Content:

Basic Math Properties:

Sum of Arithmetic Sequence: Sn = n/2 [2a + (n - 1) d] or Sn = n/2 [a1 + an]

Sum of Geometric Sequence: $Sn = \frac{a_1(1-r^n)}{1-r}$

Sum of Squares: $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$

Canonical representation of a positive integer:

$n = \prod_{i=1}^{k} p_i^{n_i}$, for all positive integers > 1.

PnC: C(n, k) = C(n-1, k-1) + C(n-1, k)

nPr = n!/(n-r)!

nCr = n!/((n-r)!r!)

Types of Problems PE0:

Use of Counter:

Property of prime:

Eg. Almost.c

```
bool almost(long m)
{
  long counter = 0;
  for (long i = 2; (double)i <= sqrt((double)m); i += 1)
  {
    while (m % i == 0)
    {
      m = m / i;
      counter += 1;
    }
  }
  if (((counter == 2) && (m > 1)) || ((m == 1) && (counter == 3)))
    {
      return true;
    }
  return false;
}
```

Tree Recursion

Eg. stone.c

```
long base(long k); // Generate the Base Case which is the input n in stone.
void stone(long n, long k)
{
      If (k == 0)
      {
            cs1010_println_long(n);
      }
```

```
    long p = powten(k - 1); // Get 10 to the power of k - 1.
    stone(n, k - 1);
    stone(n - p + 2 * p, k - 1);
    }
}
```

Changing order of Digits
Construction of long
Bubble Sort using Recursion

Eg. Largest.c: input:123 output: 321

```
//Sorted one time, separate fn to repeat the action.
long bubble_once(long n)
{
    if (((n >= 0) && (n < 10)) || ((n < 0)&&(n > -10)))
    {
        return n;
    }
    if (n % 10 > n / 10 % 10)
    {
        return n / 10 % 10 + 10 * bubble_once(n / 10 - n / 10 % 10 + n % 10);
    }
    return n % 10 + 10 * bubble_once(n / 10);
}
```

Construction of integer

Eg. simple.c   input: 11223 output: 123

By recursion:

```
long simple(long n)
{
    if (((n >= 0) && (n < 10)) || ((n < 0)&&(n > -10)))
    {
        return n;
    }
    if (n % 10 == n / 10 % 10)
    {
        return simple(n / 10);
    }
    return n % 10 + 10 * simple(n / 10);
}
```

By While-loop:

```
long simple(long n)
{
    long result = 0;
    while (n != 0)
    {
        if (n % 10 != n / 10 % 10)
        {
            result = result *10 + n % 10;
        }
        n = n / 10;
    }
    return result;
}
```

Arithmetic Calculation

Eg. pattern.c, fraction.c (太长了不想写)

Eg. Given an integer n, count *the total number of digit* 1 *appearing in all non-negative integers less than or equal to* n. Leetcode #233

Input: 1, output: 1; input: 13, output: 6

```
long countDigitOne(long n) {
    if (n == 0) {
        return 0;
    }
    int k = countDigits(n); // Count the number of digits in n
    long x = powten(k - 1); // 10^(k-1)
    long y = x / 10; // 10^(k-2)
    if (n < 10) {
        return 1; // Special case for single-digit numbers
    }
    if (n / x > 1) {
        // The leftmost digit is greater than 1
        return x + (n / x) * (k - 1) * y + countDigitOne(n % x);
    }
    if (n / x == 1) {
        // The leftmost digit is 1
        return y * (k - 1) + n % x + 1 + countDigitOne(n % x);
    }
    // The leftmost digit is 0
    return countDigitOne(n % x);
}
```

Constructing Geometric Pattern

Eg. square.c

图形切成不同部分

```
void print_border(long width); //省略
void print_space(long width); // 省略
void draw_square(long row, long width) {
  if ((row == 1) || (row == width))
  {
    print_border(width);
  }
  else if ((row == 2) || (row == width - 1))
  {
    cs1010_print_string("#");
    print_space(width - 2);
    cs1010_print_string("#");
  }
  else
  {
    cs1010_print_string("# ");
    draw_square(row - 2, width - 4);
    cs1010_print_string(" #");
  }
}
int main()
{
  long n = cs1010_read_long();
  for (long i = 1; i <= n; i += 1) {
    draw_square(i, n);
    cs1010_println_string("");
  }
```

Pe2

Basics:

Array Initialisation:

```
long array[3] = {1, 2, 3};
long array[3] = {0}; // All zero
long array[100] = {1, [5] = 2, 3, [99] = 8}; // The rest all 0;
long array[] = {1,2,3,4,5,}
```

Pass Array in function:

```
void foo(long array[10]);
void foo(long len, long array[]);
void foo(long* array);
```

Pointer:

Swap order:

```
// In the declaration, * mean pointer.
void swap(long *a, long *b) {
// Within the fn, * is the dereference operator (content of).
```

```
  long temp = *a;
  *a = *b;
  *b = temp;
}
```

Heap & String

```
cs1010_read_size_t();
cs1010_read_long_array(); // 1D
cs1010_read_double_array(); // 1D
cs1010_read_word(); // 1D
cs1010_read_line(); // 1D
cs1010_read_word_array(); // 2D
cs1010_read_line_array(); // 2D
putchar(); // input is of long type, which is the numerical value in ASCII table;
```

String:

```
// The three declarations below are equivalent.
char hello1[7] = {'h', 'e', 'l', 'l', 'o', '!', '\0'};
char hello2[7] = "hello!";
char hello3[] = "hello!";
char *hello4 = "hello!"; // The string literals cannot be modified. ie. hello4[2] = ‘x’;  leads to error
```

Multidimensional Array(please always remember to do failure handling):

1.Allocating a 2D Array, Non-Contiguous Memory

```
double **canvas;
size_t num_of_rows = cs1010_read_size_t();
size_t num_of_cols = cs1010_read_size_t();
canvas = calloc(num_of_rows, sizeof(double *)); // note the call to sizeof
if (canvas == NULL) {
  cs1010_println_string("unable to allocate array");
  return 1; // or other error indicator
}
for (size_t i = 0; i < num_of_rows; i += 1) {
  canvas[i] = calloc(num_of_cols, sizeof(double));
  if (canvas[i] == NULL) {
    cs1010_println_string("unable to allocate array");
    for (size_t j = 0; j < i; j += 1) {
      free(canvas[j]);
    }
    free(canvas);
    return 1; // or other error indicator
  }
}
```

Need to deallocate with multiple free calls

```
for (size_t i = 0; size_t i < num_of_rows; i += 1) {
  free(canvas[i]);
}
free(canvas);
```

2.Allocating a 2D Array, Contiguous Memory

```
double **canvas;
size_t num_of_rows = cs1010_read_size_t();
size_t num_of_cols = cs1010_read_size_t();
canvas = calloc(num_of_rows, sizeof(double *));
if (canvas == NULL) {
  cs1010_println_string("unable to allocate array");
  return 1;
}
canvas[0] = calloc(num_of_rows * num_of_cols, sizeof(double));
if (canvas[0] == NULL) {
  cs1010_println_string("unable to allocate array");
  free(canvas);
  return 1;
}

for (size_t i = 1; i < num_of_rows; i += 1) {
  canvas[i] = canvas[i-1] + num_of_cols;
}

free(canvas[0]);
free(canvas);
```

3.jagged array

Eg.

```
double *half_square[10];
for (size_t i = 0; i < 10; i += 1) {
  half_square[i] = calloc(i+1, sizeof(double));
}
```

Search

1.linear search

```
long search(long n, const long list[], long q) {
  for (long i = 0; i < n; i += 1) {
    if (list[i] == q) {
      return i;
    }
  }
  return -1;
}
```

Linear Search Variant:

```
long search(const long list[], long i, long j, long q) {
  if (i > j) {
    return -1;
  }
  long mid = (i+j)/2;
  if (list[mid] == q) {
    return mid;
  }
  long found = search(list, i, mid-1, q);
  if (found >= 0) {
    return found;
  }
  return search(list, mid+1, j, q);
}
```

2.Binary search(O(logn))

```
long search(const long list[], long i, long j, long q) {
  if (i > j) {
        return -1;
  }
  long mid = (i+j)/2;
  if (list[mid] == q) {
    return mid;
  }
  if (list[mid] > q) {
        return search(list, i, mid-1, q);
  }
    return search(list, mid+1, j, q);
}
```

Sort

1.counting sort(approximately O(n))

```
void counting_sort(size_t len, const long in[], long out[])
{
  size_t freq[MAX + 1] = { 0 };

  for (size_t i = 0; i < len; i += 1) {
    freq[in[i]] += 1;
  }

  size_t outpos = 0;
  for (long i = 0; i <= MAX; i += 1) {
    for (size_t j = outpos; j < outpos + freq[i]; j += 1) {
      out[j] = i;
    }
    outpos += freq[i];
  }
}
```

2. Counting sort on string (part of radix sort):
array is in the form of list[i][j].

```
void counting_sort(char** list, size_t n, size_t index){
    size_t count[128] = {0};
    for(size_t i = 0; i < n; i += 1)
    {
      count[list[i][index] - '\0'] += 1;
    }
    char** output = calloc(n, sizeof(char*));

    // Change count[] to the sorted position of char ‘i’.
    for(size_t i = 0; i < 128; i += 1)
    {
      count[i] += count[i - 1];
    }

    // output[sorted_pos_of_list[i][index]] = list[i].
    for(size_t i = n - 1; (long)i >= 0; i -= 1)
    {
      output[count[list[i][index] - '\0']] = list[i];
      count[list[i][index] - '\0'] -= 1;
    }

    for(size_t i = 0; i < n; i += 1)
    {
      list[i] = output[i];
    }
  }
```

3. Insertion Sort

```
void insert(long a[], size_t curr)
{
  size_t i = curr;
  long temp = a[curr];
  while (i >= 1 && temp < a[i - 1]) {
    a[i] = a[i - 1];
    i -= 1;
  }
  a[i] = temp;
}

void insertion_sort(size_t n, long a[]) {
  for (size_t curr = 1; curr < n; curr += 1) {
    insert(a, curr);
  }
}
```

4.bubble sort（O(n^2))

```
void bubble_pass(size_t last, long a[]) {
  for (size_t i = 0; i < last; i += 1) {
    if (a[i] > a[i+1]) {
      swap(a, i, i+1);
    }
  }
}
void bubble_sort(size_t n, long a[n]) {
  for (size_t last = n - 1; last > 0; last -= 1) {
    bubble_pass(last, a);
  }
}
```

RECURSION (Let magic do its job)

i) Base Case

ii) Recursive Case

General Pseudo-code:

```
out_type fn(in_type param…)
{
  if(base_case)
  {
    do something;
  }
  changes if needed;
  fn(case k - 1);
  // Below if multiple branches needed.
  changes if needed;
  fn(case k - 1);
  …
}
```

Tower of Hanoid

Pseudo:

base: move case 1 from src to des.

recursive: move k - 1 from src to place holder, move case 1 form src to des, move k - 1 from place holder to des.

Code:

```
void solve(long k, char from, char to, char hold)
{
  if(k == 1) // Base Case.
  {
    move(k, from, to); // Move can be replaced to other action.
  }
  solve(k - 1, from, hold, to); // Move k - 1 from src to place holder.
  move(k, from, to); // Move the remaining from src to des.
  solve(k - 1, hold, to, from); // Move the k - 1 from place holder to des.
}
```

Permutation (and its variants)

Basic Permutation

base: 1 element to permute

recursive:

Repeat:

swap current with current + 1.

permute the new string after the swap.

swap back in the old string.

swap current with current + 2.

...

// The outer swap change the order of an element with all elements after it, the inner recursion call allows the swap to be done on every newly permuted string.

Code:

```
void permute(char a[], size_t len, size_t curr)
{
  if(curr == len - 1)
  {
    cs1010_println_string(a);
    return;
  }
  for(size_t i = 0; i < len; i += 1)
  {
    swap(a, i, curr);
    permuta(a, len, curr + 1);
    swap(a, curr, i);
  }
}
```

Combination of all substring of length k (nCk)

Method of include and exclude every element.

@param remains How many elements remaining to be excluded from curr onward.

```
void combine(char* word, bool chosen, size_t remains, size_t curr, size_t end)
{
  if((long)remains == 0) // If the length has been reached.
  {
    print(word, chosen); // Print the chosen words.
  }
  if(curr != end) // If we have not reached the end.
  {
    combine(word, chosen, remains, curr + 1,  end); // Choose curr, remains unchanged
    chosen[curr] = false; // exclude curr.
    combine(word, chosen, remains - 1, curr + 1, end);
    chosen[curr] = true; // Restore.
  }
}
```

Same Algo without bool array.

```
void combinationUtil(int arr[], int n, int r, int index, int out[], int i)
{
    // Current combination is ready, print it
    if (index == r)
    {
        print(out);
        return;
    }
    // When no more elements are there to put in data[]
    if (i >= n)
        return;
    // current is included, put next at next location
    out[index] = arr[i];
    combinationUtil(arr, n, r, index+1, out, i+1);

    // current is excluded, replace it with next (Note that
    // i+1 is passed, but index is not changed)
    combinationUtil(arr, n, r, index, out, i+1);
}
```

PE problems on permutation, social and fill:
#0. stone
#1. (Group)

```
input:3    output: 1 1 1  // 3 persons in one group
                   1 1 2  // person 1 and 2 in one group, 2 in another
                   1 2 1
                   1 2 2
                   1 2 3  // All in different groups
```

```
void group(size_t current, size_t num_groups, size_t partition[], size_t n)
{
  if(current == n)
  {
      print(partition, n);
    return;
  }
  for(size_t i = 1; i <= num_groups; i += 1)
  {
    partition[current] = i;
    group(current + 1, num_groups, students, n);
  }
  partition[current] = num_groups + 1;
  group(current + 1, num_groups + 1, students, n);
}
```

#2.Path
py07
print all path from person i to person j based on triangle contact map.

```
input: 1                        output: 0 2 1 // 0 contacts 2, who contacts 1
       01
       111  i = 0, j = 1
```

```
void path(char** matrix, size_t i, size_t j, size_t n, size_t degree, bool* visited,
size_t* string)
  {
    if(i == j)
    {
      print_path(string, degree);
      return;
    }
    for(size_t middle = 0; middle < n; middle += 1)
    {
      if(middle != i && is_contact(matrix, i, middle) && !visited[middle])
      {
        visited[middle] = true;
        string[degree] = middle;
        path(matrix, middle, j, n, degree + 1, visited, string);
        visited[middle] = false;
      }
    }
  }
```

#3. Cluster (count the number of cluster based on trianglular contact map)
py08 Q5 (different from given solution, I find using bool array easier)

```
long num_cluster(char** map, size_t n, size_t curr, bool* chosen, long cluster)
{
  if(curr == n)
  {
    return cluster;
  }
  chosen[curr] = true;
  for(size_t i = 0; i < n; i += 1)
  {
    if(i != curr && !chosen[i] && is_contact(map, i, curr))
    {
      chosen[i] = true;
      cluster = num_cluster(map, n, i, chosen, cluster);
    }
  }
  for(size_t i = 0; i < n; i += 1)
  {
    if(!chosen[i])
    {
      cluster += num_cluster(map, n, i, chosen, cluster);
    }
  }
  return cluster;
}
```

#4. substring.

Other Recursion Question:
#1. Bracket.c
valid string if empty
or start with (, [, { < and end with),], }, >, within which is also a valid string
or a valid string followed by another

```
bool is_open(char a)
{
  if(a == '(' || a == '<' || a == '{' || a == '[')
  {
    return true;
  }
  return false;
}
bool is_closed(char a, char b)
{
  if((a == '(' && b == ')') ||
      (a == '<' && b == '>') ||
      (a == '[' && b == ']') ||
      ( a == '{' && b == '}'))
  {
    return true;
  }
  return false;
}
size_t is_consumed(char* string, size_t begin)
{
  if(string[begin] == '\0')
  {
    return begin;
  }
  if(!is_open(string[begin]))
  {
    return begin;
  }
  size_t end = is_consumed(string, begin + 1);
  if(is_closed(string[begin], string[end]))
  {
    return is_consumed(string, end + 1);
  }
  return begin;
}
```

#2. Grammar
Valid if
write @@ and continue writing valid string

write %# and continue writing valid string in between
write one single char and stop

```
bool is_valid(char* word, size_t i, size_t end)
{
  if(i == end)
  {
    if(word[i] == ‘@’ || word[i] == ‘%’ || word[i] == ‘#’)
    {
      return true;
    }
  }
  if(word[i] == ‘%’ && word[end] == ‘#’)
  {
    return is_valid(word, i + 1, j - 1);
  }
  if(word[i] == ‘@’ && word[i + 1] == ‘@’)
  {
    return is_valid(word, i + 2, j);
  }
  return false;
}
```

Introductory Dynamic Programming

nCk/ walk.c dp method

```
int nCk(int n, int k)
{
    int C[n + 1][k + 1];
    int i, j;
    for (i = 0; i <= n; i++) {
        for (j = 0; j <= min(i, k); j++) {
            // Base Cases
            if (j == 0 || j == i)
                C[i][j] = 1;
            // Calculate value using
            // previously stored values
            else
                C[i][j] = C[i - 1][j - 1] + C[i - 1][j];
        }
    }
    return C[n][k];
}
```

Alternative (the math way)

```
long count_step(long x, long y, long step){
  for (long i = 0; i < x; i += 1){
    step = step * (x + y - i) / (i + 1);
  }
  return step;
}
```

Helper Function

Length:

```
size_t length_of(const char *str) {
  size_t i = 0;
  while(str[i] != '\0')
  {
    i += 1;
  }
  return i;
}
```

Helper Code

power:

```
long power(long x, long n) {
    long result = 1;
    while (n > 0) {
        if (n % 2 == 1) {
            result *= x;
        }
        x *= x;
        n /= 2;
    }

    return result;
}
```

gcd:

```
long compute_gcd(long a, long b)
{
  if (a >= b)
  {
    if (b != 0)
    {
      return compute_gcd(b, a%b);
    }
    return a;
  }

  return compute_gcd(b, a);
}

Is_prime: (more efficient)
bool is_prime(long number)
{
  if (number <= 1){
    return false;
  }
  if (number <= 3){
    return true;
  }
  if ((number % 2 == 0) || (number % 3 == 0)){
    return false;
  }
  for (long i = 5; i * i <= number; i += 6){
    if ((number % i == 0) || (number % (i + 2) == 0)){
      return false;
    }
  }
  return true;
}


digits:
current_digit = n % 10;
Second_last_digit = n / 10 % 10;

Number of digit;
long num_digit(long n)
{
  if (n < 10)
  {
    return 1;
  }
  return 1 + num_digit(n / 10);
}

bool is_contact(char** map, size_t a, size_t b)
{
  if(((a > b) && (map[a][b] == '1')) ||
      ((b > a) && map[b][a] == '1'))
  {
    return true;
  }
  return false;
}
```