在线实验部分

User **邹林壮's** Information

Register time : 2024-10-14 21:47:09
School : 湖南大学
Major : 计算机科学与技术
Grade : 大三
Contact Info :
Last login : 2024-11-24 17:19:48
Accepts : 10 / Submits : 60

Solved problems

10000 10001 10013 10027 10043 10054 10104 10388 11208 13775

Try but failed problems

10013 算法分析与设计实验报告

第_1_次在线实验

姓名	邹林壮	学号	202208040412	班级	计科拔尖 2201 班
时间	2024.11.23	地点	院楼 432		
实验名称	10013 Calendar 计算给定天数后的日期和星期				
实验目的	通过本实验，深入理解和掌握日期计算的方法，特别是如何根据给定的天数计算出相应的日期和星期。同时，通过实际编程练习，提高对 C++语言的运用能力，以及对算法性能的分析能力。				
实验原理	根据公历的规定，能被 4 整除的年份是闰年，但是能被 100 整除而不能被 400 整除的年份不是闰年。因此，1700、1800、1900 和 2100 年不是闰年，而 1600、2000 和 2400 年是闰年。给定自公元 2000 年 1 月 1 日以来的天数，我们可以计算出对应的日期和星期。				
实验步骤	**① 确定输入输出** 输入由各行组成，每行包含一个正整数，即自公元 2000 年 1 月 1 日以来经过的天数 输出为对每个天数输出一行包含日期和星期几的行，格式为“YYYY-MM-DD DayOfWeek” 约束为-1 结束，生成的日期不会晚于 9999 年 Time Limit: 1000ms, Special Time Limit:2500ms, Memory Limit:32768KB **② 判断是否为闰年** 定义函数 pan(int y)，用于判断给定的年份 y 是否为闰年。 年份是 4 的倍数且不是 100 的倍数或者是 400 的倍数，则是闰年返回 1，否则返回 0。 **③ 输入 d，进行两次循环得到年月日** 读取输入的天数 d， 使用第一次循环计算经过的完整年数，并从 d 中减去相应的天数，直到 d 小于一年中的天数， 使用第二次循环计算经过的完整月数，并从 d 中减去相应的天数，直到 d 小于一个月中的天数， 经过上面两次循环得到剩余的天数 d，即为本月中的日期。 **④ 输出结果** 依次输出，注意月份和日期的补零操作，以及其+1 操作，因为数组是从 0 开始的，而真实值是从 1 月开始的。				

关键代码

```
bool pan(int y){
    return (y%4==0&&y%100!=0)||(y%400==0);
}
char
week[7][10]={"Saturday","Sunday","Monday","Tuesday","Wednesday","Thu
rsday","Friday"};
int year[2]={365,366};
int
month[2][12]={31,28,31,30,31,30,31,31,30,31,30,31,31,29,31,30,31,30,
31,31,30,31,30,31};
int ny,nm,nd,nw;
int d;
//2000.1.1 是周六
int main(){

    while(cin>>d){
        ny=0,nm=0,nd=0,nw=0;
        if(d==-1)return 0;
        nw=d%7;//得到星期表示
        for(ny=2000;d>=year[pan(ny)];ny++){
            d-=year[pan(ny)];
        }
        for(nm=0;d>=month[pan(ny)][nm];nm++){
            d-=month[pan(ny)][nm];
        }
        printf("%d-%02d-%02d %s\n",ny,nm+1,d+1,week[nw]);
    }
}
```

测试结果

(1)正确性：

796863	xzxg001	10013	GNU C++	Accepted	1136KB	359ms	663B	2024-11-17 20:25:55.0

正确，可以正确判断闰年，并顺利地得到正确日期，直到-1 停止循环。

(2)复杂度：

时间复杂度:

1.判断闰年的操作：时间复杂度为 O(1)。

2.计算日期和星期的操作：循环的迭代次数取决于输入的天数 d 和当前年份 ny。在最坏情况下，ny 可能需要增加到 9999，这意味着循环可能会执行大约 8000 次（从 2000 到 9999）。因此，这个循环的时间复杂度是 O(T)， T 是从 2000 年到 9999 年的年数，它是一个常数，因此可以将整体时间复杂度简化为 O(1)

则针对每一个输入，时间复杂度为 O(1)

空间复杂度:

数组 week、year 和 month:这些数组的大小是固定的,因此空间复杂度为 O(1)。

总的时间复杂度为 O(1)

实验心得	1.通过本次实验，我对日期计算的方法以及普通的模拟有了更深入的理解。 2.学会了如何判定闰年，并根据给定的天数计算出相应的日期和星期。 3.学会了使用数组来存储，并结合判断进行选择元素。		
实验得分		助教签名	

附录：完整代码

```
#include<iostream>

using namespace std;
bool pan(int y){
    return (y%4==0&&y%100!=0)||(y%400==0);
}
char
week[7][10]={"Saturday","Sunday","Monday","Tuesday","Wednesday","Thursday","Friday
"};
int year[2]={365,366};
int
month[2][12]={31,28,31,30,31,30,31,31,30,31,30,31,31,29,31,30,31,30,31,31,30,31,30
,31};
int ny,nm,nd,nw;
int d;
//2000.1.1是周六
int main(){

    while(cin>>d){
        ny=0,nm=0,nd=0,nw=0;
        if(d==-1)return 0;
        nw=d%7;//得到星期表示
        for(ny=2000;d>=year[pan(ny)];ny++){
            d-=year[pan(ny)];
        }
        for(nm=0;d>=month[pan(ny)][nm];nm++){
            d-=month[pan(ny)][nm];
        }
        printf("%d-%02d-%02d %s\n",ny,nm+1,d+1,week[nw]);
    }
}
```

10388 算法分析与设计实验报告

第_1_次在线实验

姓名	邹林壮	学号	202208040412	班级	计科拔尖 2201 班
时间	2024.11.23	地点	院楼 432		
实验名称	10388 高级模运算				
实验目的	通过本实验，深入理解和掌握模运算的性质和快速幂算法，特别是如何高效计算多个数的乘积之和模 M 的值。同时，通过实际编程练习，提高对 C++语言的运用能力，以及对算法性能的分析能力。				
实验原理	模运算是数论中的一个重要概念，它在密码学、计算机算法等领域有着广泛的应用。快速幂算法是一种用于高效计算幂运算的方法，其核心思想是将幂运算分解为一系列的平方和乘法操作，从而减少计算量。				
实验步骤	⑤ **确定输入输出** 输入为一个数字 M（1≤M≤45000）。第二行是数字 H（1≤H≤45000）表示参加游戏的人数， 接着 H 行，每行两个数 Ai 和 Bi（1≤Ai，Bi≤231） 输出为输出一个数字，（$A_1^{B_1}+ A_2^{B_2} + \ldots + A_n^{B_n}$）mod M 的值 约束为 1≤M≤45000 1≤H≤45000 1≤A_i，B_i≤2^{31} Time Limit: 1000ms, Special Time Limit:2500ms, Memory Limit:32768KB ⑥ **快速幂算法** 主要思路就是通过二分进行优化，每次将要底数平方，结合数的移位操作判断得到是否 ans 需要乘以这个数，并且在过程中注意取模操作来缩小乘法的规模。 ;一些，如果把 n 写作二进制为 $(n_t n_{t-1}\cdots n_1 n_0)_2$，那么有: $n = n_t 2^t + n_{t-1}2^{t-1} + n_{t-2}2^{t-2} + \cdots + n_1 2^1 + n_0 2^0$ 针对每一组数进行快速幂操作 ⑦ **输出结果** 输出累加和 ans 模 M 的结果。				

关键代码	`ll binpow(ll x,ll n){`
	`  x=x%m;`
	`  ll ans=1;`
	`  while(n>0){`
	`  if(n&1)ans=ans*x%m;`
	`  x=x*x%m;`
	`  n>>=1;`
	`  }`
	`  return ans;`
	`}`				

<table>
<tr><td>测试结果</td><td colspan="3">(1)正确性：
 
797003 xzxg001 10388 GNU C++ Accepted 1116KB 406ms 379B 2024-11-19 22:34:30.0
797002 xzxg001 10388 GNU C++ Time Limit Exceeded 1120KB 1015ms 319B 2024-11-19 22:32:55.0
正确,记得开 ios::sync_with_stdio(false);cin.tie(0);cout.tie(0);否则会因为输入输出超时
(2)复杂度：
时间复杂度：针对每一组数据，采用快速幂算法，其时间复杂度为 O(logB)，其中 B 是 Bi 的最大值，整个程序的时间复杂度主要取决于快速幂算法的调用次数。
一共有 H 组数据，因此最终的时间复杂度为 O(HlogB)
空间复杂度：只使用了固定大小的变量，为 O(1)。</td></tr>
<tr><td>实验心得</td><td colspan="3">1.通过本次实验，我对模运算和快速幂算法有了更深入的理解。
2.学会了如何利用快速幂算法提高幂运算的效率，从而优化程序的性能。
3.并且比较了 cin、cout 与 scanf、printf 的时间效率的区别，并且会使用 ios::sync_with_stdio(false);cin.tie(0);cout.tie(0);来提高输入输出的性能</td></tr>
<tr><td>实验得分</td><td></td><td>助教签名</td><td></td></tr>
</table>

附录：完整代码

```
#include<iostream>
using namespace std;
#define ll long long
int m,h;
int ans;
ll binpow(ll x,ll n){
    x=x%m;
    ll ans=1;
    while(n>0){
        if(n&1)ans=ans*x%m;
        x=x*x%m;
        n>>=1;
    }
    return ans;
}
int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    cin>>m>>h;
    int a,b;
    while(h--){
        cin>>a>>b;
        ans+=binpow(a,b);}
    cout<<ans%m;}
```

11208 算法分析与设计实验报告

第_1_次在线实验

姓名	邹林壮	学号	202208040412	班级	计科拔尖 2201 班
时间	2024.11.23	地点	院楼 432		
实验名称	11208 Fibonacci Numbers 计算第 n 个斐波那契数				
实验目的	通过本实验，深入理解和掌握斐波那契数列的生成方法，特别是如何高效计算第 n 个斐波那契数。同时，通过实际编程练习，提高对 C++语言的运用能力，以及对算法性能的分析能力。				
实验原理	斐波那契数列是一个经典的数列，其中每个数等于前两个数的和，初始两个数分别为 0 和 1。本实验通过迭代的方式计算斐波那契数列，对于较大的 n 值，使用斐波那契计算公式和斐波那契计算性质和规律来高效计算第 n 个斐波那契数的前 4 位和后 4 位。				
实验步骤	⑧ **确定输入输出** 输入为第 i 个斐波那契数 输出为第 i 个斐波那契数，如果大于 8 位，则显示前四位和后四位 约束为如果大于 8 位，则显示前四位和后四位，小于等于 8 位，直接输出，i 为 $0\text{-}10^8$ Time Limit: 1000ms, Special Time Limit:2500ms, Memory Limit:65536KB ⑨ 求解前四十位 前四十位是八位以内的斐波那契数列，可以根据斐波那契数列通项公式直接进行计算 $F_n = \frac{1}{\sqrt{5}}[(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n]$ ⑩ 求解前四位 由于 n 越大，Fn 中的减项就越小，因此在计算前四位时可以忽略，前四位可以直接由通项公式求解出， 记我们要算 s 的前四位 d，s 的长度为 len，则 $$s = d.xxx * 10^{\{len-4\}}$$ 两边取 lg，得 $$\lg s = \lg d.xxx + len - 4$$ 从而得到 $$d.xxx = 10^{\lg s + 4 - len}$$ 使用 double 类型，并且只保留前四位 据斐波那契数的通项，我们可以求出当 n 足够大时，s 的近似值即为加项 $$\begin{cases} \lg s = n * \lg \frac{1+\sqrt{5}}{2} - \lg\sqrt{5} \\ d = (int)10^{\lg s - (int)\lg s + 3} \end{cases}$$				

<table>
<tr><td></td><td>根据此推导出的公式求解求解后四位
⑪ 求解后四位
由斐波那契数列的性质我们可以得到以下递推矩阵：
$$\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix} * \begin{bmatrix}f(n)\\ f(n-1)\end{bmatrix} = \begin{bmatrix}f(n+1)\\ f(n)\end{bmatrix}$$
因此
$$\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^{n-1} * \begin{bmatrix}1\\ 0\end{bmatrix} = \begin{bmatrix}f(n)\\ f(n-1)\end{bmatrix}$$
根据此推导，得出结论，只有 a[0][0],a[0][1]在循环过程中有用，令 a[0][0]=a,a[0][1]=b 则可以表示为
t=a;
a=(a+b)
b=t;
但是直接根据这个计算会超时,因此需要结合斐波那契数列后四项的计算性质进行优化，即后四位每 15000 位会循环，则一开始对 15000 取模操作就可以，但是需要注意 0、1、2 时的值。
⑫ 输出结果
最后输出只需要注意对后四位进行补 0 操作即可。</td></tr>
<tr><td>关键代码</td><td>

```
int first4(int n){
    double logs=n*log10((1+sqrt(5))/2)-log10(sqrt(5));
    return (int)pow(10,logs-(int)logs+3);
}
int last4(int n){
    int a=1,b=1;
    int t;
    n=n%15000;
    if(n==0)return 0;
    if(n==1)return 1;
    if(n==2)return 1;
    while(n>2){
        t=a%10000;
        a=(a+b)%10000;
        b=t%10000;
        n--;
    }
    return a%10000;
}
int main(){
    fib[0]=0;
    fib[1]=1;
    for(int i=2;i<40;i++)fib[i]=fib[i-1]+fib[i-2];
    int n;
```

</td></tr>
</table>

<table>
<tr><td></td><td>

```
    while(cin>>n){
        if(n<40)cout<<fib[n]<<endl;
        else {
            cout<<first4(n)<<"...";
            printf("%04d\n",last4(n));

        }
    }
}
```

</td></tr>
<tr><td>测试结果</td><td>

(1)正确性：

796866	xzxg001	11208	GNU C++	Accepted	1164KB	15ms	625B	2024-11-17 20:47:35.0
796865	xzxg001	11208	GNU C++	Wrong Answer	1156KB	31ms	588B	2024-11-17 20:36:13.0
796864	xzxg001	11208	GNU C++	Time Limit Exceeded	1152KB	1015ms	527B	2024-11-17 20:26:39.0

正确，直接根据通项公式计算会超时，结合斐波那契数列后四项的计算性质进行优化，对 15000 取模操作可以实现时间上的优化，并且注意 0、1、2 时的值，就可以通过此题。

(2)复杂度：

时间复杂度:

1. 迭代计算斐波那契数列的时间复杂度为 O(n)，因为需要迭代计算到第 n 个斐波那契数。
2. 对于超过 8 位的斐波那契数，使用公式高效计算前 4 位，时间复杂度 O(1)和后 4 位时间复杂度是 O(15000)，也是常数级，因此这部分的时间复杂度为 O(1)。

针对每一个输入，时间复杂度为 O(1)

空间复杂度:

斐波那契数列数组：代码中定义了一个大小为 40 的数组来存储斐波那契数列的前 40 个数，这部分的空间复杂度为 O(1)。

总的空间复杂度为 O(1)

</td></tr>
<tr><td>实验心得</td><td>

1. 通过本次实验，我对斐波那契数列的计算方法有了更深入的理解。
2. 学会了如何使用迭代和对数方法来高效计算斐波那契数列的大数问题。
3. 学习到如何推导斐波那契数列的通项公式，并且使用行列式求解斐波那契数列的迭代过程，推导出后四项具有循环规律。
4. 学习到了如何省略部分不重要信息实现近似求解。

</td></tr>
<tr><td>实验得分</td><td></td><td>助教签名</td><td></td></tr>
</table>

附录：完整代码

```
#include<iostream>
#include<cmath>
using namespace std;
int fib[40];
int first4(int n){
    double logs=n*log10((1+sqrt(5))/2)-log10(sqrt(5));
    return (int)pow(10,logs-(int)logs+3);
```

```
}
int last4(int n){
    int a=1,b=1;
    int t;
    n=n%15000;
    if(n==0)return 0;
    if(n==1)return 1;
    if(n==2)return 1;
    while(n>2){
        t=a%10000;
        a=(a+b)%10000;
        b=t%10000;
        n--;
    }
    return a%10000;
}
int main(){
    fib[0]=0;
    fib[1]=1;
    for(int i=2;i<40;i++)fib[i]=fib[i-1]+fib[i-2];
    int n;
    while(cin>>n){
        if(n<40)cout<<fib[n]<<endl;
        else {
            cout<<first4(n)<<"...";
            printf("%04d\n",last4(n));

        }
    }
}
```

13775 算法分析与设计实验报告

第_1_次在线实验

姓名	邹林壮	学号	202208040412	班级	计科拔尖 2201 班
时间	2024.11.23	地点	院楼 432		
实验名称	13775 Multiple of 7 判断特定模式数字是否为 7 的倍数				
实验目的	通过本实验，深入理解和掌握数字模式识别和同余性质的应用，特别是如何判断一个特定模式的数字是否为 7 的倍数。同时，通过实际编程练习，提高对 C++语言的运用能力，以及对算法性能的分析能力。				
实验原理	根据同余的性质，我们可以将一个特定模式的数字（如 abab...）分解为若干个部分，然后通过观察这些部分的同余性质来判断整个数字是否为 7 的倍数。由于数据范围较大，最大可接受的时间复杂度为 O(logn)，但无法直接循环求解实现，因此需要利用数字循环的性质来简化问题。				
实验步骤	⑬ 确定输入输出 输入为三个整数 a、b、n，分别代表数字的循环 abab…和长度 输出为如果整数是 7 的倍数，则输出“Yes”，否则输出“No” 约束为 1<=a<=9，0<=b<=9，1<=n<=10^9. **Time Limit:** 1000ms, **Special Time Limit:**2500ms, **Memory Limit:**65536KB ⑭ 构造同余 $abab...\bmod 7$ $=(a\times(10^1+10^3...+10^{n-1})+b\times(10^0+10^2...+10^{n-2}))\bmod 7$ $or\ (a\times(10^0+10^2...+10^{n-1})+b\times(10^1+10^3...+10^{n-2}))\bmod 7$ $=(a\times(10^1+10^3...+10^{n-1})\bmod 7+b\times(10^0+10^2...+10^{n-2})\bmod 7)\bmod 7$ $or\ (a\times(10^0+10^2...+10^{n-1})\bmod 7+b\times(10^1+10^3...+10^{n-2})\bmod 7)\bmod 7$ 因此可以将问题转化为 $(10^0+10^2...+10^{2n})\bmod 7=?$ 接着可以得到 $f(k)=10^k\bmod 7$ $f(0)=1, f(1)=3, f(2)=2, f(3)=6, f(4)=4, f(5)=5,$ $f(6)=1, 10^6\bmod 7=1$ $f(k)=10^k\bmod 7=10^{k-6}10^6\bmod 7\ (k\geqslant 6)=(10^{k-6}\bmod 7*10^6\bmod 7)\bmod 7$ $=f(k-6)*f(6)\bmod 7=f(k-6)$				

<table>
<tr>
<td></td>
<td>
$$(10^0+10^2...+10^{2n})\bmod 7$$
$$=(f(0)+f(2)...+f(2n))\bmod 7$$
$$=(f(0)+f(2)...+f(2n\bmod 6))\bmod 7$$
$$=(k(f(0)+f(2)+f(4))+x_1f(0)+x_2f(2))\bmod 7$$
最后得到结论
$$(10^0+10^2...+10^{2n})\bmod 7$$
$$=(k(f(0)+f(2)+f(4))+x_1f(0)+x_2f(2))\bmod 7$$
$$=(x_1f(0)+x_2f(2))\bmod 7\,((x_1,x_2)=(0,0),(1,0),(1,1))$$
$$(10^1+10^3...+10^{2n+1})\bmod 7$$
$$=(k(f(1)+f(3)+f(5))+x_1f(1)+x_2f(3))\bmod 7$$
$$=(x_1f(1)+x_2f(3))\bmod 7\,((x_1,x_2)=(0,0),(1,0),(1,1))$$
得到结论后，根据 n 对 6 取模的结果，使用 div 数组计算整个数字的同余值。
⑮ 输出结果
根据计算出的同余值判断是否为 7 的倍数，并输出相应的结果
</td>
</tr>
<tr>
<td>关键代码</td>
<td>

```
int main(){
    int div[6]={1,3,2,6,4,5};
    cin>>t;
    while(t--){
        cin>>a>>b>>n;
        n=n%6;
        int ans;
        switch(n){
            case 0:ans=0;
                break;
            case 1:ans=a*div[0];
                break;
            case 2:ans=a*div[1]+b*div[0];
                break;
            case 3:ans=a*(div[2]+div[0])+b*div[1];
                break;
            case 4:ans=a*(div[3]+div[1])+b*(div[2]+div[0]);
                break;
            case 5:ans=b*(div[3]+div[1]);
                break;
        }
        if(ans%7)cout<<"No"<<endl;
        else cout<<"Yes"<<endl;
    }
```

</td>
</tr>
</table>

<table>
<tr><td></td><td>}</td></tr>
<tr><td>测试结果</td><td>(1)正确性：

正确
(2)复杂度：
时间复杂度：因为每个测试用例的判断只涉及判断加六个数相加的计算，为O(n%6)，因此为O(1)。
空间复杂度：因为只使用了固定大小的变量，所以为O(1)。</td></tr>
<tr><td>实验心得</td><td>1.通过本次实验，我对同余性质和数字模式识别有了更深入的理解。
2.学会了如何利用同余性质简化问题，提高算法效率。
3.这扩展了我解决问题的思路，可以通过数学知识进行推导计算，找到其中的规律，从而可以直接对数据进行求解，实现对问题的优化。</td></tr>
<tr><td>实验得分</td><td></td><td>助教签名</td><td></td></tr>
</table>

附录：完整代码

```
#include<bits/stdc++.h>
using namespace std;
int n,t,a,b;
int main(){
    int div[6]={1,3,2,6,4,5};
    cin>>t;
    while(t--){
        cin>>a>>b>>n;
        n=n%6;
        int ans;
        switch(n){
            case 0:ans=0;
                break;
            case 1:ans=a*div[0];
                break;
            case 2:ans=a*div[1]+b*div[0];
                break;
            case 3:ans=a*(div[2]+div[0])+b*div[1];
                break;
            case 4:ans=a*(div[3]+div[1])+b*(div[2]+div[0]);
                break;
            case 5:ans=b*(div[3]+div[1]);
                break;
        }
        if(ans%7)cout<<"No"<<endl;
        else cout<<"Yes"<<endl;
    }
}
```

10027 算法分析与设计实验报告

第_2_次在线实验

姓名	邹林壮	学号	202208040412	班级	计科拔尖 2201 班
时间	2024.11.23	地点	院楼 432		
实验名称	10027 Longest Ordered Subsequence Extentio 单调递增最长子序列问题				
实验目的	通过本实验，深入理解和掌握动态规划算法和贪心算法的设计思想，特别是通过优化动态规划元素内容来优化算法，并且通过特别思路技巧和处理，将时间复杂度减小到$O(nlogn)$。同时，通过实际编程练习，提高对 C++语言的运用能力，以及对算法性能的分析能力。				
实验原理	单调递增最长子序列问题是求一个序列中最长的严格递增子序列的长度。可以通过对 dp 数组存储含义进行优化，并通过二分搜索进行查询优化，从而实现$O(nlogn)$的时间复杂度。也可以通过耐心排序，耐心排序是一种基于贪心策略的算法，通过维护一个辅助数组（伪堆）来记录当前找到的最小尾元素，从而实现$O(nlogn)$的时间复杂度。				
实验步骤	**⑯ 确定输入输出** 输入为序列 N 的长度。第二行包含序列的元素 N 个整数 输出为单调递增最长子序列 约束为 1 <= N <= 50000 **Time Limit:** 1000ms, **Special Time Limit:**2500ms, **Memory Limit:**65536KB **⑰ 使用耐心排序 $O(nlogn)$，并证明最小堆数=LIS 长度** 耐心排序可以找到耐心游戏的最小堆数，证明**耐心排序的最小堆数 = LIS 的长度** **引理 1：最小堆数 >= LIS 长度** 证明：假设我们有一个最长递增子序列：c1 < c2 < ... < cn。 如果我们知道牌 c(i) 的位置，那么牌 c(i+1) 在哪里? 首先，c(i+1) 不可能和 c(i) 放在同一堆中，因为 c(i) 下面所有的牌都必须小于 c(i) 本身（游戏规则的设定）。 其次，c(i+1) 不能放在 c(i) 左边的一堆上，否则 c(i) 会放在那堆的上面（耐心排序原理）。 因此，我们知道牌 c(i+1) 一定位于 c(i) 右侧的某个堆中，这意味着 LIS 的长度至多就是耐心排序的最小堆数。 **引理 2：最小堆数 <= LIS 的长度** 证明：再次假设我们有一个 LIS: c1 < c2 < ... < cn 。 首先我们考虑一张牌，c(i)。这张牌必须大于 c(i) 左侧牌堆上的顶牌 c(i-1)，否则 c(i) 将被放入那堆的上面。然后我们来考虑 c(i-1)，这张牌必须大于 c(i-1) 左侧的牌堆顶牌。因此我们可以看到，每堆中必定有一张牌可以连起来并形成递增顺序（但不一定是最长的，这就是为什么最小堆数最多				

<table>
<tr><td></td><td>是 LIS 的长度)。
因此：最小堆数 = LIS 的长度
通过以上两个引理，我们知道最小桩数必须等于 LIS 的长度才能同时满足两个引理。因此，最小堆数 = LIS 的长度。
⑱ 通过记录最小结尾元素值来优化
对该算法进行优化，容易看出 i-1 到 i 的循环中，a[i]的值起关键作用。如果 a[i]能够扩展到序列 a[0:i − 1]的最长递增子序列的长度，则 k=k+1，否则 k 不变。设 a[0:i − 1]中长度为 k 的最长可以扩展递增子序列的结尾元素是 a[j](0 <= j < i − 1)，则当 a[i] >= a[j]时可以扩展，否则不可以扩展。
如果存在多个长度为 k 的递增子序列，只需要递增子序列中结尾元素的最小值 b[k]，因此将 b[k]作为序列 a[0:i − 1]中所有长度为 k 的递增子序列中的最小结尾元素值。
增强假设后，在 i-1 到 i 的循环中，当 a[i] >= b[k]时，k = k + 1, b[k] = a[i]，否则 k 值不变；当 a[i] < b[k]时，如果 a[i] < b[1]，则应该将 b[1]的值更新为 a[i]，如果 b[1] <= a[i] <= b[k]，则二分搜索查找下标 j，使得 b[j − 1] <= a[i] < b[j]，此时 b[1:j − 1]和 b[j + 1:k]的值不变，b[j]的值更改为 a[i]

⑲ 输出结果
给定序列的最长有序子序列的长度。</td></tr>
<tr><td>关键代码</td><td>1. 使用耐心排序优化

```
int lengthOfLIS(vector<ll> a){
    int size=0;
    vector<ll> q(a.size());
    int i,j;
    for(int x:a){
        i=0,j=size;
        while(i<j){
            int m=i+(j-i)/2;
            if(q[m]<x)i=m+1;
            else j=m;
        }
        q[i]=x;
        size=max(i+1,size);
    }
    return size;
}
```

2. 通过记录最小结尾元素值来优化

```
ll lengthOfLIS(int a[]){
    b[1]=a[0];
    ll k=1;
    for(ll i=1;i<n;i++){
        if(a[i]>b[k]){
            b[++k]=a[i];
```
</td></tr>
</table>

```
7.          }else{
8.              b[std::lower_bound(b,b+k,a[i])-b]=a[i];
9.  /*          //二分查找算法，用于在已排序的范围内查找第一个不小于给定值的元素
10.             ForwardIterator lower_bound(ForwardIterator first,
11.                                 ForwardIterator last,
12.                                 const T& value);
13.             返回的是一个地址，-b 得到该元素所在位置
14. */
15.         }
16.     }
17.     return k;
18. }
```

测试结果

(1)正确性：

在 acm 平台和洛谷上进行测试，时间和空间都满足要求，正确。

(2)复杂度：

时间复杂度:

1. 二分查找：对于每个元素 x 在数组 a 中，代码使用二分查找在 q 数组中找到合适的位置插入 x。二分查找的时间复杂度为 O(logk)，其中 k 是 q 数组的当前大小，即已找到的递增子序列的长度。

总体时间复杂度：由于每个元素都需要进行一次二分查找，总体时间复杂度为 O(nlog k)，其中 n 是数组 a 的长度。由于 k 最多为 n，因此时间复杂度最差为 $O(nlogn)$。

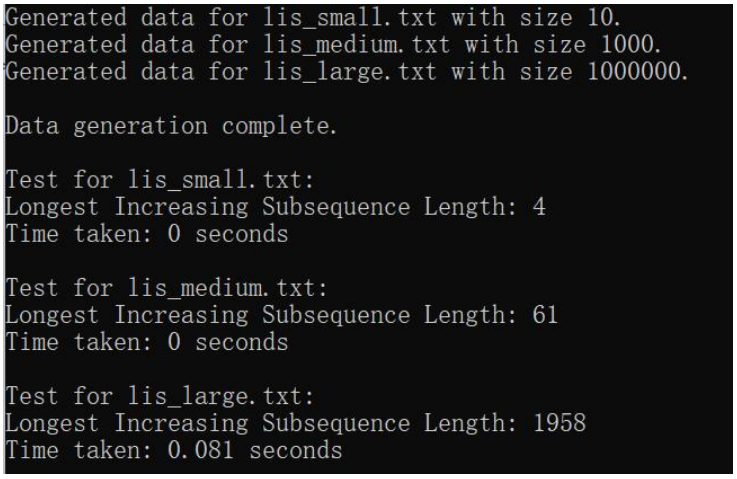
空间复杂度:

辅助数组：代码中使用辅助数组 q 或者 b，其大小与输入数组 a 相同，因此空间复杂度为 O(n)

(3)构造了不同规模数据集并进行图像化演示:

生成不同规模和大小的数据

```
generate_test_data("lis_small.txt", 10, 100);        // 小规模
generate_test_data("lis_medium.txt", 1000, 10000); // 中规模
generate_test_data("lis_large.txt", 1000000, 1000000); // 大规模
```

	
实验心得	1.动态规划的有效性与状态选择有关，选择不同的状态其效率不同，可以通过改变 dp 所存储的内容，从而改变状态空间，在一定程度上可以优化算法效率。 2.通过这道题，我深入理解了动态规划的应用，通过记录前面状态的选择情况，新状态通过常数级查询完成对新状态下的选择情况，并且有最优子结构的证明，动态规划法很严谨，在很多问题上都可以应用。

实验得分		助教签名	

附录：完整代码

1.优化 dp 实现

```
//https://zhuanlan.zhihu.com/p/670544975
#include<iostream>
#include<vector>
using namespace std;
#define ll long long
const int N=5e5+10;
int a[N],b[N];
int n;
ll lengthOfLIS(int a[]){
    b[1]=a[0];
    ll k=1;
    for(ll i=1;i<n;i++){
        if(a[i]>b[k]){
            b[++k]=a[i];
        }else{
            b[std::lower_bound(b,b+k,a[i])-b]=a[i];
            //二分查找算法，用于在已排序的范围内查找第一个不小于给定值的元素
/*          ForwardIterator lower_bound(ForwardIterator first,
                        ForwardIterator last,
                        const T& value);
            返回的是一个地址，-b 得到该元素所在位置
*/
```

```
        }
    }
    return k;
}
int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    cin>>n;
    for(ll i=0;i<n;i++)cin>>a[i];
    ll ans=lengthOfLIS(a);
    cout<<ans<<endl;
}
```

2.耐心排序解决

```
#include<bits/stdc++.h>

using namespace std;
#define ll long long

int lengthOfLIS(vector<ll> a){
    int size=0;
    vector<ll> q(a.size());
    int i,j;
    for(int x:a){
        i=0,j=size;
        while(i<j){
            int m=i+(j-i)/2;
            if(q[m]<x)i=m+1;
            else j=m;
        }
        q[i]=x;
        size=max(i+1,size);
    }
    return size;
}
int main(){
    ll n;
    cin>>n;
    vector<ll> a(n);
    for(int i=0;i<n;i++)cin>>a[i];
    ll ans=lengthOfLIS(a);
    cout<<ans<<endl;
}
```

10043 算法分析与设计实验报告

第_2_次在线实验

姓名	邹林壮	学号	202208040412	班级	计科拔尖 2201 班
时间	2024.11.23	地点	院楼 432		
实验名称	10043 Bottle taps 啤酒瓶盖收集				
实验目的	通过本实验，掌握动态规划在状态压缩问题中的应用，特别是如何在多状态组合下，使用动态规划有效求解最小化问题。通过编码实践，深入理解状态表示、转移方程及优化方法在实际问题中的重要性。				
实验原理	验中，Petrov 想以最小成本收集指定的瓶盖，问题可以抽象为一个集合覆盖最小花费问题。瓶盖编号可以表示为二进制位，利用状态压缩技术，动态规划求解每种状态的最小花费。 关键思想: 1.状态压缩:用一个整数的二进制位表示瓶盖的收集状态。例如,状态 0110 表示 Petrov 已经收集了编号 2 和 3 的瓶盖。 2.转移方程: 考虑单件购买或套餐购买，更新当前状态的最小花费。 转移方程: $$dp[state\|new\_items] = \min(dp[state\|new\_items], dp[state] + cost)$$ 其中 state 是当前状态,new_items 是购买的物品集合,dp[i] 表示收集状态为 i 时的最小花费				
实验步骤	⑳ **确定输入输出** 输入为可用抽头的数量 N，N 行包含带水龙头的瓶子的价格一个卖出水龙头的报价数量 M,接着 M 行中每行的第一个数字是该组的价格，第二个数字是该组中的点击数，然后是集合中的抽头编号 输出为获得必要水龙头上的最低金额 约束为 1 <= N <= 20 0 <= M <= 100 所有价格均为正整数，不超过 1000 Time Limit: 1000ms, Special Time Limit:2500ms, Memory Limit:32768KB ㉑ **初始化与输入** 初始化并输入单件商品价格数组、套餐信息结构体、动态规划数组和瓶盖数量 n ㉒ **设计状态压缩动态规划方案** 动态规划的核心思想是通过状态压缩计算每种状态下的最小花费: 状态表示: 用整数 state 的二进制表示记录当前已收集的瓶盖状态，用 dp[state]表示达到 state 状态的最小花费。 状态转移方程: 当前状态为 state,目标是更新通过购买商品或选择套餐后的新状态的最小花				

<table>
<tr><td></td><td>费:
单件商品购买:
$$dp[state|(1 << i)] = \min(dp[state|(1 << i)], dp[state] + \text{singlePrice}[i])$$
其中 $(1 << i)$ 表示选择瓶盖编号 i。
套餐购买:
$$dp[state|\text{packages[j].items}] = \min(dp[state|\text{packages[j].items}], dp[state] + \text{packages[j].price})$$
其中 <code>packages[j].items</code> 表示套餐 j 覆盖的瓶盖状态。
最终目标是找到 dp[target]，即完全覆盖目标瓶盖状态的最小花费

㉓ 动态规划实现
对每个状态 state，若当前状态不可达 (dp[state] == INF)，跳过；否则，尝试通过动态规划表示单件购买或套餐更新状态。
最后，如果 dp[target] == INF，表示目标状态不可达，输出 -1；否则输出 dp[target]，即满足目标瓶盖需求的最小花费。

㉔ 输出结果
输出获得目标瓶盖集合的最小花费</td></tr>
<tr><td>关键代码</td><td><pre>
while (true) {
        // 输入商品数量
        if (scanf("%d", &n) == EOF || n == 0) break; // EOF 或 n = 0 表示结束
        int m = 1 << n; // 状态总数
        fill(dp, dp + m, INF); // 初始化 dp 数组为无穷大

        // 输入单件商品的价格并初始化 dp
        for (int i = 0; i < n; ++i) {
            scanf("%d", &singlePrice[i]);
            dp[1 << i] = singlePrice[i]; // 每件商品单独购买的价格
        }

        // 输入套餐信息
        int k;
        scanf("%d", &k);
        for (int i = 0; i < k; ++i) {
            scanf("%d", &packages[i].price);
            int w;
            scanf("%d", &w);
            packages[i].items = 0;
            while (w--) {
                int temp;
                scanf("%d", &temp);
                --temp;
                packages[i].items |= (1 << temp);
</pre></td></tr>
</table>

```
            }
        }

        // 输入目标商品集合
        int target = 0;
        int w;
        scanf("%d", &w);
        while (w--) {
            int temp;
            scanf("%d", &temp);
            --temp;
            target |= (1 << temp);
        }

        // 更新 DP 状态
        dp[0] = 0; // 初始状态
        for (int i = 0; i < m; ++i) {
            if (dp[i] == INF) continue; // 如果当前状态不可达，跳过

            // 尝试购买单件商品更新 DP
            for (int j = 0; j < n; ++j) {
                dp[i | (1 << j)] = min(dp[i | (1 << j)], dp[i] +
singlePrice[j]);
            }

            // 尝试使用套餐更新 DP
            for (int j = 0; j < k; ++j) {
                dp[i | packages[j].items] = min(dp[i |
packages[j].items], dp[i] + packages[j].price);
            }
        }

        // 找到满足目标的最小花费
        int minCost = dp[target];
        printf("%d\n", (minCost == INF ? -1 : minCost));
    }
```

测试结果

(1)正确性：

798696	xzxg001	10043	GNU C++	Accepted	5200KB	218ms	2243B	2024-11-23 12:45:37.0

正确

(2)复杂度：

时间复杂度:

1.单件商品购买更新:对于每个状态 i，遍历 n 件商品。

每次转移时计算新状态的最小花费,转移复杂度为 O(n),因为共有2^n个状态,

所以单件商品购买部分的总复杂度为：$O(n \times 2^n)$

	2.套餐购买更新:对于每个状态 i，尝试用 m 个套餐进行状态转移，每次转移时计算新状态的最小花费，转移复杂度为 O(m)，因为共有$\mathbf{2^n}$个状态，所以套餐部分的总复杂度为: $\mathbf{O(m \times 2^n)}$ 因此总的时间复杂度为 $O\big((n+m)\times 2^n\big)$ 空间复杂度: **1. 动态规划数组 dp:**长度为$\mathbf{2^n}$，每个状态存储一个整数，其空间复杂度为 O($\mathbf{2^n}$) **2. 单件商品价格数组 singlePrice:**存储 n 件商品的价格，其空间复杂度为 O(n) **3. 套餐信息数组 packages:**存储 m 个套餐的信息，每个套餐包含价格和物品集合，每个套餐存储一个整数表示物品集合和一个整数表示价格，空间复杂度为 O(m) 总而言之，总的空间复杂度为 $O(2^n+n+m)\approx O(2^n)$
实验心得	1.我体会到状态表示的设计对动态规划的简洁性和效率至关重要，动态规划的核心在于如何将问题分解为状态并定义状态转移方程。在本题中，使用二进制的每一位表示一个瓶盖的收集状态，设计状态压缩后的 DP 数组 dp[state]，表示当前状态下的最小花费。这种设计充分利用了二进制位的特性，既简洁又高效 2.我对动态规划的核心思想有了更深的认识: 最优子结构: 本题中，收集指定瓶盖的最小花费可以通过之前状态的最小花费递推得到，满足动态规划的最优子结构性质。 无后效性: 一个状态的最小花费仅依赖于之前的状态，与转移路径无关，确保了 DP 的有效性。 实际意义的广泛性: 类似的状态压缩动态规划方法还可以应用于子集覆盖问题、背包问题等
实验得分		助教签名	

附录：完整代码

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <vector>
#include <algorithm>
using namespace std;

#define INF 0x5fffffff
const int N = 105;   // 最大商品数量
const int M = (1 << 20); // 最大状态数量（2^20）

int singlePrice[N]; // 单件商品的价格
```

```
struct Package {
    int price, items; // 套餐的价格和包含的商品
} packages[N]; // 套餐信息
int dp[M]; // 动态规划数组
int n;

int main() {
    while (true) {
        // 输入商品数量
        if (scanf("%d", &n) == EOF || n == 0) break; // EOF 或 n = 0 表示结束
        int m = 1 << n; // 状态总数
        fill(dp, dp + m, INF); // 初始化 dp 数组为无穷大

        // 输入单件商品的价格并初始化 dp
        for (int i = 0; i < n; ++i) {
            scanf("%d", &singlePrice[i]);
            dp[1 << i] = singlePrice[i]; // 每件商品单独购买的价格
        }

        // 输入套餐信息
        int k;
        scanf("%d", &k);
        for (int i = 0; i < k; ++i) {
            scanf("%d", &packages[i].price);
            int w;
            scanf("%d", &w);
            packages[i].items = 0;
            while (w--) {
                int temp;
                scanf("%d", &temp);
                --temp;
                packages[i].items |= (1 << temp);
            }
        }

        // 输入目标商品集合
        int target = 0;
        int w;
        scanf("%d", &w);
        while (w--) {
            int temp;
            scanf("%d", &temp);
            --temp;
            target |= (1 << temp);
```

```
        }

        // 更新 DP 状态
        dp[0] = 0; // 初始状态
        for (int i = 0; i < m; ++i) {
            if (dp[i] == INF) continue; // 如果当前状态不可达，跳过

            // 尝试购买单件商品更新 DP
            for (int j = 0; j < n; ++j) {
                dp[i | (1 << j)] = min(dp[i | (1 << j)], dp[i] + singlePrice[j]);
            }

            // 尝试使用套餐更新 DP
            for (int j = 0; j < k; ++j) {
                dp[i | packages[j].items] = min(dp[i | packages[j].items], dp[i] +
packages[j].price);
            }
        }

        // 找到满足目标的最小花费
        int minCost = dp[target];
        printf("%d\n", (minCost == INF ? -1 : minCost));
    }
    return 0;
}
```

10054 算法分析与设计实验报告

第_2_次在线实验

姓名	邹林壮	学号	202208040412	班级	计科拔尖 2201 班
时间	2024.11.23	地点	院楼 432		
实验名称	10054 拦截导弹				
实验目的	通过本实验，深入理解和掌握动态规划算法和贪心算法的设计思想，特别是如何通过耐心排序解决单调递增最长子序列问题.以及通过优化动态规划元素内容来优化算法，并且通过特别思路技巧和处理，将时间复杂度减小到$O(nlogn)$。同时，通过实际编程练习，提高对 C++语言的运用能力，以及对算法性能的分析能力。				
实验原理	单调递增最长子序列问题是求一个序列中最长的严格递增子序列的长度。耐心排序是一种基于贪心策略的算法，通过维护一个辅助数组（伪堆）来记录当前找到的最小尾元素，从而实现$O(nlogn)$的时间复杂度。也可以通过对 dp 数组存储含义进行优化，并通过二分搜索进行查询优化，从而实现$O(nlogn)$的时间复杂度。				
实验步骤	**㉕ 确定输入输出** 输入为导弹依次飞来的高度 h 输出为这套系统最多能拦截的导弹数 mp 和要拦截所有导弹最少要配备这种导弹拦截系统的套数 约束为 h<30000 Time Limit: 1000ms, Special Time Limit:2500ms, Memory Limit:32768KB **㉖ 求解最多能拦截的导弹数** 这里有两种方法: **方法一：使用最长单调不升子序列** 将拦截的导弹的高度提出来成为原高度序列的一个子序列，根据题意这个子序列中的元素是单调不增的（即后一项总是不大于前一项），我们称为单调不升子序列。本问所求能拦截到的最多的导弹，即求最长的单调不升子序列，和我之前做的最长单调上升子序列方法一致，这里记录长度为 i 的最大结尾元素，从而实现 dp 优化，以 O(nlogn)解决问题 **方法二：根据 Dilworth 定理转换** 狄尔沃斯定理亦称偏序集分解定理，该定理断言：对于任意有限偏序集，其最大反链中元素的数目必等于最小链划分中链的数目。此定理的对偶形式亦真，它断言：对于任意有限偏序集，其最长链中元素的数目必等于其最小反链划分中反链的数目。 该定理在该问题上可以理解成：把序列分成不上升子序列的最少个数，等于序列的最长上升子序列长度。把序列分成不降子序列的最少个数，等于序列的最长下降子序列长度。 因此可以**根据求解第二问的序列的最长上升子序列长度转换为不上升子序列**				

	的最少个数。 **㉗ 求解配备这种导弹拦截系统的最少套数** 针对第二问也有两种方法: **方法一：ro_num 函数，采用贪心算法** 对于每个数，既可以把它接到已有的导弹拦截后面，也可以建立一个新系统。要使子序列数最少，应尽量不建立新序列。 另外，应让每个导弹系统的**末尾尽可能大**，这样能接的数更多。因为一个数若能接到小数后面，必然能接到大数后面，反之则不成立。根据贪心策略，可总结出如下贪心算法流程: 从前往后扫描每个数，对于当前数 1. 若现有子序列的结尾都小于它，则创建新子序列。 2. 否则，将它放到结尾大于等于它的最小数后面。 **贪心证明** 我们可以知道，证明 A=B，可证 A≤B 且 A≥B。 记 A 为贪心解，B 为最优解。 1.贪心解能覆盖所有数，且形成的都是不升序列，因此合法。由定义，B≤A。 2.假设最优解对应的方案和贪心方案不同,从前往后找到第一个不在同一序列的数 x。假设贪心解中 x 前面的数是 a，最优解中 x 前面的数是 b，a 后面的数是 y，由于贪心会让当前数接到大于等于它的最小数后面，所以$x,y \le a \le b$。此时，在最优解中，把 x 一直到序列末尾，和 y 一直到序列末尾交换位置，这样做不影响正确性，也不增加序列个数，但会使 x 在最优解和贪心解中所处的位置相同。由于序列中的数是有限的，只要一直做下去，一定能使最优解变为贪心解。因此 A≤B，则 A=B，即我们的贪心解等于最优解。 **方法二：问题等价于求解最长单调递增子序列长度** 很自然地将问题转换为之前实验解决地问题，通过二分查找来加快 b[i]长度为 i 的最小结尾元素位置的定位，从而将算法时间效率提高至 O(nlogn)。 如果存在多个长度为 k 的递增子序列,只需要递增子序列中结尾元素的最小值 b[k]，因此将 b[k]作为序列 a[0:i − 1]中所有长度为 k 的递增子序列中的最小结尾元素值。 增强假设后，在 i-1 到 i 的循环中，当 a[i] >= b[k]时，k = k + 1,b[k] = a[I],否则 k 值不变；当 a[i] < b[k]时，如果 a[i] < b[1],则应该将 b[1]的值更新为 a[i],如果 b[1] <= a[i] <= b[k],则二分搜索查找下标 j,使得 b[j − 1] <= a[i] < b[j],此时 b[1:j − 1]和 b[j + 1:k]的值不变，b[j]的值更改为 a[i] **㉘ 输出结果** 输出这套系统最多能拦截的导弹数和要拦截所有导弹最少要配备这种导弹拦截系统的套数

关键代码	**1.求解最多能拦截的导弹数：**
	方法一：
	`int afind(int i,int mp){`
	`  int l=1,r=mp;`
	`  while(l<r){`
	`  int mid=(l+r)/2;`
	`  if(dp[mid]<a[i])r=mid;`
	`  else l=mid+1;`

```
    }
    return r;
}
int maxdp_q(){
    dp[1]=a[0];
    int mp=1;
    for(int i=1;i<n;i++){
        if(a[i]<=dp[mp])dp[++mp]=a[i];
        else dp[afind(i,mp)]=a[i];
//  for(int i=1;i<n;i++)cout<<dp[i]<<" ";
//  cout<<endl;
    }
    return mp;
}
    int mp=maxdp_q();
    cout<<mp<<" ";
```

方法二:

```
int ro_num(){
    int k=1;
    m[1]=a[0];
    for(int i=0;i<n;i++){
    int st=1;
    while(st<=k&&a[i]>m[st])st++;
    m[st]=a[i];
    k=max(st,k);
    }
    return k;
}
    cout<<n-ro_num();
```

2.**求解配备这种导弹拦截系统的最少套数**

方法一:

```
int ro_num(){
    int k=1;
    m[1]=a[0];
    for(int i=0;i<n;i++){
    int st=1;
    while(st<=k&&a[i]>m[st])st++;
    m[st]=a[i];
    k=max(st,k);
    }
    return k;
}
```

```
    cout<<ro_num();
```

方法二:

```
ll lengthOfLIS(int a[]){
    b[1]=a[0];
    ll k=1;
    for(ll i=1;i<n;i++){
        if(a[i]>b[k]){
            b[++k]=a[i];
        }else{
            b[std::lower_bound(b,b+k,a[i])-b]=a[i];
        }
    }
    return k;
}
    cout<<lengthOfLIS(a);
```

测试结果

(1)正确性:

在洛谷和 acm 平台上进行测试，在时间和空间上均满足题目要求，正确。

(2)复杂度:

时间复杂度分析

1.最多能拦截的导弹数（最长单调不升子序列）:

方法一：动态规划:

对于每个导弹高度 a[i]，我们需要在 dp 数组中找到合适的位置插入 a[i]。使用二分搜索，最坏情况下，需要遍历 logn 次，时间复杂度为 O(logn)。

由于我们需要对每个导弹高度执行此操作，总的时间复杂度为 O(nlogn)。

方法二：Dilworth 定理转换:

根据 Dilworth 定理，最长不升子序列的长度等于最长递增子序列的长度。这种方法实际上将问题转化为求解最长递增子序列，求解策略和方法一类似，时间复杂度与方法一相同，为 $O(nlogn)$。

2.要拦截所有导弹最少要配备的导弹拦截系统套数（最长单调递增子序列）:

方法一：贪心算法:

对于每个导弹高度 a[i],我们可能需要在 m 数组中找到合适的位置插入 a[i],这可以通过线性搜索完成，时间复杂度为 O(k)，其中 k 是当前已找到的递增子序列的长度。

由于 k 的最大值为 n，总的时间复杂度为 $O(n^2)$。

方法二：二分查找:

对于每个导弹高度 a[i]，我们使用二分查找在 b 数组中找到合适的位置插入 a[i]，时间复杂度为 O(log k)，其中 k 是当前已找到的递增子序列的长度。

由于 k 的最大值为 n，总的时间复杂度为 $O(nlogn)$。

空间复杂度分析

	1.最多能拦截的导弹数（最长单调不升子序列）: **方法一：动态规划**: 需要一个大小为 n 的 dp 数组来存储中间结果，因此空间复杂度为 O(n)。 **方法二：Dilworth 定理转换**: 同样需要一个大小为 n 的数组来存储中间结果，空间复杂度为 O(n)。 **2.要拦截所有导弹最少要配备的导弹拦截系统套数（最长单调递增子序列）**: **方法一：贪心算法**: 需要一个大小为 n 的 m 数组来存储中间结果，因此空间复杂度为 O(n)。 **方法二：二分查找**: 需要一个大小为 n 的 b 数组来存储中间结果，因此空间复杂度为 O(n)。
实验心得	1.通过本次实验，我对最长递增子序列（LIS）问题有了更深入的理解，特别是在解决实际问题中的应用。我学会了如何将一个看似复杂的问题转化为 LIS 问题，并使用动态规划和贪心算法来解决。 2.在实验过程中，我深刻体会到了算法优化的重要性。通过对 dp 数组存储含义的优化和二分搜索的应用，我成功将时间复杂度降低到了 O(n log n)，这在处理大规模数据时显得尤为重要。此外，我也学习到了狄尔沃斯定理，并理解了如何根据 Dilworth 定理转换进行问题转换，这对我的算法思维是一次极大的提升。学会了 3.在编写代码的过程中，我意识到了代码的可读性和效率同样重要。良好的代码结构和注释可以大大提高代码的可维护性，而高效的算法则可以提升程序的运行速度。这次实验让我更加注重代码质量，也让我更加热爱算法设计。

实验得分		助教签名	

附录：完整代码

```
#include<iostream>

using namespace std;
const int N=1e6+10;
#define ll long long
int n=0;
int a[N],dp[N],m[N],b[N];

int afind(int i,int mp){
    int l=1,r=mp;
    while(l<r){
        int mid=(l+r)/2;
        if(dp[mid]<a[i])r=mid;
        else l=mid+1;
    }
    return r;
}
int maxdp_q(){
```

```
    dp[1]=a[0];
    int mp=1;
    for(int i=1;i<n;i++){
        if(a[i]<=dp[mp])dp[++mp]=a[i];
        else dp[afind(i,mp)]=a[i];
//  for(int i=1;i<n;i++)cout<<dp[i]<<" ";
//  cout<<endl;
    }
    return mp;
}
ll lengthOfLIS(int a[]){
    b[1]=a[0];
    ll k=1;
    for(ll i=1;i<n;i++){
        if(a[i]>b[k]){
            b[++k]=a[i];
        }else{
            b[std::lower_bound(b,b+k,a[i])-b]=a[i];
        }
    }
    return k;
}
int ro_num(){
    int k=1;
    m[1]=a[0];
    for(int i=0;i<n;i++){
    int st=1;
    while(st<=k&&a[i]>m[st])st++;
    m[st]=a[i];
    k=max(st,k);
    }
    return k;
}

int main(){
    while(cin>>a[n])n++;



    int mp=maxdp_q();
    cout<<mp<<" ";
    cout<<lengthOfLIS(a);
}
```

10104 算法分析与设计实验报告

第_2_次在线实验

姓名	邹林壮	学号	202208040412	班级	计科拔尖 2201 班
时间	2024.11.23	地点	院楼 432		
实验名称	10104 病毒代码检测				
实验目的	通过本实验，熟悉自动机算法，特别是 AC 自动机在字符串模式匹配问题中的应用。通过设计一种高效的自动机结构，检测无限长二进制代码中是否存在病毒片段，提高对字符串算法的理解和对算法复杂度的分析能力。				
实验原理	病毒代码检测问题可以转化为一个字符串匹配问题，其中需要判断给定的二进制串是否包含任何已知的病毒代码段。Trie 树是一种用于存储字符串集合的树形数据结构，而失败函数用于在 Trie 树中处理重叠的模式串匹配问题。				
实验步骤	㉙ **确定输入输出** 输入为病毒代码段的数目 n，接着 n 行每一行都包括一个非空的 01 字符串，表示一个病毒代码段 输出为 TAK——假如存在这样的代码； NIE——如果不存在 约束条件: n<=30000 **Time Limit:** 1000ms, **Special Time Limit:**2500ms, **Memory Limit:**32768KB ㉚ **构建 Trie 树** 依次将每个病毒片段插入 Trie 树，构造节点存储所有病毒代码段并标记末端节点。 从 Trie 树的根节点（编号 0）开始，按照字符串 s 的字符逐一构建路径，如果当前字符 s[i] 的对应子节点不存在，创建新的节点并更新 tot，继续向下移动到当前字符对应的节点，最后将路径的末端节点标记为病毒代码段的末尾 (en[p] = true) ㉛ **构建失败指针** 使用广度优先搜索（BFS）构建失败指针，令每个节点跳转到其最长的后缀匹配节点。 bfs 遍历 Trie 树节点，如果当前节点 x 的子节点存在，将子节点的失败指针设置为 x 的失败指针的对应子节点，如果失败指针指向的节点是病毒末尾节点，则当前子节点也标记为病毒末尾节点，将子节点入队。如果当前节点的子节点不存在，则设置子节点为当前节点失败指针的对应子节点。 ㉜ **DFS 检测死循环**: 对自动机进行深度优先搜索，检查从初始状态开始是否存在一个环路，即无限安全代码。 如果当前节点正在访问 (v[x] == 1)，说明存在环，返回 true。如果当前节点已经访问完成 (v[x] == -1)，说明从该节点出发是安全的，返回 false，标记当前节点为正在访问 (v[x] = 1)，遍历当前节点的两个子节点：如果子				

	节点不是病毒末尾节点，递归检查子节点，若发现环路，立即返回 true。 若当前节点访问完成且安全，标记为 v[x] = -1 **㉝ 输出结果** 根据 DFS 的结果，输出 "TAK" 或 "NIE"。

关键代码	queue<int> q;
	const int maxn=30030;
	int trie[maxn][2],fail[maxn],n,tot,v[maxn];
	char s[maxn];
	bool en[maxn];
	inline void insert(){
	int len=strlen(s),p=0;
	for(int i=0;i<len;i++){
	int ch=s[i]-'0';
	if(!trie[p][ch])trie[p][ch]=++tot;
	p=trie[p][ch];
	}
	en[p]=true;
	}
	inline void built(){
	for(int i=0;i<2;i++){if(trie[0][i])q.push(trie[0][i]);}
	while(q.size()){
	int x=q.front();q.pop();
	for(int i=0;i<2;i++){
	if(trie[x][i]){
	fail[trie[x][i]]=trie[fail[x]][i];q.push(trie[x][i]);
	if(en[trie[fail[x]][i]]) en[trie[x][i]]=1;}
	else trie[x][i]=trie[fail[x]][i];
	}
	}
	}
	inline bool dfs(int x){
	if(v[x]==1) return true;
	if(v[x]==-1) return false;
	v[x]=1;
	for(int
	i=0;i<=1;i++){if(!en[trie[x][i]]){if(dfs(trie[x][i]))return 1;}}
	v[x]=-1;
	return false;
	}

<table>
<tr><td>测试结果</td><td colspan="3">
(1)正确性：


正确。
(2)复杂度：
时间复杂度:
1.构建 Trie 树:构建 Trie 树,每个字符的插入操作耗时 O(1),构建整个 Trie 树的时间复杂度为 O(L)。
2.构建失败指针: 使用 BFS，每个节点访问一次，并处理其所有子节点，Trie 树的节点数不超过 L，每个节点的失败指针处理和子节点更新操作为 O(1)，因此，构建失败指针的时间复杂度为 O(L)。
3.检测无限安全代码:dfs 遍历自动机，判断是否存在环,而每个节点在访问过程中可能需要递归访问其两个子节点，每个节点最多被访问两次（进入和离开），且访问的过程中仅涉及常数级别的操作（如判断标记、递归子节点等操作），Trie 树的节点数为 n,最大为 L，因此，DFS 的时间复杂度为 O(L)。
总而言之，整体的时间复杂度为 O(L)。
空间复杂度:
1.Trie 树:每个节点存储 2 个子节点指针,最多 L 个字符对应的路径节点需要存储。子节点指针的总大小为 2×L，空间复杂度为 O(L)。
2.失败指针：每个节点存储一个失败指针 fail[x],节点数为 L,失败指针的总大小为 L，空间复杂度为 O(L)。
3.标记和辅助数组：en[maxn] 标记每个节点是否是病毒代码段的末尾节点;v[maxn] 标记每个节点的访问状态,每个数组大小为 L,队列 q 的最大大小为 L，辅助数组和队列的空间复杂度均为 O(L)。
总而言之，整体的空间复杂度为 O(L)
</td></tr>
<tr><td>实验心得</td><td colspan="3">1.通过本次实验，我对 Trie 树和失败函数有了更深入的理解。我学会了如何使用 Trie 树来存储和匹配字符串,以及如何通过失败函数来处理重叠的模式串匹配问题。
2.在实验过程中，我深刻体会到了算法优化的重要性。通过对 Trie 树和失败函数的构建和优化,我成功地将问题转化为一个可解的字符串匹配问题。此外,我也加深了如何使用深度优先搜索来解决复杂的问题。</td></tr>
<tr><td>实验得分</td><td></td><td>助教签名</td><td></td></tr>
</table>

附录：完整代码

```
#include<iostream>
#include<cstdio>
#include<cstring>
#include<cmath>
#include<queue>
#include<algorithm>
#define ll long long

using namespace std;
```

```
queue<int> q;
const int maxn=30030;
int trie[maxn][2],fail[maxn],n,tot,v[maxn];
char s[maxn];
bool en[maxn];
inline void insert(){
    int len=strlen(s),p=0;
    for(int i=0;i<len;i++){
        int ch=s[i]-'0';
        if(!trie[p][ch])trie[p][ch]=++tot;
        p=trie[p][ch];
    }
    en[p]=true;
}
inline void built(){
    for(int i=0;i<2;i++){if(trie[0][i])q.push(trie[0][i]);}
    while(q.size()){
        int x=q.front();q.pop();
        for(int i=0;i<2;i++){
            if(trie[x][i]){
            fail[trie[x][i]]=trie[fail[x]][i];q.push(trie[x][i]);
            if(en[trie[fail[x]][i]]) en[trie[x][i]]=1;}
            else trie[x][i]=trie[fail[x]][i];
        }
    }
}
inline bool dfs(int x){
    if(v[x]==1) return true;
    if(v[x]==-1) return false;
    v[x]=1;
    for(int i=0;i<=1;i++){if(!en[trie[x][i]]){if(dfs(trie[x][i]))return 1;}}
    v[x]=-1;
    return false;
}

int main(){
    cin>>n;
    for(int i=1;i<=n;i++){scanf("%s",s);insert();}
    built();
    if(dfs(0))cout<<"TAK"<<endl;
    else cout<<"NIE"<<endl;
    return 0;
}
```