CS1010E: Programming Methodology

Tutorial 06: Recursion

TBA

1. Discussion Questions

(a) [Linear Recursion] What is/are the output of code fragments below?

i.
```c
int foo(int a, int b) {
  if(b == 0) return a;
  return foo(a+1, b-1);
}
int main() {
  printf("%d", foo(10, 15));
}
```
i. _____

ii.
```c
int foo(int a, int b) {
  if(b == 0) return 0;
  return a + foo(a, b-1);
}
int main() {
  printf("%d", foo(10, 15));
}
```
ii. _____

iii.
```c
int D = 0, R = 0;
void foo(int a, int b) {
  if(a < b) R = a;
  else {
    D++; foo(a-b, b);
  }
}
int main() {
  int a = 27, b = 4;
  foo(a, b);
  printf("%d", b*D+R);
}
```
iii. _____

2. Program Analysis

(a) [Non-Linear Recursion] What is/are the output of code fragments below?

i.
```
int S = 0;
int foo(int x, int y) {
  S++;
  if(y == 0)
    return 1;
  else if(y%2)
    return x*foo(x, y/2)*foo(x, y/2);
  else
    return foo(x, y/2)*foo(x, y/2);
}
int main() {
  int n = foo(2, 30);
  printf("%d %d", n, S);
}
```

i. 1073741824 63 (x^y in $2^{\lfloor log_2(y)+2 \rfloor}$ steps)

ii.
```
int S = 0;
int foo(int x, int y) {
  int t; S++;
  if(y == 0)
    return 1;
  else if(y%2) {
    t = foo(x, y/2); return x * t * t;
  } else {
    t = foo(x, y/2); return t * t;
  }
}
int main() {
  int n = foo(2, 30);
  printf("%d %d", n, S);
}
```

ii. 1073741824 6 (x^y in $\lfloor log_2(y) + 2 \rfloor$ steps)

iii.
```
int S = 0;
int foo(int n) {
  S++; return (n==0) ? 1 : foo(n-1) + foo(n-1);
}
int main() {
  int n = foo(30);
  printf("%d %d", n, (S+1)/2);
}
```

iii. 1073741824 1073741824 (2^n in $(2^{n+1} - 1)$ steps)

iv.
```
int foo(int n) {
  return (n>100) ? n-10 : foo(foo(n+11));
}
int main() {
  printf("%d", foo(99));
}
```

iv. 91

v.
```
void foo(int n) {
  if(n == 0) return;
  if(n%2==0) printf("%d ", n%10);
  foo(n/10);
  if(n%2==1) printf("%d ", n%10);
}
int main() {
  foo(1234567890);
}
```

[]

v. 0 8 6 4 2 1 3 5 7 9

(b) [Complex Recursion] What is/are the output of code fragments below?

i.
```
int foo(int m, int n) {
  if(m == 0) {
    return n + 1;
  } else if(n == 0) {
    return foo(m-1, 1);
  } else {
    return foo(m-1, foo(m, n-1));
  }
}
int main() {
  printf("%d", foo(2, 3));
}
```

i. 9

3. Designing a Solution

(a) [Combinatorial] Carrol is a robot that lives in a town where the houses are arranged in nice blocks similar to Manhattan, New York. The aerial view of the town can be simplified to look like Diagram 1. The road going from north-south (or south-north, depending on your point-of-view) is called Street and the road going from east-west is called Avenue. Every Street and Avenue are numbered starting from 0 with Carrol living at a house at the corner of Street #0 and Avenue #0.

Carrol wants to know how many *shortest path* are there from her current position to her house. For instance, Diagram 1 shows Carrol at Street #2 and Avenue #3. There are **eight (8)** possible shortest path to her house. Write your program below (note: *the function prototypes are meant to help you, utilize them properly*):

```
int home(int str, int ave) {
  /* str: current street number, ave: current avenue number */
  if(str == 0 || ave == 0) {
    return 1;
  }
  return home(str - 1, avenue) + home(str, ave - 1);
  // or simplistically:
  // return (str == 0 || ave == 0) ? 1 : home(str - 1, ave) + home(str, ave - 1);




}
```

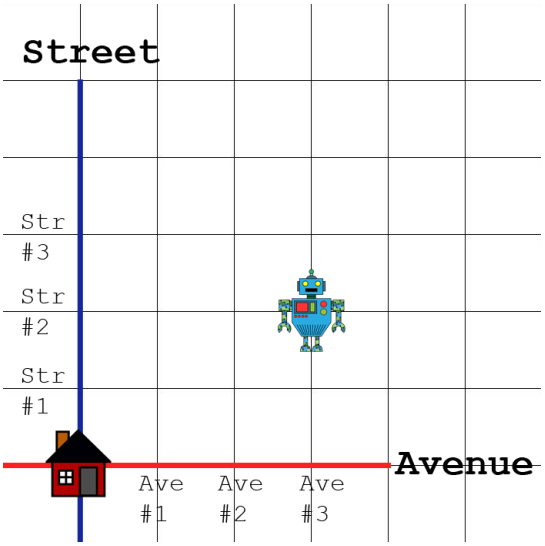

Diagram 1: Example town with position of Carrol at Street #2 and Avenue #3.

(b) [Combinatorial] Given M number of Apples and N number of Plates, how many *unique* ways can I place these M Apples on N Plates. In this problem, we assume that empty plates are allowed and that two arrangements are *equal* if for every Plate in one arrangement, there is another Plate in the second arrangement (*may not be in the same position*) that contains exactly the same number of Apple. Two equal arrangements are not considered unique. For instance, the arrangement "5 1 1" and "1 5 1" are equal.

Consider 7 Apple and 3 Plates. The *unique* arrangements are:

- 0 0 7
- 0 1 6
- 0 2 5
- 1 1 5
- 0 3 4
- 1 2 4
- 1 3 3
- 2 2 3

As such, there are 8 ways of placing 7 Apples on 3 Plates. Another possibilities are: 15 ways of placing 8 Apples on 4 Plates and 11 ways of placing 6 Apples on 6 Plates. Your answer should be done using recursion (i.e. *do not use any combinatorial functions such as permutation*). Write your program below:

```
int place(int apple, int plate) {
  /* apple: current number of apples, plate: current number of plates */
  if(apple <  0) {        // Base Case: no more apples
    return 0;
  } else if(plate == 1) { // Base Case: only one plate left
    return 1;
  }
  return place(apple, plate-1) + place(apple-plate, plate);
  // or simplistically:
  // return (apple < 0 ? 0 : (plate == 1 ? 1 : place(apple, plate-1) + place(apple-plate
, plate)));

}
```

Hint: *consider the following cases below in parallel.*

- There may be empty plates: *In this case, we can place* M *Apples on the other* N-1 *Plates*
- There are no empty plates: *In this case, we can consider placing an Apple on every Plate first before placing the remaining* M-N *Apples on the same number of plates*

4. Challenge

(a) [Exhaustive Search] Singapore coin is actually well designed. To give the *minimum* number of coins for change, you can always start by giving the largest denomination before giving the smaller denomination. We call this method, *greedy* method (since we always greedily give the largest possible value first). This is not always true in general. For instance, given a set of coin denominations of: 1 cents, 15 cents, and 25 cents, and given the sum of 30 cents, the greedy method will not work simply because the change given will be [25, 1, 1, 1, 1, 1] instead of the minimun [15, 15].

Your task is to write a program to find the minimum number of coins, given exactly **three (3)** coin denominations[1]. The solution will be recursive in nature.

i. Write a code for finding the minimum number of coins used for change. Use the following template:

```
int change(int C1, int C2, int C3, int amount) {
  /* C1,C2,C3: coin denominations, amount: amount to be paid */
  int nMin, n1, n2, n3; n1 = n2 = n3 = -1;
  if(amount >= C1) n1 = 1 + change(C1, C2, C3, amount - C1); // try give C1 coin
  if(amount >= C2) n2 = 1 + change(C1, C2, C3, amount - C2); // try give C2 coin
  if(amount >= C3) n3 = 1 + change(C1, C2, C3, amount - C3); // try give C3 coin
  nMin = n1; // find the minimum among the 3 possibilities
  if(n2 != -1 && (nMin == -1 || n2 < nMin)) nMin = n2;
  if(n3 != -1 && (nMin == -1 || n3 < nMin)) nMin = n3;
  return (nMin == -1) ? 0 : nMin;
}
```

Hint: *imagine giving out a change in* C1 *denomination, that is 1 coin given. how many coins must be given for the remaining value? is there any function to solve this problem? once you do this for all three coin denominations, you know 3 things: 1) how many coins needed if you give C1 coin, 2) how many coins needed if you give C2 coin, and 3) how many coins needed if you give C3 coin. what can you do with this information?*

ii. Write a code to find the how many of each coin denominations are used for the change (*in any order*). Use the following template:

```
void coins(int C1, int C2, int C3, int amount) {
  /* C1,C2,C3: coin denominations, amount: amount to be paid */
  int nMin = change(C1, C2, C3, amt);
  while(nMin > 0) { // try giving coin and see if it is part of solution
    if(change(C1, C2, C3, amt-C1) == nMin - 1) {
      printf("%d cents\n", C1); amt -= C1;
    } else if(change(C1, C2, C3, amt-C2) == nMin - 1) {
      printf("%d cents\n", C2); amt -= C2;
    } else if(change(C1, C2, C3, amt-C3) == nMin - 1) {
      printf("%d cents\n", C3); amt -= C3;
    } nMin--; // coin given
  }
}
```

Hint: *let's say you know the minimum number of coins to be given. if you are actually giving a coin denomination* C1 *(i.e.* C1 *is in the given change), what should be the minimum number of coins to be given now? you do not actually require recursion for this, but this method from the hint is actually highly inefficient.*

[1] Since you have not yet learned about array at this point. You can expand this concept involving an *arbitrary* number of coin denominations using array.