1. Merge k Sorted Lists

Hard

Merge *k* sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

**Example:**

Input:  
[  
 1->4->5,  
 1->3->4,  
 2->6  
]  
Output: 1->1->2->3->4->4->5->6

**解法1**

暴力求解。先将所有节点的值存在数组中，再排序，最后构建新的链表

**解法2**

k-路归并。设置k个工作指针，每次从k个节点中挑出最小的一个节点

/\*\*  
 \* Definition for singly-linked list.  
 \* struct ListNode {  
 \* int val;  
 \* ListNode \*next;  
 \* ListNode(int x) : val(x), next(NULL) {}  
 \* };  
 \*/  
class Solution {  
public:  
 ListNode\* mergeKLists(vector<ListNode\*>& lists) {  
 vector<ListNode\*>v;  
 ListNode \*head = new ListNode(-1), \*cur = head;  
 for(int i = 0; i < lists.size(); ++i){  
 if(lists[i] != NULL)v.push\_back(lists[i]);  
 }  
 while(v.size() > 1){  
 int index = 0;  
 for(int i = 1; i < v.size(); ++i){  
 if(v[i]->val < v[index]->val){  
 index = i;  
 }  
 }  
 cur->next = v[index];  
 cur = cur->next;  
 v[index] = v[index]->next;  
 if(v[index] == NULL)v.erase(v.begin() + index);  
 }  
 if(v.size() == 1)cur->next = v[0];  
 return head->next;  
 }  
};

**解法3**

在2的基础上将工作指针存储为**优先队列**

1. 优先队列默认按照大根堆实现，比较函数是“>”的逻辑，但是内部使用的是less函数模板
2. 指明比较方式时需要使用三参数定义式，其中容器参数不能省略，通常使用vector

* priority\_queue<ListNode\*, vector<ListNode\*>, cmp>q;

1. 比较规则通过重载函数体实现

* struct cmp{  
   bool operator()(ListNode \*a, ListNode \*b){  
   return a->val > b->val;  
   }  
  };

时间复杂度：

空间复杂度：

class Solution {  
public:  
 struct cmp{  
 bool operator()(ListNode \*a, ListNode \*b){  
 return a->val > b->val;  
 }  
 };  
 ListNode\* mergeKLists(vector<ListNode\*>& lists) {  
 vector<ListNode\*>v;  
 ListNode \*head = new ListNode(-1), \*cur = head;  
 priority\_queue<ListNode\*, vector<ListNode\*>, cmp>q;  
 for(int i = 0; i < lists.size(); ++i){  
 if(lists[i] != NULL)q.push(lists[i]);  
 }  
 while(!q.empty()){  
 ListNode \*temp = q.top();  
 q.pop();  
 cur->next = temp;  
 cur = cur->next;  
 temp = temp->next;  
 if(temp != NULL)q.push(temp);  
 }  
 return head->next;  
 }  
};

**解法4**

做k-1次2-路归并。用一个栈存储归并好链表的头节点

实际效率不高，和解法3相差极大

时间复杂度：

class Solution {  
public:  
 ListNode\* mergeKLists(vector<ListNode\*>& lists) {  
 stack<ListNode\*>s;  
 for(int i = 0; i < lists.size(); ++i)if(lists[i] != NULL)s.push(lists[i]);  
 if(s.size() == 0)return NULL;  
 while(s.size() > 1){  
 ListNode \*p = s.top();  
 s.pop();  
 ListNode \*q = s.top();  
 s.pop();  
 s.push(merge2(p, q));  
 }  
 return s.top();  
 }  
 ListNode\* merge2(ListNode \*l1, ListNode \*l2){  
 ListNode \*head = new ListNode(-1), \*cur = head;  
 ListNode \*p = l1, \*q = l2;  
 while(p != NULL && q != NULL){  
 if(p->val < q->val){  
 cur->next = p;  
 p = p->next;  
 }else{  
 cur->next = q;  
 q = q->next;  
 }  
 cur = cur->next;  
 }  
 if(p != NULL)cur->next = p;  
 if(q != NULL)cur->next = q;  
 ListNode \*res = head->next;  
 delete head;  
 return res;  
 }  
};

**解法5**

分治法。

用队列实现。每次将新合并的节点放在队尾。

时间复杂度：

实际运行效率远远优于k-1次两路归并，与使用优先队列的k-路归并相当

class Solution {  
public:  
 ListNode\* mergeKLists(vector<ListNode\*>& lists) {  
 queue<ListNode\*>q;  
 for(int i = 0; i < lists.size(); ++i)if(lists[i] != NULL)q.push(lists[i]);  
 if(q.size() == 0)return NULL;  
 while(q.size() > 1){  
 ListNode \*p1 = q.front();  
 q.pop();  
 ListNode \*p2 = q.front();  
 q.pop();  
 q.push(merge2(p1, p2));  
 }  
 return q.front();  
 }  
 ListNode\* merge2(ListNode \*l1, ListNode \*l2){  
 ListNode \*head = new ListNode(-1), \*cur = head;  
 ListNode \*p = l1, \*q = l2;  
 while(p != NULL && q != NULL){  
 if(p->val < q->val){  
 cur->next = p;  
 p = p->next;  
 }else{  
 cur->next = q;  
 q = q->next;  
 }  
 cur = cur->next;  
 }  
 if(p != NULL)cur->next = p;  
 if(q != NULL)cur->next = q;  
 ListNode \*res = head->next;  
 delete head;  
 return res;  
 }  
};