程序设计基础

王子轩 2023011307

PART1 DS

1 数组`array`

1.1 初始化和遍历

```
//存储在栈上，值是随机的
int arr[5];
//存储在堆上，一般会初始化为{0}
int* arr1 = new int[5];
cout<<arr[0]<<endl;
cout<<arr1[0]<<endl;
delete[] arr1;
/*
output:
4199008
0
*/
void traverse(int *nums, int size) {
    int count = 0;
    // 通过索引遍历数组
    for (int i = 0; i < size; i++) {
        count += nums[i];
        cout<<nums[i]<<" ";
    }
    cout<<endl;
}
```

1.2 插入和删除

```
//在数组的所索引 index 处插入元素 num
void insert(int *nums, int size, int num, int index) {
    // 把索引 index 以及之后的所有元素向后移动一位
    for (int i = size - 1; i > index; i--) {
        nums[i] = nums[i - 1];
    }
    // 将 num 赋给 index 处的元素
    nums[index] = num;
}
/* 删除索引 index 处的元素 */
void remove(int *nums, int size, int index) {
    // 把索引 index 之后的所有元素向前移动一位
    for (int i = index; i < size - 1; i++) {
        nums[i] = nums[i + 1];
    }
}
```

2 列表 vector or list

2.1 初始化和访问

```
/* 初始化列表 */
// 需注意，C++ 中 vector 即是本文描述的 nums
// 无初始值
vector<int> nums1;
/*
output:segmentation fault
*/
// 有初始值
vector<int> nums = { 1, 3, 2, 5, 4 };
/* 访问元素 */
int num = nums[1];  // 访问索引 1 处的元素
/* 更新元素 */
nums[1] = 0;  // 将索引 1 处的元素更新为 0
```

2.2 插入与删除

```
/* 清空列表 */
nums.clear();
/* 在尾部添加元素 */
nums.push_back(1);
nums.push_back(3);
nums.push_back(2);
nums.push_back(5);
nums.push_back(4);
/* 在中间插入元素 */
nums.insert(nums.begin() + 3, 6);  // 在索引 3 处插入数字 6
/* 删除元素 */
nums.erase(nums.begin() + 3);      // 删除索引 3 处的元素
```

2.3 遍历

```
template <typename T>
void printNum(int num) {
    std::cout << num << std::endl;
}
/* 通过索引遍历列表 */
void traversebyindex(vector<T> nums,void (*func)(T)){
    for (int i = 0; i < nums.size(); i++) {
        func(nums[i]);
    }
}
//调用方式
traversebyindex(nums, printNum);
/* 直接遍历列表元素 */
void traversebyindex(vector<T> nums,int* count){
    for (int num : nums) {
        count += num;
}
```

2.4 拼接

```
/* 拼接两个列表 */
vector<int> nums1 = { 6, 8, 7, 10, 9 };
// 将列表 nums1 拼接到 nums 之后
nums.insert(nums.end(), nums1.begin(), nums1.end());
```

2.5 列表的实现（array）

```
/* 列表类 */
class MyList {
  private:
    int *arr;              // 数组（存储列表元素）
    int arrCapacity = 10; // 列表容量
    int arrSize = 0;       // 列表长度（当前元素数量）
    int extendRatio = 2;    // 每次列表扩容的倍数
  public:
    /* 构造方法 */
    MyList() {
        arr = new int[arrCapacity];
    }
    /* 析构方法 */
    ~MyList() {
        delete[] arr;
    }
    /* 获取列表长度（当前元素数量）*/
    int size() {
        return arrSize;
    }
    /* 获取列表容量 */
    int capacity() {
        return arrCapacity;
    }
    /* 访问元素 */
    int get(int index) {
        // 索引如果越界，则抛出异常，下同
        if (index < 0 || index >= size())
            throw out_of_range("索引越界");
        return arr[index];
    }
    /* 更新元素 */
    void set(int index, int num) {
        if (index < 0 || index >= size())
            throw out_of_range("索引越界");
        arr[index] = num;
    }
    /* 在尾部添加元素 */
    void add(int num) {
        // 元素数量超出容量时，触发扩容机制
        if (size() == capacity())
            extendCapacity();
        arr[size()] = num;
        // 更新元素数量
        arrSize++;
```

```
    }
    /* 在中间插入元素 */
    void insert(int index, int num) {
        if (index < 0 || index >= size())
            throw out_of_range("索引越界");
        // 元素数量超出容量时，触发扩容机制
        if (size() == capacity())
            extendCapacity();
        // 将索引 index 以及之后的元素都向后移动一位
        for (int j = size() - 1; j >= index; j--) {
            arr[j + 1] = arr[j];
        }
        arr[index] = num;
        // 更新元素数量
        arrSize++;
    }
    /* 删除元素 */
    int remove(int index) {
        if (index < 0 || index >= size())
            throw out_of_range("索引越界");
        int num = arr[index];
        // 将索引 index 之后的元素都向前移动一位
        for (int j = index; j < size() - 1; j++) {
            arr[j] = arr[j + 1];
        }
        // 更新元素数量
        arrSize--;
        // 返回被删除的元素
        return num;
    }
    /* 列表扩容 */
    void extendCapacity() {
        // 新建一个长度为原数组 extendRatio 倍的新数组
        int newCapacity = capacity() * extendRatio;
        int *tmp = arr;
        arr = new int[newCapacity];
        // 将原数组中的所有元素复制到新数组
        for (int i = 0; i < size(); i++) {
            arr[i] = tmp[i];
        }
        // 释放内存
        delete[] tmp;
        arrCapacity = newCapacity;
    }
    /* 将列表转换为 Vector 用于打印 */
    vector<int> toVector() {
        // 仅转换有效长度范围内的列表元素
        vector<int> vec(size());
        for (int i = 0; i < size(); i++) {
            vec[i] = arr[i];
        }
        return vec;
    }
};
```

3 链表Linkedlist

3.1 链表的建立，插入，删除，访问，查找

```
#include <iostream>
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(nullptr) {}
};
class Linkedlist {
public:
    ListNode* head;  // 链表头指针
    int count;       // 链表节点计数器
public:
    Linkedlist() : head(nullptr), count(0) {}
    // 构造函数：接收头节点指针
    Linkedlist(ListNode* ptr) : head(ptr), count(1) {}
    // 析构函数：销毁链表，释放内存
    ~Linkedlist() {
        clear();
    }
    // 插入节点：在节点 n0 后插入新节点 P
    void insert(ListNode* n0, ListNode* P);
    // 删除节点：删除 n0 后的第一个节点
    void remove(ListNode* n0);
    // 查找目标值
    int find(int target);
    // 访问链表中的节点，返回指定索引的节点
    ListNode* access(int index);
    // 遍历链表并输出
    void traverse() const;
    // 清空链表
    void clear();
    // 获取链表的节点数
    int size() const { return count; }
};
void Linkedlist::insert(ListNode* n0, ListNode* P) {
    if (n0 == nullptr || P == nullptr) return;
    P->next = n0->next;
    n0->next = P;
    count++;  // 更新节点计数
}
void Linkedlist::remove(ListNode* n0) {
    if (n0 == nullptr || n0->next == nullptr) return;  // 空链表或已是尾节点
    ListNode* toDelete = n0->next;
    n0->next = toDelete->next;
    delete toDelete;
    count--;  // 更新节点计数
}
int Linkedlist::find(int target) {
    ListNode* current = head;
    int index = 0;
    while (current != nullptr) {
        if (current->val == target) {
```

```
            return index;
        }
        current = current->next;
        index++;
    }
    return -1;  // 未找到目标
}
ListNode* Linkedlist::access(int index) {
    if (index < 0 || index >= count) return nullptr;
    ListNode* current = head;
    for (int i = 0; i < index; ++i) {
        current = current->next;
    }
    return current;
}
void Linkedlist::traverse() const {
    ListNode* current = head;
    while (current != nullptr) {
        std::cout << current->val << " ";
        current = current->next;
    }
    std::cout << std::endl;
}
void Linkedlist::clear() {
    while (head != nullptr) {
        ListNode* toDelete = head;
        head = head->next;
        delete toDelete;
    }
    count = 0;
}
int main() {
    // 初始化链表 1 -> 3 -> 2 -> 5 -> 4
    Linkedlist mylinkedlist;
    ListNode* n0 = new ListNode(1);
    mylinkedlist.head = n0;
    ListNode* n1 = new ListNode(3);
    ListNode* n2 = new ListNode(2);
    ListNode* n3 = new ListNode(5);
    ListNode* n4 = new ListNode(4);
    n0->next = n1;
    n1->next = n2;
    n2->next = n3;
    n3->next = n4;
    mylinkedlist.traverse();
    // 插入节点 0 到 n2 后
    ListNode* p = new ListNode(0);
    mylinkedlist.insert(n2, p);
    mylinkedlist.traverse();
    // 删除节点 n3 后的节点
    mylinkedlist.remove(n3);
    mylinkedlist.traverse();
    // 查找值为 0 的节点
    std::cout << "Index of 0: " << mylinkedlist.find(0) << std::endl;
    // 访问索引为 1 的节点
    ListNode* node = mylinkedlist.access(1);
```

```
    if (node != nullptr) {
        std::cout << "Node at index 1: " << node->val << std::endl;
    }
    std::cout << "Size of list: " << mylinkedlist.size() << std::endl;
    // 清空链表
    mylinkedlist.clear();
    std::cout << "Size of list after clearing: " << mylinkedlist.size() <<
std::endl;
}
```

3.2 双向链表

```
/* 双向链表节点结构体 */
struct ListNode {
    int val;          // 节点值
    ListNode *next;  // 指向后继节点的指针
    ListNode *prev;  // 指向前驱节点的指针
    ListNode(int x) : val(x), next(nullptr), prev(nullptr) {}  // 构造函数
};
```

4 栈`stack`

4.1 STL库接口

```
#include <stack>
/* 初始化栈 */
stack<int> stack;
/*基本操作*/
/* 元素入栈 */
stack.push(1);
stack.push(3);
stack.push(2);
stack.push(5);
stack.push(4);
/* 访问栈顶元素 */
int top = stack.top();
/* 元素出栈 */
stack.pop(); // 无返回值
/* 获取栈的长度 */
int size = stack.size();
/* 判断是否为空 */
bool empty = stack.empty();
```

4.2 基于`Linkedlist`实现

```
/* 基于链表实现的栈 */
class LinkedListStack {
  private:
    ListNode *stackTop; // 将头节点作为栈顶
    int stkSize;        // 栈的长度
  public:
    LinkedListStack() {
```

```
        stackTop = nullptr;
        stkSize = 0;
    }
    ~LinkedListStack() {
        // 遍历链表删除节点，释放内存
        freeMemoryLinkedList(stackTop);
    }
    /* 获取栈的长度 */
    int size() {
        return stkSize;
    }
    /* 判断栈是否为空 */
    bool isEmpty() {
        return size() == 0;
    }
    /* 入栈 */
    void push(int num) {
        ListNode *node = new ListNode(num);
        node->next = stackTop;
        stackTop = node;
        stkSize++;
    }
    /* 出栈 */
    int pop() {
        int num = top();
        ListNode *tmp = stackTop;
        stackTop = stackTop->next;
        // 释放内存
        delete tmp;
        stkSize--;
        return num;
    }
    /* 访问栈顶元素 */
    int top() {
        if (isEmpty())
            throw out_of_range("栈为空");
        return stackTop->val;
    }
    /* 将 List 转化为 Array 并返回 */
    vector<int> toVector() {
        ListNode *node = stackTop;
        vector<int> res(size());
        for (int i = res.size() - 1; i >= 0; i--) {
            res[i] = node->val;
            node = node->next;
        }
        return res;
    }
};
```

4.3 基于 array 的实现

```
/* 基于数组实现的栈 */
class ArrayStack {
  private:
    vector<int> stack;
  public:
    /* 获取栈的长度 */
    int size() {
        return stack.size();
    }
    /* 判断栈是否为空 */
    bool isEmpty() {
        return stack.size() == 0;
    }
    /* 入栈 */
    void push(int num) {
        stack.push_back(num);
    }
    /* 出栈 */
    int pop() {
        int num = top();
        stack.pop_back();
        return num;
    }
    /* 访问栈顶元素 */
    int top() {
        if (isEmpty())
            throw out_of_range("栈为空");
        return stack.back();
    }
    /* 返回 Vector */
    vector<int> toVector() {
        return stack;
    }
};
```

5 队列 queue

5.1 STL库接口

```
#include <queue>
/* 初始化队列 */
queue<int> queue;
/* 元素入队 */
queue.push(1);
queue.push(3);
queue.push(2);
queue.push(5);
queue.push(4);
/* 访问队首元素 */
int front = queue.front();
/* 元素出队 */
queue.pop();
```

```
/* 获取队列的长度 */
int size = queue.size();
/* 判断队列是否为空 */
bool empty = queue.empty();
```

5.2 基于 Linkedlist 实现

```
/* 基于链表实现的队列 */
class LinkedListQueue {
  private:
    ListNode *front, *rear; // 头节点 front ，尾节点 rear
    int queSize;
  public:
    LinkedListQueue() {
        front = nullptr;
        rear = nullptr;
        queSize = 0;
    }
    ~LinkedListQueue() {
        // 遍历链表删除节点，释放内存
        freeMemoryLinkedList(front);
    }
    /* 获取队列的长度 */
    int size() {
        return queSize;
    }
    /* 判断队列是否为空 */
    bool isEmpty() {
        return queSize == 0;
    }
    /* 入队 */
    void push(int num) {
        // 在尾节点后添加 num
        ListNode *node = new ListNode(num);
        // 如果队列为空，则令头、尾节点都指向该节点
        if (front == nullptr) {
            front = node;
            rear = node;
        }
        // 如果队列不为空，则将该节点添加到尾节点后
        else {
            rear->next = node;
            rear = node;
        }
        queSize++;
    }
    /* 出队 */
    int pop() {
        int num = peek();
        // 删除头节点
        ListNode *tmp = front;
        front = front->next;
        // 释放内存
        delete tmp;
        queSize--;
```

```
        return num;
    }
    /* 访问队首元素 */
    int peek() {
        if (size() == 0)
            throw out_of_range("队列为空");
        return front->val;
    }
    /* 将链表转化为 Vector 并返回 */
    vector<int> toVector() {
        ListNode *node = front;
        vector<int> res(size());
        for (int i = 0; i < res.size(); i++) {
            res[i] = node->val;
            node = node->next;
        }
        return res;
    }
};
```

5.3 基于 array 实现

```
/* 基于环形数组实现的队列 */
class ArrayQueue {
  private:
    int *nums;        // 用于存储队列元素的数组
    int front;        // 队首指针，指向队首元素
    int queSize;      // 队列长度
    int queCapacity; // 队列容量

  public:
    ArrayQueue(int capacity) {
        // 初始化数组
        nums = new int[capacity];
        queCapacity = capacity;
        front = queSize = 0;
    }

    ~ArrayQueue() {
        delete[] nums;
    }

    /* 获取队列的容量 */
    int capacity() {
        return queCapacity;
    }

    /* 获取队列的长度 */
    int size() {
        return queSize;
    }

    /* 判断队列是否为空 */
    bool isEmpty() {
        return size() == 0;
```

```
    }

    /* 入队 */
    void push(int num) {
        if (queSize == queCapacity) {
            cout << "队列已满" << endl;
            return;
        }
        // 计算队尾指针，指向队尾索引 + 1
        // 通过取余操作实现 rear 越过数组尾部后回到头部
        int rear = (front + queSize) % queCapacity;
        // 将 num 添加至队尾
        nums[rear] = num;
        queSize++;
    }

    /* 出队 */
    int pop() {
        int num = peek();
        // 队首指针向后移动一位，若越过尾部，则返回到数组头部
        front = (front + 1) % queCapacity;
        queSize--;
        return num;
    }

    /* 访问队首元素 */
    int peek() {
        if (isEmpty())
            throw out_of_range("队列为空");
        return nums[front];
    }

    /* 将数组转化为 Vector 并返回 */
    vector<int> toVector() {
        // 仅转换有效长度范围内的列表元素
        vector<int> arr(queSize);
        for (int i = 0, j = front; i < queSize; i++, j++) {
            arr[i] = nums[j % queCapacity];
        }
        return arr;
    }
};
```

6 STL库

`map`

```
#include <iostream>
#include <map>
int main() {
    std::map<int, std::string> myMap;
    // 插入元素
    myMap.insert(std::make_pair(1, "one"));
    myMap[2] = "two";
    myMap[3] = "three";
```

```
    // 查找元素
    auto it = myMap.find(2);
    if (it != myMap.end()) {
        std::cout << "Found: " << it->first << " -> " << it->second << std::endl;
    }
    // 遍历 map
    for (const auto& pair : myMap) {
        std::cout << pair.first << " -> " << pair.second << std::endl;
    }
    // 删除元素
    myMap.erase(2);
    // 输出大小
    std::cout << "Size after erase: " << myMap.size() << std::endl;
    return 0;
}
```

unordered_map

```
#include <unordered_map>//C++11特性
//也可以使用<map>，底层是红黑树实现
/* 初始化哈希表 */
unordered_map<int, string> map;

/* 添加操作 */
// 在哈希表中添加键值对 (key, value)
map[12836] = "小哈";
map[15937] = "小啰";
map[16750] = "小算";
map[13276] = "小法";
map[10583] = "小鸭";
//对于map重复插入Key一样的会自动替换
/* 查询操作 */
// 向哈希表中输入键 key ，得到值 value
string name = map[15937];

/* 删除操作 */
// 在哈希表中删除键值对 (key, value)
map.erase(10583);
```

tuple

```
#include <tuple>
tuple<int, double, string> myTuple(1, 3.14, "Hello");
std::tuple<int, double, std::string> t1(1, 3.14, "hello");
std::tuple<int, double, std::string> t2(1, 3.14, "world");
std::tuple<int, double> t3(1, 3.14);
if (t1 < t2)
    std::cout << "t1 < t2\n";
//输出:
//t1 < t2
std::tuple<int, double> getValues() {
    return {42, 3.14};
}
//可以用于返回多个值
```

```
std::tie(x, y) = getValues();
std::cout << "x: " << x << ", y: " << y << std::endl;
//输出：  x: 42, y: 3.14
 std::tuple<int, double, std::string> myTuple(42, 3.14, "hello");
std::tie(a,b,c) = myTuple;
//a: 42, b: 3.14, c: hello
//访问
get<idx>(myTuple);
//感觉可以用结构体来代替
```

string

```
#include <iostream>
#include <string>

int main() {
    std::string str1;  // 默认构造函数，创建一个空字符串
    std::string str2 = "Hello, world!";  // 使用字符串字面量初始化
    std::string str3("Another string");  // 使用 C-style 字符串初始化
    std::string str4(5, 'A');  // 创建一个包含5个字符'A'的字符串

    std::cout << "str2: " << str2 << std::endl;
    std::cout << "str3: " << str3 << std::endl;
    std::cout << "str4: " << str4 << std::endl;
}
/*
str2: Hello, world!
str3: Another string
str4: AAAAA
*/
std::string str = "Hello";
std::cout << "First character: " << str[0] << std::endl;  // 使用下标
std::cout << "Second character: " << str.at(1) << std::endl;  // 使用 at() (会进行越界检查)
std::string str1 = "Hello";
std::string str2 = "World!";
std::string result = str1 + " " + str2;  // 使用 + 运算符
std::cout << result << std::endl;

str1.append(" everyone!");  // 使用 append 方法
std::cout << str1 << std::endl;
```

```
//字符串流处理函数
#include <sstream>
string data = "hello world 1.34 !"
isstringstream iss(data);
string word1;
string word2;
```

```
double val;
char C;
iss >> word1 >> word2 >> val >> C;
//字符串处理函数
#include <cstring>
//返回不包括结尾字符'\0'的字符串的长度
strlen(const char* str);
//按照字典序返回字符串的大小顺序
strcmp(const char*str1, const char*str2);
strncmp(const char* str1, const char* str2, size_t n);
//字符串拷贝函数
strcpy(char* dest, const char* src);
```

tree

```
//实现最简单的树结构
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr);
}
/* 初始化二叉树 */
// 初始化节点
TreeNode* n1 = new TreeNode(1);
TreeNode* n2 = new TreeNode(2);
TreeNode* n3 = new TreeNode(3);
TreeNode* n4 = new TreeNode(4);
TreeNode* n5 = new TreeNode(5);
// 构建节点之间的引用（指针）
n1->left = n2;
n1->right = n3;
n2->left = n4;
n2->right = n5;
/* 插入与删除节点 */
TreeNode* P = new TreeNode(0);
// 在 n1 -> n2 中间插入节点 P
n1->left = P;
P->left = n2;
// 删除节点 P
n1->left = n2;
```

`set`

```
//set是基于红黑树实现的，默认按照<实现比较，重载即可
#include <set>
structure State {
    //最好有一个构造函数
}
<set>State visited;
//使用set最重要的功能就是要排除重复
State new_state(···);
if(visited.find() == visited.end()) {
    visited.insert(new_state);
    //···
}
```

PART2 文件

文件读取

```
std::ifstream infile("input.txt");
infile.open("example.txt");
infile.close()
if(infile)//infile可以提供一个bool值来表示文件是否打开成功
{
    //···
}
//采用“>>”逐个读取，读取操作与流操作完全一致
while (infile >> word) {
    std::cout << word << std::endl;
}
//按行读取
std::string line;
while (std::getline(infile, line)) {
    std::cout << line << std::endl;
}
//文件指针的移动ios_base::beg开头位置,ios_base::cur当前位置
//seek（移动字节数，操作符）
infile.seek(-4, ios_base::end);
//读文件一直到文件末尾
while(!fin.eof()){
    //···
}

//二进制文件的读入
infile.open("inputfile", std::ios::binary);
infile.read((char*)data_buf_ptr, data_buf_size);
```

文件写入

```
//覆盖模式
std::ofstream outfile("output.txt");
//追加模式
std::ofstream outfile("output.txt", std::ios::app);
outfile.close();
//直接用<<运算符写入
outfile<<"Hello world!"<<std::endl;
//用二进制办法写入
char data[] = "Hello world!";
outfile.write(data, sizeof(data));
//二进制办法打开并且追加
std::ofstream outfile("example.bin", std::ios::binary | std::ios::app);
```

文件的课件例题：

> task 1：从文件中读取指令并输出

```
void read_msg_from(char msg[][20])
{
    ifstream file;
    file.open("prompt.txt");
    file>>msg[0]>>msg[1]>>msg[2];
}
```

> task 2：指令控制存储的计数器

```
int main()
{
    int last_cnt = 0, last_res = 0;
    ifstream status("tmp.res");
    if(status)
        status >> last_cnt >> last_res;
    status.close();

    for(; last_cnt < 10000; ){
        int data;
        cin >> data;
        if(data == 0){//输入指令0查看当前结果
            cout << last_res << endl;
            continue;
        }
        if(data == -1){//输入指令-1退出程序
            ofstream fout("tmp.res");
            fout << last_cnt << ' ' << last_res;
            fout.close();
            break;
        }
        //其他输入将被记入累加的数
        last_res += data;
        last_cnt ++;
    }
    if (last_cnt == 10000)
```

```
        cout << "Finished! Final result = " << last_res << endl;
    return0;
}
```

> task 3: 词频统计

对给定文本中出现的英文单词进行词语频率统计

> task 4: 成绩排序题目

分别从三门课程的成绩单中统计人名以及相应的分数，并计算总成绩

PART3 ALGO

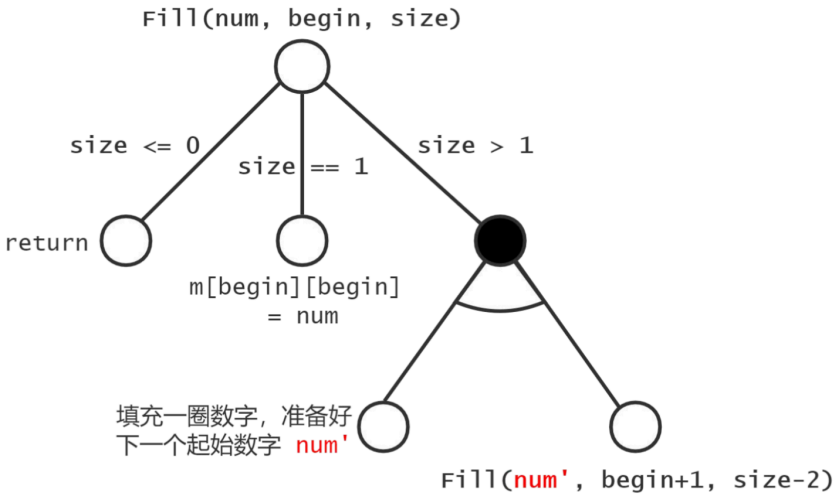
- 递归、减治
- 搜索
- 分治
- 动态规划
- 枚举、回溯
- 多步决策

减治 & 递归

> 1~N*N的自然数逆时针旋转填充

用与或图来分析算法思路

函数Fill(number, begin, size)表示将number开头的数，从位置(begin, begin)开始填写，矩阵大小为size*size




```
#include <iostream>
#include <iomanip>
using namespace std;
int m[6][6] = {0};
```

```
void show()
{
    for(int i = 0; i < 6; i++){
        for(int j = 0; j < 6; j++)
            cout<<setw(2)<<m[i][j]<< ' ';
        cout<<endl;
    }
}
void Fill(int num, int begin, int size);
int main(){
    Fill(1,0,6);
    show();
}
```

```
void Fill(int num, int begin, int size){
    if(size <= 0) return;
    if(size == 1) {
        m[begin][begin] = num;
        return;
    }
    for(int j=0; j<size-1 ;j++){
        m[begin+j][begin]               = num+j;
        m[begin+size-1][begin+j]        = size-1+num+j;
        m[begin+size-1-j][begin+size-1] = 2*(size-1)+num+j;
        m[begin][begin+size-1-j]        = 3*(size-1)+j;
    }
    Fill(num+4*(size-1),begin+1, size-2);
}
```

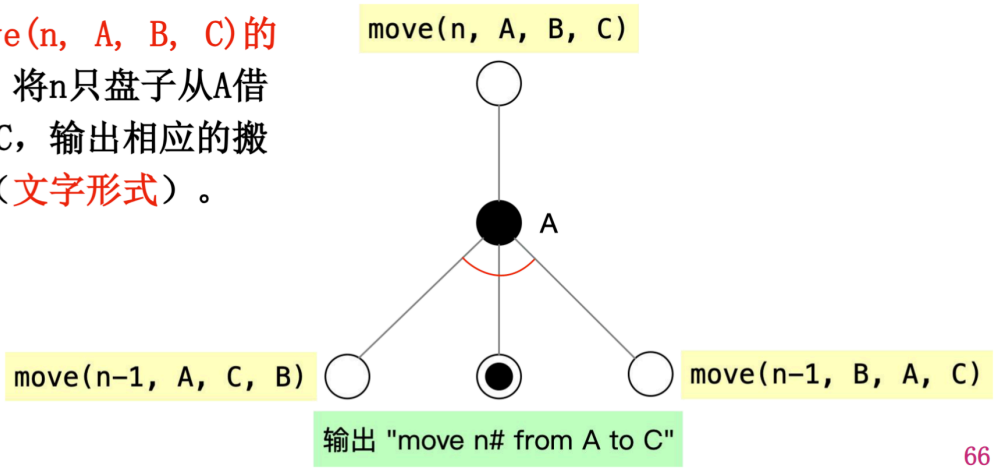
> 斐波那契数列-DP(查表)

```
int FIB[20]; //备忘录表
int fib(int n)
{
    if (FIB(n))
        return FIB[n];
    cout<< "fib(" << n << ")";
    if (n < 2)
        FIB[n] = n;
    else
        FIB[n] = fib(n-1) + fib(n-2);
    return FIB[n];
}
```

> 汉诺塔

思路是实现函数`move(n, A, B, C)`，即将n只盘子从A借助B搬到C；

6、函数move(n, A, B, C)的功能是：将n只盘子从A借助B搬到C，输出相应的搬移方法（文字形式）。




```
#include <iostream>
using namespace std;
void move(int n, char a, char b, char c)
{
    if(!n)
        return;
    move(n-1, a, c, b);
    cout<<"move "<<n<<"#"<<"from "<<a<<"to "<<c<<endl;
    move(n-1, b, a, c);
}
int main()
{
    int n;
    cin>>n;
    char A = 'A';
    char B = 'B';
    char C = 'C';
    move(n, A, B, C);
    return 0;
}
```

> 逆向哈诺塔问题，根据输入判定是否成功`move n# from A to B`

描述：文件中第一行为指令的数量，后面为指令，要求判定是否可以达成要求;

思路：模拟;栈结构

```
#include <fstream>
#include <iostream>
#include <vector>
#include <stack>//用于模拟汉诺塔
#include <algorithm>
#include <sstream>//用于按行读取时进行字符串流操作
#include <unordered_map>
using namespace std;
unordered_map<char, stack<int>> tower;
struct move
{
    int obj;
    char ori;
```

```
    char des;
};
bool is_valid_moves(const vector<struct move>& moves, int n){
    //初始化三根塔的状态
    tower['A'] = stack<int>();
    tower['B'] = stack<int>();
    tower['C'] = stack<int>();
    for(int i = n; i > 0; i--)
        tower['A'].push(i);
    //处理每一个移动
    for(auto mv : moves){
        int obj = mv.obj;
        char ori = mv.ori;
        char des = mv.des;
        //is valid?
        if(tower[ori].top() != obj)
            return false;
        else
            if(!tower[des].empty() && tower[des].top() < obj)
                return false;
        tower[ori].pop();
        tower[des].push(obj);
    }
    //检查最终状态是否合法
    for(int i = 1; i <= n; ++i){
        if(tower['C'].top() != i)
            return false;
        tower['C'].pop();
    }
}
int main()
{
    ifstream input("in.txt");
    int n;
    input >> n;
    vector<struct move> moves;
    string line;
    //读取第一行最后的换行符
    getline(input, line);
    while(getline(input, line)){
        istringstream iss(line);
        string temp;
        int obj;
        char ori;
        char des;
        char tempchar;
        //move n# from A to B
        iss >> temp >> obj >> tempchar >> temp >> ori >> temp >> des;
        struct move v = {obj, ori, des};
        moves.push_back(v);
    }
    input.close();

    if (is_valid_moves(moves, n)) {
        cout << "YES" << endl;
    } else {
```

```
        cout << "NO" << endl;
    }
    return 0;
}
```

二分搜索

分治的核心思路是主定理，降低整体复杂度

> 基础二分搜索策略

对长度为n的`nums`数组，查找并返回元素`target`在其中的索引

```
int binarySearch(vector<int> &nums, int target)
{
    //双闭区间法[i, j]
    int i = 0, j = nums.size() - 1;
    while (i <= j) {
        int m = i + (j - i) / 2; // 计算中点索引 m, 不用(i+j) / 2是担心越界
        if (nums[m] < target)    // 此情况说明 target 在区间 [m+1, j] 中
            i = m + 1;
        else if (nums[m] > target) // 此情况说明 target 在区间 [i, m-1] 中
            j = m - 1;
        else // 找到目标元素，返回其索引
            return m;
    }
    // 未找到目标元素，返回 -1
    return -1;
}
```

> task 1: **查找**插入点索引 利用二分查找+线性查找，

对于有重复元素的情形，需要返回重复元素的最小索引

思路：先二分查找到数组中的一个target，再线性查找到最左边的target

```
int binarySearchInsertion(vector<int> &nums, int target)
{
    int i = 0, j = nums.size() - 1;//[0, n-1]
    while( i <= j ){
        int m = i + (j - i)/2;
        if(nums[m] < target){
            i = m + 1; // target in [m + 1, j]
        } else if (nums[m] > target){
            j = m - 1; // target in [i, m - 1]
        } else{ //这里非常难理解，要注意，在target = nums[m]的时候还是要进行搜索范围的缩小，
            j = m - 1; // target in [i, ]
        }
    }
    //返回插入点i
    return i;// i in [0, nums.size()]
}
```

> task 2: 二分搜索查找边界

复用task 1代码，分别实现左右边界的查找;给数组和target，没有找到就返回-1

```
/* 二分查找最左一个 target */
int binarySearchLeftEdge(vector<int> &nums, int target) {
    // 等价于查找 target 的插入点
    int i = binarySearchInsertion(nums, target);
    // 未找到 target ，返回 -1
    if (i == nums.size() || nums[i] != target) {
        return -1;
    }
    // 找到 target ，返回索引 i
    return i;
}
/* 二分查找最右一个 target */
int binarySearchRightEdge(vector<int> &nums, int target) {
    // 转化为查找最左一个 target + 1
    int i = binarySearchInsertion(nums, target + 1);
    // j 指向最右一个 target ，i 指向首个大于 target 的元素
    int j = i - 1;
    // 未找到 target ，返回 -1
    if (j == -1 || nums[j] != target) {
        return -1;
    }
    // 找到 target ，返回索引 j
    return j;
}
```

如图 10-7 所示，查找完成后，指针 i 指向最左一个 `target + 1`（如果存在），而 j 指向最右一个 `target` ，**因此返回 j 即可。**

 img

排序问题

```
//helper 函数
void printVec(vector<int> &nums){
    int n = nums.size();
    for(int i =0; i < n; i++){
        cout << nums[i] << " ";
    }
    cout<<endl;
    return;
}
```

- 选择排序$\sigma(n^2)$

```
void selectionSort(vector<int> &nums) {
    int n = nums.size();
    //此时未排序的区间是[i, n - 1]
    for(int i = 0; i < n -1; i++){
        int k = i;
        for(int j = i + 1; j < n; j++)
            if(nums[j] < nums[i])
                k = j;//记录最小值的索引
        swap(nums[i], nums[k]);
        printVec(nums);
    }
}
```

- 快速排序->基于分治策略的算法 $\sigma(nlog(n))$

思路：确定哨兵达到的效果是`[left,pivot - 1] <= pivot <= [pivot + 1, right]`->对分出来的两个子数组进行排序，递归地完成

```
/* pivot 划分, 区间为[left, right] */
int partition(vector<int> &nums, int left, int right){
    //以left作为基准数字
    int i = left, j = right;
    while( i < j ) {
        while(i < j && nums[j] >= nums[left])
           j--;
        while(i < j && nums[i] <= nums[left])
           i++;
        swap(nums[i], nums[j]);
    }
    swap(nums[i], nums[left]);//将基准数字交换到两个子数组的分界线
    return i;//返回基准数的索引
}
void quickSort(vector<int> &nums, int left, int right){
    //子数组长度为1时候停止递归
    if(left >= right)
        return;
    int pivot = partition(nums, left, right);
    printVec(nums);
    quickSort(nums, left, pivot - 1 );
    quickSort(nums, pivot + 1, right);
}
```

- 冒泡排序$\sigma(n^2)$

```
void bubbleSort(vector<int> &nums) {
    // 外循环：未排序区间为 [0, i]
    for (int i = nums.size() - 1; i > 0; i--) {
        // 内循环：将未排序区间 [0, i] 中的最大元素交换至该区间的最右端
        for (int j = 0; j < i; j++) {
            if (nums[j] > nums[j + 1]) {
                // 交换 nums[j] 与 nums[j + 1]
                // 这里使用了 std::swap() 函数
                swap(nums[j], nums[j + 1]);
                printVec(nums);
```

```
            }
        }
    }
}
```

- 归并排序
- 插入排序
- 计数排序

DP

> 对于给定数字串，插入`k`个`*`号使得乘积最大、和最大

思路：考虑在位置上的第i个插入，具有最优子结构的特点；前缀和和前缀积？简化计算

> 爬楼梯问题：给定一个共有 n 阶的楼梯，你每步可以上 1 阶或者 2 阶，请问有多少种方案可以爬到楼顶?

思路：`dp[i] = dp[i - 1] + dp[i - 2]` 具有最优子结构

- 解法1：记忆化搜索（自顶向下的方法）

```
/* 记忆化搜索 */
int dfs(int i, vector<int> &mem) {
    // 已知 dp[1] 和 dp[2] ，返回之
    if (i == 1 || i == 2)
        return i;
    // 若存在记录 dp[i] ，则直接返回之
    if (mem[i] != -1)
        return mem[i];
    // dp[i] = dp[i-1] + dp[i-2]
    int count = dfs(i - 1, mem) + dfs(i - 2, mem);
    // 记录 dp[i]
    mem[i] = count;
    return count;
}

/* 爬楼梯：记忆化搜索 */
int climbingStairsDFSMem(int n) {
    // mem[i] 记录爬到第 i 阶的方案总数，-1 代表无记录
    vector<int> mem(n + 1, -1);
    return dfs(n, mem);
}
```

- 解法2：纯DP，自底向上方法，从子问题逐步构建大问题的解

```
/* 爬楼梯：动态规划 */
int climbingStairsDP(int n) {
    if (n == 1 || n == 2)
        return n;
```

```
    // 初始化 dp 表，用于存储子问题的解
    vector<int> dp(n + 1);
    // 初始状态：预设最小子问题的解
    dp[1] = 1;
    dp[2] = 2;
    // 状态转移：从较小子问题逐步求解较大子问题
    for (int i = 3; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n];
}
```

> 直方格地图上的最小距离问题：给定一个 n×m 的二维网格 `grid`，网格中的每个单元格包含一个非负整数，表示该单元格的代价。机器人以左上角单元格为起始点，每次只能向下或者向右移动一步，直至到达右下角单元格。请返回从左上角到右下角的最小路径和。

思路：`dp[n][m]` ,满足最优子结构,递推关系是`dp[n][m] = max(dp[n - 1][m], dp[n][m - 1]) + grid[i][j]`

```
/* 最小路径和：动态规划 */
int minPathSumDP(vector<vector<int>> &grid) {
    int n = grid.size(), m = grid[0].size();
    // 初始化 dp 表
    vector<vector<int>> dp(n, vector<int>(m));
    dp[0][0] = grid[0][0];
    // 状态转移：首行
    for (int j = 1; j < m; j++) {
        dp[0][j] = dp[0][j - 1] + grid[0][j];
    }
    // 状态转移：首列
    for (int i = 1; i < n; i++) {
        dp[i][0] = dp[i - 1][0] + grid[i][0];
    }
    // 状态转移：其余行和列
    for (int i = 1; i < n; i++) {
        for (int j = 1; j < m; j++) {
            dp[i][j] = min(dp[i][j - 1], dp[i - 1][j]) + grid[i][j];
        }
    }
    return dp[n - 1][m - 1];
}
```

> 钢条切割盈利问题

> 矩阵链的最少乘法问题

```
#include <iostream>
#include <vector>
#include <climits>
```

```cpp
using namespace std;

// 计算矩阵链乘法的最小标量乘法次数
int matrixChainOrder(const vector<int>& p, int n) {
    // M[i][j] 表示计算矩阵链 Ai到Aj所需的最小乘法次数
    vector<vector<int>> M(n, vector<int>(n, 0));

    // l 为链的长度，从2开始，直到n
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i < n - len + 1; i++) {
            int j = i + len - 1;
            M[i][j] = INT_MAX;
            // 找到一个k使得 M[i][k] + M[k+1][j] + p[i-1]*p[k]*p[j] 最小
            for (int k = i; k < j; k++) {
                int q = M[i][k] + M[k+1][j] + p[i] * p[k+1] * p[j+1];
                if (q < M[i][j]) {
                    M[i][j] = q;
                }
            }
        }
    }

    // 返回从A1到An的最小乘法次数
    return M[0][n-1];
}

int main() {
    // 矩阵链的尺寸数组
    vector<int> p = {10, 20, 30, 40, 30}; // p数组表示矩阵链的维度
    int n = p.size() - 1;  // n表示矩阵链的个数

    // 输出最小标量乘法次数
    cout << "最小标量乘法次数: " << matrixChainOrder(p, n) << endl;

    return 0;
}
```

> 0-1背包问题：给定 n 个物品，第 i 个物品的重量为 wgt[i−1]、价值为 val[i−1] ，和一个容量为 cap 的背包。每个物品**只能选择一次**，问在限定背包容量下能放入物品的最大价值？

思路：构建二维数组dp(n +1) * (cap + 1) 表示容量为cap的背包在对n个物品。最优子结构：

第二步：找出最优子结构，进而推导出状态转移方程

当我们做出物品 i 的决策后，剩余的是前 $i-1$ 个物品决策的子问题，可分为以下两种情况。

- **不放入物品** i：背包容量不变，状态变化为 $[i-1, c]$。
- **放入物品** i：背包容量减少 $wgt[i-1]$，价值增加 $val[i-1]$，状态变化为 $[i-1, c-wgt[i-1]]$。

```cpp
/* 0-1 背包：动态规划 */
int knapsackDP(vector<int> &wgt, vector<int> &val, int cap) {
    int n = wgt.size();
    // 初始化 dp 表
    vector<vector<int>> dp(n + 1, vector<int>(cap + 1, 0));
```

```
    // 状态转移
    for (int i = 1; i <= n; i++) {
        for (int c = 1; c <= cap; c++) {
            if (wgt[i - 1] > c) {
                // 若超过背包容量，则不选物品 i
                dp[i][c] = dp[i - 1][c];
            } else {
                // 不选和选物品 i 这两种方案的较大值
                dp[i][c] = max(dp[i - 1][c], dp[i - 1][c - wgt[i - 1]] + val[i -
1]);
            }
        }
    }
    return dp[n][cap];
}
```

多步决策

- 迭代加深搜索

> 人鬼过河问题：3人3鬼，船载2人以内，任意岸边鬼必须 小于等于 人数，求解过河方案

思路：（R,G）代表当前某一个岸上的人鬼数对，我们在x-y坐标系网格上可以看到所有的安全状态

```
#include <iostream>
#include <vector>
using namespace std;
int count;//全局变量count记录当前状态索引
struct state //状态（人数，鬼数），也可以用来表船的状态
{
    int R;
    int G;
};
vector<struct state> S;
vector<int> choice;//记录每一步的决策
void transfer_state();//状态转移函数
void display(){//输出渡河状态
    for(int i = 1; i <= count; ++i)
        cout << i << ": (" << S[i].R << "," << S[i].G << ")" << endl;
}
int main()
{
    transfer_state();
    display();
    return 0;
}
void transfer_state()
{
    vector<struct state> d = {{0,0}, {1,0}, {2,0}, {1,1}, {0,2}, {0,1}};
    //状态初始化
    count = 1;
    S.resize(20, {3,3});
    choice.resize(20, 0);
    state initial_state = {3,3};
    S[0] = initial_state;
    while(!(S[count].R == 0 && S[count].G == 0)){
```

```
        int fx = 1;
        if (count % 2 == 1)
            fx = -1;
        int i;
        for(i = choice[count + 1] + 1; i <= 5; i++){
            //遍历并试探
            int u = S[count].R + fx * d[i].R;
            int v = S[count].G + fx * d[i].G;
            //减枝
            if( u > 3 || v > 3 || u < 0 || v < 0) continue;//越界 (1)
            if (!(u == 3 || u == 0 || u == v)) continue;//不满足约束->安全性检查：两
边都需要安全 (2)
            //避免走回重复的状态节点->只考虑东岸的所有状态的集合，通过遍历来排除重复 (3)
            bool is_repeat = false;
            for(int j = count - 1; j >= 1; j -= 2)
                if(S[j].R == u && S[j].G == v)
                    is_repeat = true;
            if(is_repeat) continue;
            //试探
            count++;
            S[count].R = u;
            S[count].G = v;
            //记录解
            choice[count] = i;
            break;
        }
        //回退
        if(i > 5){
            count--;
        }
    }
}
```

- 递归求解方法

> task： 统一算法框架求解**所有可能**的过河方案（人鬼过河问题,n = 3, m = 4）

```
#include <iostream>
#include <iomanip>
#include <algorithm>
using namespace std;
struct position {int x,y;};
position dxy[] = {{1,0}, {0,1}, {1,1}, {2,0}, {0,2}};//所有可能的状态转移量
struct state {
    int dir;//将方向纳入状态的考量
    position pos;
};
state start = {-1, {3, 3}}, goal = {1,{0, 0}};//定义状态转移的初态和末态
state path[100];//100在此题目中被认为是一个足够大的量;path结构体数组在多个函数体内部共享,用
来存储经过的路径,也可以看作是状态路径栈.
int num;//全局变量用来存储有多少个可能的解法
int minStep;//全局变量来存储最短的路径
bool isEq(state st1, state st2){
    return (st1.dir == st2.dir) && (st1.pos.x == st2.pos.x) && (st1.pos.y ==
st2.pos.y);
```

```
}
bool isValid(state st, int step){//剪枝函数
    if(st.pos.x < 0 || st.pos.x > 3 || st.pos.y < 0 || st.pos.y > 3)
        return false;
    if(st.pos.x != 0 && st.pos.x != 3 && st.pos.x != st.pos.y)
        return false;
    for(int i = step - 1; i >= 0; i--)
        if(isEq(st, path[i]))
            return false;
    return true;
}
state getNewstate(state st, int k, int step){//根据分支序号k获取此状态的后继状态
    state next_st = {-st.dir, {st.pos.x + st.dir * dxy[k].x, st.pos.y + st.dir *
dxy[k].y}};
    return next_st;
}
bool isDone(state st){//判断终止条件
    if(isEq(st, goal))
        return true;
    return false;
}
void logStep(state st, int step){//将状态中的节点记录到path中
    path[step] = st;
}
void printStep(state st){//将某个节点的信息打印在屏幕上
    cout << setw(2) << st.dir << "(" << st.pos.x << st.pos.y << ")" ;
}
void printPath(int step){//将路径信息打印出来
    for(int i = 0; i <= step; i++)
        printStep(path[i]);
    cout << endl;
}
void Jump(state st, int step){
    //判断递归终止条件
    if(isDone(st)){
        num++;
        minStep = min(minStep, step);
        cout << num << ": ";
        printPath(step);
        return;
    }
    //遍历每一种决策分支
    for(int k = 0; k < sizeof(dxy) / sizeof(dxy[0]) ; k++){
        state next_state = getNewstate(st, k, step);
        if(!isValid(next_state, step + 1))
            continue;//不符合的解会提前跳出，因此不需要再回溯了
        logStep(next_state, step + 1);
        Jump(next_state, step + 1);
    }
}
int main()
{
    num = 0;
    minStep = 1000;
    logStep(start, 0);
    Jump(start, 0);
```

```
    cout << "minStep:" << minStep << endl;
    return 0;
}
```

```
1: -1(33) 1(22)-1(32) 1(30)-1(31) 1(11)-1(22) 1(02)-1(03) 1(01)-1(11) 1(00)
2: -1(33) 1(22)-1(32) 1(30)-1(31) 1(11)-1(22) 1(02)-1(03) 1(01)-1(02) 1(00)
3: -1(33) 1(31)-1(32) 1(30)-1(31) 1(11)-1(22) 1(02)-1(03) 1(01)-1(11) 1(00)
4: -1(33) 1(31)-1(32) 1(30)-1(31) 1(11)-1(22) 1(02)-1(03) 1(01)-1(02) 1(00)
minStep:11
```

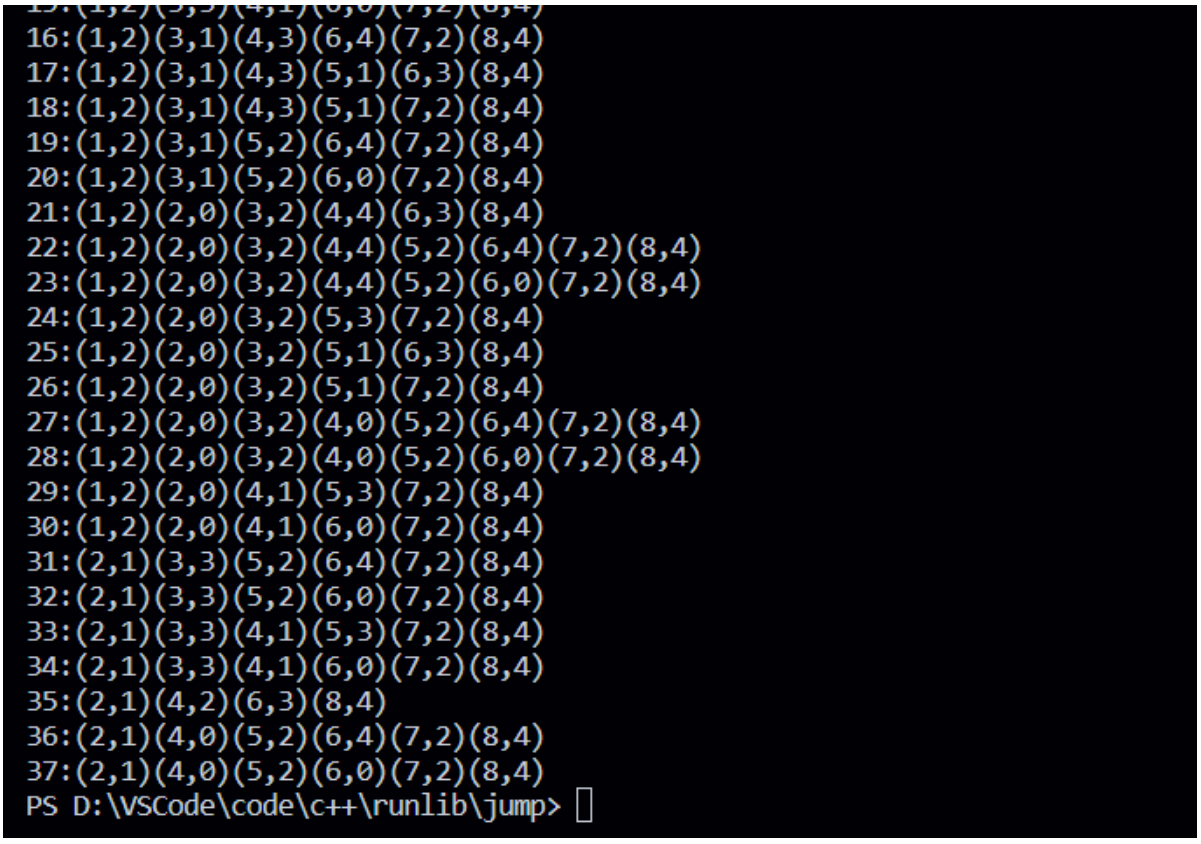
> 跳马问题：采用枚举+递归方式，列举所有可能的完成方案，实际上就是深度优先搜索

思路：跳马操作，可以视为位置的变化（代数化）`(x,y)->(x + dx, y + dy)`

dx dy的实现可以用平行数组或者结构体数组来实现

在用递归解题的时候可以多使用全局变量，方便递归函数。效果如图所示，由于题目本身的特性，避免了图上的环存在（必须向右走）

```
#include <iostream>
using namespace std;
struct position {int x, y;};
//满足题目的走法，用结构体数组来存储
position dxy[4] = {{1,2},{2,1},{2,-1},{1,-2}};
position start_pos = {0,0}, goal_pos ={8, 4};
position position_path[100];
int num;
void Jump(position pos, int step);
int main() {
    num = 0;
    Jump(start_pos, 0);
    return 0;
}
void Jump(position pos, int step){
    //判断是否达到目标,实际上这里可以
    if((pos.x == 8) && (pos.y == 4)){
        num++;//找到了新的方案
        cout << num << ":";
        for(int i = 0; i < step; i++)
            cout << "(" << position_path[i].x << "," << position_path[i].y <<")"
;
        cout << endl;
    }
    //选择
    for(int k = 0;k < 4; k++){
        //试探
        position next_pos = {pos.x + dxy[k].x, pos.y + dxy[k].y};
        if( next_pos.x > 8 || next_pos.x < 0 || next_pos.y < 0 || next_pos.y > 4)
            continue;
        position_path[step] = next_pos;
        Jump(next_pos, step + 1);
    }
}
```



> 下楼问题: 每次可以一步走两步走三步走，求出**所有的**方案

- 有待补充

> 最后一次作业题：gpt: BFS解决general 过河问题的最快解法：

BFS具有的特点就是它是通过一层一层搜索来实现的，因此对于没有权重的图可以可以搜出最近的path，而dfs难以实现这一点。（ps: 对于有权重的图，BFS需要配合其他的算法才能找到最短的路径解法）

```
BFS(graph, start):
    // 创建一个队列
    queue = []
    // 创建一个访问标记数组
    visited = []
    // 将起始节点放入队列，并标记为已访问
    queue.push(start)
    visited[start] = true
    while queue is not empty:
        // 从队列中取出当前节点
        current = queue.pop()
        // 访问当前节点的邻居
        for each neighbor in graph[current]:
            if not visited[neighbor]:
                visited[neighbor] = true
                queue.push(neighbor)
```

```
#include <iostream>
#include <queue>
```

```cpp
#include <set>
#include <tuple>
#include <algorithm>
using namespace std;
struct State {
    //lw: left wolves rw: left sheep bp: boat place st: what steps now
    int lw, ls, bp, st;
    //状态的构造函数
    State(int lw, int ls, int bp, int st) : lw(lw), ls(ls), bp(bp), st(st) {};
    //重载<运算符
    bool operator<(const State &other) const {
        return tie(lw, ls, bp, st) < tie(other.lw, other.ls, other.bp, other.st);
    }
};
bool is_valid(int wolves, int sheep) {
    return sheep >= wolves || sheep == 0; // 保证狼数不超过羊数，除非羊数为0
}
bool is_finished(int wolves, int sheep) {
    return (wolves == 0 && sheep == 0);
}
int bfs(int n, int m)
{
    queue<State> q;
    set<State> visited;
    //初始化状态
    State init_state(n, n, 0, 0);
    q.push(init_state);
    visited.insert(init_state);

    while(!q.empty()) {
        State current = q.front();
        q.pop();
        if(is_finished(current.lw, current.ls)) return current.st;
        if(current.bp == 0) {
            //双层遍历求所有的送法
            for(int w = 0; w <= min(m, current.lw); ++w){
                for(int l = 0; l <= min(m - w, current.ls); ++l){
                    int nlw = current.lw - w, nls = current.ls - l;
                    int nrw = n - current.lw, nrs = n - current.ls;
                    //不符合条件提前剪枝
                    if(w == 0 && l == 0) continue; //船上至少一只羊
                    if(!is_valid(w, l)) continue; // 船上yang >= lang
                    if(!is_valid(nlw, nls)) continue; // 左岸yang >= lang
                    if(!is_valid(nrw, nrs)) continue;
                    State new_state(nlw, nls, 1, current.st + 1);
                    if(visited.find(new_state) == visited.end()){
                        visited.insert(new_state);
                        q.push(new_state);
                    }
                }
            }
        }
        else {
            //双层遍历求所有的送法
            for(int w = 0; w <= min(m, n - current.lw); ++w){
                for(int l = 0; l <= min(m - w, n - current.ls); ++l){
```

```
                    int nrw = n - current.lw, nrs = n - current.ls;
                    int nlw = current.lw + w, nls = current.ls + l;
                    //不符合条件提前剪枝
                    if(w == 0 && l == 0) continue; //船上至少一只羊
                    if(!is_valid(w, l)) continue; // 船上yang >= lang
                    if(!is_valid(nlw, nls)) continue; // 左岸yang >= lang
                    if(!is_valid(nrw, nrs)) continue; // 右岸yang >= lang
                    State new_state(nlw, nls, 0, current.st + 1);
                    if(visited.find(new_state) == visited.end()){
                        visited.insert(new_state);
                        q.push(new_state);
                    }
                }
            }
        }
    }
    return -1;
}

int main() {
    int n, m;
    cin >> n >> m;

    int result = bfs(n, m);
    cout << result << endl;

    return 0;
}
```

- 之前课堂讲解的基于递归和循环的搜索都是base on DFS的！具有很明显的“找到一个解”而非关心这个解是否是最优的（基于最短路径来讨论什么是最优的）

```
dfs(start_node)
    //创建一个空栈 stack
    stack []
    //创建一个 visited 集合，用来记录已访问的节点
    visited []
    //将 start_node 压入栈 stack
    stack.push(start_node)
    while(!stack.empty())
        current_node = stack.pop()
        visited.insert(current_node)
        for(neighbor : current_node.neighbors)
            if(!neighbor in visited)
                stack.push(neighbor)
```