 29 0A 09 70 61 72 73 65 72 2E 61 64 64 5F 61 | _)..parser.add_a
000000015 | 72 67 75 6D 65 6E 74 28 27 66 69 6C 65 27 2C 20 | rgument('file',
000000016 | 68 65 6C 70 3D 27 66 69 6C 65 20 74 6F 20 62 65 | help='file to be
000000017 | 20 70 61 72 73 65 64 27 29 0A 0A 09 61 72 67 73 | parsed')...args
000000018 | 20 3D 20 76 61 72 73 28 70 61 72 73 65 72 2E 70 | = vars(parser.p

Octal

Octal is base 8, so 3 bits per digit, and was also used for humans to interact with binary data. It's mostly fallen into disuse, but hangs out in a few places (Unix file permissions)

0	1	1	1	1	1	1	0	0	1	1	1
3			7			4			7		

$3 * 8^3 + \quad 7 * 8^2 + \quad 4 * 8^1 + \quad 7 * 8^0 \quad = 2023$

In decimal, you have a 1's place, a 10's place, a 100's place. These are powers of 10 (10^0, 10^1, 10^2, 10^3, ...)

In octal, each place is 8x the previous one, so 1, 8, 64, 512...

In hexadecimal, it's 16^0=1, 16^1=16, 16^2=256, 16^3=4096, ...

And in binary, of course, it's powers of 2.

000	0
001	1
010	2
011	3

010	4
101	5
110	6
111	7

Java: `0x` and `0` prefixes

In Java, we usually specify numbers in decimal (base 10). Java actually lets you write numbers in hex or octal.

An integer prefixed with `0x` is hexadecimal.

An integer prefixed with `0` is octal.

```
System.out.println(0777);
> 511

System.out.println(03747);
> 2023

System.out.println(0x100);
> 256
```

One More Recursive Trace Exercise (5 minutes)

```
public int p(int r, int c) {
  if (c == 0) {
    return 1;
  } else if (c > r) {
    return 0;
  } else {
    return p(r-1, c) + p(r-1, c-1);
  }
}
```

What is printed?

```
System.out.println(p(3, 0));
System.out.println(p(3, 1));
System.out.println(p(3, 2));
System.out.println(p(3, 3));
```

Hint: Write out equations:

p(3, 0) = 1
p(3, 1) = p(2, 1) + p(2, 0)
p(3, 2) = ???
p(3, 3) = ???
p(2, 0) = ???
p(2, 1) = ???
p(2, 2) = ???
p(2, 3) = ???
p(1, 0) = ???
p(1, 1) = ???
p(1, 2) = ??? ...

One More Recursive Trace Exercise – ANSWERS

```
public int p(int r, int c) {
  if (c == 0) {
    return 1;
  } else if (c > r) {
    return 0;
  } else {
    return p(r-1, c) + p(r-1, c-1);
  }
}
```

What is printed?

```
System.out.println(p(3, 0));
> 1

System.out.println(p(3, 1));
> 3

System.out.println(p(3, 2));
> 3

System.out.println(p(3, 3));
> 1
```

One More Recursive Trace Exercise – ANSWERS

Write out the equations, starting with what you need to solve. Base cases, handle directly.

```
p(3, 0) = 1
p(3, 1) = p(2, 1) + p(2, 0)
p(3, 2) = p(2, 2) + p(2, 1)
p(3, 3) = p(2, 3) + p(2, 2)

p(2, 0) = 1
p(2, 1) = p(1, 1) + p(1, 0)
p(2, 2) = p(1, 2) + p(1, 1)
p(2, 3) = 0

p(1, 0) = 1
p(1, 1) = p(0, 1) + p(0, 0)
p(1, 2) = 0

p(0, 0) = 1
p(0, 1) = 0
```

Then, working backward from simplest to most complicated, do the substitutions.

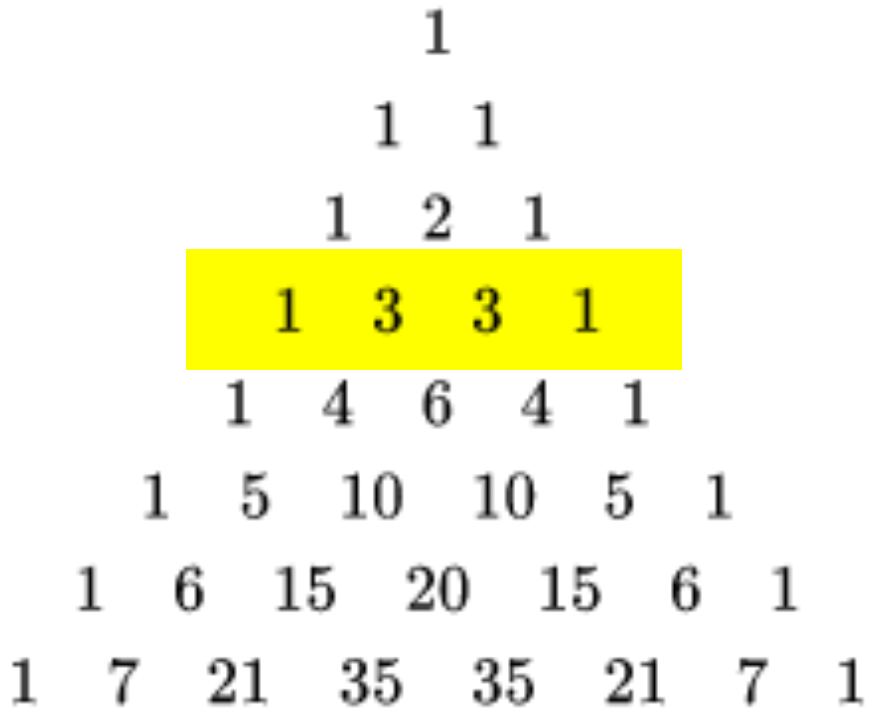
```
p(0, 0) = 1
p(0, 1) = 0

p(1, 0) = 1
p(1, 1) = p(0, 1) + p(0, 0) = 1
p(1, 2) = 0

p(2, 0) = 1
p(2, 1) = p(1, 1) + p(1, 0) = 2
p(2, 2) = p(1, 2) + p(1, 1) = 1
p(2, 3) = 0

p(3, 0) = 1
p(3, 1) = p(2, 1) + p(2, 0) = 2 + 1 =
3
p(3, 2) = p(2, 2) + p(2, 1) = 1 + 2 =
3
```

Pascal's Triangle



Pascal's Triangle, aka the Yang Hui Triangle or Khayyam Triangle, was discovered 1,000+ years ago and has applications in algebra, combinatorics, and probability theory.

`p(r, c)` calculates the value of Pascal's Triangle ar row `r` and column `c`

Pascal's Triangle has many interesting mathematical properties. One is that it gives the coefficients in the expansion of any binomial expression.

$$
\begin{aligned}
(a+b)^0 &= 1 \\
(a+b)^1 &= a+b \\
(a+b)^2 &= a^2+2ab+b^2 \\
(a+b)^3 &= a^3+3a^2b+3ab^2+b^3 \\
(a+b)^4 &= a^4+4a^3b+6a^2b^2+4ab^3+b^4 \\
(a+b)^5 &= a^5+5a^4b+10a^3b^2+10a^2b^3+5ab^4+b^5
\end{aligned}
$$

It can also be used to calculate the probability of a winning hand of cards!