论文编号：

基于插入排序的优化排序算法

张诚睿

1.同济大学 软件学院，上海 202212

摘要：插入排序是一种简单的排序算法，在此基础上，前人引入了折半搜索、希尔排序等方式提高插入排序的效率，但这些优化并未使排序的平均时间复杂度有稳定的数量级上的降低。本文研究了基于链表结构的二分查找算法，引入了跳表的逻辑结构，同时提高了插入排序的比较与交换效率，使得在略微增加空间开销的情况下总时间复杂度有稳定的数量级级别的优化。

关键词: 插入排序；算法；链表

Optimal sorting algorithm based on insertion sorting

Chengrui Zhang

1.Tongji University School of Software, Shanghai 202212

Abstract: Insertion sort is a simple sort algorithm. On this basis, predecessors introduced half search, Hill sort and other methods to improve the efficiency of insertion sort, but these optimizations did not reduce the average time complexity of the sort by a stable order of magnitude. This paper studies the binary search algorithm based on linked list structure, introduces the logical structure of jump table, and improves the comparison and exchange efficiency of insertion sorting, so that the total time complexity has a stable order of magnitude optimization when the space overhead is slightly increased.

Key Words: Insertion sort; Algorithm; Linked list

1 引 言

排序是一种常见的的数据处理方式，针对于此的算法也有许多不同的设计，它们各有优劣。众多排序算法之中，插入排序是一种较为直观的算法[1]，它易于理解且具有稳定性，但时间效率一般。因此，提高插入排序的时间效率非常重要。

常见的两种优化方案是折半插入法和希尔排序法，折半插入法优化了搜索比较的过程，希尔排序法使情况趋于最好待排序状态，两者都一定程度上降低了时间复杂度，但折半插入法未能在数量级上对时间效率做出优化，希尔排序法破坏了数据稳定性[2]。

为了在插入排序的基础上将时间复杂度降低到O(nlogn)且保持排序稳定性，需要引入类似跳表的数据结构。这样的做法增加了空间开销，但规模不大，相比于时间效率上的提高，是可接受的。

本文在分析插入排序的优化过程后，提出了一种全新的排序方式：Sprung 排序。该排序算法利用跳表结构，满足快速查找与插入元素的需求，大大提高了算法效率，并且让算法的时间复杂度稳定在 O(nlogn)。Sprung 算法还能尽可能地避免出现极端情况，对待排序数据也没有像基数排序那样的特殊要求[3]，是一种非常稳定的且广泛适用的高效排序方式。

2 插入排序

2.1 插入排序流程

插入排序清晰直观，算法的整个过程可拆分为三个部分。

第一个部分为初始状态，序列中每个元素均为待排序状态。此时选中第一个元素，认为它已经是有序的，剩余元素仍处于待排序状态。序列状态如图 1 所示。

图 1 插入排序初始状态

第二个部分为中间状态，序列中部分元素已经被排好序。插入排序每次会选取一个待排序元素插入到已有序序列中，形成新的有序序列，因此序列中前半部分是已有序序列，后半部分为待排序序列。序列状态如图 2 所示。

图 2 插入排序中间状态

第三个部分为完成状态，主程序经过 n-1 次插入排序流程后，序列中所有元素均被加入到已有序序列中，整个序列按关键码大小完成排序。序列状态如图 3 所示。

图 3 插入排序完成状态

2.2 中间状态操作

在插入排序中间状态每次进行插入操作时，都需

要挑选待排序序列的第一个元素作为插入对象，与已有序序列的每个元素从后向前逐个比较，直到自身比前一个元素大时才停止，否则一直交换到序列的第一位。

以图 4 待排序序列为例，进行插入排序。

图 4 待排序序列

开始排序时，认为首个元素已经是有序的，以绿色标出，后续序列是待排序的，用橙色标出。此时选中首个待排序元素（关键码 4），将它与已有序序列最后一个元素比较，如图 5。

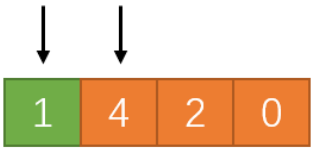

图 5 第一次插入排序

关键码 4 大于关键码 1，因此不需要交换，首次插入排序已完成，如图 6。

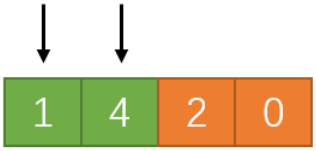

图 6 第一次插入排序完成

再次挑选待排序序列的第一个元素（关键码 2），在已有序序列中从后向前比较，不断交换位置直到找到插入位置时停止，如图 7。

图 7.1 第二次插入排序初始指针

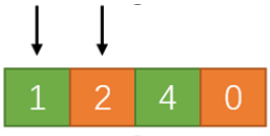

图 7.2 第二次插入排序交换过程

图 7.3 第二次插入排序完成状态

重复上述过程直到所有元素都在已有序序列中（图 8），排序完成。

图 8 排序结果

2.3 插入排序的效率

从插入排序的过程可以看出，核心步骤有两个：交换和比较。每做一次比较就会根据结果进行一次数据交换，在最坏情况下每次比较后都需要交换数据，这样总的关键码比较次数 KCN 与元素交换次数 RMN 分别为：

$$KCN=\sum_{i=1}^{n-1}\frac{n(n-1)}{2}\approx\frac{n^2}{2}$$

$$RMN=\sum_{i=1}^{n-1}(i+2)=\frac{(n+4)(n-1)}{2}\approx\frac{n^2}{2}$$

从上面的分析可知，插入排序的运行时间和待排序元素的初始排列顺序密切相关。若待排序元素序列中出现各种可能排列的概率相同，则可取上述最好情况和最坏情况的平均情况。在平均情况下的关键码比较次数和元素交换次数约为$\frac{n^2}{4}$，因此，插入排序的时间复杂度为 $O(n^2)$。

由插入排序的过程分析，也可直观的了解到插入排序是具有稳定性的。

3 折半插入排序

对插入排序的效率进行分析后得知，降低比较次数或排序次数的数量级是提高总体算法效率的核心。针对已有序序列的部分，可以使用折半搜索（二分查找）的方式加快寻找插入位置这一步骤，将比较次数大大降低[4]。

3.1 二分查找的实现

二分查找对查找范围有两条要求，一是序列有序排列，而是序列采用线性存储结构。对于插入排序的已有序序列完全符合二分查找的前提要求，因此可以用二分查找法提高查找效率。

插入排序算法中整个序列是数组存储的，使用二分查找需要给定数组的起末位置 begin 和 end，寻找中间位置 middle=(begin+end)/2，如图 9，将序列分成两个子序列。假设数组从小到大排列，则若中间位置关键码小于查找元素关键码，向右侧 middle+1 至 end 区间继续寻找，反之向左侧 begin 至 middle-1 区间寻找，直至找到相等关键码或区间 begin 大于 end 时结束，此时 begin 即为被插入位置。

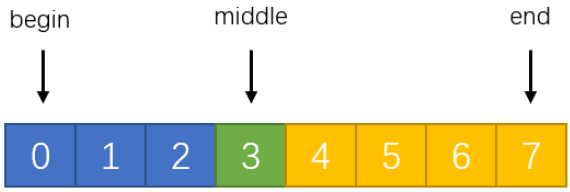

图 9　二分查找子序列划分

找到插入位置后，现将插入位置后的元素块整体向后移动一位，再将操作元素插入当前位置，如图 10。

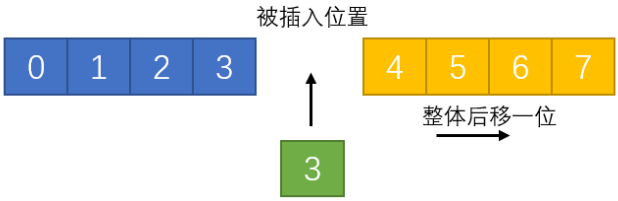

图 10　二分查找插入过程

3.2 折半插入的效率

折半搜索比顺序搜索快，所以折半插入排序的平均性能要好于直接插入排序。它所需要的关键码比较次数与待排序元素序列的初始排列无关，仅依赖于元素个数。根据折半搜索的过程分析可知，在插入第 i 个元素时，需要经过[$\log_2 i$]+1 次关键码的比较，才能确定它应插入的位置。因此，将 n 个元素用折半插入排序所进行的关键码比较次数为：

$$\sum_{i=1}^{n-1}([log_2\, i]+1) \approx n\, log_2\, n$$

4　希尔排序

希尔排序(Shell Sort)又称为缩小增量排序(diminishing-increment sort)，是 1959 年由 D.L.Shell 提出来的[5]。希尔排序利用了插入排序对较有序序列的额外高效特点，通过缩小增量的方法使总过程的时间复杂度接近于插入排序的最优情况。

4.1 希尔排序的实现

希尔排序的基本思想是对距离为 gap 的元素子序列进行插入排序，完成后再缩小 gap 值，对新的子序列进行插入排序，直到 gap 等于 1，对整个序列进行一遍插入排序。

在首次设定 gap 时，常见的方式是选取总长度 n/2 的值，如图 11。

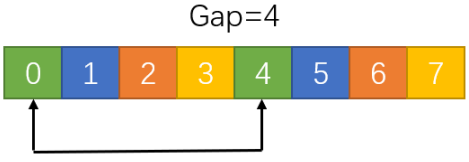

图 11 希尔排序首次分组

对各组分别进行组内插入排序后，将 gap 再次缩小为 gap/2，使子序列增长，如图 12。

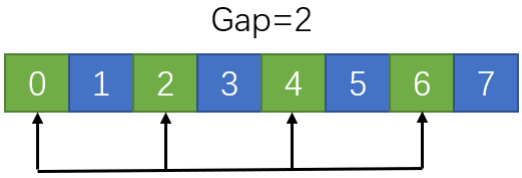

图 12 希尔排序第二次分组

对各组分别进行组内插入排序，重复 gap 缩小的步骤，直到 gap 等于 1 时，全序列都在同一组内，对整个序列进行一次插入排序，如图 13。

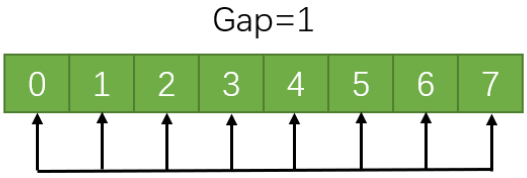

图 13 希尔排序最后一次分组

此时，整个序列都是有序的，完成了排序过程。

4.2 希尔排序的效率

希尔排序并未改变插入排序的比较方式和交换方式，而是利用插入排序对较有序序列额外高效的特点提高了算法的时间效率。插入排序在最好情况下时间复杂度为 O(n)，希尔排序利用分组让每次排序开始时序列情况尽可能地接近最优状态。

对增量 gap 的取值有很多不同方案，不同的取值方法对不同序列的排序性能影响很大，因此对希尔排序的时间复杂度还没有人可以给出严谨的数学证明，只有在特定情况下或大量实验统计才能大致判断希尔排序的时间效率。在 Knuth 所著的《计算机程序设计技巧》第 3 卷中，他用大量实验统计资料得出在数据量很大时，关键码平均比较次数和元素平均移动次数约在 $n^{1.25}$~$1.6n^{1.25}$。我们普遍认为希尔排序的时间复杂度下限为 O(nlogn)，但实际上若序列原有顺序不利于排序时，希尔排序的时间复杂度容易接近 $O(n^2)$，这对于追求稳定时间效率的开发者来说并不是可接受的。

同时，分组排序的方式还打乱了元素的稳定性，使得希尔排序不再具有插入排序原有的稳定性，在需要相等数据保持前后关系的情境下，希尔排序便不再适用。

5　Sprung 排序

折半插入排序和希尔排序都利用了插入排序自身的特点做出了优化，但折半插入排序没有对时间复杂度做出数量级级别的优化，希尔排序将时间复杂度下限降低到 O(nlogn)，但不能保证平均时间复杂度可以稳定在 O(nlogn)。同时希尔排序不具有稳定性，在使用上受到限制。

为了能够将插入排序时间复杂度优化至 O(nlogn)，

并且保持排序的稳定性，下面将提出基于链表结构的 Sprung 排序算法。

5.1 二分查找链表

链表可以高效插入新结点，这种结构本身就是对插入排序的交换过程的优化，对每次插入结点的操作时间复杂度均为 O(1)，是最佳的优化结果。但链表不便于查找插入位置，因此将链表优化为二分查找是提高总过程时间效率的关键。

若有如图 14 的逻辑搜索结构，可以实现类似二分查找的方式，从顶层开始寻找，逐层下降至最后一层。每一层结点个数是下一层的一半，因此设待搜索序列的链表结点有 n 个，则该搜索结构的时间复杂度为 O(logn)。

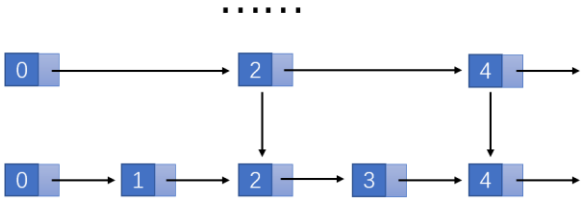

图 14 二分查找链表搜索逻辑结构

为了简化操作，现将链表中的结点（以关键码为 int 类型为例）设计结构如表 1。

表 1 链表结点数据结构

int key	Node* down	Node* lnk
关键码	向下链接指针	平行链接指针

5.2 跳表结构

最底层的链表是完整的数据链表，每个数据之间用平行链接指针 lnk 相连，向下链接指针 down 为空。上层链表每层数量依次减少 1/2，一层之间用平行链接指针 lnk 相连，邻近层用向下链接指针 down 相连，它指向与自身同关键码的低层结点。两个指针域结合可以让跳表满足平行访问和逐层访问两个需求，方便查找特定元素，如图 15 所示。

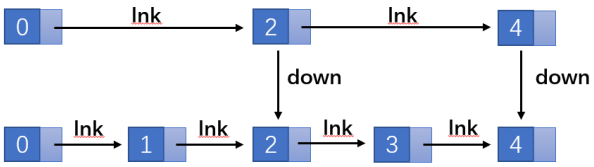

图 15 跳表指针结构

5.3 跳表搜索

搜索值为 target 的元素时，从最高层的首个结点开始，沿平行链接指针 lnk 向后搜索，当下一节点值 k 小于等于 target 时，访问当前结点的 down，在下一层进行搜索，直到在最后一层找到 target。若找不到则停留在 target 插入位置的前一个节点处。

以搜索 target=3 为例，搜索过程如图 16 中的橙色标注。

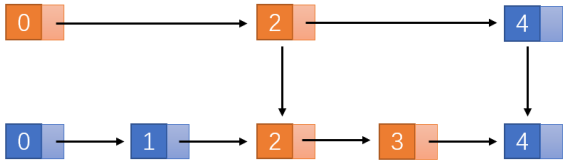

图 16 跳表搜索过程

5.4 跳表插入

理想的跳表每一层节点数量是低一层节点数量的一半，搜索时间复杂度为 O(logn)。在插入新节点时，首先需要在最底层链表插入，此时若不在高层链表中插入，则会导致跳表的理想状态遭到破坏。随着插入结点越来越多，跳表的搜索性能大幅下降，在极端情况下（图 17），甚至可能导致跳表结构趋近于单链表。这会导致我们所做出的优化趋于无效，因此，跳表插入新结点时必须考虑是否在高层链表中同时插入。

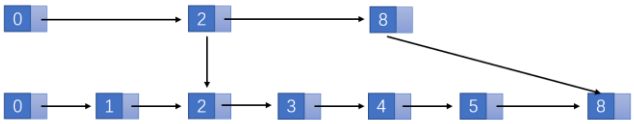

图 17 趋于极端情况的跳表

如果每次插入新结点时都要调整整个跳表为理想跳表，可以预想到这会大量破坏高层链表的结构，整个调整操作几乎会重塑高层索引结构，涉及的结点增删也会很多，这不利于在插入排序频繁插入新结点的情况下提高效率。因此，我们需要放弃绝对理想的跳表，改用较理想的跳表结构。

较理想的跳表结构应当使得每一层的结点数量的是低一层结点数量的一半左右，在强求理想跳表效率极低的情况下，我们只要求跳表大致满足理想跳表的结构，这样既可以让插入操作拥有极低的时间开销，又可以维持高效的搜索结构，每层的结点数量如表 2 所示。

表 2 跳表每层结点数量设计

层高	理想结点数量	真实结点数量
n	$log_2 n$	$log_2 n + k$
3	0.25n	0.25n+k
2	0.5n	0.5n+k
1	n	n

为了让每层结点数量基本维持减半的特点，在插入新结点时，应自底向上判定是否插入。判断的方法随机判断，自底向上添加一层的概率为 0.5，添加两层的概率为 0.25，添加三层的概率为 0.125，以此类推。可以用如下函数来返回添加层数。

```
int LevelRan()
{
    for (int i=0;i<Max_Height;i++)
    {
```

```
            int judge=rand()%2;
            if (judge)
                return i;
        }
        return Max_Height-1;
    }
```

每次添加新结点时都要先调用上述函数获取添加层数，若添加层数为 0，则只在最底层插入，若添加层数不为 0，则按层高向上插入。如图 18，添加了关键码为 6 的结点，随机层高为 1，可见插入后跳表基本特征不变，对其查找性能影响不大。

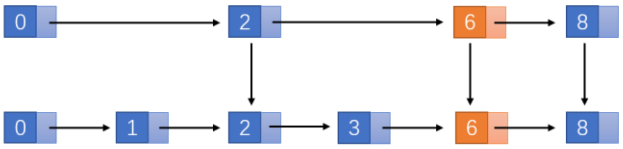

图 18 跳表插入样例

设底层链表有 n 个结点，则生成随机层高函数的平均时间复杂度为 O(logn)，插入操作平均时间复杂度也为 O(logn)，插入后的跳表搜索操作平均时间复杂度保持在 O(logn)。因此，采用随机层高的方式插入跳表是合理高效的。

5.5 基于插入排序的跳表使用

插入排序开始时需要选定首个结点，以它为基准建立一个跳表，排序过程中需要不断对跳表进行元素搜索与插入操作，不断动态调整跳表的结构。完成排序后，高层结点可以删除，排序的结果在最底层链表中，只需遍历一次即可输出。

5.6 建立跳表

在管理跳表的类中应当存在私有成员 Node* head 记录首个结点，以此为访问跳表的入口。

首个结点应当拥有最高的层数，所以需要首先根据待排序序列的长度 n 确定最高层数为[log2n]，然后再建立首个结点。

首次建立跳表的结构如图 19。

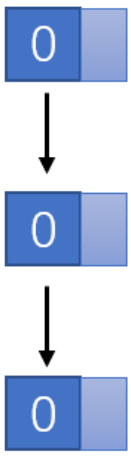

图 19 最大层高为 3 的初始跳表

从图 19 中可以看出，插入排序过程中首个结点应当拥有最高的层数来保证对每一层的索引。如果排序过程中不断有新的元素被插入到首个结点的位置，会导致过多结点有很大的层高，这打破了理想跳表的结构，所以为了使跳表总体结构稳定，首个结点应该是整个序列中关键码最小的元素，以此在整个排序过程中可以保持不动。

为了保证首个结点是全待排序序列中最小的元素，首先要遍历整个序列，挑选出最小值，将它与第一个元素交换，将其设定为首个结点初始化跳表，然后开始基于跳表的插入排序。

这一步的伪代码过程如下：

```
for (int i=0;i<n;i++)
{
    /*找出最小值，并记录*/
}
/*交换最小值与首个元素*/
/*建立跳表的首个结点，高度为 log2n*/
```

5.7 Sprung 排序过程

首结点建立过程已经在上文阐明，不再赘述，结果如图 20.1（数字为关键码）。

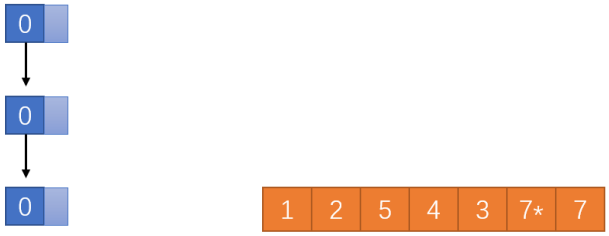

图 20.1 建立首结点状态

拥有首结点后，在待排序序列中选出第一个元素进行插入排序，首先查找其插入位置，插入时随机生成层高为 0，如图 20.2。

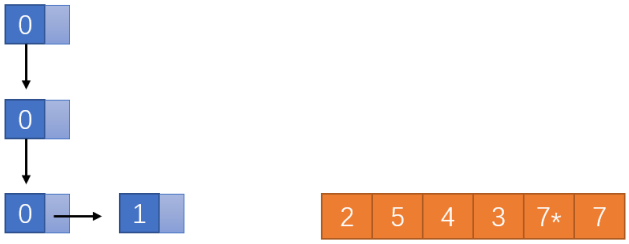

图 20.2 首次插入排序状态

重复上述过程，将 2、5、4、3、7*均插入到跳表中，一种可能的结果如图 20.3。

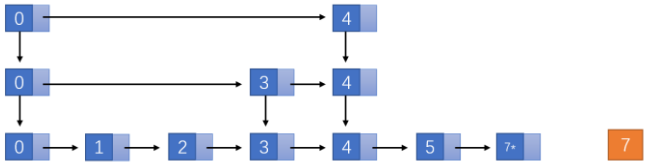

图 20.3 排序中间过程

最后一步将 7 添加至跳表中，最底层链表顺序即为关键码从小到大的顺序。此时若确定不再增加新的待排序序列，可以从顶层开始逐层删除除最底层外的所有结点。

5.8 Sprung 排序伪代码

为了实现应用于 Sprung 算法的跳表结构，首先应设计跳表操作相关的函数，其中包括：

（1）void init(int key, int height)函数，根据传入的最大层高建立首结点。

（2）Node* Find(int key)函数，查找 key 关键码的插入位置，并返回插入位置的前一个结点，在主程序中记录。

（3）int LevelRan()函数，随机生成当前待插入结点的层高，每层高度出现的可能性以 1/2 衰减，不超过 Max_Height 最大层高。

（4）void Insert(Node* pos, int key, int height)函数，在 pos 结点后插入关键码为 key 的新结点，并按照 height 逐层插入。

```
void Sprung(int* array, int n, Node* head)
{
    for (int i=0;i<n;i++)
    {
        /* 找出最小值，并记录 */
    }
    /* 交换最小值与首个元素 */
    /* 建立跳表的首个结点，高度为 log2n */
    for (int i=1;i<n;i++)
    {
        /* 记录 array[i]的值 */
        /* 在跳表中查找插入位置 */
        /* 随机层高 height=LevelRan() */
        /* 根据随机层高和插入位置将元素插入跳表 */
    }
}
```

5.9 Sprung 排序的效率与稳定性

Sprung 排序算法没有改变插入排序的两大核心步骤：交换与比较，只是对这两个步骤的具体实现做了优化。

设所有待排序元素数量为 n。交换步骤中，由于采用跳表，每次插入的操作包含随机层高生成和每层插入两个步骤，总的的时间复杂度是 O(logn)。

$$\mathbf{RMN}=\sum_{i=1}^{n} log_2\ n = n \cdot log_2 n$$

比较步骤中，跳表始终是接近理想跳表的结构，因此搜索插入位置的时间复杂度是 O(logn)。

假设每次在各个位置插入是等可能的，总的比较次数的数量级为。

$$\mathbf{KCN}=\sum_{i=1}^{n} log_2\ n = n \cdot log_2 n$$

首次建立结点还需要先遍历序列挑选出最小值，这一步的时间复杂度为 O(n)

综上所述，全过程平均时间复杂度和最好、最坏时间复杂度是相同的，都是

$$\mathbf{O}(n\ log_2 n+n\ log_2 n+n)=\mathbf{O}(n\ log_2 n)$$

从跳表的建立和添加可以看出，Sprung 排序算法并未对插入排序的过程做出改变，只优化了具体实现的过程。因此，Sprung 算法具有排序稳定性。

6 结 论

6.1 Sprung 算法的优缺点

现行常用的排序算法平均时间复杂度往往在 $O(n \log n)$，最差情况往往贴近于 $O(n^2)$，而 Sprung 排序算法几乎在任何情况下都可以保持 $O(n \log n)$的时间复杂度，具有极强的时间稳定性。在空间上也具有排序稳定性。时间与空间上极强的稳定性让 Sprung 算法可以适用于更多不同的实际情景，对关键码的数据类型也没有额外的要求。

Sprung 算法最大的缺点在于需要 O(n)的额外空间来构建跳表结构。但现代计算机发展迅速，使用者支配的空间往往比较富裕，算法所需的 O(n)额外空间不会成为使用障碍。

现代计算机使用者更加追求稳定的时间效率，并不追求空间的极致利用，这也正是 Sprung 排序算法所追求的目标。

6.2 Sprung 算法的适用情景

根据对 Sprung 算法的时间、空间效率分析，可知该算法使用时需要与问题规模相当的额外存储空间，只有在可用空间相对富裕或没有内存限制时，适合使用 Sprung 算法进行排序操作。

对于待排序数据的类型，Sprung 算法也没有额外的要求，只要关键码可以比较大小就可以使用该算法。对原有跳表的比较方法只需添加次关键码，就可以实现二级排序或多级排序。

参考文献

[1] 翁子盛.常见排序算法及其主要应用[J].科学技术创新,2019(26):75-76.

[2] Mohanty Sachi Nandan and Tripathy Pabitra Kumar. Data Structure and Algorithms Using C++:A Practical Implementation[M]. Hoboken, NJ, USA : John Wiley & Sons, Inc., 2021

[3] 张震.排序算法性能分析及基数排序算法的应用[J].时代农机,2017,44(06):36+39.

[4] 殷人昆. 数据结构（用面向对象方法与 C++语言描述）（第 3 版）[M].清华大学出版社, 2021

[5] 赵欢. 计算机科学概论[M].人民邮电出版社, 2014