The following materials have been collected from the numerous sources such as Stanford CS106 and Harvard CS50 including my own and my students over the years of teaching and experiences of programming. Please help me to keep this tutorial up-to-date by reporting any issues or questions. Please send any comments or criticisms to idebtor@gmail.com. Your assistances and comments will be appreciated..

PSet 03 – Recursion

Table of Contents

Getting Started - Recursion

Recursion in computer science is a method where the solution to a problem depends on solutions to **smaller instances** of the same problem (as opposed to iteration). The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science. An infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions. Most computer programming languages support recursion by allowing a function to call itself within the program text.
(Resources: https://en.wikipedia.org/wiki/Recursion & www.codingBat.com)

Recursive algorithm is expressed in terms of

1. **base case(s)** for which the solution can be stated **non-recursively**,
2. **recursive case(s)** for which the solution can be expressed in terms of a smaller version of itself.

Part I: Implementing Recursion Functions

1. Take a look at the sample solution of Example 1 ~ 6 in `recursion.cpp` and **recusrionDriver.cpp.**

2. Implement the rest of examples while following the patterns of the program including comment sections in **recursion.cpp.** Well written comments are required such that that we can clearly see both how the algorithms work and what the testing results are.

Example 1: Factorial

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fact}(n-1) & \text{if } n > 0 \end{cases}$$

factorial(n)

input: integer *n* such that *n* >= 0
output: [*n* × (*n*-1) × (*n*-2) × … × 1]
   1. if *n* is 0, **return** 1
   2. otherwise, **return** [*n* × factorial(*n*-1)]
end factorial

factorial(1) → 1
factorial(2) → 2
factorial(3) → 6
factorial(8) → 40320
factorial(12) → 479001600
factorial(20) → 2432902008176640000

Hint:
First, detect the "base case", a case so simple that the answer can be returned immediately (here when n==1). Otherwise make a recursive call of `factorial(n-1)` (towards the base case). Assume the recursive call returns a correct value and fix that value up to make our result.

Code:

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

or

```
long long factorial(int n) {
   return n == 0 ? 1 : n * factorial(n - 1);
}
```

Example 2: GCD (Great Common Divisor)

$$\gcd(x, y) = \begin{cases} x & \text{if } y = 0 \\ \gcd(y, \text{remainder}(x, y)) & \text{if } y > 0 \end{cases}$$

Recurrence relation for greatest common divisor, where `x%y` expresses the remainder of `x/y`:

$$\begin{aligned} gcd(x, y) &= gcd(y, x\%y) & if\ y \ne 0 \\ &= gcd(x, 0) = x & if\ y = 0 \end{aligned}$$

gcd(x, y)

input: integer *x*, *y* such that *x* >= *y*, *y* > 0
output: gcd of x and y
    1. if y is 0, **return** x
    2. otherwise, **return** [*gcd* (*y*, *x%y*)]
end gcd

Ex: Computing the recurrence relation for x = 27 and y = 9

```
gcd(27, 9)   = gcd(9, 27 % 9)
             = gcd(9, 0)
```

Ex: Computing the recurrence relation for x = 111 and y = 259

```
gcd(111, 259)   = gcd(259, 111% 259)
                = gcd(259, 111)
                = gcd(111, 259% 111)
                = gcd(111, 37)
                = gcd(37, 111% 37)
                = gcd(37, 0)
                = 37
```

Code:

```
int gcd(int x, int y) {
  if (y == 0) return x;
  return gcd(y, x % y);
}
```

or

```
int gcd(int x, int y) {
   return y == 0 ? x : gcd(y, x % y);
}
```

Example 3: Fibonacci

The fibonacci sequence is a famous bit of mathematics, and it happens to have a recursive definition. The first two values in the sequence are 0 and 1 (essentially 2 base cases). Each subsequent value is the sum of the previous two values, so the whole sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21 and so on.

Define a recursive `fibonacci(n)` method that returns the nth fibonacci number, with n=0 representing the start of the sequence.

fibonacci(0) → 0
fibonacci(1) → 1
fibonacci(2) → 1
fibonacci(3) → 2
fibonacci(4) → 3
fibonacci(5) → 5
fibonacci(11) → 89
fibonacci(33) → 3524578
fibonacci(44) → 701408733 (It takes some time to compute.)

Code:

```
long long fibonacci(int n) {
  if (n == 0 || n == 1) return n;
  return fibonacci(n-1) + fibonacci(n-2);
```

```
}
```

or

```
long long fibonacci(int n) {
    return n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2);
}
```

Example 4: Bunny Ears

We have a number of bunnies and each bunny has two big floppy ears. We want to compute the total number of ears across all the bunnies recursively (without loops or multiplication).

bunnyEars(0) → 0
bunnyEars(1) → 2
bunnyEars(2) → 4
bunnyEars(3) → 6
bunnyEars(234) → 468

Hint:
First detect the base case (`bunnies == 0`), and in that case just return 0. Otherwise, make a recursive call to `bunnyEars(bunnies-1)`. Trust that the recursive call returns the correct value, and fix it up by adding 2.

Code:

```
int bunnyEars(int bunnies) {
if (bunnies == 0) return 0;

  // Recursive case: otherwise, make a recursive call with bunnies-1
  // (towards the base case), and fix up what it returns.
  return 2 + bunnyEars(bunnies-1);
}
```

or

```
int bunnyEars(int bunnies) {
   return bunnies == 0 ? 0 : 2 + bunnyEars(bunnies - 1);
}
```

Example 5: Funny Ears

We have bunnies and funnies standing in a line, numbered 1, 2, ... The odd bunnies (1, 3, ..) have the normal 2 ears. The even funnies (2, 4, ..) we'll say have 3 ears, because they each have a raised foot. Recursively return the number of "ears" in the bunny and funny line 1, 2, ... n (without loops or multiplication).

funnyEars(0) → 0
funnyEars(1) → 2
funnyEars(2) → 5
funnyEars(3) → 7
funnyEars(4) → 10
funnyEars(9) → 22
funnyEars(10 → 25

Code:

```
int funnyEars(int funnies) {
  // your code here
}
```

Last updated: 3/18/2020

Example 6: Triangle

We have triangle made of blocks. The topmost row has 1 block, the next row down has 2 blocks, the next row has 3 blocks, and so on. Compute recursively (no loops or multiplication) the total number of blocks in such a triangle with the given number of rows.

triangle(0) → 0
triangle(1) → 1
triangle(2) → 3
triangle(3) → 6
triangle(4) → 10
triangle(7) → 28

Code:

```
int triangle(int rows) {
// your code here
}
```

Example 7: Sum of digits

Given a non-negative int n, return the sum of its digits recursively (no loops). Note that mod (%) by 10 yields the rightmost digit (126 % 10 is 6), while divide (/) by 10 removes the rightmost digit (126 / 10 is 12).

sumDigits(126) → 9
sumDigits(12) → 3
sumDigits(1) → 1
sumDigits(10110) → 3
sumDigits(235) → 10

Code:

```
int sumDigits(int n) {
  // your code here
}
```

Example 8: Count 8

Given a non-negative int n, return the count of the occurrences of 8 as a digit, so for example 818 yields 2. (no loops). Note that mod (%) by 10 yields the rightmost digit (126 % 10 is 6), while divide (/) by 10 removes the rightmost digit (126 / 10 is 12).

count8(818) → 2
count8(8) → 1
count8(123) → 0
count8(881238) → 3
count8(48581) → 2
count8(888586198) → 5
count8(99899) → 1

Code:

```
int count8(int n) {
// your code here
}
```

Example 9: Power N

Given base and n that are both 1 or more, compute recursively (no loops) the value of base to the n power, so powerN(3, 2) is 9 (3 squared).

powerN(2, 5) → 32
powerN(3, 1) → 3
powerN(3, 2) → 9
powerN(3, 3) → 27
powerN(10, 2) → 100
powerN(10, 3) → 1000

Code:

```
long long powerN(int base, int n) {
  // your code here
}
```

Example 10: Recursive Binary Search

The binary search algorithm is a method of searching a sorted array for a single element by cutting the array in half with each recursive pass. The trick is to pick a midpoint near the center of the array, compare the value at that point with the key being searched and then responding to one of three possible conditions: the key is found at the midpoint, the data at the midpoint is greater than the data being searched for, or the data at the midpoint is less than the key being searched for.

Recursion is used in this algorithm because with each pass a new array is created by cutting the old one in half. The binary search procedure is then called recursively, this time on the new (and smaller) array. Typically, the array's size is adjusted by manipulating a beginning and ending index. The algorithm exhibits a logarithmic order of growth because it essentially divides the problem domain in half with each pass.

Step 1: Implement binsearch.cpp while setting #if 1

Use a skeleton code binsearch.cpp provided first to implement the recursive binary search.

In this code, a user may want to simply call the function with three parameters such as **binary_search(list, key, size)** at user's convenience. Once you get the user's initial call, then you call **_binary_search(list, key, lo, hi)** recursively. As you notice that all the recursive operations will be done in **_binary_search()** with four parameters, not in **binary_search().**

Check the following code and implement it in **binsearch.cpp** skeleton code provided. You may test this functionality first before you proceed the Step 1.

Code:

```
/*
   This implements a binary search recursive algorithm.
   INPUT: list is an array of integers SORTED in ASCENDING order,
          key is the integer to search for,
          lo is the minimum array index,
          hi is the maximum array index
   OUTPUT: an array index of the key found in the list
          if not found, return a modified index where it could be found.
*/

int _binary_search(int *list, int key, int lo, int hi) {
   DPRINT(cout << "key=" << key << " lo=" << lo << " hi=" << hi << endl;);

    cout << "your code here \n";

    return 0;
}

int binary_search(int *list, int key, int size) {
   DPRINT(cout << ">binary_search: key=" << key << " size=" << size << endl;);

   int answer = _binary_search(list, key, 0, size);
```

Last updated: 3/18/2020

```
    DPRINT(cout << "<binary_search: answer=" << answer << endl;);
    return answer;
}

#if 1
int main(int argc, char *argv[]) {
    int list[] = { 3, 5, 6, 9, 11, 12, 15, 18, 19, 20 };        // test set 1
    // int list[] = { 13, 15, 16, 19, 21, 22, 25, 28, 29, 30 };  // test set 2
    // int list[] = { 53, 55, 56, 59, 61, 62, 65, 68, 69, 70 };  // test set 3
    int size = sizeof(list) / sizeof(list[0]);

    cout << "list: ";
    for (int i = 0; i < size; i++)
      cout << list[i] << " ";
    cout << endl;

    // randomly generate numbers to search between
    // list[0] = 3 and list[size-1] = 20, inclusively.
    // do this by 'size' times.
    // print the results as shown in binsearchx.exe.

    cout << "your code here \n";
}
#endif
```

The **main()** function is provided to test both **binary_search()** and **_binary_search()** functions which is a part of binsearch.cpp.

● If you set #if 1, then this entire block of **main()** function will be included during the compilation such that you can test this file itself.

● If you want to find how to use **rand(), srand()** and a random number between a and b, refer to this web site. In Korean.

How to test the algorithm:

We want to generate random numbers or keys to test **binary_search().**

● A key generated randomly should be between the first and last values of the list or or [3 .. 20] in this case.

● Repeat the test by the number of elements in the list.

● Display the result as shown below:

● The first line in this result shows that 8 is not found in the list. If it is found, it should be found at the list index [3]. The sixth line shows that 5 is found at the list index [1] and so on.

Take-away: How long does the `_binary_search()` take with n items?

In one call to **`_binary_search()`,** we eliminate at least half the elements from consideration. Hence, it takes $log_2\ n$ (the base 2 logarithm of n) **`_binary_search()`** calls to pare down the possibilities to one. Therefore **`_binary_search()`** takes time proportional to $log_2\ n.$

Take-away: Then, how long does the `binary_search()` take with n items?

Step 2: Recursion.cpp using binsearch.cpp while setting #if 0

Now, you continue implementing the menu option 10 in recursion.cpp. We want to use **`binary_search()`** but not **`main()`** in binsearch.cpp. You may exclude the **`main()`** while setting **`#if 0`** in binsearch.cpp. Then we can compile two files (recursion.cpp and binsearch.cpp) and create one executable called recursion.exe. In summary,

- Since you finish testing of binsearch.cpp with the **`main()`,** you exclude the main off by setting <mark>`#if 0`</mark>, not to include it during compilation since other file (recursion.cpp) have a **`main()`**.

- Therefore, the build command is

```
g++ recursionDriver.cpp recursion.cpp binsearch.cpp mergesort.cpp
                          -I../../include -L../../lib -lnowic -o recursion
```

Submitting your solution

- Include the following line at the top of your every source file with your name signed.
  On my honour, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.
  Signed: _____   Section: _____   Student Number: _____
- Make sure your code **compiles** and **runs** right before you submit it. Every semester, we get dozens of submissions that don't even compile. Don't make "a tiny last-minute change" and assume your code still compiles. You will not get sympathy for code that "almost" works.
- If you only manage to work out the problem sets partially before the deadline, you still need to turn it in. However, don't turn it in if it does not compile and run.
- Place your source files in the folder you and I are sharing.
- After submitting, if you realize one of your programs is flawed, you may fix it and submit again if it is **before the deadline**. You will have to resubmit any related files together, even if you only change one. You may submit as often as you like. **Only the last version** you submit before the deadline will be graded.

Files to submit and Grade

Submit the following files.
- **`recursion.cpp`, `recursionDriver.cpp` – 1.5 point**
- **`binsearch.cpp` – 1.5 point**

Upload your file **pset3** folder in Piazza.

Due

- Due: 11:55 pm