# Contents

## 阅读须知

本次课设报告中的代码前均引入了ATL/base/include/cardinal.hpp头文件，为避免重复在这里一次性给出。

同样，solution类行为均为处理一组输入数据。将在 main 函数多次调用solution类实现多组数据。这里一并给出。

本人使用C++11高级模板技巧原创ATL通用算法模板库github开源地址：https://github.com/OrbitZore/ATL

```cpp
#include <bits/stdc++.h>
using namespace std;
using INT=int;
//#define int long long
#define pb push_back
#define eb emplace_back
#define all(a) (a).begin(),(a).end()
template<class T>
using refT=reference_wrapper<T>;
template<class T>
using crefT=reference_wrapper<const T>;
auto &_=std::ignore;
using ll=long long;
template<class T>
using vec=vector<T>;
template<bool B,class T=void>
using enableif_t=typename enable_if<B,T>::type;
//以下为工具宏
#define DEF_COULD(name,exp) \
template<class U> \
struct name{\
    template<class T>\
    constexpr static auto is(int i)->decltype(exp,true){return true;}\
    template<class T>\
    constexpr static bool is(...){return false;}\
    static const bool value=is<U>(1);\
};
#define DEF_CAN(name,exp) DEF_COULD(can##name,exp)
#define ENABLE(T,name) enableif_t<can##name<T>::value>(1)
#define ENABLEN(T,name) enableif_t<!can##name<T>::value>(1)
#define FOR_TUPLE enableif_t<i!=tuple_size<T>::value>(1)
#define END_TUPLE enableif_t<i==tuple_size<T>::value>(1)

#define DEF_INF(name,exp)\
constexpr struct{\
    template<class T>\
    constexpr operator T()const {return numeric_limits<T>::exp();}\
} name;
//定义SFINAE工具
DEF_CAN(Out,(cout<<*(T*)(0))) DEF_CAN(For,begin(*(T*)(0)))
//定义极限工具变量
DEF_INF(INF,max) DEF_INF(MINF,min)
//以下为读入tuple
template<size_t i,class T>
```

```cpp
auto operator>>(istream& is,T &r)->decltype(END_TUPLE,is){
    return is;
}
template<size_t i=0,class T>
auto operator>>(istream& is,T &r)->decltype(FOR_TUPLE,is){
    is>>get<i>(r);
    return operator>> <i+1>(is,r);
}
//以下为仿C++20 std::format实现
template<class T>
auto __format(ostream &os,const char *c,const T& cv)->decltype(ENABLE(T,Out),c+1){
    os << cv;
    while (*c!='}') c++;
    return c+1;
}
template<size_t i,class T>
auto __format(ostream &os,const char *c,const T& cv)->decltype(ENABLEN(T,For),END_TUPLE,c+1){
    return c;
}
template<size_t i=0,class T>
auto __format(ostream &os,const char *c,const T& cv)->decltype(ENABLEN(T,For),FOR_TUPLE,c+1){
    while (*c!='{') os << *c++;
    c=__format(os,c,get<i>(cv));
    return __format<i+1>(os,c,cv);
}
template<class T>
auto __format(ostream &os,const char *c,const T& cv)->decltype(ENABLEN(T,Out),ENABLE(T,For),c+1){
    const char *ct=c+1;
    if (cv.size()==0){
        while (*ct!='}') ct++;
        ct++;
        while (*ct!='}') ct++;
    }else{
        for (auto &i:cv){
            const char *cc=c+1;
            while (*cc!='{') os << *cc++;
            cc=__format(os,cc,i);
            while (*cc!='}') os << *cc++;
            ct=cc;
        }
    }
    return ct+1;
}
void _format(ostream &os,const char *c){
    while (*c!='{'&&*c!='\0') os<< *c++;
}
template<class T,class ...Args>
void _format(ostream &os,const char *c,const T &a,Args&& ...rest){
    while (*c!='{'&&*c!='\0') os<< *c++;
    if (*c=='{') c=__format(os,c,a);
    _format(os,c,forward<Args>(rest)...);
}
```

```cpp
template<class ...Args>
string format(const char *c,Args&& ...rest){
    ostringstream os;
    _format(os,c,forward<Args>(rest)...);
    return os.str();
}
template<class ...Args>
ostream& print(const char *c,Args&& ...rest){return _format(cout,c,forward<Args>(rest)...),cout;}

#ifdef LOCAL
#define debug(...) cerr<<format(__VA_ARGS__)
#else
#define debug(...) cerr
#endif
//以下为读入多维数组实现
template<class T,class ...Args>
struct Rtar{
    T& a;tuple<Args...> n;
    Rtar(T& a,tuple<Args...> n):a(a),n(n){}
};
template<class T,class ...Args>
Rtar<T,Args&...> rtar(T &a,Args&... rest){
    return Rtar<T,Args&...>(a,tie(rest...));
}
template<size_t i,class U,class ...Args,class T=tuple<Args&...>>
auto operator>>(istream& is,Rtar<U,Args&...> r)->decltype(END_TUPLE,is){
    return is>>r.a;
}
template<size_t i=0,class U,class ...Args,class T=tuple<Args&...>>
auto operator>>(istream& is,Rtar<U,Args&...> r)->decltype(FOR_TUPLE,is){
    r.a=typename decay<U>::type(get<i>(r.n));
    for (auto &w:r.a)
        operator>> <i+1>(is,Rtar<decltype(w),Args&...>(w,r.n));
    return is;
}
//以下为取最小最大值并赋值函数实现
template<class T1,class T2>
bool cmin(T1 &a,const T2 b){return a>b?a=b,1:0;}
template<class T1,class T2>
bool cmax(T1 &a,const T2 b){return a<b?a=b,1:0;}
template<class T1,class T2,class ...T3>
bool cmin(T1 &a,const T2 b,const T3 ...rest){return cmin(a,b)|cmin(a,rest...);}
template<class T1,class T2,class ...T3>
bool cmax(T1 &a,const T2 b,const T3 ...rest){return cmax(a,b)|cmax(a,rest...);}
//控制输入多组数据
bool MULTIDATA=false;
struct solution{
    //...略
    void scan(){
        //...略
    }

    void solve(){
        //...略
    }
```

```
};


INT main(){
    //取消同步，加速cin,cout
    cin.tie(0);
    ios::sync_with_stdio(false);
    int T=1;
    if (MULTIDATA) cin>>(T);
    //多组数据处理
    while (T--){
        auto a=unique_ptr<solution>(new solution());
        a->scan();
        a->solve();
        if (!cin.good()) break;
    }
    return 0;
}
```

## D1T1: 复杂度分析 (A)

### 1. 内容和目的

```
for(i=1;i<n;i++)
  for(j=1;j<i;j++)
    for(k=1;k<j;k++)
      printf("\n");
```

分析代码printf语句执行了多少次，执行完毕后i+j+k等于多少

### 2. 分析与设计实现

**假设都能进入循环**　根据 for 循环定义可知，第一层循环了 $n-1$ 次，第二层循环了 $i-1$ 次，第三层循环了 $j-1$ 次。可知printf语句总共执行了 $\sum_{i=1}^{n-1}\sum_{j=1}^{i-1}\sum_{k=1}^{j-1}1$ 次，计算得 $\frac{n^3-6n^2+11n-6}{6}$，化为计算多项式 $(((n-6)*n+11)*n-6)/6$

同理可知执行完毕后 $i=n, j=n-1, k=n-2$，相加为 $3n-3$

**假设不能进入循环**　即 $n\leq 2$，不会进入第三层循环，此时访问 k 的值为 UB，执行 0 次

### 3. 性能分析

此算式显然为常数复杂度，即 O(1)

### 4. 小结

探明了数学和计算机的联系

## D1T2: 复杂度分析 (B)

### 1. 内容和目的

```
i=1;
while(i++<n){
  j=1;
  while(j++<i){
    k=1;
    while(k++<j)
      printf("\n");
  }
}
```

分析代码printf语句执行了多少次，执行完毕后i+j+k等于多少

### 2. 分析与设计实现

**假设能进入循环**　相比 D1T1，此问题循环实现为先比较后累加。可写出printf执行次数的求和式 $\sum_{i=2}^{n}\sum_{j=2}^{i}\sum_{k=2}^{j}1$

计算可得 $(n-1)*n*(n+1)/6$

执行完毕后显然 $i=n+1, j=n+1, k=n+1$，即 $i+j+k=3n+3$

**当 $n\leq 1$ 时**　无法进入循环

**当 $n=2$ 时**　可以进入循环

### 3. 性能分析

此算式显然为常数复杂度，即 O(1)

### 4. 小结

探明了数学和计算机的联系

## D1T3:Josephus 问题 (1)

### 1. 内容和目的

n 个人排成一圈，按顺时针方向依次编号 1，2，3...n。从编号为 1 的人开始顺时针”一二”报数，报到 2 的人退出圈子。这样不断循环下去，圈子里的人将不断减少。最终一定会剩下一个人。试问最后剩下的人的编号。

要求程序模拟题意来实现。

### 2. 分析与设计实现

使用链表 (std::list) 实现，可这样实现循环，当遍历到`std::list::end()`时赋值为`std::list::begin()`

#### 核心代码

```
struct solution{
    ll n;
    void scan(){
        if (!(cin>>n)) exit(0);
    }

    void solve(){
        list<int> l(n);
        iota(all(l),1);//填充链表
        auto a1=l.begin();
        while (l.size()!=1){
            auto a2=next(a1);
            if (a2==l.end()) a2=l.begin();//回归链表首
            a1=next(a2);
            if (a1==l.end()) a1=l.begin();//回归链表首
            l.erase(a2);//踢人
        }
        cout << *l.begin() << endl;
    }
}
```

### 3. 性能分析

总共 n 个人，为排除一个人执行 2 次遍历，排除一个人时间复杂度为常数，直到最后一个人排除。显然时间复杂度为 O(n)

### 4. 小结

可以使用链表来模拟环上操作

## D1T4:Josephus 问题 (2)

### 1. 内容和目的

主要题意同 D1T3

但是，n 达到了 $2^{31}$，需要时间复杂度小于 $O(n)$ 的算法

### 2. 分析与设计实现

#### 法一

打表找规律知 $a_n = (n - 2^{\lfloor \log 2(n) \rfloor}) * 2 + 1$

#### 法二

考虑递推，设函数 $f(n)$ 为 n 个人报 2，最后一个剩下的人的下标。

则可以推出递推方程 $f(n) = (f(n-1) + m - 1)\%n + 1$

### 3. 性能分析

#### 法一

$2^n$ 可用移位 $O(1)$ 解决，所以整个计算复杂度为 $O(1)$

#### 法二

显然为 $O(n)$

### 4. 小结

训练了初中数学的找规律题目

## D1T5:Josephus 问题 (3)

### 1. 内容和目的

n 个人排成一圈，按顺时针方向依次编号 1，2，3...n。从编号为 1 的人开始顺时针”一二三....”报数，报到 m 的人退出圈子。这样不断循环下去，圈子里的人将不断减少。最终一定会剩下一个人。试问最后剩下的人的编号。

本题的数据规模更具有挑战性，尝试更通用且高效的算法。

### 2. 分析与设计实现

#### 分析

考虑针对 d1t4-Method2 进行优化，考虑到n远大于m。可能要很多次才能到达环末。此时的环可以这样看：

$$(A_{m-1}+B_1)_{\lfloor \frac{n}{m} \rfloor}+C_{n\%m} \tag{0}$$

Tips:A表示跳过的人，B表示淘汰的人，+号表示链接，下标表示重复多少次

考虑一次递推淘汰掉所有B，即调用，$g(x)$ 为一式子

$$f(n,m)=g(f(n-\frac{n}{m},m)) \tag{1}$$

去掉 $B$ 之后的环视图应该是下图，注意我们得到了下一层函数返回的剩下的人的位置其实是从C处开始的，还需要正确处理坐标的偏移量。

$$(A_{m-1})_{\lfloor \frac{n}{m} \rfloor}+\underbrace{C_{n\%m}}_{\text{start here}} \tag{2}$$

考虑处理坐标的偏移量，即求 $(2)$ 式到 $(0)$ 的 $g(x)$

$$g(x)=\begin{cases} x+\lfloor \frac{n}{m} \rfloor & ,x<n\%m,(3) \\ x-n\%m+(x-n\%m-1)/(m-1) & ,x\geq n\%m,(4) \end{cases}$$

$(3)$：当 $x$ 落在 $(2)$ 中的C上时，补上 $\lfloor \frac{n}{m} \rfloor$ 个B的坐标转换式即可完成映射

$(4)$：当 $x$ 落在 $(2)$ 中的A上时，先在不考虑B，然后在补上B的影响

考虑边界情况，$f(1,m)=1$

发现 $m=1$ 时 $(3)$ 未定义，此时 $f(n,1)=n$

又发现 $n\le m$ 时此式无法进行，可使用 d1t4-Method2 方法 $f(n,m)=f(n,1)$

#### 核心代码

```
int f(int n,int m){
    if (m==1) return n;
    if (n==1) return 1;
    if (n<=m) return (f(n-1,m)+m-1)%n+1;
    int w=n%m,e=n-n/m,l=n/m*m;
    int a=f(e,m);
    if (a<=w) return a+l;
    a-=w;a+=(a-1)/(m-1);
    return a;
}
```

### 3. 性能分析

当 $n=m$ 时，显然为 $O(m)$

当 $n>m$ 时，每一轮 $n$ 变为 $n-\lfloor\frac{n}{m}\rfloor$。最坏情况设 $m=2$，$n$ 变为 $\lceil\frac{n}{2}\rceil$，不断的除 2，显然为 $O(\log(n))$

叠加在一起为 $O(m+\log(n))$

### 4. 小结

探索了美妙的数学，将数学知识和计算机编程语言巧妙地结合在一起

## D1T6:Josephus Problem

### 1. 内容和目的

题意同 D1T3, 但是第 n 轮淘汰喊`x[n]`的人

`x[n] = (x[n-1] * A + B)% M.`

## 2. 分析与设计实现

使用二叉搜索树上区间和维护这个人环，初始设所有人为 1，删除操作即设某人为 0。使用树上的查找第 k 大查找存活的第 k 个人。

而寻找淘汰的第 k 个人可以使用数学方法直接计算出来。

### 数据结构设计

```
struct segtree{
    unique_ptr<segtree> lson,rson;
    int s,fsum,t;//fsum维护子树s域的总和
    static int getfsum(unique_ptr<segtree> &ptr){return ptr?ptr->fsum:0;}
    template<class U>
    segtree(const U& op,const U& ed){//从升序容器迭代器区间构造树
        ll cnt=ed-op,mid=cnt>>1;
        s=op[mid];t=1;
        if (0<mid) lson.reset(new segtree(op,op+mid));
        if (mid+1<cnt) rson.reset(new segtree(op+mid+1,ed));
        pushup();
    }
    void pushup(){
        fsum=t+getfsum(lson)+getfsum(rson);
    }
    void unset(const ll &l){//修改第l个人为0
        if (l<=getfsum(lson)) lson->unset(l);
        else if (l<=fsum-getfsum(rson)) {t=0;}
        else rson->unset(l-fsum+getfsum(rson));
        pushup();
    }
    int get(const ll &l){//查找第l个人是谁
        if (l<=getfsum(lson)) return lson->get(l);
        else if (l<=fsum-getfsum(rson)) return s;
        else return rson->get(l-fsum+getfsum(rson));
    }
};
```

### 核心代码

```
void solve(){
    vec<int> v(n+1);
    for (int i=1,j=1;i<=n;i++){
        j=(j+x-1)%(n-i+1)+1;//计算将要淘汰的第k个人
        v[i]=tree->get(j);//寻找第k个人并存储
        tree->unset(j);//设置某人为0
        x=(x*a+b)%M;//计算
    }
    for (int i=1;i<=m;i++){//输出
        int q;cin>>q;
        cout << v[q] << " ";
```

```
    }
    cout << endl;
}
```

### 3. 性能分析

总共 n 轮，一轮执行一次完全二叉树树顶到树下。考虑到此二叉树建树过程确保了左右子树大小差不超过 1，意味着叶子节点俩俩之间的高度差小于等于 1，显然高度为 $O(\log n)$ 级别。时间复杂度为 $O(n \log n)$。

### 4. 小结

训练了手写二叉搜索树的能力

## D1T7: 交集

### 1. 内容和目的

取俩递增无相同元素数列重复项为一个新递增数列。

### 2. 分析与设计实现

考虑使用双指针维护。

i维护a数组，j维护b数组。

先让i从0到n-1循环，确保得到的a数组元素递增。然后每次循环如果b[j]<a[i]令j++。确保b[j]>=a[i]。又因为此为递增过程，b[j]会在符合b[j]>=a[i]下最小，判断b[j]==a[i]我们就找到了每个a[i]对应的b[j]，此时插入vector末尾即可，因为a[i]递增，得到的vector也是递增的。

#### 核心代码

```
vector<int> ans;
int j=0;
for (int i=0;i<n;i++){
    while (i+1<a.size()&&a[i]==a[i+1]) i++;//过滤掉重复
    while (j<b.size()&&a[i]>b[j]) j++;
    if (a[i]==b[j]) ans.push_back(a[i]);
}
```

### 3. 性能分析

遍历了一遍 a 数组和 b 数组，显然为 $O(n)$。

### 4. 小结

探明了双指针在维护数组合并时的妙用。

## D1T8: 大爱线性表

### 1. 内容和目的

实现一个数组，支持序列反转和删除首个元素。

### 2. 分析与设计实现

基于数组，使用双指针维护当前区间即可。为了方便，此处使用对称的全闭区间`[i,j]`

#### 核心代码

处理操作

```
for (auto op:ops){
    if (op=='R') who=!who; //序列反转标记
    else{
        if (i>j){//区间不存在,
            print("error\n");
            return ;
        }
        if (who==0) i++;//没有反转，i即为首元素
        else j--;//反转了，j为首元素
    }
}
```

输出

```
if (who==0){
    cout << "[";
    for (int l=i;l<=j;l++){//正序输出
        cout << v[l];
        if (l!=j) cout  << ",";
    }
    cout << "]";
```

```
}
else{
    cout << "[";
    for (int l=j;l>=i;l--){//反序输出
        cout << v[l];
        if (l!=i) cout  << ",";
    }
    cout << "]";
}
```

### 3. 性能分析

一次循环，显然 $O(n)$

### 4. 小结

继续探明了双指针的用途

## D2T1: 单词检查 (1)- 顺序表实现

### 1. 内容和目的

许多应用程序，如字处理软件，邮件客户端等，都包含了单词检查特性。单词检查是根据字典，找出输入文本中拼错的单词，我们认为凡是不出现在字典中的单词都是错误单词。不仅如此，一些检查程序还能给出类似拼错单词的修改建议单词。例如字典由下面几个单词组成：

bake cake main rain vase

如果输入文件中有词 vake，检查程序就能发现其是一个错误的单词，并且给出 bake, cake 或 vase 做为修改建议单词。

修改建议单词可以采用如下生成技术：1. 在每一个可能位置插入‘a-‘z’中的一者 2. 删除单词中的一个字符 3. 用‘a’-’z’中的一者取代单词中的任一字符

很明显拼写检查程序的核心操作是在字典中查找某个单词，如果字典很大，性能无疑是非常关键的。

你写的程序要求读入字典文件，然后对一个输入文件的单词进行检查，列出其中的错误单词并给出修改建议。

课程设计必须采用如下技术完成并进行复杂度分析及性能比较。- 朴素的算法，用线性表维护字典 - 使用二叉排序树维护字典 - 采用 hash 技术维护字典

本题要求使用顺序表实现。

## 2. 分析与设计实现

此次课设为简化代码将使用 std::string 以及 std::vector 实现线性表字典。

我们将不使用修改建议单词生成技术。使用动态规划算法实现查找字典中单词是否满足 12 条件，使用匹配计数实现判断 3 条件。

### 动态规划状态设计

以下假设字符串a的大小大于字符串b的大小

为删掉a中的任意一个字符即等价于在b中任意位置插入任意一个字符，即 12 子问题是等价的。

我们只需解决在a中最多删一个字符能否匹配上b

设f[n][i]为在a[0:n+i]串中删除了i次，匹配到b[0:n]了没有

转移见代码：核心动态规划判断函数

```
int cmp(const string &A,const string &B){
    const string& a=A.size()>B.size()?A:B;
    const string& b=A.size()>B.size()?B:A;
    if (a.size()-b.size()!=1) return false;//显然ab大小差为1
    vector<array<int,2>> DP(b.size()+1,{false,false});
    array<int,2>* const dp=&DP[1];//保证dp[-1]有定义
    dp[-1]={true,false};
    for (int i=0;i<b.size();i++){
            if (b[i]==a[i])//如果当前位置能匹配上，只能是a没删除的情况下转移
                dp[i][0]|=dp[i-1][0];
            if (b[i]==a[i+1])//如果能和删除后的位置匹配上，可以是删掉a[i]，也可以是之前就删掉了的
                dp[i][1]|=dp[i-1][0]||dp[i-1][1];
    }
    return DP[b.size()][1];
}
```

### 匹配计数

核心代码

```
if (a.size()==b.size()){//长度相等
    int k=0;
    for (int i=0;i<a.size();i++)//逐位匹配
        if (a[i]==b[i]) k++;//匹配计数
    return k==a.size()-1;//如果匹配计数比长度小一，就是修改一个字母使其相等
}
```

### 3. 性能分析

设单词长度为 $L$，字典大小 $n$，查询 $q$ 次。

插入单词 $O(L)$，插入 $n$ 次单词。为 $O(Ln)$

匹配计数和动态规划状态统计显然均为 $O(L)$，查询单词为 $O(L)$，查询 $q$ 次，共为 $O(Lq)$

则时间复杂度为 $O(Ln+Lq)$

### 4. 小结

实现了动态规划的算法

## D2T2: 单词检查 (2)- 二叉排序树实现

### 1. 内容和目的

内容同 D2T1，要求使用二叉排序树实现。

### 2. 分析与设计实现

即把 D2T1 的字典存放数据结构换成二叉搜索树

#### 二叉搜索树数据结构设计

```
template<class T>
struct tree{
    T str;int i;//key-value设计，str为字符串，i为进入字典序号
    unique_ptr<tree> ch[2];//左右子树，使用unique_ptr封装可解决析构问题
    tree(T&& str,int i):str(str),i(i){}
    tree(const T& str,int i):str(str),i(i){}
    static void insert(unique_ptr<tree>& t,const T& a,int i){//插入
        if (!t) {t.reset(new tree(a,i));return ;}
        if (a<t->str)  insert(t->ch[0],a,i);
        else insert(t->ch[1],a,i);
        t->maintain();
    }
    static tree* search(unique_ptr<tree>& t,const T& a){//搜索
        if (!t) return nullptr;
        if (t->str==a) return t.get();
        if (a<t->str) return search(t->ch[0],a);
        return search(t->ch[1],a);
    }
```

```
    static void back_search(unique_ptr<tree>& t){//后序遍历
        if (!t) return ;
        back_search(t->ch[0]);
        back_search(t->ch[1]);
        print("{} ",t->str);
    }
};
```

### 核心代码

```
unique_ptr<tr> t;
string a;int i=0;
while (getline(cin,a),a!="#") tr::insert(t,a,i++);//读入
tr::back_search(t);//后序遍历
cout << endl;
string a;
while (getline(cin,a),a!="#"){
    if (tr::search(t,a)){
        print("{} is correct\n",a);
    }else{
        print("{}:",a);
        for (auto &x:find(a))//find函数使用单词生成技术并尝试匹配字典，返回所有候选单词
            print(" {}",x.second->str);
    }
}
```

### find 函数具体实现

```
vec<pair<int,tr*>> find(const string &a){
    vec<pair<int,tr*>> vs;
    auto inserter=[&](const string &a){//插入工具函数
        auto x=tr::search(t,a);
        if (x) vs.pb(make_pair(x->i,x));
    };
    for (int i=0;i<=a.size();i++){
        for (char c='a';c<='z';c++){
            string tmp;tmp.reserve(a.size()+1);
            tmp+=a.substr(0,i);
            tmp+=c;
            tmp+=a.substr(i,a.size()-i);
            inserter(tmp);//插入字符
        }
        if (i!=a.size()){//删除字符
            string tmp;tmp.reserve(a.size()-1);
            tmp+=a.substr(0,i);
            tmp+=a.substr(i+1,a.size()-i-1);
            inserter(tmp);

            tmp=a;
            for (char c='a';c<='z';c++){//修改单个字符
```

```
                tmp[i]=c;
                inserter(tmp);
            }
        }
    }
    sort(all(vs),[](const pair<int,tr*> &a,
                    const pair<int,tr*> &b){
        return a.first<b.first;
    });//按照插入顺序排序
    vs.resize(unique(all(vs))-vs.begin());//去重
    return vs;
}
```

### 3. 性能分析

$n$ 次插入二叉搜索树，一次平均时间复杂度 $O(L\log n)$，最坏时间复杂度 $O(Ln)$。显然建立字典阶段，平均 $O(Ln\log n)$，最坏 $O(Ln^2)$

总共 $q$ 次查询，单词生成技术时间复杂度为 $O(L^2)$ 生成了 $2L+1$ 个单词，需要进行二叉搜索树search方法，此方法平均时间复杂度 $O(L\log n)$，最坏时间复杂度 $O(Ln)$，总共平均 $O(q(L^2+L^2\log n))=O(qL^2\log n))$，最坏 $O(qL^2n)$

共平均 $O((Ln+qL^2)\log n)$，最坏 $O(Ln^2+qL^2n)$

### 4. 小结

训练了手写二叉搜索树能力

## D2T13: 单词检查 (3)- Hash 表实现

### 1. 内容和目的

同 D2T1，将数据结构换成了 hash 表

### 2. 分析与设计实现

hash 函数实现将采用常见的 BKDRHash，将修改 seed 以做性能比较

### 线性地址法

```cpp
const int MAXN=1024*1024;
unsigned int zhash(const string& str){
    unsigned int sum=0;
    for (int i=0;i<str.size();i++)
        sum=sum*97+str[i]-'a';
    return sum%MAXN;
}

template<class T,class U>//key-value类型
struct shash{
    using vT=pair<T,U>;
    vector<unique_ptr<vT>> s;//桶
    shash(){s.resize(MAXN);}
    void insert(const T& a,const U& b){//插入key-value
        auto i=zhash(a);
        while (s[i]) (i+=1)%=MAXN;
        s[i].reset(new vT(a,b));
    }
    vT* search(const T& a){//搜索key，返回pair
        auto i=zhash(a);
        bool f;
        while (s[i]&&(f=s[i]->first!=a)) (i+=1)%=MAXN;
        return f?nullptr:s[i].get();
    }
};
using tr=shash<string,int>;
```

### 链地址法

```cpp
const int MAXN=1024*1024;
unsigned int zhash(const string& str){
    unsigned int sum=0;
    for (int i=0;i<str.size();i++)
        sum=sum*97+str[i]-'a';
    return sum%MAXN;
}

template<class T,class U>
struct tree{
    using vT=pair<T,U>;
    vector<forward_list<unique_ptr<vT>>> s;
    tree(){s.resize(MAXN);}
    void insert(const T& a,const U& b){
        auto i=zhash(a);
        s[i].push_front(unique_ptr<vT>(new vT(a,b)));
    }
    vT* search(const T& a){
        auto i=zhash(a);
        for (auto &str:s[i])
            if (str->first==a) return str.get();
        return nullptr;
```

```
    }
};
using tr=tree<string,int>;
```

### 3. 性能分析

| | seed=51 | seed=97 | seed=131 | seed=173 |
|---|---|---|---|---|
| 线性地址法 | 时间超限 | 372ms | 419ms | 491ms |
| 链地址法 | 491ms | 484ms | 423ms | 511ms |
| std::hash | 451ms | —— | —— | —— |

可见 seed 只要不是太小，性能比肩 std::hash

相比线性地址法，链地址法更加稳定。实际运用当中应该尽量使用链地址法。

### 4. 小结

实现了哈希表，进行了简单的性能分析并给出了哈希表选型指导性建议

## D3T1: 后缀表达式求值

### 1. 内容和目的

为了便于处理表达式，常常将普通表达式（称为中缀表示）转换为后缀{运算符在后，如X/Y写为XY/表达式。在这样的表示中可以不用括号即可确定求值的顺序，如：(P+Q)*(R - S)*→*PQ+RS - 。后缀表达式的处理过程如下：扫描后缀表达式，凡遇操作数则将之压进堆栈，遇运算符则从堆栈中弹出两个操作数进行该运算，将运算结果压栈，然后继续扫描，直到后缀表达式被扫描完毕为止，此时栈底元素即为该后缀表达式的值。

### 2. 分析与设计实现

使用栈模拟即可

### 核心代码

```
stack<double> s;
void scan(){
    string str;cin>>str;
    while (str!="@"){
        try{
            double a=stod(str);//尝试转为数字
            s.push(a);
        }catch(const exception &e){//输入不是数字，是操作符
            double opl,opr;
            opr=s.top();s.pop();//获取右操作数
            opl=s.top();s.pop();//获取左操作数
            s.push([&,opl,opr]()->double{//应用操作符
                if (str=="/"){
                    return opl/opr;
                }else if (str=="*"){
                    return opl*opr;
                }else if (str=="+"){
                    return opl+opr;
                }else if (str=="-"){
                    return opl-opr;
                }
                return 0;
            }());
        }
        cin>>str;
    }
}

void solve(){//执行完毕后，栈顶元素即为所求答案
    cout << round(s.top()) << endl;
}
```

## 3. 性能分析

显然 $O(n)$

## 4. 小结

训练使用了栈数据结构。

## D3T2: 中缀表达式转后缀表达式

### 1. 内容和目的

输入一个中缀表达式，编程输出其后缀表达式，要求输出的后缀表达式的运算次序与输入的中缀表达式的运算次序相一致。为简单起见，假设输入的中缀表达式由＋（加）、-（减）、×（乘）、/（除）四个运算符号以及左右圆括号和英文字母组成，其中算术运算符遵守先乘除后加减的运算规则。假设输入的中缀表达式长度不超过 300 个字符，且都是正确的，即没有语法错误，并且凡出现括号其内部一定有表达式，即内部至少有一个运算符号。

中缀表达式转后缀表达式的方法：

1. 遇到操作数：直接输出（添加到后缀表达式中）
2. 栈为空时，遇到运算符，直接入栈
3. 遇到左括号：将其入栈
4. 遇到右括号：执行出栈操作，并将出栈的元素输出，直到弹出栈的是左括号，括号不输出。
5. 遇到其他运算符：加减乘除：弹出所有优先级大于或者等于该运算符的栈顶元素，然后将该运算符入栈
6. 最终将栈中的元素依次出栈，输出。

### 2. 分析与设计实现

按题意模拟即可，不过我们发现如果使输入被括号包裹，则可以去掉操作 6 以简化代码。

#### 核心代码

```
map<char,int> PRI={//操作符优先级
    {'+',1},
    {'-',1},
    {'*',2},
    {'/',2},
};
struct solution{
    string str;
    void scan(){
        getline(cin,str);
        str="("+str+")";
    }
    stack<char> stc;
    void solve(){
        for (auto c:str){
            if (isalpha(c)) putchar(c);//字母直接输出
            else if (c=='(') stc.push(c);//左括号入栈
            else if (c==')') {
```

```
                while (stc.top()!='(')//右括号-出栈到左括号
                    putchar(stc.top()),stc.pop();
                stc.pop();//左括号出栈
            }else{
                if (PRI.find(c)!=PRI.end())//是操作符
                    while (PRI.find(stc.top())!=PRI.end()&&
                        PRI[c]<=PRI[stc.top()])//是操作符且优先级比当前操作符大
                        putchar(stc.top()),stc.pop();//出栈
                stc.push(c);//当前操作符入栈
            }
        }
    }
};
```

### 3. 性能分析

显然 $O(n)$

### 4. 小结

深化了对栈和中、后缀表达式的理解。

## D3T3: 二叉树的创建和文本显示

### 1. 内容和目的

编一个程序，读入先序遍历字符串，根据此字符串建立一棵二叉树（以指针方式存储）。例如如下的先序遍历字符串：

A ST C # # D 10 # G # # F # # #

各结点数据（长度不超过 3），用空格分开，其中“#”代表空树。建立起此二叉树以后，再按要求输出二叉树。

输出解释：对于每组数据，显示对应的二叉树，然后再输出一空行。输出形式相当于常规树形左旋 90 度。见样例。注意二叉树的每一层缩进为 4，每一行行尾没有空格符号。

- 输入 1

```
A ST C # # D 10 # G # # F # # #
```

- 输出 1

```
A
              F
         D
                   G
              10
     ST
         C

         6
     5
4
         3
     2
         1
```

## 2. 分析与设计实现

从输入字符串建立二叉树，然后对每一个节点新开一个记录深度的域，按照这个域中序遍历输出即可。注意到此为顺时针旋转 90 度，题目要求逆时针旋转 90 度。倒序输出即可。

### 二叉树数据结构设计

```cpp
struct tree{
    string a;
    int deep;
    unique_ptr<tree> ch[2];
    tree(int deep=0):deep(deep){
        if (!cin.good()) exit(0);//没输入了立即退出
        cin>>a;
        if (a!="#"){//边建树，边维护深度(deep)域
            ch[0].reset(new tree(deep+1));
            ch[1].reset(new tree(deep+1));
        }
    }
    void print(vec<string>& vs){//输出到vs中
        string out;//一行的输出内容
        if (a=="#") return ;
        ch[0]->print(vs);
        for (int i=0;i<deep;i++)
            out+="    ";
        out+=a;
        vs.eb(move(out));//中序遍历输出
        ch[1]->print(vs);
    }
};
```

### 核心代码

```
struct solution{
    tree t;
    void solve(){
        vec<string> s;
        t.print(s);
        reverse(all(s));//倒置
        print("{{}\n}\n",s);//输出
    }
};
```

### 3. 性能分析

### 4. 小结

深化了对二叉树的理解

## D3T4: 表达式树的创建与输出

### 1. 内容和目的

编一个程序，读入先序遍历字符串，根据此字符串建立一棵二叉树（以指针方式存储），请注意的是，我们保证该树一定是表达式树（见教材 5.2 5.8）。

例如下面的先序遍历字符串：

+ 13 # # * 5 # # 9 # #

运算符只可能是加减乘除，数值为小于等于 100，各结点用空格分开，其中“#”代表空树。

### 2. 分析与设计实现

注意到括号仅在操作符时输出，我们可以在中序遍历到操作符时于递归左右子树前后输出括号。这样就处理好了加括号的问题。

#### 数据结构设计

```
struct tree{
    string a;
    int deep;
    unique_ptr<tree> ch[2];
    tree(int deep=0):deep(deep){
```

```
        if (!cin.good()) exit(0);
        cin>>a;
        if (a!="#"){
            ch[0].reset(new tree(deep+1));
            ch[1].reset(new tree(deep+1));
        }
    }
    void print(){
        if (a=="#") return ;
        bool f=ch[0]->a=="#"&&ch[1]->a=="#";//判断是不是数字，数字即叶子节点，
            叶子节点即左右子树都是#
        if (!f) cout << "(";
        ch[0]->print();
        cout<<a;
        ch[1]->print();
        if (!f) cout << ")";
    }
};
```

**核心代码**

```
struct solution{
    tree t;
    void solve(){
        t.print();
        cout<<endl;
    }
};
```

### 3. 性能分析

显然 $O(n)$

### 4. 小结

深化了对二叉树的理解

## D4T1: 表达式树的值

### 1. 内容和目的

读入表达式树的先序遍历字符串，求其值。运算符只可能是加减乘除，保证输入的每个子表达式树的结果都是整数值且可以用 C 语言的 int 类型表达。

样例输入

```
+ 13 # # * 5 # # 9 # #
* + 13 # # 5 # # 9 # #
```

样例输出

```
(13+(5*9))=58
((13+5)*9)=162
```

## 2. 分析与设计实现

先序遍历建树，然后回溯计算

### 数据结构设计

```
map<string,function<double(double,double)>> funcmap={
    {"+",plus<double>()},
    {"-",minus<double>()},
    {"*",multiplies<double>()},
    {"/",divides<double>()}
};//运算符转函数
struct tree{
    string a;
    unique_ptr<tree> ch[2];
    tree(){//构造即建树
        if (!cin.good()) exit(0);
        cin>>a;
        if (a!="#"){
            ch[0].reset(new tree);
            ch[1].reset(new tree);
        }
    }
    double calc(){//计算
        if (ch[0]->a=="#"&&ch[1]->a=="#")//叶子节点，即为数字
            return stoi(a);
        return funcmap[a](ch[0]->calc(),ch[1]->calc());//查表计算
    }
    void print(){//中序遍历输出表达式
        if (a=="#") return ;
        bool f=ch[0]->a=="#"&&ch[1]->a=="#";
        if (!f) cout << "(";
        ch[0]->print();
        cout<<a;
        ch[1]->print();
        if (!f) cout << ")";
    }
};
```

### 核心代码

```
struct solution{
    tree t;
    void scan(){

    }

    void solve(){
        int w=round(t.calc());
        t.print();
        print("={}\n",w);
    }
};
```

### 3. 性能分析

显然 $O(n)$

### 4. 小结

通过运用函数对象，更深刻的认识到了运算的本质。同时成功抽象了代码，具有良好的扩展性。

## D4T2:24 点游戏 (1)

### 1. 内容和目的

24 点游戏的玩法是这样的：任取一幅牌中的 4 张牌 (不含大小王)，每张牌上有数字（其中 A 代表 1，J 代表 11，Q 代表 12，K 代表 13)，你可以利用数学中的加、减、乘、除以及括号想办法得到 24，每张牌只能用一次。例如有四张 6，那么 6+6+6+6=24，也可以 6*6-6-6=24。但是有些牌是无法得到 24 的，比如两张 A 和两张 2。

读入表达式树的先序遍历字符串，这里的表达式树是来自 24 点游戏的真实场景，也就是对应四个数字（值在 1 到 13 之间）组成的表达式，问该表达式树能不能得到 24？

### 2. 分析与设计实现

数据结构设计同D4T1，只需在输出时判断，注意浮点数运算精度误差即可。

#### 核心代码

```
struct solution{
    tree t;
    void scan(){

    }

    void solve(){
        auto w=t.calc();
        if (fabs(w-24.0)<=1e-6){
            t.print();
            print("={}\n",w);
        }else print("NO\n");
    }
};
```

### 3. 性能分析

显然 $O(n)$

### 4. 小结

对浮点数运算有了更深刻的印象

## D4T3:24 点游戏 (2)

### 1. 内容和目的

本步骤要求给定四个数字，生成四个数字能构成的所有表达式树。

### 2. 分析与设计实现

#### 法一：递归求解

make 函数，传入一个数字集合，返回这些数字能构建的树的集合

具体实现即先从传入集合中选取子集合及其补集分别 make，再枚举运算符，循环建立所有可能的树的集合

**make 函数实现**

```
vec<shared_ptr<tree>> make(set<int> v={0,1,2,3}){
    vec<shared_ptr<tree>> ans;
    if (v.size()==1){//大小为一，就是本节点
        ans.eb(shared_ptr<tree>(new tree(*v.begin())));
    }else{
        for (int i=1;i<v.size();i++)
            for (auto str:{"+","-","*","/"}){
                for (auto& pair:choose(all(v),i)){
                    auto &a=pair.first,&b=pair.second;//选子集和补集
                    auto lsons=make(a),rsons=make(b);//递归建树
                    for (auto &l:lsons){
                        for (auto &r:rsons){//循环组合
                            auto t=shared_ptr<tree>(new tree(str));
                            t->ch[0]=l;
                            t->ch[1]=r;
                            ans.eb(t);
                        }
                    }
                }
            }
    }
    return ans;
}
```

**法二：打表**

输出答案有足足 $107520B=107kB$，考虑压缩。

本次将使用 Python LAMA 库压缩以及 binascii 库编码为十六进制

**压缩代码**

```
import lzma,binascii,sys
print(binascii.hexlify(lzma.compress(''.join(sys.stdin.readlines()).encode())))
```

**主要程序**

```
d=b"...省略压缩后字符串..."
import lzma,binascii
s = lzma.decompress(binascii.unhexlify(d))
v = [str(_) for _ in input().strip().split(' ')]
print(len(s))
print(s.decode().replace("a",v[0]).replace("b",v[1]).replace("c",v[2]).replace(
    "d",v[3])[:-1])
```

## 3. 性能分析

**法一**：

考虑本题为 4 个数的排列组合，可知复杂度为常数

### 法二

显然为常数

### 4. 小结

探索了压缩算法在打表中的运用

## D4T4:24 点游戏 (3)

### 1. 内容和目的

按 24 点游戏规则，输入四个数字，判定其能否算出 24。

### 2. 分析与设计实现

只需按照D4T3算出每个表达式后计算能否算出 24 即可。

#### 法一：递归求解

本法沿用D4T3法一

**核心代码**

```
struct solution {
    //其他方法略...
    static vec<shared_ptr<tree>> after;
    //保存0,1,2,3组成的表达式树
    void solve(){
        for (auto& tree_ptr:after){
            auto w=tree_ptr->calc([&](const string& i)->double{
                return a[stoi(i)];
            });//计算中嵌入映射函数，从0,1,2,3映射到输入
            if (fabs(w-24)<=1e-6){//计算精度
                tree_ptr->print([&](const string& i){
                    if (isdigit(i[0])) return to_string(a[stoi(i)]);//同嵌入映
                        射函数
                    return i;
                });
                print("=24\n");
                return ;
            }
        }
        print("NO\n");
```

```
    }
};
vec<shared_ptr<tree>> solution::after=solution::make();//运行前计算好
```

**法二：打表**

带压缩字符串的代码托管在 https://paste.ubuntu.com/p/8hFcnYNk4V/

```
d=b"...省略压缩后字符串..."
import lzma,binascii,sys
s = lzma.decompress(binascii.unhexlify(d))
data = sys.stdin.readlines()
for i in data:
    F=False
    v = [str(_).encode() for _ in i.strip().split(' ')]
    for exp in s.split(b'\n'):
        try:
            s2 = exp.replace(b"a",v[0]).replace(b"b",v[1]).replace(b"c",v[2]).
                replace(b"d",v[3])
            l = eval(s2)
            if abs(l-24)<=1e-6:
                print(s2.decode()+"=24")
                F=True
                break
        except Exception:
            pass
    if not F:
        print("NO")
```

### 3. 性能分析

查表，$O(Cn)$，其中 C 为很大一常数

### 4. 小结

成功实现了解 24 点的程序，体会到了 Python eval() 函数的方便

## D4T5:24 点游戏 (4)

### 1. 内容和目的

按 24 点游戏规则，输入四个数字，求出有多少种不同的解法能算出 24。

如果不同的算式被认为是解法相同，则通过下述规则得到：

1. 加法和乘法的交换律和结合律
2. a/b=a*(1/b)
3. a*1=1*a=a/1=a
4. a/a=1
5. a+0=a-0=a

## 2. 分析与设计实现

难点在于如何判断解法相同，本次课设首先将带入算式使用正则表达式对 2345 规则化简，将不考虑实现结合律，使用交换律判断是否相等

为了方便，本次实现将仅使用 Python

### 化简

核心代码

```
regexList = [
        (rb"x*1",rb"\1"),
        (rb"1*x",rb"\1"),
        (rb"x/1",rb"\1"),
        (rb"x/(1/x)",rb"(\1*\2)"),
        (rb"x*(1/x)",rb"(\1/\2)"),
        (rb"x+0",rb"\1"),
        (rb"0+x",rb"\1"),
        (rb"x-0",rb"\1"),
        (rb"0-x",rb"\1"),#x表示字表达式
]#之后将会更换至符合正则表达式语法的形式
escapeList = [rb"(",rb")",rb"*",rb"+",rb"-"]
#需要添加转义
regexListAfter = []

for g,v in regexList:
    D = rb"("+g+rb")"#手动添加括号
    for op in escapeList:
        D = D.replace(op,b"\\"+op)#添加转义
    D = D.replace(b"x",rb"(\d*|\(.*\))")#x代表数字或者括号包起来的子表达式
    regexListAfter.append((D,v))#存入List

def process(x):
    for i in range(3):#每条规则最多用4-1=3次
        for g,v in regexListAfter:
            x = re.sub(g,v,x)#运用规则
    return x
```

**运用交换律判断是否相等**

定义 $cmp(a,b)$ 为判断 a,b 表达式是否等价

同样递归判断，使用正则表达式分析字表达式

核心代码

```
expRegex = rb"\(((\d*|\(.*\))([\+\-\*/])(\d*|\(.*\)))\)"
def cmp(a,b):
    if a==b:#a=b显然等价
        return True
    aR = re.search(expRegex,a)
    bR = re.search(expRegex,b)
    if aR!=None and bR!=None:#条件1：表达式
        al,aop,ar = aR.group(1),aR.group(2),aR.group(3)
        bl,bop,br = bR.group(1),bR.group(2),bR.group(3)
        if '-'!=aop==bop!='/':#条件2：满足交换律
            return (cmp(al,bl) and cmp(ar,br))or(cmp(al,br) and cmp(ar,bl))#运
                用交换律判断
        else:
            return False
    return False
```

### 3. 性能分析

略

### 4. 小结

探明了正则表达式在符号计算中的作用。感觉在写一个编译器的词法分析部分，希望能够在编译原理中学到更多。

## D5T1: 推箱子游戏-广度优先搜索版本

### 1. 内容和目的

推箱子是一款经典游戏。这里我们玩的是一个简单版本，就是在一个 N*M 的地图上，有 1 个玩家、1 个箱子、1 个目的地以及若干障碍，其余是空地。玩家可以往上下左右 4 个方向移动，但是不能移动出地图或者移动到障碍里去。如果往这个方向移动推到了箱子，箱子也会按这个方向移动一格，当然，箱子也不能被推出地图或推到障碍里。当箱子被推到目的地以后，游戏目标达成。现在告诉你游戏开始是初始的地图布局，请问玩家至少要多少步才能达成目标？

## 2. 分析与设计实现

玩家走动时地图不变，可变状态只有玩家和箱子的坐标。以此建立搜索状态。

### 矢量支持库设计（基于 std::array）

```
#define op_array(x) \
template<class T,size_t d> \//
array<T,d>& operator x##=(array<T,d>& a,const array<T,d>& b){for (size_t i=0;i<d;i++) a[i] x##=b[i];return a;}\
template<class T,class U,size_t d>\
auto operator x##=(array<T,d>& a,const U& b)->decltype(T(declval<U>()),a){for (size_t i=0;i<d;i++) a[i] x##=b;return a;}\
template<class T,size_t d>\
array<T,d> operator x (const array<T,d>& a,const array<T,d>& b){array<T,d> k(a);k x##=b;return k;}
op_array(+) op_array(-) op_array(*) op_array(/)
template<class T>
using vec2=array<T,2>;
template<class T>
using vec3=array<T,3>;
template<class T,size_t d>
T dot(const array<T,d>& a,const array<T,d>& b){
    T ans=0;
    for (size_t i=0;i<d;i++) ans+=a[i]*b[i];
    return ans;
}
template<class T,size_t d>
T abs(const array<T,d>& a){return sqrt(dot(a,a));}
template<class T>
T crs(const vec2<T> &a,const vec2<T> &b){return a[0]*b[1]-a[1]*b[0];}
template<class T>
vec3<T> crs(const vec3<T> &a,const vec3<T> &b){return {a[1]*b[2]-a[2]*b[1],a[2]*b[0]-a[0]*b[2],a[0]*b[1]-a[1]*b[0]};}
template<class T,size_t d>
istream& operator>>(istream& is,array<T,d> &p){
    for (size_t i=0;i<d;i++) is>>p[i];
    return is;
}
```

### 状态结构体设计

```
struct status{
    vec2<int> player,box;//二维向量
    bool operator<(const status &s)const{//重载小于号以使用set记录
        if (player<s.player) return true;
        if (s.player<player) return false;
        if (box<s.box) return true;
        return false;
    }
};
```

### 方向向量设计

```
const array<vec2<int>,4> direct{
    -1,0,
    0,1,
    1,0,
    0,-1
};
const array<char,4> direct2char{
    'U',
    'R',
    'D',
    'L',
};
```

### 搜索函数实现：

```
int bfs(){
    set<status> s;//状态字典
    queue<status> q,Q;//本次使用双队列广搜记录步数
    int step=0;
    auto insert=[&](status t){s.insert(t);Q.push(t);return t.box==dest;};//插入队列工具函数
    auto notin=[&](status t){return s.find(t)==s.end();};
    if (insert(ini)) return step;
    do{
        while (q.size()){
            status t=q.front();q.pop();
            for (auto i:direct){//枚举四个方向
                status nt{t.player+i,t.box};
                if (nt.player==nt.box) nt.box+=i;
                if (vaild(nt.player)&&vaild(nt.box)){//坐标有效
                    if (notin(nt)) {//此状态以前没遍历到
                        if (insert(nt))//插入状态
                            return step;//到达返回
                    }
                }
            }
        }
        swap(q,Q);
        step++;
    }while (q.size());
    return -1;
}
```

### solution 类设计

```
struct solution{
    int n,m;
    vec<vec<char>> G;
    set<status> s;
```

```
    vec2<int> dest;
    status ini;
    void scan(){
        cin>>n>>m>>rtar(G,n,m);//打包读入
    }
    const char at(const vec2<int>& p)const {return G[p[0]][p[1]];}
    bool vaild(const vec2<int>& p)const {return 0<=p[0]&&p[0]<n&&0<=p[1]&&p[1]<
        m&&at(p)!='#';}
    //....省略bfs函数
    void solve(){
        for (int i=0;i<n;i++) for (int j=0;j<m;j++)//记录特殊点坐标
            if (G[i][j]=='X') ini.player={i,j};
            else if (G[i][j]=='*') ini.box={i,j};
            else if (G[i][j]=='@') dest={i,j};
        print("{}\n",bfs());//广搜输出
    }
};
```

### 3. 性能分析

状态空间为 $nmnm$，一次状态计算为 $\log(nmnm)$，复杂度为 $n^2m^2\log(nm)$。

### 4. 小结

训练了 C++ 代码能力。

## D5T2: 推箱子游戏-深度优先搜索版本

### 1. 内容和目的

同 T1，只不过把广度优先搜索换成了深度优先搜索。

### 2. 分析与设计实现

没什么要讲的

#### 核心代码

```
int dfs(status t,int step=0){
    if (t.box==dest) return step;
    for (int j=0;j<4;j++){
        auto& i=direct[j];
```

```
        status nt{t.player+i,t.box};
        if (nt.player==nt.box) nt.box+=i;
        if (vaild(nt.player)&&vaild(nt.box))
            if (s.find(nt)==s.end()){
                s.insert(nt);
                c.push_back(direct2char[j]);
                int a=dfs(nt,step+1);
                if (a!=-1) {
                    return a;
                }
                c.pop_back();
            }
    }
    return -1;
}
```

### 3. 性能分析

同 d5t1，为 $O(n^2m^2\log(nm))$

### 4. 小结

训练了 C++ 代码能力。

## D5T3: 带权路径长度

### 1. 内容和目的

给定一个长度为 $n$ 的数列 $a_i$ 构造 Huffman 树，求带权路径长度。

$n \in [2, 1e5], a_i \in [1, 1e9]$

### 2. 分析与设计实现

模拟构造即可，注意到最终根节点大小可能达到 $1e5 * 1e9 = 1e14$，需要使用`long long`维护节点数据域。

本次实现将使用`ATL::mint`库工具类实现自动模。

### ATL::mint 实现

```cpp
#define op_mint(op)\
_mint operator op (const _mint a)const{_mint k(*this);k op##=a;return k;}
#define cmp_mint(op)\
bool operator op (const _mint a)const{return v op a.v;}
template<class T>
struct _mint{
    T v;
    static T mod;
    _mint()=default;
    _mint(const T &a){(v=a%mod)<0&&(v+=mod);}
    _mint& operator+=(const _mint a){return (v+=a.v)>=mod&&(v-=mod),*this;}
    _mint& operator-=(const _mint a){return (v-=a.v)<0&&(v+=mod),*this;}
    _mint& operator*=(const _mint a){return (v*=a.v)%=mod,*this;}
    op_mint(+) op_mint(-) op_mint(*)
    cmp_mint(<) cmp_mint(>) cmp_mint(<=) cmp_mint(>=) cmp_mint(!=) cmp_mint(==)
};
template<class T>
T _mint<T>::mod;
template<class T>
ostream& operator<<(ostream& os,const _mint<T>& a){return os<<a.v%a.mod;}
template<class T>
istream& operator>>(istream& os,_mint<T>& a){T k;os>>k;a=_mint<T>(k);return os
    ;}
```

### solution 类实现

```cpp
template<>
long long mll::mod=1000000007;
struct solution{
    int n;
    multiset<long long> v;
    void scan(){
        cin>>n;
        if (!cin.good()) exit(0);
        for (int i=0;i<n;i++){
            long long a;cin>>a;
            v.insert(a);
        }
    }

    void solve(){
        mll sum=0;
        while (v.size()!=1){
            auto i=v.begin(),j=next(i);
            long long add=*i+*j;
            sum+=add;
            v.erase(i);v.erase(j);
            v.insert(add);
        }
        print("{}\n",sum);
    }
```

```
26 };
```

### 3. 性能分析

显然 $n\log(n)$

### 4. 小结

训练了 C++ 代码能力。

## D6T1: 自来水管道

### 1. 内容和目的

你领到了一个铺设校园内自来水管道的任务。校园内有若干需要供水的点，每两个供水点可能存在多种铺设路径。对于每一种铺设路径，其成本是预知的。

任务要求最终铺设的管道保证任意两点可以直接或间接的联通，同时总成本最低。

### 2. 分析与设计实现

显然即为最小生成树问题，套用 Kruscal 算法解决。

#### 边数据结构定义

```
template <class vT>
struct wedge{int u,v;vT w;};
template <class vT>
bool operator<(const wedge<vT> &a,const wedge<vT> &b){return a.w<b.w;}
```

#### 并查集数据结构设计

```
struct mfset:protected vector<int>{//继承自std::vector
    mfset(){}
    mfset(int size){resize(size);}
    void resize(int size){
        vector<int>::resize(size);
        iota(this->begin(),this->end(),0);//初始化，每个人都是自己的根节点
    }
```

```
    int find(int a){//找当前节点所对应的根节点
        int &b=this->operator[](a);
        return a==b?a:b=find(b);//路径压缩
    }
    bool merge(int a,int b){//合并节点，合并成功返回true，同一集合返回false
        int aa=find(a),bb=find(b);
        if (aa!=bb){
            this->operator[](bb)=aa;
            return true;
        }
        return false;
    }
};
```

**solution 类设计**

```
struct solution{
    int n,m;
    mfset s;
    vec<wedge<int>> l;
    void scan(){
        cin>>n>>m;
        if (!cin.good()) exit(0);
        s.resize(n+1);
        for (int i=0;i<m;i++){
            int u,v,w;
            cin>>u>>v>>w;
            l.pb({u,v,w});
        }
    }

    void solve(){
        sort(all(l));//升序排序边数组
        ll sum=0;
        for (auto &e:l)//升序遍历每一条边
            if (s.merge(e.u,e.v))//尝试合并
                sum+=e.w;//合并成功，统计答案
        cout << sum << endl;
    }
};
```

## 3. 性能分析

显然 $m \log m$

## 4. 小结

**训练了代码以及抽象能力**

## D6T2: 最小时间

### 1. 内容和目的

有多个城市组成一个铁路交通网络。任意两个城市之间有直连铁路，或者通过其他城市间接到达。给定某个城市，要求在 M 时间内能从该城市到达任意指定的另一城市，求最小的 M。

### 2. 分析与设计实现

显然单源最短路问题，使用 Dijkstra 算法解决。

#### 边类型及邻接表数据结构定义

```
template <class vT>
struct wedge{int u,v;vT w;};
template <template <class vT> class etT,class vT>
struct graph:public vector<vector<etT<vT>>>{//基于二重vector数组
    using eT=etT<vT>;
    using esT=vector<eT>;
    using GT=vector<esT>;
    using vector<vector<etT<vT>>>::vector;
    void addEdge(const eT &a){GT::operator[](a.u).push_back(a);}//添加单向边
    void add2Edge(eT a){addEdge(a);swap(a.u,a.v);addEdge(a);}//添加双向边
};
```

#### dijkstra 算法实现

```
template<class vT=int,class GT>
vector<vT> dijkstra(const GT &G,int i){
    using P=pair<vT,int>;
    const auto n=G.size();
    vector<vT> dis(n,0x3f3f3f3f);
    vector<bool> book(n,true);
    dis[i]=0;
    set<P> p;//堆优化
    p.insert(make_pair(0,i));
    int now=i;
    while (p.size()){
        now=p.begin()->second;p.erase(p.begin());
        if (!book[now]) continue;
        book[now]=false;
        for (auto &e:G[now])
            if (book[e.v]&&dis[e.v]>dis[now]+e.w){
                p.erase(make_pair(dis[e.v],e.v));
                dis[e.v]=dis[now]+e.w;
                p.insert(make_pair(dis[e.v],e.v));
```

```
            }
        }
        return dis;
    }
```

**solution 类设计**

```
struct solution{
    int n,m;
    graph<wedge,int> G;
    void scan(){
        cin>>n;
        if (!cin.good()) exit(0);
        G.resize(n);
        for (int i=0;i<n;i++)
            for (int j=0;j<i;j++){
                string s;int w;
                cin>>s;
                w=s=="x"?0x3f3f3f3f:stoi(s);
                G.add2Edge({i,j,w});
            }
    }

    void solve(){
        auto dis=dijkstra<int>(G,0);
        int w=MINF;
        for (auto i:dis) if (i!=0x3f3f3f3f) cmax(w,i);
        cout << w << endl;
    }
};
```

## 3. 性能分析

显然 $m \log m$

## 4. 小结

训练了代码以及抽象能力

## D6T3:2010 省赛题：Repairing a Road

### 1. 内容和目的

给定一张 n 个点 m 条边的带权无向图，你需要从点 1 走到点 n。在从点 1 开始走的时候你需要指定一条边，你的好友会帮助你修理这条边。在接下来的时间中，假设你时刻 t 走到了这条边的边界点并且接下来将经过这条边，这条边的权值即变为 $v_i * a_i^{-t}$。求你走到点 n 的最短时间

### 2. 分析与设计实现

我们先跑一遍 Floyd 算出最短路矩阵 $D$

因为只能指定一条边，不妨先枚举这一条边 $e$。显然这条边一定要经过，那么离开这条边 $e$ 时，最短路径即为 D[e.v][n]。进入这条边时设已经经过 $t$ 时间，显然 $t \geq$ D[1][e.u]。那么总时间函数 $f(t)$ 应该是

$$f(t) = t + \text{e.v} * \text{e.a}^{-t} + \text{D[e.v][n]}, t \geq \text{D[1][e.u]}$$

求此函数最小值，先求一次导数

$$f'(t) = 1 - \text{e.v} * \text{e.a}^{-t} \log(\text{e.a})$$

显然此导数单调递增，令 $f'(x) = 0$ 解得

$$t_0 = \frac{\log\left(\text{e.v} * \log(\text{e.a})\right)}{\log(\text{e.a})}$$

此即为原函数极小值点，显然也是最小值点。考虑到 $t \geq$ D[1][e.u]。

则定义域内最小值点即为 $\max(t_0, \text{D[1][e.u]})$

带入计算，全部取最小值即可。

注意到 $a = 1$ 时 $t_0$ 不成立，显然答案为 D[1][e,u] + e.w + D[e.v][n]

#### 边数据结构设计以及邻接表图数据结构设计

```
template <class vT>
struct wedge{int u,v;vT w,a;};
template <class vT>
bool operator<(const wedge<vT> &a,const wedge<vT> &b){return a.w<b.w;}
```

```
template <template <class vT> class etT,class vT>
struct graph:public vector<vector<etT<vT>>>{
    using eT=etT<vT>;
    using esT=vector<eT>;
    using GT=vector<esT>;
    using vector<vector<etT<vT>>>::vector;
    void addEdge(const eT &a){GT::operator[](a.u).push_back(a);}
    void add2Edge(eT a){addEdge(a);swap(a.u,a.v);addEdge(a);}
};
```

### Floyd 算法设计

```
template<class vT=int,class GT>
vector<vector<vT>> floyd(const GT &G){
    const auto n=G.size();
    vector<vector<vT>> dp(n,vector<vT>(n,(vT)INF));
    for (size_t i=0;i<n;i++) dp[i][i]=0;
    for (size_t k=0;k<n;k++)
        for (auto &e:G[k])
            cmin(dp[k][e.v],e.w);
    for (size_t k=0;k<n;k++)
        for (size_t i=0;i<n;i++)
            for (size_t j=0;j<n;j++)
                cmin(dp[i][j],dp[i][k]+dp[k][j]);
    return dp;
}
```

### solution 类设计

```
struct solution{
    int n,m;
    graph<wedge,double> G;
    void scan(){
        cin>>n>>m;
        if (n==0&&m==0) exit(0);
        G.resize(n);
        for (int i=0;i<m;i++){
            int u,v;double w,a;
            cin>>u>>v>>w>>a;
            G.add2Edge({--u,--v,w,a});
        }
    }

    void solve(){
        auto M=floyd<double>(G);//得到距离矩阵
        double minn=INF;
        for (auto &r:G) for (auto &e:r){//遍历边
            double a;
            cmin(minn,a=[&](){
                const double from=M[0][e.u],out=M[e.v][n-1];
                if (e.a>1){
```

```
                    const double x=max(log(e.w*log(e.a))/log(e.a),from);
                    return out+x+e.w*pow(e.a,-x);
                }else{
                    return from+out+e.w;
                }
            }());
        }
        printf("%.3f\n",minn);
    }
};
```

### 3. 性能分析

显然大头在 Floyd 中，为 $O(n^3)$

### 4. 小结

训练了代码能力以及抽象能力