如果不能在内存中完成排序，而必须在磁盘上进行，称为外部排序。

插入排序：时间复杂度

例：程序sort\_test1

// 插入排序

template<typename Comparable>

void InsertionSort(vector<Comparable>& array)

{

int j;

for (int i = 1; i < array.size(); ++i)

{

Comparable temp = array[i];

// 第i个元素与之前的0-i - 1进行比较

for (j = i; j > 0 && temp < array[j - 1]; --j)

{

// 移动数组

// 如果temp<array[j - 1]，将array[j - 1]向后移动

// 以便插入temp

array[j] = array[j - 1];

}

array[j] = temp;

}

}

希尔排序：缩减增量排序

希尔排序使用一组增量序列，并且。使用增量的一趟排序后，使得。

如果增量序列都为1，则与上述的插入排序没有任何区别。

例：程序sort\_test1

// 希尔排序

template<typename Comparable>

void ShellSort(vector<Comparable>& array)

{

// 增量序列

vector<int> gap\_array;

gap\_array.push\_back(6);

gap\_array.push\_back(3);

gap\_array.push\_back(1);

for (int k = 0; k < gap\_array.size(); ++k)

{

// gap为增量

int gap = gap\_array[k];

for (int i = gap; i < array.size(); ++i)

{

Comparable temp = array[i];

int j = i;

// array[i]与array[i - gap]比较

for (; j >= gap && temp < array[j - gap]; j -= gap)

{

// 移动数组

array[j] = array[j - gap];

}

array[j] = temp;

}

cout << "gap = " << gap << endl;

for (int i = 0; i < array.size(); ++i)

{

cout << array[i] << " ";

}

cout << endl;

}

}

gap为5,3,1得到的排序结果为：

gap = 5

35 17 11 28 12 41 75 15 96 58 81 94 95

gap = 3

28 12 11 35 15 41 58 17 94 75 81 96 95

gap = 1

11 12 15 17 28 35 41 58 75 81 94 95 96

gap为6,3,1

gap = 6

15 94 11 58 12 35 17 95 28 96 41 75 81

gap = 3

15 12 11 17 41 28 58 94 35 81 95 75 96

gap = 1

11 12 15 17 28 35 41 58 75 81 94 95 96

不同的增量序列，中间的排序过程并不相同。

希尔增量：，

最坏时间复杂度为：

Hibbard增量，排序趟数为的整数部分：

最坏时间复杂度为：

例：程序sort\_test1

堆排序：优先队列，平均时间复杂度：

// 堆排序

// 左儿子的索引值

// 与二叉堆不同的是其0位置有元素

// 二叉堆的0位置没有元素

inline int LefChild(int i)

{

return 2 \* i + 1;

}

// 下滤

template<typename Comparable>

void PercolateDown(vector<Comparable>& array, int i, int n)

{

int child;

Comparable temp = array[i];

for (; LefChild(i) < n; i = child)

{

child = LefChild(i);

// 最大堆，小值下滤

if (child != n - 1 && array[child] < array[child + 1])

{

++child;

}

if (temp < array[child])

{

array[i] = array[child];

} else

break;

}

array[i] = temp;

}

template<typename Comparable>

void HeapSort(vector<Comparable>& array)

{

// 建立堆序

for (int i = array.size() / 2; i >= 0; --i)

{

PercolateDown(array, i, array.size());

}

// 排序

for (int j = array.size() - 1; j > 0; --j)

{

// array[0]肯定是数组的最大值

// 排序开始，将array[0]与array[array.size() - 1]交换

// 这样最大值就换到了数组末，然后执行下滤操作

// 注意是前array.size() - 1项的下滤，最后一项不参考

// 这样把前array.size() - 1项的最大值移动到了array[0]位置

swap(array[0], array[j]);

PercolateDown(array, 0, j);

}

// 打印

for (int k = 0; k < array.size(); ++k)

{

cout << array[k] << " ";

}

cout << endl;

}

归并排序：最坏时间复杂度

归并排序采用递归策略，是一种分治的递归策略，有点类似快速排序。

参考：<https://www.cnblogs.com/chengxiao/p/6194356.html>

例：程序sort\_test2

快速排序：平均时间复杂度为

数组的快速排序：

1. 如果中的元素为0或1，则返回。
2. 取中的任一元素，称之为枢纽元(pivot)。
3. 将（中的其余元素）划分为两个不相交的集合：和
4. 返回{quickSort()，后跟，继而quickSort()}。

快速排序中枢纽元的选择很重要，最合适的情况是选择数组的中值，不过实际上难以获取，常用的是采用三数中值分割法来获取枢纽元。

例：程序sort\_test3

// 插入排序

// 数组长度比较小时使用插入排序

// 只排数组的一部分

template<typename Comparable>

void InsertionSort(vector<Comparable>& array, int left, int right)

{

int j;

for (int i = left + 1; i <= right; ++i)

{

Comparable temp = array[i];

// 第i个元素与之前的0-i - 1进行比较

for (j = i; j > 0 && temp < array[j - 1]; --j)

{

// 移动数组

// 如果temp<array[j - 1]，将array[j - 1]向后移动

// 以便插入temp

array[j] = array[j - 1];

}

array[j] = temp;

}

}

// 三数中值分割法

template<typename Comparable>

const Comparable& Median3(vector<Comparable>& array, int left, int right)

{

int center = (left + right) / 2;

if (array[center] < array[left])

{

swap(array[left], array[center]);

}

if (array[right] < array[left])

{

swap(array[left] , array[right]);

}

if (array[right] < array[center])

{

swap(array[right], array[center]);

}

// array[left] < array[center] < array[right]

// array[center]为枢纽元

// 交换array[center]与array[right - 1]，是因为

// array[right] > array[center]，将枢纽元放置在right - 1位置

// 这样后续的比较可以少比较一次

swap(array[center], array[right - 1]);

return array[right - 1];

}

template<typename Comparable>

void QuickSort(vector<Comparable>& array, int left, int right)

{

if (left + 10 <= right)

{

// 枢纽元

Comparable pivot = Median3(array, left, right);

int i = left;

int j = right - 1; // right位置的元素肯定比pivot大

for (; ;)

{

// 左边小于枢纽元

while (array[++i] < pivot)

{

}

// 右边大于枢纽元

while (pivot < array[--j])

{

}

if (i < j)

{

swap(array[i] , array[j]);

} else

break;

}

// 退出上面的for循环

// 说明array[i] >= array[right - 1]

// 交换它们进行排序

// array[i]存储的就是pivot

swap(array[i], array[right - 1]);

// left到i - 1肯定小于pivot

QuickSort(array, left, i - 1);

// i + 1到right肯定大于pivot

QuickSort(array, i + 1, right);

} else {

// 如果left与right之间的差距小于10

// 则执行插入排序

cout << "执行插入排序" << " left = "

<< left << " right = " << right << endl;

InsertionSort(array, left, right);

}

}

template<typename Comparable>

void QuickSort(vector<Comparable>& array)

{

QuickSort(array, 0, array.size() - 1);

for (int i = 0; i < array.size(); ++i)

{

cout << array[i] << " ";

}

cout << endl;

}

使用1000万个随机元素的文件，找出第500万个最大值，例：程序sort\_test4

解决下面的问题：

http://blog.csdn.net/gzlaiyonghao/article/details/3547776

例：程序sort\_test5