经典问题：最近公共祖先, 最长公共子序列(LCS), 最长递增子序列(LIS)

数智班

计算思维综合实践

动态规划

- 这种将一个问题分解为子问题递归求解，并且将中间结果保存以避免重复计算的办法，就叫做“动态规划”。
- 动态规划通常用来求最优解，能用动态规划解决的求最优解问题，必须满足，最优解的每个局部解也都是最优的。

例题：数字三角形

```
        7
      3   8
    8   1   0
  2   7   4   4
4   5   2   6   5
```

- 上图给出了一个数字三角形。从三角形的顶部到底部有很多条不同的路径。对于每条路径，把路径上面的数加起来可以得到一个和，和最大的路径称为最佳路径。你的任务就是求出最佳路径上的数字之和。
- 注意：路径上的每一步只能从一个数走到下一层上和它最近的左边的数或者右边的数。

解法一

- 这道题目可以用递归的方法解决。
- 基本思路是：
  - 以**D(r, j)**表示第r 行第 j 个数字(r，j 都从1 开始算)，以**MaxSum(r, j)** 代表从第 r 行的第 j 个数字到底边的最佳路径的数字之和，则本题是要求 MaxSum(1, 1) 。
  - 从某个**D(r, j)**出发，显然下一步只能走D(r+1, j)或者D(r+1, j+1)。
  - 如果走D(r+1, j)，那么得到的MaxSum(r, j)就是MaxSum(r+1, j) + D(r, j)；
  - 如果走D(r+1, j+1)，那么得到的MaxSum(r, j)就是MaxSum(r+1, j+1) + D(r, j)。
  - 所以，选择往哪里走，就看MaxSum(r+1, j)和MaxSum(r+1, j+1)哪个更大了。

解法一参考程序

```
#include <stdio.h>
#define MAX_NUM 100
int D[MAX_NUM + 10][MAX_NUM + 10];
int N;
int MaxSum(int r, int j) {
        if (r == N)
                return D[r][j];
        int nSum1 = MaxSum(r + 1, j);
        int nSum2 = MaxSum(r + 1, j + 1);
        if (nSum1 > nSum2)
                return nSum1 + D[r][j];
        return nSum2 + D[r][j];


}
main() {
        int m;
        scanf("%d", &N);
        for (int i = 1; i <= N; i ++)
                for (int j = 1; j <= i; j ++)
                        scanf("%d", &D[i][j]);
        printf("%d", MaxSum(1, 1));
}
```

Time Limit Exceed	1004	C++	2s	960K

解法一存在的问题

- 上面的程序，效率非常低，在N 值并不大，比如N=100 的时候，就慢得几乎永远算不出结果了。为什么会这样呢？
- 是因为过多的重复计算。我们不妨将对MaxSum 函数的一次调用称为一次计算。那么，每次计算MaxSum(r, j)的时候，都要计算一次MaxSum(r+1, j)，而每次计算MaxSum(r, j+1)的时候，也要计算一次MaxSum(r+1, j)。重复计算因此产生。在题目中给出的例子里，如果我们将MaxSum(r, j)被计算的次数都写在位置（r, j），那么就能得到下面的三角形：

解法一存在的问题

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

- 从上图可以看出，最后一行的计算次数总和是16，倒数第二行的计算次数总和是8。不难总结出规律，对于N 行的三角形，总的计算次数是$2^0+2^1+2^2 \ldots\ldots 2^{N-1}=2^N$。当N= 100 时，总的计算次数是一个让人无法接受的大数字。

解法一存在问题的解决

- 既然问题出在重复计算，那么解决的办法，当然就是，一个值一旦算出来，就要记住，以后不必重新计算。即第一次算出MaxSum(r, j)的值时，就将该值存放起来，下次再需要计算MaxSum(r, j)时，直接取用存好的值即可，不必再次调用MaxSum 进行函数递归计算了。
- 这样，每个MaxSum(r, j)都只需要计算1 次即可，那么总的计算次数（即调用MaxSum 函数的次数）就是三角形中的数字总数，即1+2+3+……N = N(N+1)/2

解法二

- 如何存放计算出来的MaxSum（r, j）值呢？
- 显然，用一个二维数组aMaxSum[N][N]就能解决。aMaxSum[r][j]就存放MaxSum(r, j)的计算结果。下次再需要MaxSum(r, j)的值时，不必再调用MaxSum 函数，只需直接取aMaxSum[r][j]的值即可。

解法二参考程序

```
#include <stdio.h>
#include <memory.h>
#define MAX_NUM 100
int D[MAX_NUM + 10][MAX_NUM + 10];
int N;
int aMaxSum[MAX_NUM + 10][MAX_NUM + 10];
int MaxSum(int r, int j) {
        if (r == N)
                return D[r][j];
        if (aMaxSum[r+1][j] == -1)  //如果MaxSum(r+1, j)没有计算过
                aMaxSum[r+1][j] = MaxSum(r + 1, j);
        if (aMaxSum[r+1][j+1] == -1) //如果MaxSum(r+1, j+1)没有计算过
                aMaxSum[r+1][j+1] = MaxSum(r + 1, j + 1);
        if (aMaxSum[r+1][j] > aMaxSum[r+1][j+1])
                return aMaxSum[r+1][j] + D[r][j];
        return aMaxSum[r+1][j+1] + D[r][j];
}
main() {
        int m;
        scanf("%d", & N);
//将 aMaxSum 全部置成-1, 表示开始所有的 MaxSum(r, j)都没有算过
        memset(aMaxSum, -1, sizeof(aMaxSum));
        for (int i = 1; i <= N; i ++)
                for (int j = 1; j <= i; j ++)
                        scanf("%d", & D[i][j]);
        printf("%d", MaxSum(1, 1));
}
```

Accepted	1004	C++	0.04s	5540K

解法三

- 实际上，递归的思想在编程时未必要实现为递归函数。在上面的例子里，有递推公式：

$$\text{anMaxSum}[r][j] = \begin{cases} D[r][j] & r = N \\ \text{Max}(\text{anMaxSum}[r+1][j], \text{anMaxSum}[r+1][j+1]) + D[r][j] & \text{其他情况} \end{cases}$$

- 因此，不需要写递归函数，从aMaxSum[N-1]这一行元素开始向上逐行递推，就能求得最终aMaxSum[1][1]的值了。

解法三参考程序

```
#include <stdio.h>
#include <memory.h>
#define MAX_NUM 100
int D[MAX_NUM + 10][MAX_NUM + 10];
int N;
int aMaxSum[MAX_NUM + 10][MAX_NUM + 10];
main() {
        int i, j;
        scanf("%d", & N);
        for (i = 1; i <= N; i ++)
                for (j = 1; j <= i; j ++)
                        scanf("%d", &D[i][j]);
        for (j = 1; j <= N; j ++)
                aMaxSum[N][j] = D[N][j];
        for (i = N ; i > 1 ; i --)
                for (j = 1; j < i ; j ++) {
                        if (aMaxSum[i][j] > aMaxSum[i][j+1])
                                aMaxSum[i-1][j] = aMaxSum[i][j] + D[i-1][j];
                        else
                                aMaxSum[i-1][j] = aMaxSum[i][j+1] + D[i-1][j];
                }
        printf("%d", aMaxSum[1][1]);
}
```

Accepted	1004	C++	0.01s	2920K

例题小结

- 这种将一个问题分解为子问题递归/递推求解，并且将中间结果保存以避免重复计算的办法，就叫做“动态规划”。
- 动态规划通常用来求最优解，能用动态规划解决的求最优解问题，必须满足，最优解的每个局部解也都是最优的。
- 以上题为例，最佳路径上面的每个数字到底部的那一段路径，都是从该数字出发到达到底部的最佳路径。

思考题

- 上面的几个程序只算出了最佳路径的数字之和。如果要求输出最佳路径上的每个数字，该怎么办？

动态规划解题的一般思路

- 许多求最优解的问题可以用动态规划来解决。用动态规划解题，首先要把原问题分解为若干个子问题，这一点和前面的递归方法类似。区别在于，单纯的递归往往会导致子问题被重复计算，而用动态规划的方法，子问题的解一旦求出就会被保存，所以每个子问题只需求解一次。
- 在用动态规划解题时，我们往往将和子问题相关的各个变量的一组取值，称之为一个“状态”。一个“状态”对应于一个或多个子问题，所谓某个“状态”下的“值”，就是这个“状态”所对应的子问题的解。

- 定义出什么是“状态”，以及在该“状态”下的“值”后，就要找出不同的状态之间如何迁移———即如何从一个或多个“值”已知的“状态”，求出另一个“状态”的“值”。
- 状态的迁移可以用递推公式表示，此递推公式也可被称作“状态转移方程”。

- 所有“状态”的集合，构成问题的“状态空间”。“状态空间”的大小，与用动态规划解决问题的时间复杂度直接相关。在数字三角形的例子里，一共有N×(N+1)/2 个数字，所以这个问题的状态空间里一共就有N×(N+1)/2 个状态。在该问题里每个“状态”只需要经过一次，且在每个状态上作计算所花的时间都是和N 无关的常数。
- 用动态规划解题，如何寻找“子问题”，定义“状态”，“状态转移方程”是什么样的，并没有一定之规，需要具体问题具体分析，题目做多了就会有感觉。甚至，对于同一个问题，分解成子问题的办法可能不止一种，因而“状态”也可以有不同的定义方法。不同的“状态”定义方法可能会导致时间、空间效率上的区别。

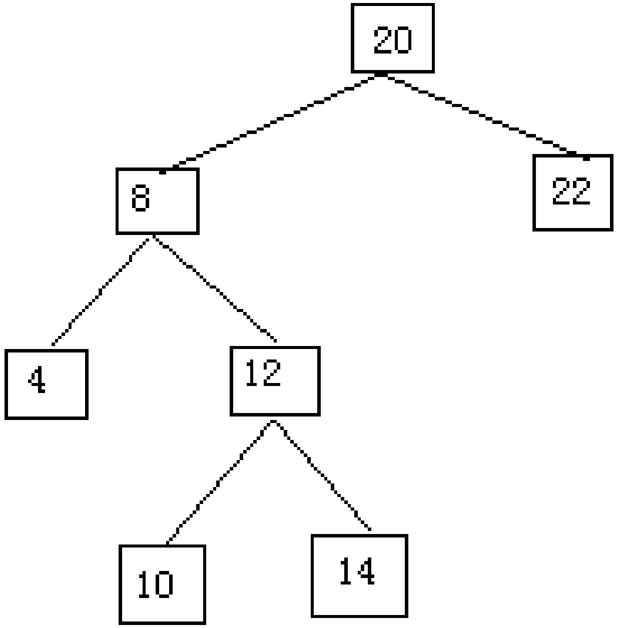
最近公共祖先

- 最近公共祖先(Least Common Ancestors)
- 对于有根树T的两个结点u、v，最近公共祖先LCA(T,u,v)表示一个结点x，满足x是u、v的祖先且x的深度尽可能大。
- 另一种理解方式是把T理解为一个无向无环图，而LCA(T,u,v)即u到v的最短路上深度最小的点。

LCA的例子

- 对于T=<V,E>
- V={1,2,3,4,5}
- E={(1,2),(1,3),(3,4),(3,5)}
- 则有：
- LCA(T,5,2)=1
- LCA(T,3,4)=3
- LCA(T,4,5)=3

如何求解



- 最直接的方法是什么?
- 这种方法有什么不足?

解答

- 从两个给定的结点出发，分别寻找父亲结点，两条回溯路线的交点就是这两个结点的最近公共祖先
- 每次遍历的结点数为两结点的最大高度
- 当树退化为一条线性结构时，每次查询最多将遍历所有结点
- 作业FOJ1031

RMQ问题(Range Minimum Query)

- 如何高效的解决LCA问题?
- 首先介绍RMQ问题。
- RMQ问题是指：对于长度为n的数列A，回答若干询问RMQ(A,i,j)($1\leq i,j\leq n$)，返回数列A中下标在[i,j]里的最小值下标。

RMQ的例子

- 对数列：5,8,1,3,6,4,9,5,7 有：
- RMQ(2,4)=3
- RMQ(6,9)=6

LCA问题向RMQ问题的转化

- 对树进行深度优先遍历，每当“进入”或回溯到某个结点时，将这个结点及其深度存入数组E和D最后一元素。同时数组R记录结点i在数组E中第一次出现的位置(事实上就是进入结点i时记录的位置)，记做R[i]。如果结点E[i]的深度记做D[i]，易见，这时**LCA(T,u,v)**，就等价于求E[RMQ(D,R[u],R[v])]，(R[u]<R[v])。
- 易知，转化后得到的数列长度为树的结点数的两倍减一，所以转化后的RMQ问题与LCA问题的规模同次。

LCA转RMQ例子

- 对于T=<V,E> V={1,2,3,4,5} E={(1,2),(1,3),(3,4),(3,5)}
- 数列E[i]为：1,2,1,3,4,3,5,3,1
- R[i]为：1,2,4,5,7
- D[i]为：0,1,0,1,2,1,2,1,0
- 于是有：
- LCA(T,5,2) = E[RMQ(D,R[2],R[5])] = E[RMQ(D,2,7)] = E[3] = 1
- LCA(T,3,4) = E[RMQ(D,R[3],R[4])] = E[RMQ(D,4,5)] = E[4] = 3
- LCA(T,4,5) = E[RMQ(D,R[4],R[5])] = E[RMQ(D,5,7)] = E[6] = 3

RMQ问题求解

- RMQ问题是求给定区间中的最值问题。
- 最简单的算法是O(n)的，但是对于查询次数很多(如100万次)，O(n)的算法效率不够。
- Sparse Table算法可以在O(nlogn)的预处理以后实现O(1)的查询效率。
- 下面把Sparse Table算法分成预处理和查询两部分来说明(以求最小值为例)。

RMQ预处理

- 预处理使用DP的思想，令f(i, j)表示[i, i+2^j - 1]区间中的最小值，我们可以开辟一个数组专门来保存f(i, j)的值。
- 例如，f(0, 0)表示[0,0]之间的最小值,就是num[0], f(0, 2)表示[0, 3]之间的最小值, f(2, 4)表示[2, 17]之间的最小值。
- 注意, 因为f(i, j)可以由f(i, j - 1)和f(i+2^(j-1), j-1)导出, 而递推的初值(所有的f(i, 0) = i)都是已知的
- 所以我们可以采用自底向上的算法递推地给出所有符合条件的f(i, j)的值。

RMQ查询

- 假设要查询从m到n这一段的最小值, 那么我们先求出一个最大的k, 使得k满足$2^k \le (n - m + 1)$。
- 于是我们就可以把[m, n]分成两个(部分重叠的)长度为2^k的区间: [m, m+2^k-1], [n-2^k+1, n];
- 而我们之前已经求出了f(m, k)为[m, m+2^k-1]的最小值, f(n-2^k+1, k)为[n-2^k+1, n]的最小值。
- 我们只要返回其中更小的那个, 就是我们想要的答案, 这个部分算法的时间复杂度是O(1)的。
- 例如, rmq(0, 11) = min(f(0, 3), f(4, 3)), k=3。

具体代码

```
#include<iostream>
#include<math.h>
using namespace std;
#define MAXN 10000
#define mmin(a, b)   ((num[a]<num[b])?(a):(b))
int num[MAXN] = {1,5,3,2,7,9,3,6,7,0,6,8};
int f1[MAXN][100];

void dump(int n){ //测试输出所有的f(i, j)
    int i, j;
    for(i = 0; i < n; i++){
        for(j = 0; i + (1<<j) - 1 < n; j++){
            printf("f[%d, %d] = %d\t", i, j, f1[i][j]);
        }
        printf("\n");
    }
    for(i = 0; i < n; i++)printf("%d ", num[i]);
    printf("\n");
}
```

```
void st(int n){ //sparse table算法
    int i, j, k, m;
    k = (int) (log((double)n) / log(2.0));
    for(i = 0; i < n; i++) f1[i][0] = i; //递推的初值
    for(j = 1; j <= k; j++){ //自底向上递推
        for(i = 0; i + (1 << j) - 1 < n; i ++){
            m = i + (1 << (j - 1)); //求出中间的那个值
            f1[i][j] = mmin(f1[i][j-1], f1[m][j-1]);
        }
    }
}
int rmq(int i, int j){ //查询i和j之间的最值,注意i是从0开始的
    int k = (int)(log(double(j-i+1)) / log(2.0)), t; //用对2去对数的方法求出k
    t = mmin(f1[i][k], f1[j - (1<<k) + 1][k]);
    printf("rmq(%d, %d) = %d @ pos(%d)\n", i, j, num[t], t);
}
int main(){
    int i, j;
    st(12); //初始化
    dump(12); //测试输出所有f(i, j)
    while(scanf("%d%d", &i, &j) != EOF){
        rmq(i, j);
    }
    return 0;
}
```

LCA问题的Tarjan离线算法

- 利用并查集优越的时空复杂度，我们可以实现LCA问题的O(n+Q)算法，这里Q表示询问的次数。
- Tarjan算法基于深度优先搜索的框架，对于新搜索到的一个结点，首先创建由这个结点构成的集合，再对当前结点的每一个子树进行搜索，每搜索完一棵子树，则可确定子树内的LCA询问都已解决。其他的LCA询问的结果必然在这个子树之外，这时把子树所形成的集合与当前结点的集合合并，并将当前结点设为这个集合的祖先。之后继续搜索下一棵子树，直到当前结点的所有子树搜索完。

- 这时把当前结点也设为已被检查过的，同时可以处理有关当前结点的LCA询问，如果有一个从当前结点到结点v的询问，且v已被检查过，则由于进行的是深度优先搜索，当前结点与v的最近公共祖先一定还没有被检查，而这个最近公共祖先的包涵v的子树一定已经搜索过了，那么这个最近公共祖先一定是v所在集合的祖先。

Tarjan算法的伪代码

```
LCA(u)
{
    Make-Set(u)
    ancestor[Find-Set(u)]=u
    对于u的每一个孩子v
    {
        LCA(v)
        Union(u)
        ancestor[Find-Set(u)]=u
    }
    checked[u]=true
    对于每个(u,v)属于P
    {
        if checked[v]=true
        then {
            回答u和v的最近公共祖先为 ancestor[Find-Set(v)]
        }
    }
}
```

- 由于是基于深度优先搜索的算法，只要调用LCA(root[T])就可以回答所有的提问了，这里root[T]表示树T的根，假设所有询问(u,v)构成集合P。Make-Set，Find-Set，Union是对并查集的操作。
- 作业FOJ1628

最长公共子序列

- 子序列:
  - 设 X=< x1, x2,---, xm>，若有 1≤i1< i2< --- <ik≤m，使得 Z=< z1, z2,---, zk> = < xi1, xi2,---, xik>，则称 Z 是 X 的子序列，记为 Z<X。
  - 如X=<A,B,C,B,D,A,B>，Z=<B,C,B,A>
- 公共子序列:
  - 设 X,Y 是两个序列，且有 Z<X 和 Z<Y，则称 Z 是 X 和 Y 的公共子序列。
- 最长公共子序列:
  - 若 Z<X、Z<Y，且不存在比 Z 更长的 X 和 Y 的公共子序列，则称 Z 是 X 和 Y 的最长公共子序列，记为 Z∈LCS(X , Y)。最长公共子序列往往不止一个。
  - 如X=<A,B,C,B,D,A,B>，Y=<B,D,C,A,B,A>则Z=<B,C,B,A>，Z’=<B,C,A,B>， Z”=<B,D,A,B>，均属于 LCS(X , Y)，即 X,Y 有 3 个 LCS。

如何找出 一个最长公共子序列?

- Brute-force 法:
  - 列出 X 的所有长度不超过 n(即Y)的子序列,
  - 从长到短逐一进行检查,看其是否为 Y 的子序列,
  - 直到找到第一个最长公共子序列。
- 由于 X 共有 2^m 个子序列,故此方法对较大的 m 没有实用价值。

- 是否能使用动态规划法?如何使用?

分析

- 令 Xi=<x1,...,xi>即 X 序列的前 i 个字符 $(1 \le i \le m)$(前缀)
- Yj=<y1,...,yj>即Y序列的前j个字符 $(1 \le j \le n)$(前缀)
- 假定 Z=<z1,...,zk>∈LCS(X , Y)。
- 若 xm=yn(最后一个字符相同)，则不难用反证法证明：该字符必是 X 与 Y 的任一最长公共子序列 Z(设长度为 k)的最后一个字符，即有 zk = xm = yn 且显然有 Zk-1∈LCS(Xm-1 , Yn-1)
- 即 Z 的前缀 Zk-1 是 Xm-1 与 Yn-1 的最长公共子序列。

- 若 xm≠yn，则亦不难用反证法证明：
- 要么 Z∈LCS(Xm-1, Y)，要么 Z∈LCS(X , Yn-1)。
- 由于 zk≠xm 与 zk≠yn 其中至少有一个必成立,若 zk≠xm 则有 Z∈LCS(Xm-1 , Y)。类似的，若 zk≠yn 则有 Z∈LCS(X , Yn-1)。
- ∴若 xm=yn,则问题化归成求Xm-1与Yn-1的 LCS，LCS(X , Y)的长度等于 LCS(Xm-1 , Yn-1)的长度加 1
- 若 xm≠yn 则问题化归成求Xm-1与Y的 LCS及X与Yn-1的 LCS，LCS(X , Y)的长度为：max{LCS(Xm-1 , Y)的长度, LCS(X , Yn-1)的长度}

- 求 LCS(Xm-1,Y)的长度与 LCS(X,Yn-1)的长度这两个问题不是相互独立的:
- ∵两者都需要求 LCS(Xm-1,Yn-1)的长度，因而具有重叠性。另外两个序列的 LCS 中包含了两个序列的前缀的 LCS，故问题具有最优子结构性质 ⇒ 考虑用动态规划法。
- 引进一个二维数组 C，用 C[i,j]记录 Xi 与 Yj 的 LCS 的长度，如果我们是自底向上进行递推计算,那么在计算 C[i,j]之前，C[i-1,j-1], C[i-1,j]与 C[i,j-1]均已计算出来。
- 此时我们根据 X[i]=Y[j]还是 X[i]≠Y[j],就可以计算出C[i,j]。若 X[i]=Y[j]，则执行 C[i,j]←C[i-1,j-1]+1；若 X[i]≠Y[j]，则根据：
    - 若C[i-1,j]≥C[i,j-1]，则C[i,j]取C[i-1,j]；否则C[i,j]取C[i,j-1]。

- 容易得到下面的递归公式：

- c[i,j]=0 如果i=0或者j=0
- c[i,j]=c[i-1,j-1]+1 如果xi=yj
- c[i,j]=min(c[i,j-1], c[i-1,j]) 如果xi!=yj

- 作业FOJ1160

最长递增子序列

- 设L=<a1,a2,…,an>是n个不同的实数的序列，L的递增子序列是这样一个子序列Lin=<ak1,ak2,…,akm>，其中k1<k2<…<km且ak1<ak2<…<akm。求最大的m值。
- 转化为LCS问题求解
- 设序列X=<b1,b2,…,bn>是对序列L=<a1,a2,…,an>按递增排好序的序列。那么显然X与L的最长公共子序列即为L的最长递增子序列。
- 算法的时间复杂度为O(nlogn)＋O(n^2)＝O(n^2)

动态规划的思路

- 设f(i)表示L中以ai为末元素的最长递增子序列的长度。则有如下的递推方程：
- **f(i)=max(max(f(j)|aj<ai,j<i)+1,1)**
- 这个递推方程的意思是，在求以ai为末元素的最长递增子序列时，找到所有序号在i前面且小于ai的元素aj，即j<i且aj<ai。如果这样的元素存在，那么对所有aj，都有一个以aj为末元素的最长递增子序列的长度f(j)，把其中最大的f(j)选出来，那么f(i)就等于最大的f(j)加上1，即以ai为末元素的最长递增子序列，等于以使f(j)最大的那个aj为末元素的递增子序列最末再加上ai；如果这样的元素不存在，那么ai自身构成一个长度为1的以ai为末元素的递增子序列。

作业FOJ1348

```
public void lis(float[] L)
{
    int n = L.length;
    int[] f = new int[n];//用于存放f(i)值；
    f[0] = 1;//以第a1为末元素的最长递增子序列长度为1；
    for (int i = 1;i < n;i++) { //循环n-1次
        f[i] = 1;//f[i]的最小值为1；
        for (int j = 0;j < i;j++) { //循环i 次
            if (L[j] < L[i] && f[j] > f[i] - 1)
                f[i] = f[j] + 1;//更新f[i]的值。
        }
    }
    System.out.println(f[n-1]);
}
```

改进算法

- 在计算每一个f(i)时，都要找出最大的f(j)(j<i)来，由于f(j)没有顺序，只能顺序查找满足aj<ai最大的f(j)，如果能将让f(j)有序，就可以使用二分查找，这样算法的时间复杂度就可能降到O(nlogn)。
- 于是想到用一个数组B来存储子序列的最大递增子序列的最末元素，即有
- B[f(j)] = aj
- 在计算f(i)时，在数组B中用二分查找法找到满足j<i且B[f(j)]=aj<ai的最大的j，并将B[f[j]+1]置为ai。
- 算法的时间复杂度为O(nlogn)

作业FOJ1506

```
lis1(float[] L)
{
        int n = L.length;
        float[] B = new float[n+1];//数组B；
        B[0] = -10000;//把B[0]设为最小，假设任何输入都大于-10000；
        B[1] = L[0];//初始时，最大递增子序列长度为1的最末元素为a1
        int Len = 1;//Len为当前最大递增子序列长度，初始化为1；
        int p, r, m;//p,r,m分别为二分查找的上界，下界和中点；
        for (int i = 1;i < n;i++) {
                p = 0;
                r = Len;
                while (p <= r) { //二分查找最末元素小于ai+1的长度最大的最大递增子序列；
                        m = (p + r) / 2;
                        if (B[m] < L[i]) p = m + 1;
                        else r = m - 1;
                }
                B[p] = L[i];//将长度为p的最大递增子序列的当前最末元素置为ai+1;
                if (p > Len) Len++;//更新当前最大递增子序列长度；


        }
        System.out.println(Len);
}
```

证明

- 命题1：每一次循环结束数组B中元素总是按递增顺序排列的。
- 证明：用数学归纳法，对循环次数 i 进行归纳。
- 当i=0时，即程序还没进入循环时，命题显然成立。
- 设i<k时命题成立，当i=k时，假设存在j1<j2，B[j1]>B[j2]，因为第i次循环之前数组B是递增的，因此第i次循环时B[j1]或B[j2]必有一个更新，假设B[j1]被更新为元素ai，由于ai=B[j1]> B[j2]，按算法ai应更新B[j2]才对，因此产生矛盾；假设B[j2]被更新，设更新前的元素为s，更新后的元素为ai，则由算法可知第i次循环前有B[j2]＝s< ai< B[j1]，这与归纳假设(当i<k时，j1<j2,B[j1]<B[j2]=s)矛盾。
- 命题得证。

- 命题2：B[c]中存储的元素是当前所有最长递增子序列长度为c的序列中，最小的最末元素，即设当前循环次数为i，有
- $B[c]=\{aj \mid f(k)=f(j)=c \wedge k,j \le i+1 \rightarrow aj \le ak\}$
- (f(i)为与上文中的f(i)含义相同)。
- 证明：程序中每次用元素ai更新B[c]时(c=f(i))，设B[c]原来的值为s，则必有ai<s，不然ai就能接在s的后面形成长度为c+1的最长递增子序列，而更新B[c+1]而不是B[c]了。所有B[c]中存放的总是当前长度为 c 的最长递增子序列中，最小的最末元素。

- 命题3：设第i次循环后得到的p为p(i)，那么p(i)为以元素ai为最末元素的最长递增子序列的长度。
- 证明：只须证p(i)等于改进算法中的f(i)。
- 显然一定有p(i)≤f(i)。假设p(i)<f(i)，那么有两种情况，第一种情况是由二分查找法找到的p(i)不是数组B中能让ai接在后面成为新的最长递增子序列的最大的元素，由命题1和二分查找的方法可知，这是不可能的；第二种情况是能让ai接在后面形成长于p(i)的最长递增子序列的元素不在数组B中，由命题2可知，这是不可能的，因为B[c]中存放的是最末元素最小的长度为c的最长递增子序列的最末元素，若ai能接在长度为L(L> p(i))的最长递增子序列后面，就应该能接在B[L]后面，那么就应该有p(i)=L，与L> p(i)矛盾。因此一定有p(i)＝f(i)，命题得证。

作业

- 作业FOJ1031
- 作业FOJ1628
- 作业FOJ1160
- 作业FOJ1348
- 作业FOJ1506