优先队列讲稿

一、优先队列概述

普通的队列是一种先进先出（FIFO）的数据结构，元素从队尾插入（push），从队首删除(pop)。

而在优先队列中，元素被赋予优先级，当访问元素时，具有最高优先级的元素最先删除，所以优先队列不再是先进先出，而是优先级高的先出，类似医院中的急诊处理——病情越严重，优先级越高，因而，最危重的病人需要排在最前面，从而最早得到处理。优先队列按优先权出列顺序分两种：优先权最小的先出队称为最小队，反之则成为最大队。

优先队列执行的操作有：1）查找；2）插入一个新元素；3）删除。在最小队中，查找操作用来搜索优先权最小的元素；删除操作用来删除该元素；在最大队中，查找操作用来搜索优先权最大的元素，删除操作用来删除该元素。优先队列中的元素可以有相同的优先权。查找与删除操作可根据任意优先权进行。

C++的 STL 中提供优先队列的容器：priority_queue，在这个容器中，插入、删出、查找操作的时间复杂度都是 $O(\log_2 n)$的。而手工自定义的优先队是一种被称为“堆”的数据结构，后面又介绍。

【引例】集合维护

设计一种数据结构来维护一个整数集合，支持下面 3 中操作，要求每个操作都尽量地快。

1、把一个整数 x 加入到集合中。

2、询问集合最小元素的值。

3、删除集合中最小的元素。

【样例】

9　　//N<=100 000：有 N 个操作 1 20 2 1 30 1 10 2 3 2 3 2	20　//依次是每个查询操作 10 20 30

【讲解】

因为本题的查找和删出的都是集合的最小元素，所以可以用 STL 的最小优先队列来维护集合。

1、STL 之优先队列的定义

定义 STL 的优先队列需要指明优先级，显然本题应以元素值作为优先级，且**值越大的优先级越低**。

定义方法 1：缺省优先级

#include<queue> **priority_queue<int>q;**	缺省优先级为：元素值小的优先级低，队列中队首元素是值最大的，先出队列，即缺省定义的是一个**最大队**。

定义方法 2：仿函数自定义优先级

```
struct mycmp
{
  bool operator ()(int a, int b)  //如果 a 比 b 的优先级低则返回 true
  {
       return  a > b;  //值大的优先级低
  }
};
priority_queue< int, vector<int>, mycmp >pq;
                元素类型   容器        比较函数
```

定义方法 3、元素是结构体的优先队列定义

```
struct node
{
    int x, y;
    friend bool operator< (node a, node b) //通过重载<操作符来比较元素中的优先级不能重载>。
    {
        return a.x > b.x;   //结构体中，成员 x 值大的优先级低
    }
};
priority_queue<node>q;    //定义方法
```

也可以这样定义：

```
struct node{ int x, y; }; //结构体，作为队列元素
struct mycmp   //自定义比较函数
{
    bool operator ()(node a, node b)  //如果 a 比 b 的优先级小则返回 true
    {
        return  a.x > a.x;  //结构体中，成员 x 值大的优先级低
    }
};
priority_queue<node,vector<node>,mycmp>q;    //定义方法
```

还有其他定义方法，请自行查找有关资料。

2、优先队列的操作（因为优先队列内部用平衡树实现，所以其维护的时间复杂度为 $O(\log_2 n)$）

操作	含义	时间复杂度
q.empty();	如果队列为空返回真（true）	O(1)
int sz=q.size();	返回优先队列中拥有的元素个数	O(1)
q.push(x);	加入一个元素	$O(\log_2 n)$
q.pop();	删除对顶元素	$O(\log_2 n)$
x=q.top();	返回优先队列队顶元素	O(1)

3、用优先队列解答本题（时间复杂度为 $O(n\log_2 n)$）

```
struct mycmp   //最小队的优先级定义
{
    bool operator()(int a,int b){ return a>b; }   //值大的优先级低
};
int main()
{
    priority_queue< int, vector<int>, mycmp > pq;   //定义队首元素最小的优先队列
    scanf("%d",&N);
    int op,x;
    for(int i=1;i<=N;i++)
    {
        scanf("%d",&op);
        if(op==1) { scanf("%d",&x); pq.push(x); }      //入队操作
        else if(op==2 && !pq.empty()) { printf("%d\n",pq.top());} //取当前优先级最高的元素
        else if(op==3 && !pq.empty()) {  pq.pop();}     //出队操作
    }
    return 0;
}
```

二、优先队列的应用

例 1、数列处理器

一个整数序列处理器，支持如下命令操作，现在请你编程模拟这个处理器。

1、ADD x： 向整数序列加入一个整数 x；

2、DEL： 删除整数序列中的第 K 小的数（把序列中的数以非降排序后的第 K 个位置的数），如果 K 大于当前序列的整数个数，则不执行；

3、QUERY： 查询序列中第 K 小的数，如果 K 大于当前序列个数，则输出“Error”；

【样例】

2　　　//K ADD 10　//若干条命令，命令条数不超过 50000。 ADD 3 ADD 5 DEL ADD 8 QUERY ADD 4 DEL QUERY	8　//查询的结果 8

【讲解】

◆定义一个最大队 A，用于维护当前序列中的前 K 小元素；如果 A 中元素个数为 K，则队首 A.top() 就是第 K 小的元素；

◆定义一个最小队 B，用于维护当前序列中的其他元素；那么队首 B.top() 就是第 K+1 小的元素；

数据结构定义如下：

```
struct cmp1  //最大队：值小的优先级低
{
    bool operator()(int a,int b)
    {return a<b;}
};
struct cmp2 //最小队：值大的优先级低
{
    bool operator()(int a,int b)
    {return a>b;}
};
priority_queue<int,vector<int>,cmp1>A;
priority_queue<int,vector<int>,cmp2>B;
int main()
{
  scanf("%d",&K);  //输入 K
  char op[10];
  while(scanf("%s",op)==1)  //还有命令存在
  {
     if(op[0]=='A') do_ADD();
     else if(op[0]=='D') do_DEL()
     else if(op[0]=='Q') do_QUERY()
  }
  return 0;
}
```

各命令的执行如下：

```
void do_ADD()
{
    读入 x 并把 x 加入最大队 A 中；
    如果 A 中的元素个数为 K+1，则把 A 的顶元素 push 到 B 中，然后删出 A 的顶元素。
}
void do_DEL()
{
    如果 A 中有 K 个元素则：删出 A 的顶元素；然后把 B 的顶元素 push 到 A 中；删出 B 的顶元素；
}
void do_QUERY()
{
    如果 A 中元素有 K 个元素，则输出其顶部元素。
}
```

因为维护优先队列的时间复杂度为 $\log_2 N$，所以算法的时间复杂度(O(N$\log_2$N))，其中 N 代表命令条数。

例 2 序列合并

有两个各包含 N 个整数的序列 A 和 B，在 A 和 B 中各取一个数相加可以得到 N^2 个和，求这 N^2 个和中最小的 N 个。

【样例】

3 //N: 1<=N<=100000 2 6 6 //序列 A 1 4 8 //序列 B	3 6 7 //前 N 个最小数

【讲解】

二路归并算法：在归并排序中，每次合并需要把两个有序表合成一个，可以做到 O(n)的时间复杂度。

多路归并算法：把多个有序表合并成一个有序表，下面来探讨多路多路归并的高效算法。

面对 $N<=100000$ 规模，显然暴力生成 N^2 个和生成出来，再排序是不可能的。如果把 A[i],B[i]递增排序后，把 N^2 个和组织成如下的 N 个递增有序表，构成 N^2 矩阵：

表 1、	A[1]+B[1] <= A[1]+B[2] <= ... <= A[1]+B[N]
表 2、	A[2]+B[1] <= A[2]+B[2] <= ... <= A[2]+B[N]
……	……
表 N、	A[N]+B[1] <= A[N]+B[2] <= ... <= A[N]+B[N]

第 i 行第 j 列的元素为：

A[i]+B[j]

算法实现就是：先把矩阵中的第一列的所有元素：A[1]+B[1]、A[2]+B[1]+...+A[N]+B[1]进入最小队；然后每次都输出**顶元素**，并删除这个元素，如果该元素元素是来自上述矩阵序列的第 i 行的第 j 个元素，则把第 i 行的下一个元素 A[i]+B[j+1]插入队中。直到输出 N 个元素为止。

在定义优先队列时，队列中的元素应包含 3 个成员：元素值 A[i]+B[j]、矩阵的行 i 和列 j。

数据结构定义	算法框架（时间复杂度：O(N*log$_2$N)）

<pre>struct data
{
   int V,i,j;   //V=A[i]+B[j]
   friend bool operator<(data a,data b)
   {
      return a.V>b.V;
   }
};
priority_queue<data> pq;</pre>
	<pre>void solve()
	{
	    输入数据;
	    把 A[]和 B[]由小到大排序;
	    for(int i=1;i<=N;i++) //第 1 列元素进入队列
	      pq.push((data){A[i]+B[1],i,1});
	    for(int k=1;k<=N;k++)
	    {
	       data t=pq.top();
	       pq.pop();
	       printf("%d ",t.V); //输出队顶元素
	       int i=t.i,j=t.j;
	       pq.push((data){A[i]+B[j+1],i,j+1});
	    }
	}</pre>

另外，本题也可以利用二分猜答案的思想，做到$O(N(\log_2 N)^2)$的时间复杂度。

优先队列动态维护集合的思想常用来优化贪心、动态规划等算法，来看下面的例子。

例 3、智力大冲浪[2]（题库 P1590）

有 N 个智力小游戏，游戏者从第 0 时刻开始，完成第 i 个小游戏需要 Ci 分钟，且这个游戏必须在规 Ti 分钟前完成。一个游戏一旦开始，就必须到完成为止，中间不能有任何停顿，一个人在同一个时间段，只能做一个游戏，不能同时做两个游戏。最后的胜利者是完成游戏最多的人！

作为参赛者，小沐很想赢得冠军，请帮他算算，怎样安排游戏的顺序才能完成最多的游戏。

【样例】

5　　//N 个游戏 2 4 5 4 3　　//C[i] 5 5 8 10 11　　//T[i]	3

【数据范围】

对于 100%的数据：N<=50000；1<=Ci,Ti<=2 000 000 000

【讲解】

数据规模巨大排列问题，只能考虑贪心或动态规划。

本题可采用贪心算法，贪心策略为：按期限 Ti 由小到大排序，然后这个顺序来依次考察每个游戏，对于当前考察的游戏 i，有如下两种情况：

- **情况 1：**设已经选择的游戏需要的总时间为 tot，如果 tot+C[i]<=T[i]，则选择游戏 i；
- **情况 2：**tot+C[i]>T[i]，设已经选择的游戏中花费时间最长是游戏 j，如果 C[j]>C[i]，则可用游戏 i 替换游戏 j。因为替换后，选择的游戏总量不变，但已经选择的游戏花费的总时间 tot 会减少 C[j]-C[i]，为后面的游戏选择留下更多的时间，所以划算些！

算法框架：

```
1、读入 P[i].C 和 P[i].T
2、把游戏按 P[i].T 由小到大排序
3、tot=0;   //已经选择的游戏完成的总时间
3、for(int i=1;i<=N;i++) //依次考察每个游戏
   {
          if(tot+P[i].C<=P[i].T)
             选择该游戏(ans++; tot+= C[i]);
          else
          {
             查找已经选择的游戏中，用时最多的游戏 j;
             如果 P[j].C>P[i].C，则用游戏 i 替换 j (tot-=P[j].C-P[i].C)
          }
      }
```

注意到上面算法中，如果用顺序查找已选择的游戏中用时最多的游戏 j 的话，那么算法的时间复杂度为 $O(N^2)$，只能通过 50%的数据，所以需要优化算法。

“查找已经选择的游戏中，**完成时间最大**的游戏 j”，可以用一个最大队的来维护已经选游戏的集合，以每个游戏所需时间 c 为队列元素，C 小的优先级低。这时：

- 选择任务 i 后，把 C[i]压入优先队列中；时间复杂度为 $O(\log_2 N)$
- 已选择任务中，所需时间最大的为队顶元素 pq.top()，如果其 pq.top()>C[i]，则删除 pq.top()，插入 C[i]。时间复杂度为 $O(\log_2 N)$

所以，算法总时间复杂度为：$O(N\log_2 N)$，期望得分 100 分

由此，可以看出：优先队列在凡是涉及查找最大、最小、删除最大、最小的算法中，能很好地优化算法！！

例 4、滑动窗口（P1511）

给你一个长度为 N 的数组，一个长为 K 的滑动的窗体从最左移至最右端，你只能见到窗口的 K 个数，每次窗体向右移动一位，你的任务是找出窗口在各位置时的最大值和最小值。如下表长度为 3 的窗口：

窗口位置	窗口中的最小值	窗口中的最大值
[1 3 -1] -3 5 3 6 7	-1	3
1 [3 -1 -3] 5 3 6 7	-3	3
1 3 [-1 -3 5] 3 6 7	-3	5
……		

【样例】

8 3 //N,K 1 3 -1 -3 5 3 6 7 //数组	-1 -3 -3 -3 3 3 //窗口最小值 3 3 5 5 6 7 //窗口最大值

【讲解】

问题描述为：

f[i]=min{ a[j] | i-K+1<=j<=i } 实质是：在窗口[i-K+1,i]内选一个最小元素

g[i]=max{ a[j] | i-K+1<=j<=i } 实质是：在窗口[i-K+1,i]内选一个最大元素

最后需要输出 f[K]..f[n]和 g[K]..g[n]。

把窗口里的元素看成一个集合 A，随着窗口向右滑动，对集合的维护不外乎就是如下 3 种操作：

1）插入：窗口右边外的元素插入到集合 A 中；

2）删除：窗口左边的元素从集合 A 中删除；

3）查找：查找集合 A 中的最小元素或最大元素。

穷举+查找算法当然是正确的，时间复杂度为：O(N*K)，对与 N 达到百万的规模，显然无法通过。

如果用优先队列来实现集合的这 3 种维护操作，可以做到每次维护操作的时间复杂度$O(\log_2 N)$。

下面的代码以求窗口的最小值为例：

```
struct D  //队列中的元素；
{
   int v;  //v 是元素值
   int id     // id 表示元素对应数组中的下标
};
struct cmp1  //最小队的优先级定义
{
  bool operator()(data a,data b)
  {
      return a.v>b.v;
  }
};
```

```
int main()
{
   scanf("%d%d",&N,&K);;
   for(int i=1;i<=N;i++)
      scanf("%d",&a[i]);

    priority_queue<D,vector<D>,cmp1>A;

    for(int i=1;i<=N;i++)  //a[i]进入窗口
    {
      while(!A.empty() && i-A.top().id+1>K)
          A.pop();  //队顶元素不在窗口中

       A.push((D){i,a[i]});  //a[i]进入窗口

       if(i>=K)   //输出窗口中的最小值
          printf("%d ",A.top().v);
    }
    printf("\n");
    …………  //维护窗口的最大值类似
    return 0;
}
```

这个代码的时间复杂度：每次维护的时间复杂度是 $\log_2 N$，总共进行了 N 次维护，所以总时间复杂度是 $O(N*\log_2 N)$的。

例 5、烽火传递（P1513）

在某两座城市之间有 N 个烽火台，每个烽火台发出信号都有一定的代价。为了使情报准确的传递，在 M 个烽火台中至少要有一个发出信号。现输入 N、M 和每个烽火台发出的信号的代价，请计算总共最少需要花费多少代价，才能使敌军来袭之时，情报能在这两座城市之间准确的传递!!!

【输入样例】

5 3 //N,M 1 2 5 6 2 //a[i]	4 //最小代价

【数据范围】

1 <= m < n <= 1,000,000 每个烽火台的代价不超过 10000。

【讲解】

显然的动态规划：

状态定义： f(i)=消息传到第 i 个烽火台，且第 i 个必须发信号的情况下，需要的最少代价。

边界分析： f(0)=0;

最后的答案： Ans=f(n+1) //n+1 目的城市，其代价 a[n+1]=0;

状态转移方程： f(i)=**min{ f(j) | j>=0 && i-j<=m }** + a[i]

用顺序查找来实现这个状态转移方程的时间复杂度为 O(n*m)，不能通过全部数据。

优化： 分析方程 f(i)=**min{ f(j) | j>=0 && i-j<=m }** + a[i]，红色部分就是在窗口[i-m,i-1]中选择一个最小的 f[j]，且随着 i 向后走，窗口中不断有元素进和出，所以可以用优先队列中的最小队来维护窗口中的元素。由于优先队列的每次维护时间复杂度是 $\log_2 m$ 的，所以这时的时间复杂度为 $O(n\log_2 m)$。

决策是用顺序选择：O(nm)

```
void solve()
{
  int i,j;
  f[0]=0;    ////技巧，虚拟一个 0
  a[n+1]=0;
  for(i=1;i<=n+1;i++)   //技巧，虚拟一个 n+1
  {
    LL t=inf;
    for(j=i-1;j>=0 && i-j<=m;j--) //顺序查找
        t=min(t,f[j]);
    f[i]=t+a[i];
  }
  cout<<f[n+1]<<'\n';
}
```

用优先队列维护决策表：$O(n\log_2 m)$

```
struct data   //最小队元素定义
{
    LL v;
    int id;
    friend bool operator <(data a,data b)
    {
        return a.v>b.v;
    }
};
void solve()
{
   priority_queue<data>q;  //最小队
   f[0]=0;
   a[n+1]=0;
   q.push((data){f(0),0});
  for(int i=1;i<=n+1;i++)
  {
    while(!q.empty()&&i-q.top().id>m)
        q.pop();
    LL t=q.top().v;
    f[i]=t+a[i];
    q.push((data){f[i],i});
  }
   cout<<f[n+1]<<'\n';
}
```

动态规划还可以优化图论中的最优路径算法，隐式图的 BFS 等算法，以后回陆续学习！