STL算法复杂度索引

$O(nlog_2n)$

1.sort()

`sort永远滴神！`

$O(log_2n)$

1.lower_bound()和upper_bound()

`set由于是有序的，直接使用std::lower_bound()的效率接近O(n)，需要使用`

`std::set::lower_bound()` 参考

lower_bound(begin,end,num)：从数组的begin位置到end-1位置二分查找第一个大于或等于num的数字，找到返回该数字的地址，不存在则返回end。通过返回的地址减去起始地址begin,得到找到数字在数组中的下标。

upper_bound(begin,end,num)：从数组的begin位置到end-1位置二分查找第一个大于num的数字，找到返回该数字的地址，不存在则返回end。通过返回的地址减去起始地址begin,得到找到数字在数组中的下标。

O(n)

1.next_permutation()

`按照字典序产生序列，因此若须产生全序列，则须事先进行排序，prev_mermutation()同`

代码实例

```
#include<cstdio>
#include<algorithm>
using namespace std;
int main(void){
    int arr[] = {1, 2, 3, 4};
    do
    {
        printf("%d%d%d%d\n", arr[0], arr[1], arr[2], arr[3]);
    }
    while(next_permutation(arr, arr + 4));
    return 0;
}
```

2.fill()

转载

> 一维数组代码实例

```
#include <algorithm>
#include <stdio.h>
using namespace std;
int v[10];
int main(int argc, char const *argv[])
{
    fill(v,v+10,-1);
    for (int i = 0; i <10; ++i)
        printf("%d ",v[i]);

    return 0;
}
```

> 二维数组代码实例

```
#include <algorithm>
#include <stdio.h>
using namespace std;
int v[10][10];
int main(int argc, char const *argv[])
{
    fill(v[0],v[0]+10*10,-1);
    for (int i = 0; i <10; ++i){
        for (int j = 0; j <10 ; ++j){
            printf("%d ",v[i][j] );
        }
        printf("\n");
    }
    return 0;
}
```

补充：

vector.assign()会释放原本空间进行赋值 函数原型是：

1:void assign(const_iterator first,const_iterator last);

2:void assign(size_type n,const T& x = T());

第一个相当于个拷贝函数，把first到last的值赋值给调用者；（注意区间的闭合） 第二个把n个x赋值给调用者；

3.reverse()

> 代码实例

```
class Solution {
public:
    vector<vector<int>> flipAndInvertImage(vector<vector<int>>& A) {
        int col = A.size();
        int row = A[0].size();
        for(int i=0;i<col;i++)
        {
            reverse(A[i].begin(),A[i].end());//传两个迭代器
        }
        for(int i=0;i<col;i++)
        {
            for(int j=0;j<row;++j)
            {
                A[i][j]=!A[i][j];
            }
        }
        return A;
    }
};
```

4.unique()

1.只有两个参数，且参数类型都是迭代器：

```
iterator unique(iterator it_1,iterator it_2);
```

这种类型的unique函数是我们最常用的形式。其中这两个参数表示对容器中[it_1，it_2)范围的元素进行去重(注：区间是前闭后开，即不包含it_2所指的元素),返回值是一个迭代器，它指向的是去重后容器中不重复序列的最后一个元素的下一个元素。

> vector离散化（使用前先sort）

```
vector.erase(unique(vector.begin(),vector.end()),vector.end())
```

2.有三个参数，且前两个参数类型为迭代器，最后一个参数类型可以看作是bool类型：

该类型的unique函数我们使用的比较少，其中前两个参数和返回值同上面类型的unique函数是一样的，主要区别在于第三个参数。这里的第三个参数表示的是自定义元素是否相等。也就是说通过自定义两个元素相等的规则，来对容器中元素进行去重。

```
iterator unique(iterator it_1,iterator it_2,bool MyFunc);
```

5.random_shuffle()

> generator by default (1)

template void random_shuffle (RandomAccessIterator first, RandomAccessIterator last);

> specific generator (2)

template void random_shuffle (RandomAccessIterator first, RandomAccessIterator last, RandomNumberGenerator&& gen);

1.快速排序面对较为有序的数列的效率较低。 而打乱顺序能有效 提升快排速度的稳定性。

2.在随机化算法方面的价值。

> 补充

srand函数是随机数发生器的初始化函数。原型:void srand(unsigned seed);

用法:它初始化随机种子，会提供一个种子，这个种子会对应一个随机数，如果使用相同的种子后面的rand()函数会出现一样的随机数，如: srand(1); 直接使用1来初始化种子。不过为了防止随机数每次重复，常常使用系统时间来初始化，即使用 time函数来获得系统时间，它的返回值为从 00:00:00 GMT, January 1, 1970 到现在所持续的秒数，然后将time_t型数据转化为(unsigned)型再传给srand函数，即: srand((unsigned) time(&t)); 还有一个经常用法，不需要定义time_t型t变量,即: srand((unsigned) time(NULL)); 直接传入一个空指针，因为你的程序中往往并不需要经过参数获得的数据。

进一步说明下：计算机并不能产生真正的随机数，而是已经编写好的一些无规则排列的数字存储在电脑里，把这些数字划分为若干相等的N份，并为每份加上一个编号用srand()函数获取这个编号，然后rand()就按顺序获取这些数字，当srand()的参数值固定的时候，rand()获得的数也是固定的，所以一般srand的参数用time(NULL)，因为系统的时间一直在变，所以rand()获得的数，也就一直在变，相当于是随机数了。只要用户或第三方不设置随机种子，那么在默认情况下随机种子来自系统时钟。如果想在一个程序中生成随机数序列，需要至多在生成随机数之前设置一次随机种子。 即：只需在主程序开始处调用srand((unsigned)time(NULL)); 后面直接用rand就可以了。不要在for等循环放置srand((unsigned)time(NULL));

> 用途

当某个数据类型的不等号不具备传递性(比如一个数据类型有5个int成员，其中3个较大时，这个元素就较大)，可以先打乱数组，然后遍历每个元素，依次比较是否比其他所有数据大，小于等于时break进入下一轮比较，可以证明整体复杂度为O(logn)，证明

6.nth_element()

void nth_element(_RAIter, _RAIter, _RAIter);

void nth_element(_RAIter, _RAIter, _RAIter, _Compare);

void nth_element(_RandomAccessIterator __first, _RandomAccessIterator __nth,_RandomAccessIterator __last)

void nth_element(_RandomAccessIterator __first, _RandomAccessIterator __nth,_RandomAccessIterator __last, _Compare __comp)

nth_element(arr,arr+x,arr+n);

可以将x在[0,n]范围内，将第x小的数字移动到arr[x]上，其余比arr[x]大的，在x后面，比arr[x]小的，在x前面。

> ## O(1)

1.__builtin_popcount(x):

返回x二进制表示下1的个数

2.__builtin_clz(x)：

返回x二进制下32位的前导零个数 leading zero

3.__builtin_ctz(x)：

返回x二进制下32位的后缀零个数 trailing zero