函数

2022 年 8 月 16 日

1. 输入一个正整数 n，输出从 1 到 n 范围内的全部素数。

```
#include <stdio.h>

int is_prime(int n)
// 判断正整数n是否素数
{
    if(n == 1) return 0; // 1不是素数
    for(int i = 2; i < n; i++) // 从2开始测试到n-1，蛮力法
    {
        if(n % i == 0) // 能整除
        {
            return 0; // 返回假，注意return的使用技巧
        }
    }
    return 1; // 返回真
}

int main()
{
    int input_num = 0;
    printf("Please input a positive integer:\n");
    scanf("%d", &input_num);

    for(int n = 1; n <= input_num; n++) // 从1枚举到input_num
    {
        if(is_prime(n)) // 如果是
        {
            printf("%d\t", n);
        }
    }

    return 0;
}
```

2. 设一个数组包含 12 个元素，每一个元素都由用户输入。先把数组的最小元素与第一个元素交换，然后把数组的最大元素与最后一个元素交换。如果出现多个并列最大元素或者并列最小元素，只取第一个最大或者最小元素来交换。

```
#include<stdio.h>
#include<string.h>

void swap(int *address_1, int *address_2) // void表示函数无返回值
{
    int temp; // 作为交换的媒介
    temp = (*address_1); // 把address_1指向的对象给到temp
    (*address_1) = (*address_2); // 把address_2指向的对象的值，赋给address_1指向的对象
    (*address_2) = temp; // 把temp的值给到address_2指向的对象
}

// 以下传入指针作为参数，有利于修改函数外面定义的变量
void adjust(int *array_address, int array_len)
{
    int loc_4_max, loc_4_min; // 用于保存目前找到的最大值、最小值的下标
    loc_4_max = loc_4_min = 0; // 假想最大值、最小值在0号单元

    for(int i = 1; i < array_len; i++) // 扫描数组
    {
        if(array_address[loc_4_min] > array_address[i]) // 发现更小
            loc_4_min = i; // 更新下标
    }

    swap(array_address + 0, array_address + loc_4_min); // 传入首元素地址和最小元素的地址

    for(int i = 1; i < array_len; i++) // 扫描数组
    {
        if(array_address[loc_4_max] < array_address[i]) // 发现更大
```

```c
            loc_4_max = i; // 更新下标
    }

    swap(array_address + array_len - 1, array_address + loc_4_max); // 传入末尾元素地址和最大元素地址
}


int main()
{
    int a[10], i;

    printf("Input 10 integers:\n");

    for(i = 0; i < 10; i++)
        scanf("%d", a + i); // 注意数组名自动转化为首元素地址

    printf("The input array:\n");
    for(i = 0; i < 10; i++)
        printf("%d\t", a[i]);
    printf("\n");

    adjust(a, 10); // 传入数组名，它自动转化为首元素地址

    printf("The adjusted array:\n");
    for(i = 0; i < 10; i++)
        printf("%d\t", a[i]);

    printf("\n");

    return 0;
}
```

3. 编程求超过正整数 n 的最小素数。

```c
#include <stdio.h>
int min_prime_greater_than(int n)
{
    while(1) // 循环条件永远成立，因此，只能用跳转语句离开循环
    {
        n++; // 考虑下一个
        int i; // 用于表示可能的约数
        for(i = 2; i < n; i++) // 枚举可能的约数
        {
            if(n % i == 0) // 整除
            {
                break; // 跳出内层循环
            }
        }
        if(i == n) return n; // 并没有执行break语句，说明找不到约数，因此是素数
    }
}

int main()
{
    int n;
    printf("input an integer:\n");
    scanf("%d", &n);
    int m = min_prime_greater_by(n); // 函数返回比n大的最小素数，并保存在m中
    printf("the next prime is %d\n", m);
    return 0;
}
```

4. 用随机函数产生 10 个互不相同的两位正整数, 存放至数组中, 输出这 10 个数并输出其中的素数。

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int is_prime(int n)
{
    if(n == 1) return 0;
    for(int i = 2; i < n; i++)
    {
        if(n % i == 0)
        {
            return 0;
        }
    }
    return 1;
}

int main()
{
    srand(time(NULL));

    // initialize candidate numbers, only the last 90 are meaningful
    int candidate_numbers[100] = {0};
    for(int i = 0; i < 99; i++)
    {
        candidate_numbers[i] = i;
    }

    // randomly select 2-digit numbers which are different from each other
    int cursor = 10; // cursor is always the start of the candidate set
    for(int i = 1; i <= 10; i++)
    {
        // since each number occupies a location, selecting a random location means selecting a random number
        int rand_location = rand() % (99 - cursor + 1) + 10;
        // remove the selected number from the candidate set, also put it in our result set, below is a common swapping operation
        int temp = candidate_numbers[cursor];
        candidate_numbers[cursor] = candidate_numbers[rand_location];
        candidate_numbers[rand_location] = temp;
        // maintain
        cursor++; // cursor表示被选集的开头
    }

    // print the selected numbers
    printf("The 10 randomly selected numbers are:\n");
    for(int i = 10; i < 20; i++)
    {
        printf("%d\t", candidate_numbers[i]);
    }
    printf("\n");

    // print the prime numbers
    printf("Among them, below are prime numbers:\n");
    for(int i = 10; i < 20; i++)
    {
        int cand_num = candidate_numbers[i];
        if(is_prime(cand_num))
        {
            printf("%d\t", cand_num);
        }
    }
    printf("\n");

    return 0;
}
```

5. 先输入一个 4×4 矩阵，然后输出一个 4×1 矩阵，然后输出它们的乘积。

```c
#include <stdio.h>
#include <stdlib.h>
// matrix multiplications
// 一个*号声明一级指针，两个*号声明二级指针
// 一维数组的元素用一级指针来访问；二维数组的元素用二级指针来访问
// 因此，访问二维数组使用二级指针
void compute_product_of_two_matrices(double **result_mtx, double **a, int m, int l, int n, double **b)
// a must be an m*l matrix, b must be an l*n matrix
{
    for(int i = 0; i < m; i++)
    {
        for(int j = 0; j < n; j++)
        {
            // result_mtx[i]表示result所指的i号单元，
            // 因为result_mtx是int **型的，所以，它所指的i号单元（行单元）是int *型的，即result_mtx[i]是int *型的
            // 同理可得，result_mtx[i][j]是result_mtx[i]所指的j号单元（元素单元），
            // 因为result_mtx[i]是int *型的，所以result_mtx[i][j]是int型的
            // 于是result_mtx[i][j]就是i号行，j号列的元素
            result_mtx[i][j] = 0;
            for(int k = 0; k < l; k++)
            {
                result_mtx[i][j] += a[i][k] * b[k][j]; // 定义法
            }
        }
    }
}

// matrices' square
// 同上，访问二维数组，用二级指针
void compute_square_of_square_matrix(double **result_mtx, double **sq_a, int n)
{
    return compute_product_of_two_matrices(result_mtx, sq_a, n, n, n, sq_a);
}

// print matrices
void print_matrices(double **a, int m, int n)
{
    for(int i = 0; i < m; i++)
    {
        for(int j = 0; j < n; j++)
        {
            printf("%.4lf\t", a[i][j]);
        }
        printf("\n");
    }
}

// 下面创建二维数组，并使“指针a所指的对象”指向新创建的二维数组
// 注意这里使用三级指针
// 注意到C语言的函数按值传递，因此，当我们要修改一个变量时，就必须传入它的地址
// 我们要修改用于访问二维数组的二级指针，就必须传入它的地址，因此，必须传入三级指针
void create_matrices(double ***p, int m, int n)
{
    // p是三级指针，因此(*p)是二级指针，正好用于访问数组
    // 下面语句，*p是二级指针，右边类型转换之后也得到二级指针，两边同型，可进行赋值
    (*p) = (double**)malloc(m * sizeof(double*));
    if(!(*p))
    {
```

```c
        printf("Insufficient memory\n");
        exit(-1);
    }
    for(int i = 0; i < m; i++)
    {
        // (*p)[i]是一级指针，右边也是一级指针，同型可以赋值
        (*p)[i] = (double*)malloc(n * sizeof(double));
        if(!(*p)[i])
        {
            printf("Insufficient memory.\n");
            exit(-1);
        }
    }
}

void input_scalar(double **a, int m, int n)
{
    for(int i = 0; i < m; i++)
    {
        for(int j = 0; j < n; j++)
        {
            scanf("%lf", &a[i][j]);
        }
    }
}

void free_matrices(double **a, int m)
{
    // 释放每一行，此时需要用到a指向的空间的信息，因此a指向的空间仍需保留
    for(int i = 0; i < m; i++)
    {
        free(a[i]);
    }
    // 释放a指向的空间
    free(a);
}

int main()
{

    double **input_a, **input_b, **result_matrix;

    printf("Input a square 4*4 matrix:\n");
    create_matrices(&input_a, 4, 4);
    printf("******\n");
    input_scalar(input_a, 4, 4); // 传入input_a的地址，以便修改它
    printf("******\n");
    print_matrices(input_a, 4, 4);

    printf("Input a square 4*1 matrix:\n");
    create_matrices(&input_b, 4, 1); // 传入input_b的地址，以便修改它
    input_scalar(input_b, 4, 1);
    print_matrices(input_b, 4, 1);

    create_matrices(&result_matrix, 4, 1); // 传入result_matrix的地址，以便修改它


    compute_product_of_two_matrices(result_matrix, input_a, 4, 4, 1, input_b);
    print_matrices(result_matrix, 4, 1);

    // 回收三个矩阵所占的空间
    free_matrices(input_a, 4);
    free_matrices(input_b, 4);
    free_matrices(result_matrix, 4);
    return 0;
}
```

6. 输入一个正整数 n，然后输入一个 n 阶方阵，如果它可逆，则输出它的逆矩阵，否则输出 “它不可逆”。

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define EPS 0.001 // 精度

float cut_tail(float t)
{
    return (float)((int)t);
}

int close_to_integers(float t)
{
    if(fabs(cut_tail(t) - t) < EPS)
        return 1;
    else
        return 0;
}

void print_matrix(float **a, int m, int n)
{
    for(int i = 0; i < m; i++)
    {
        for(int j = 0; j < n; j++)
        {
            float t = a[i][j];
            if(close_to_integers(t))
            {
                t = cut_tail(t);
                printf("%d\t", (int)t);
            }
            else
                printf("%.3g\t", t);
        }
        printf("\n");
    }
}

// 求初等变换之后，对角线上非零元素的个数
int non_zero_row_num(float **a, int n)
{
    int count = 0;
    for(int i = 0; i < n; i++)
    {
        for(int j = i; j < 2 * n; j++)
        {
            if(fabs(a[i][j]) > EPS)
            {
                count++;
                break;
            }
        }
    }

    return count;
}

void elementarily_transform(float **a, int n)
// preconditions: a must be a n * 2n matrix, of the form [A, I] where I is a unit matrix
// postconditions: of the form [I, A^{-1}] where I is a unit matrix
{
    // deal with bottom-left parts
    for(int i = 0; i < n - 1; i++) // deal with each column
    {
        // select the minimum head
        int min_head_at = -1;
        for(int j = i; j < n - 1; j++)
        {
            if(fabs(a[i][j]) > EPS)
            {
                min_head_at = j;
                break;
            }
        }

        if(min_head_at == -1) // all 0 column, skip
        {
            continue;
        }

        for(int j = i + 1; j < n; j++)
        {
            if(fabs(a[i][j]) > EPS && fabs(a[i][j]) < fabs(a[i][min_head_at])) // avoid 0s in the diagnal line
            {
              min_head_at = j;
            }
        }

        if(min_head_at != i)
        {
            for(int j = i; j < 2 * n; j++)
            {
                // swap a[i][j] with a[min_head_at][j]
                float temp = a[i][j];
                a[i][j] = a[min_head_at][j];
                a[min_head_at][j] = temp;
            }
        }

        for(int j = i + 1; j < n; j++) // for each row
        {
            float quotient = a[j][i] / a[i][i];
            a[j][i] = 0;
            for(int k = i + 1; k < 2 * n; k++)
            {
                a[j][k] -= quotient * a[i][k];
            }
        }
        printf("having computed 1 step:\n");
        print_matrix(a, n, 2 * n);
    }

    // transform the left half into a unit matrix
    for(int i = n - 1; i >= 0; i--) // for each column
    {
        if(fabs(a[i][i]) < EPS)
        {
            continue;
        }
        float quotient = a[i][i];
        a[i][i] = 1;
        for(int j = i + 1; j < 2 * n; j++)
        {
```

```c
            a[i][j] /= quotient;
        }

        for(int j = i - 1; j >= 0; j--)
        {
            float quotient = a[j][i];
            a[j][i] = 0;
            for(int k = i + 1; k < 2 * n; k++)
            {
                a[j][k] -= a[i][k] * quotient;
            }
        }
        printf("having computed 1 step:\n");
        print_matrix(a, n, 2 * n);
    }
    return;
}

int main()
{
    // input
    int n = 0;
    printf("Input the dimension of your matrix:");
    scanf("%d", &n);

    float **a = (float **)malloc(n * sizeof(float*));
    if(!a)
    {
        printf("Not enough space for allocating a %d * %d matrix\n", n, n);
        return 1;
    }

    for(int i = 0; i < n; i++)
    {
        a[i] = (float*)malloc(n * sizeof(float));
        if(!a[i])
        {
            printf("Not enough space for allocating the %d-th row\n", i + 1);
            return 1;
        }

        printf("Input the %d-th row\n", i + 1);
        for(int j = 0; j < n; j++)
        {
            printf("Input the %d-th element:\n", j + 1);
            scanf("%f", &a[i][j]);
        }
    }
    printf("Inputs completed\n");

    printf("The input matrix is:\n");
    print_matrix(a, n, n);

    // construct [A, I]
    float **extended_a = (float**)malloc(n * sizeof(float*));
    if(!extended_a)
    {
        printf("not enough memory\n");
        return 1;
    }

    for(int i = 0; i < n; i++)
    {
        extended_a[i] = (float*)malloc(2 * n * sizeof(float));
        if(!extended_a[i])
        {
            printf("not enough memory\n");
            return 1;
        }
    }


    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < n; j++)
        {
            extended_a[i][j] = a[i][j];
        }
        for(int j = n; j < 2 * n; j++)
        {
            if(j - n == i)
            {
                extended_a[i][j] = 1.0;
                continue;
            }
            extended_a[i][j] = 0.0;
        }
    }

    printf("extended matrix:\n");
    print_matrix(extended_a, n, 2 * n);


    // elementary transformations
    printf("starts...\n");
    elementarily_transform(extended_a, n);

    int rank = non_zero_row_num(extended_a, n);
    printf("The rank of the input matrix is: %d\n", rank);
    if(rank < n)
    {
        printf("This matrix is not inversible.");
    }
    else
    {
        printf("results:\n");
        print_matrix(extended_a, n, 2 * n);
    }

    // free memory
    for(int i = 0; i < n; i++)
    {
        free(extended_a[i]);
    }
    free(extended_a);

    for(int i = 0; i < n; i++)
```

```c
    {
        free(a[i]);
    }
    free(a);
    return 0;
}
```

7. 对“以org_str为首字符地址的字符串”进行擦除操作。从该字符串下标为start的字符开始，擦除连续的num个字符（不需要考虑非法输入）。

   运行示例：

```
Input the original string:
abcdefghijk
Input the index from which we erase:
5
Input the number of characters which will be erased:
3
After characters are erased, the result is abcdeijk.
```

```
Input the original string:
ABCDEFG
Input the index from which we erase:
2
Input the number of characters which will be erased:
3
After characters are erased, the result is ABFG.
```

```c
#include<stdio.h>
#include<string.h>

void erase(char* org_str, int start, int num);
//对“以org_str为首字符地址的字符串”进行擦除操作。从该字符串下标为start的字符开始，擦除连续的num个字符。
//不需要考虑非法输入

int main()
```

```c
{
    char org_str[1024];
    int start, num;
    printf("Input the original string:\n");
    gets(org_str);
    printf("Input the index from which we erase:\n");
    scanf("%d", &start);
    printf("Input the number of characters which will be erased:\n");
    scanf("%d", &num);
    erase(org_str, start, num);
    printf("After characters are erased, the result is %s.\n", org_str);
    return 0;
}


void erase(char* org_str, int start, int num)
{
    int first_index_of_tail = start + num; // 待擦除的子串结束后紧接的字符的下标
    int tail_length = strlen(org_str) - (start + num); // 原字符串减去前一段，再减去擦除的那一段，得到紧接那一段

    // 移动元素（字符）
    for(int i = first_index_of_tail; i <= first_index_of_tail + tail_length; i++)
    // <= because we want to copy '\0'
    {
        org_str[i - num] = org_str[i];
    }

}
```

8. 在“以 org_str 为首元素地址的字符串”中，按“从尾到头”的顺序查找“以 str 为首元素地址的字符串”。如果找到，返回首次发生匹配时最末尾字符的下标；否则返回-1（不需要考虑非法输入）。

   运行示例：

```
Input the searched string:
abcdeabhiabxyz
Input the string to be located:
```

```
ab
Last occurence at index: 10


Input the searched string:
abcdefg
Input the string to be located:
re
Not found


Input the searched string:
abcdefccdef
Input the string to be located:
bcdef
Last occurence at index: 5
```

```c
#include <stdio.h>
#include <string.h>

int rfind(char* org_str, char* str);
//在“以org_str为首元素地址的字符串”中，按“从尾到头”的顺序查找“以str为首元素地址的字符串”。
//如果找到，返回首次发生匹配的下标；否则返回-1（不需要考虑非法输入）．

int main()
{
    char org_str[1024];
    char str[1024];
    printf("Input the searched string:\n");
    gets(org_str);
    printf("Input the string to be located:\n");
    gets(str);
    int index = rfind(org_str, str);
    if(index != -1)
    {
        printf("Last occurence at index: %d\n", index);
    }
    else
    {
        printf("Not found\n");
    }
    return 0;
}

int rfind(char* org_str, char* str)
{
    for(int i = strlen(org_str) - 1; i >= strlen(str) - 1; i--) // 从最后一个有效字符开始往前搜
    {
        int found = 1; // 表示找到了子串
        for(int j = strlen(str) - 1, k = i; j >= 0; j--, k--) // 从子串最末端元素开始往前搜，逐个逐个匹配
        {
            if(str[j] != org_str[k]) // 出现不匹配
            {
                found = 0; // 找不到子串
                break; // 退出内层循环
            }
        }
        if(found) return i; // 找到就返回下标
    }
    return -1;
}
```

9. 对“以 org_str 为首字符地址的字符串”进行替换操作。从该字符串下标为 start 的字符开始，替换连续的 num 个字符，使之变为以 new_subtr 为首元素地址的字符串（不需要考虑非法输入）。

   运行示例：

```
Input the original string:
abcdefghijklmn
Input the new substring:
RST
Input the index from which replacement occurs:
5
```

```
Input the number of characters which will be replaced:
2
After replacements, the result is abcdeRSThijklmn.
```

```
Input the original string:
abcdefghijklmn
Input the new substring:
RST
Input the index from which replacement occurs:
4
Input the number of characters which will be replaced:
7
After replacements, the result is abcdRSTlmn.
```

```
Input the original string:
abcdefghijklmn
Input the new substring:
RST
Input the index from which replacement occurs:
5
Input the number of characters which will be replaced:
3
After replacements, the result is abcdeRSTijklmn.
```

```c
#include<stdio.h>
#include<string.h>

void replace(char* org_str, int start, int num, char* new_substr);
//对“以org_str为首字符地址的字符串”进行替换操作。
//从该字符串下标为start的字符开始，替换连续的num个字符，使之变为以new_subtr为首元素地址的字符串。
//不需要考虑非法输入
int main()
{
    char org_str[1024];
    int start, num;
    char new_substr[1024];
    printf("Input the original string:\n");
    gets(org_str);
    printf("Input the new substring:\n");
    gets(new_substr);
    printf("Input the index from which replacement occurs:\n");
    scanf("%d", &start);
    printf("Input the number of characters which will be replaced:\n");
    scanf("%d", &num);
    replace(org_str, start, num, new_substr);
    printf("After replacements, the result is %s.\n", org_str);
    return 0;
}


void replace(char* org_str, int start, int num, char* new_substr)
{
    // org_str is divided into 3 parts: head, mid, tail
    if(strlen(new_substr) < num)//new substring shorter
    {//move the tail forward
        int forward = num - strlen(new_substr);//the number of positions forward
        int tail_length = strlen(org_str) - (start + num);
        int first_index_of_tail = start + num;
        for(int i = first_index_of_tail; i <= first_index_of_tail + tail_length; i++)
        // <= because we also need to copy '\0'
        {
            org_str[i - forward] = org_str[i];
        }
    }
    else if(strlen(new_substr) > num)//new substring longer
    {//move the tail backward
        int backward = strlen(new_substr) - num;
        int tail_length = strlen(org_str) - (start + num);
        int last_index_of_tail = strlen(org_str) - 1;
        for(int i = last_index_of_tail + 1; i >= last_index_of_tail + 1 -
             tail_length; i--)
        // +1 because we also need to copy '\0'
        {
            org_str[i + backward] = org_str[i];
        }
    }
    // copy new_substr into org_str
    for(int i = start, j = 0; i < start + strlen(new_substr); i++, j++)
    {
        org_str[i] = new_substr[j];
    }
}
```