LIS 最长上升子序列详解

一. 摘要

关于LIS部分，重点讲一下LIS的概念定义和理解，以及求LIS的两种方法，分别是$O(n^2)$的动态规划方法，$O(nlogn)$的二分+贪心法，最后附上几道非常经典的LIS的例题。

二. LIS的定义

最长上升子序列（$Longest\ Increasing\ Subsequence$），简称$LIS$。有些情况求的是**最长不下降子序列**，二者区别就是序列中是否可以有相等的数。

假设我们有一个序列b数组，当$b1 < b2 < ... < bN$的时候，我们称这个序列是上升的。

对于给定的一个序列$a1, a2, ..., aN$，我们也可以从中得到一些上升的子序列$a[i1], a[i2], ..., a[iK]$，这里$1 <= i1 < i2 < ... < ik <= N$，但必须按照从前到后的顺序。比如，对于序列$1, 7, 3, 5, 9, 4, 8$，我们就会得到一些上升的子序列，如$(1, 7, 9), (3, 4, 8), (1, 3, 5, 8)$等等，而这些子序列中最长的（如子序列$1, 3, 5, 8$），它的长度为$4$，因此该序列的最长上升子序列长度为$4$。

略有点好抽象，详细地解释下，如下：

首先需要知道，**子串**和**子序列**的概念，我们以**字符串子串**和**字符串子序列**为例，更为形象，也能顺带着理解字符串的子串和子序列

（1）字符串子串指的是字符串中**连续**的n个字符，如$abcdefg$中，ab，cde，fg，$abcdefg$等都属于它的字串。

（2）字符子序列指的是字符串中**不一定连续但先后顺序一致**的n个字符，即不可改变其前后顺序。如$abcdefg$中，$acdg$，bdf属于它的子序列，而bac，$dbfg$则不是，因为它们与字符串的字符顺序不一致。

知道了这个，数组的子序列就很好明白了。这样的话，最长上升子序列也很容易明白了。

归根结底还是子序列，然后子序列中，按照上升顺序排列的最长的就是我们最长上升子序列了

还有一个非常重要的问题：请大家用集合的观点来理解这些概念，子序列、公共子序列以及最长公共子序列都不唯一，但很显然，**对于固定的数组，虽然LIS序列不一定唯一，但LIS的长度是唯一的。**再拿我们刚刚举的栗子来讲，给出序列$1, 7, 3, 5, 9, 4, 8$，易得最长上升子序列长度为4，这是确定的，但序列可以为$(1, 3, 5, 8)$，也可以为$(1, 3, 5, 9)$。

三. LIS的求解方法

这里详细介绍一下求LIS的两种方法，分别是$O(n^2)$的动态规划，$O(nlogn)$的二分+贪心法

解法1：动态规划DP：

我们都知道，动态规划的一个特点就是当前解可以由上一个阶段的解推出【无后效性】， 由此，把我们要求的问题简化成一个更小的子问题。子问题具有相同的求解方式，只不过是规模小了而已。

最长上升子序列就符合这一特性。我们要求n个数的最长上升子序列，可以求前$n-1$个数的最长上升子序列，再跟第n个数进行判断。
求前$n-1$个数的最长上升子序列，可以通过求前$n-2$个数的最长上升子序列……直到求前1个数的最长上升子序列，此时LIS当然为1。

让我们举个例子：求 $2,7,1,5,6,4,3,8,9$ 的最长上升子序列。我们定义dp数组$i \in [1,n]$来表示**前i个数以当前元素结尾的最长上升子序列长度。**

前1个数，2前面没有数字，$dp[1]=1$，当前的最长上升子序列为2；

前2个数，7前面有2，小于7，$dp[2]=dp[1]+1=2$， 当前的最长上升子序列为$2,7$

前3个数，在1前面没有比1更小的，1自身组成长度为1的子序列 $dp[3]=1$，子序列为1

前4个数，在5前面有2小于5，$dp[4]=dp[1]+1=2$，子序列为$2,5$
在5前面有1小于5，$dp[4]=dp[3]+1=2$，子序列为$2,5$

前5个数，6前面有$2,1,5$，均小于6，其中最大的是$dp[5]=dp[4]+1=3$，子序列为$2,5,6$

前6个数， 4前面有$2,1$，均小于4， 其中最大的是$dp[6]=dp[1]+1=2$，子序列为$2,4$

前7个数，3前面有$2,1$，均小于3， 其中最大的是$dp[7]=dp[1]+1=2$，子序列为$2,3$

前8个数， 8前面有$2,7,1,5,6,4,3$，均小于8，其中最大的$dp[8]=dp[5]+1=4$，子序列为$2,5,6,8$

前9个数， 9前面有$2,7,1,5,6,4,3,8$，均小于9，其中最大的$dp[9]=dp[8]+1=5$，子序列为
$2,5,6,8,9$

将dp数组中求出其中的最大值，我们可以看出这9个数的LIS为$dp[9]=5$

执行策略：

1.输入原始数组a数组 和 存放以各个元素结尾的LIS长度dp数组 及相关变量

2.初始化dp数组，全部赋值为1

3.双重循环，第一层循环遍历每一个元素i，第二层循环遍历i元素前面的元素j

4.如果$a[i]>a[j]$，则表明当前元素可以拼接到第j个元素后面，即$dp[i]=dp[j]+1$
但此时不是100%更新，$dp[j]$非常小可能还不如原来的$dp[i]$,需要和$dp[i]$原始的值去做比较，需要更新取最大。
如果$a[i]<a[j]$, 则表明当前元素不可以拼接到第j个元素后面，即$dp[i]$保持不变，那就不用写了

5.最后求dp数组中的最大值即可

总结一下，$dp[i]$**就是以$a[i]$结尾的，找$a[i]$之前的并且比a[i]要小的元素a[x]，在其最长上升子序列的基础上**
$dp[x]+1$
当$a[i]$之前没有比$a[i]$更小的数时，$dp[i]=1$。
所有的$dp[i]$里面最大的就是最长上升子序列。

```cpp
#include<bits/stdc++.h>
using namespace std;
int a[5005] , n , dp[5005];
int main(){

    cin >> n;
    for(int i=1;i<=n;i++){
        cin >> a[i];
        dp[i]=1;
    }
    // dp
    for(int i=1;i<=n;i++){
        for(int j=1;j<i;j++){
            if(a[i]>a[j]){ // 当前元素大于前面的元素，即可以进行拼接
                dp[i] = max(dp[i] , dp[j]+1);
            }
        }
    }
    int mmax = 0;
    for(int i=1;i<=n;i++) mmax = max(mmax , f[i]);
    cout << mmax;

    return 0;
}

```

这个算法的时间复杂度为$O(n^2)$，并不是最优的算法。在限制条件苛刻的情况下，这种方法行不通。

解法2：贪心+二分

首先，为什么会有贪心的思路呢?
大家可以思考一下怎么样把LIS的长度扩大?
对于一个上升子序列，显然其结尾元素越小，越有利于在后面接其他的元素，也就越可能变得更长。
所以我们需要尽量把构成LIS选择的数据尽可能的降低
例如 $1-2-3$ 和 $9-10-11$，它们的LIS均为3，但是很明显前一个更具备可操作性，更有可扩展性

那如何实现呢?

新建一个 b 数组，b 数组**存放目前构成最长上升子序列的元素，但注意这是可能存在的序列！！！，不一定是正确的最长上升子序列**
只需要维护 b 数组，需要先取变量 a 数组
如果 $a[i]$ $>$当前 b 数组最后一个元素【也是构成LIS下的最大值】，就把 $a[i]$ 接到当前 b 数组后面
否则，就用 $a[i]$ 取更新 b 数组。具体方法是，在 b 数组中找到第一个**大于等于**$a[i]$ 的元素 $b[j]$，用 $a[i]$ 去更新 $b[j]$

以下序列 a 数组= $3, 1, 2, 6, 4, 5, 10, 7$为例，求LIS长度。

定义一个 b 数组来储存可能的排序序列，len为LIS长度。我们依次把 $a[i]$ 有序地放进 b 数组 里，i的范围就从1 n表示第i个数。

$a[1] = 3$，因为是第一个元素，无脑把3放进$b[1]$，此时 b 数组为3，此时$len = 1$，末尾是3

$a[2] = 1$，因为 1 比 3 小，所以可以把$b[1]$中的3替换为1，此时$b[1] = 1$，此时$len = 1$，最小末尾是1

$a[3] = 2$，因为 2 大于 1 ，就把 2 放进去，len++ , $b[2] = 2$，此时b 数组为$1, 2$ ，$len = 2$

$a[4] = 6$，因为 6 大于 2，就把 6 放进去，len++ , $b[3] = 6$，此时b 数组为$1, 2, 6$ ，$len = 3$

$a[5] = 4$，4 在 2 和 6 之间，比 6 小，可以把 6 替换为 4 ，此时b 数组为$1, 2, 4$，$len = 3$

$a[6] = 5$，因为 5 大于 4，就把 5 放进去，len++ , $b[4] = 5$，此时b 数组为$1, 2, 4, 5$ ，$len = 4$

$a[7] = 10$，因为 10 大于 5，就把 10 放进去，len++ , $b[5] = 10$，此时b 数组为$1, 2, 4, 5, 10$ ，$len = 5$

$a[8] = 7$，7 在 5 和 10 之间，比 10 小，可以把 10 替换为7，此时 b 数组为$1, 2, 4, 5, 7$，$len = 5$

注意1：

最终我们得出LIS长度为 5 ，**但是，但是！！！b 数组中的序列并不一定是正确的最长上升子序列。**
在这个例子中，我们得到的1 2 4 5 7 恰好是正确的最长上升子序列
下面我们再举一个例子：有以下序列 $a = 1, 4, 7, 2, 5, 9, 10, 3$ ，求LIS长度。

大家可以去推导一下
最终LIS答案是 5 ，b 数组更新最后为$1, 2, 3, 9, 10$

但是！！这里的$1, 2, 3, 9, 10$很明显不是正确的最长上升子序列。
因此，b 序列并不一定表示最长上升子序列，它**只表示达到最长子序列长度下的排好序的最小序列。**
这有什么用呢？我们最后一步 3 替换 5 并没有增加最长子序列的长度，但很明显降低了数值大小，更有利于后续数字的加入，在于记录最小序列，代表了一种“最可能性”，只是此种算法为计算LIS而进行的一种替换。

注意2：

如果 $a[i] >$当前 b 数组最后一个元素【也是构成LIS下的最大值】，就把 $a[i]$ 接到当前 b 数组后面
否则，就用 $a[i]$ 取更新 b 数组。具体方法是，在 b 数组中找到第一个大于等于$a[i]$ 的元素 $b[j]$，用 $a[i]$ 去更新 $b[j]$

这里为什么是**大于等于**呢？

举个例子，b数组目前为$1, 4, 7$，如果下一个遍历的元素是 4 ，如果是大于，则替换完变成$1, 4, 4$，很明显不是LIS，所以只能是替换掉4，这样还是$1, 4, 7$，仍然能保持LIS，所以是大于等于

优化：

但是如果从头到尾扫一遍 b 数组的话，时间复杂度仍是$O(n^2)$。
我们注意到 b 数组内部一定是单调不降的，是有序的，不需要移动，只需要替换元素即可。所以我们可以利用二分查找插入的位置，找出第一个大于等于 $a[i]$ 的元素。
二分 b 数组的时间复杂度的 $O(lgn)$ ，求LIS长度的算法复杂度降为了 $O(nlogn)$

可以使用二分STL模板中的$lower_bound$**优化最长上升子序列**，可以快速找到第一个大于等于目标数字的地址

$$lower\_bound(\text{数组名称} + \text{起点下标}, \text{数组名称} + \text{终点下标，目标值})$$

但是得到的是地址，需要通过 地址 - 地址 得到对应的下标，即：

$$int \quad t = lower\_bound(\text{数组名称} + \text{起点下标}, \text{数组名称} + \text{终点下标，目标值}) - \text{数组名称};$$

得到下标，直接替换即可，$b[t] = a[i]$

代码：

```
#include<bits/stdc++.h> // 万能头文件
using namespace std;
#define int long long
int n , a[100005] , b[100005];//b数组 储存单调递增可能构成的数列
signed main(){

    cin >> n;
    for(int i=1;i<=n;i++) cin >> a[i];
    // a[1]是第一个，必然保存，先存到b数组中
    b[1] = a[1];
    int id = 1;
    for(int i=2;i<=n;i++){
        if( a[i] > b[id] ){
            id++;
            b[id] = a[i];
        }else{
            // a[i]小于当前b数组中的最大值
            // 替换掉 b数组中第一个大于等于a[i]的数字
            // 降低 数字的大小，更有益提升递增单调递增的长度
            // lower_bound函数的应用，注意减去的b是地址。地址 - 地址 = 下标。
            int t = lower_bound( b+1 , b+id+1 , a[i] ) - b;
            b[t] = a[i];
        }
    }
    cout << id;
    return 0;
}

```

四. Dilworth狄尔沃斯定理 ----- 记住即可

不上升子序列的个数等于最长上升子序列的长度。
不下降子序列的个数等于最长下降子序列的长度。