数据结构归纳

上海洛谷网络科技有限公司
山东大学泰山学堂
—扶苏—

洛谷
www.luogu.com.cn

contents

vector

经典的可变长容器数组

vector 的内存分配机制

- vector 在内存上有两个参量：容量 capacity 和元素个数 size。以下将 capacity 简写为 c，size 简写为 s。他们永远满足 s<=c。vector 的实际空间占用是 capacity。
- 可以用 capacity() 和 size() 两个函数查看这两个参量。

```cpp
#include <vector>
#include <iostream>

int main() {
  std::vector<int> ovo;
  std::cout << ovo.size() << ' ' << ovo.capacity() << std::endl;
  ovo.push_back(1); ovo.push_back(2); ovo.push_back(3);
  std::cout << ovo.size() << ' ' << ovo.capacity() << std::endl;
}
```

运行结果：
0 0
3 4

- 修改这两个参量的对应函数为 reserve() 和 resize()。我们将会在晚上介绍他们。

vector 的内存分配机制

- 每次向 vector中 push_back 时，如果当前 s < c，则直接在容器的下一个位置添加元素；如果当前 s = c，则触发扩容机制：再申请一段大小为 kc 的连续内存，将当前 c 个元素拷贝过去，然后再按 s < c 添加元素。其中 k 是一个大于 1 的实数。
- g++ 中 k=2。
- 拷贝代价为均摊 O(1)。证明：设当前容器大小为 n，则上次扩容的拷贝次数为 n/k，上上次拷贝扩容的次数为 n/k^2，……，总的拷贝代价为 $n/k + n/k^2 + n/k^3 + \cdots = n(1/k + 1/k^2 + \cdots)$ 。
- 括号里的内容是个无穷级数，收敛于一个与 k 有关的常数。当 k=2 时，它收敛于 1（准确地说，括号内的内容永远小于 1）

B3665 小清新数据结构题

- 给你 n 条数据，第 i 条数据有 s_i 个数字。
- 有 q 次询问，每次给定 x, y，求第 x 条数据的第 y 个数字是多少。
- $n, s_i, q \leq 3 \times 10^6$，且 s_i 之和不超过 3×10^6。
- $2s$，有 O2。

B3665 小清新数据结构题

- 注意到 n 和 s_i 的最大值都是 3×10^6，我们不能开一个 3e6*3e6 的数组，不然会直接完蛋。
- 通过刚才对 vector 内存分配机制的分析，容易发现 vector 的 capacity 不会超过实际元素个数的 k 倍（2 倍）。
- 所以开一个 vector 数组，记录每条信息即可，空间消耗是合理的。

```cpp
#include <array>
#include <vector>
#include <iostream>
#include <algorithm>

const int maxn = 5000006;
std::array<std::vector<unsigned int>, maxn> a;

int main() {
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);
  int n, q;
  std::cin >> n >> q;
  for (int i = 1, x; i <= n; ++i) {
    std::cin >> x;
    for (unsigned y; x; --x) {
      std::cin >> y;
      a.at(i).push_back(y);
    }
  }
  unsigned ans = 0;
  for (int x, y; q; --q) {
    std::cin >> x >> y;
    ans ^= a.at(x).at(y - 1);
  }
  std::cout << ans << std::endl;
}
```

stack

先入后出的线性数据结构

栈

- 栈是一个线性数据结构。
- 考虑一个上下大小相同的木桶：向木桶里放入大小与桶口相同的物品，先被放进去的物品只能后被拿出来。栈就是一个这样的木桶。
- 栈只有一个出入口（类比木桶的桶口），它只需要支持再这一侧压入/取出元素。

std::stack

STL 提供了 std::stack 实现了栈，**但是其底层内存分配逻辑为 std::deque**。这就是说，std::stack 数组的空间消耗可能高于你的预期。

操作	功能
std::stack<int> stk;	声明一个存储 int 的 stack
stk.push(x)	把 x 压入栈
stk.pop();	弹出栈顶
stk.top();	查询栈顶的值（但不弹出）

std::stack 不支持遍历。

算法竞赛中一般认为 std::basic_string 的内存分配逻辑是优于 std::deque 的，事实上 vector 也可以实现 stack 的所有功能：

std::vector	std::stack
vec.push_back(x);	stk.push(x);
vec.pop_back();	stk.pop();
vec.back();	stk.top();

P1241 括号序列

给定一个括号串，按如下的方式配对：

1. 从左到右扫描该字符串
2. 对于一个右括号，找它左侧第一个未配对的左括号，若左右括号对应，则配对成功，否则配对失败。

对每个配对失败的括号，在它旁边添加一个对应括号使之配对成功。输出最后的字符串。

串长不超过 1000000。

P1241

- stack 的经典应用就是处理括号平衡问题。
- 用栈维护当前未配对的左括号。
- 扫描字符串时，遇到左括号入栈，遇到右括号则查看栈顶，如果对应，则配对成功，将左括号弹栈否则配对不成功。

- 有两种配对不成功的括号：扫描时配对失败的右括号、扫描结束后还在栈里的左括号。不要忘记后者。

洛谷

```cpp
std::string s;
std::cin >> s;
std::vector<int> stk;
std::bitset<500> balance;
for (int i = 0; i < s.size(); ++i) {
  char u = s[i];
  if (u == '(' || u == '[') {
    stk.push_back(i);
  } else if (u == ')') {
    if (stk.size() && s[stk.back()] == '(') {
      balance.set(i); balance.set(stk.back());
      stk.pop_back();
    }
  } else {
    if (stk.size() && s[stk.back()] == '[') {
      balance.set(i); balance.set(stk.back());
      stk.pop_back();
    }
  }
}
```

递归的调用栈

系统在进行函数递归时是借助系统栈运行的。

在冯诺依曼体系结构的计算机中，指令和数据在内存中处于同一地位，拥有一个独有的地址。

每调用一次函数，系统会将该函数的内部的局部变量压入系统栈中。同时会记录当前执行的指令。

我们用如下的代码进行演示：

```
void foo(int x) {
  int a = 1;
  foo(a - 1);
}
```

系统栈

Windows 系统的默认系统栈大小只有 1M（1024 K）

所有的局部变量（包括 main 函数内部的）在实际被声明时都会被加入系统栈。

当被压入系统栈的元素大小超过 1M，就会发生 stack overflow，发生 segmentation fault（RE）。

在评测时，系统可用的栈空间大小与内存限制相同，同时其内存占用也被计入程序的内存占用。

系统栈

在竞赛时，如果下发的样例非常大，有可能在本机测试时爆栈。需要在编译时手动指定栈空间大小。在编译命令中加入

-Wl,--stack=512000000

以上命令的含义是将程序可用的系统栈大小调整为 512MB，仅对 Windows 生效。

注意：

1. W 必须大写
2. ,和--之间不能加空格
3. 请务必背熟这一指令
4. 如果在 PowerShell 中使用这一编译开关，需要将其用引号括起来

递归的调用栈

递归的函数结构组织成一棵树，称为递归树。

我们来手画一下递归计算肥波纳妾数列第五项的递归树。

不难发现，递归调用栈所维护的就是递归树上当前节点到根的链上的所有递归函数信息。

此外，当我们对一棵树进行 dfs 遍历时，可以发现递归树刚好就是所遍历的树。

P1612 [yLOI2018] 树上的链

给定一棵树，点有点权 w[u] 和参数 c[u]，要求对每个点 u，找到一条链，满足这条链：

1. 所有点都在 u 到根的简单路径上
2. u 是链的端点
3. 链上的点权和不大于 c[u]。

n = 1e5，点权和参数都是 1e9。

P1612 [yLOI2018] 树上的链

我们可以用栈在 dfs 的同时维护当且节点到根的链上的节点：

进入递归时将当前节点的信息入栈，回溯时弹栈。这样从栈顶到栈底就是当前节点到根的信息。

有了这个栈，我们只需要在栈上找到最长的后缀，满足后缀的权值和不大于 c[u] 即可。这个问题可以二分完成。

具体来说，每次入栈当前节点到根的前缀和 s[u]，二分出最小的 l，满足 s[u] – s[l] <= c[u] 即可。

时间复杂度 O(n log n)

```cpp
int main() {
  int n = read();
  for (int i = 2; i <= n; ++i) {
    e.at(p.at(i) = read()).push_back(i);
  }
  std::generate_n(w.begin() + 1, n, read);
  std::generate_n(c.begin() + 1, n, read);
  stk.push_back(0);
  dfs(1);
  for (int i = 1; i <= n; ++i) std::cout << ans.at(i) << " \n"[i == n];
}

void dfs(const int u) {
  stk.push_back(w.at(u) + stk.back());
  int ret = 0;
  for (int l=0,r=stk.size()-1,mid=(l+r)>>1;l<=r;mid=(l+r)>>1)
    if(stk.back()-stk.at(mid)<=c.at(u)){
        r = (ret = mid) - 1;
      } else {
        l = mid + 1;
      }
  ans.at(u) = stk.size() - ret - 1;
  for (auto v : e[u]) dfs(v);
  stk.pop_back();
}
```

queue /kju:/

先入先出的线性数据结构

队列

考虑排队进行核酸检测：

不断有人排到队伍的尾部，而在队伍头部的人做完核酸将离开队伍。

队列维护的就是这样的一个过程。它有一个出口和一个入口，要支持从入口压入元素、出口弹出元素。

std::queue 和 std::deque

STL 提供了 std::queue 实现了队列，**其底层内存分配逻辑为 std::deque**。

操作	功能
std::queue<int> que;	声明一个存储 int 的 queue
que.push(x)	把 x 压入队列
que.pop();	弹出队头
que.front();	查询队头的值（但不弹出）
que.back();	查询队尾

能从两端插入删除的队列称为双端队列，可以直接使用 std::deque。

操作	功能
std::deque<int> deq;	声明一个存储 int 的 deq
deq.push_back(x)	把 x 压入队尾
deq.pop_back();	弹出队尾
deq.push_front(x);	把 x 压入队头
deq.pop_front();	弹出队头

[NOIP2017PJ] 棋盘

给定一个 m*m 的棋盘。上面有 n 个带颜色的位置，为红色或黄色。其余位置无色

你每次可以走一个四联通的格子（上下左右），但是目标格子必须有颜色。如果颜色和当前格子相同不花钱，否则花一块钱

你还可以用两块钱使用魔法，让某个白色格子变成红色或黄色，持续时间为一步，不可以连续使用。

求从左上角走到右下角的最小花费。

m = 100, n = 1000

[NOIP2017PJ] 棋盘

如果没有魔法的条件，这是一道非常简单的 BFS 题。

加上魔法怎么做？

考虑给 bfs 的状态升维数：用 d[x][y][0/1/2] 表示当前在第 x 行 y 列的格子，本轮没有用魔法/用魔法变成红色/用魔法变成黄色的最小花费。

转移需要仔细一点。

这里选出本题是想提醒大家，不是只有最普通的方格遍历才适用 bfs。很多时候可以通过升维对情况做出更细致的限制，更好的表示我们当前的状态，然后使用 bfs 求出答案。我们称之为隐式图的遍历。

合并傻子 加强版

我们会在讲完堆以后讲解这个问题。

list

链式线性数据结构

链表

考虑一个文档编辑器，我们不仅可以在文档的末尾添加或者删除字符，也应当支持在文档的中间添加字符。可以用链表维护这个功能。

链表是由若干个**节点**按顺序构成的。每个节点有两个信息，被称为数据域和指针域。数据域存放该节点对应的信息，指针域指向该节点的下一个（上一个）节点。

根据指针域的不同，链表分为单向链表和双向链表；根据是否成环，链表分为循环链表和普通链表。

STL 提供了 std::list 作为双向链表，但是效率不佳。

[CSP-J2021]小熊的果篮

给你一个长度为 n 的 01 串。

把这个串里的所有连续 0 串、连续 1 串称作块，例如：

00111100101011

不同的颜色表示不同的块。

会进行若干次操作，每次会删除当前序列里所有块的开头元素，直到序列被删空。

注意每次删除操作结束后可能会出现块的合并

求每次删除的位置在初始串中的下标。

n = 2e5

[CSP-J2021]小熊的果篮

考虑用链表把所有的块组织起来。

每次不断地跳链表就可以找到每个块的开头，然后修改这个块。

注意到每次跳链表就会有一个位置被删除，而我们一共删了 n 个位置，所以只会跳 O(n) 次链表。

[CSP-J2021]小熊的果篮

注意每次一旦把某一块删空了则需要把下一块和上一块合并起来，否则复杂度是不正确的，因为我们的跳链表次数是由删除次数保证的。如果会不断地跳空链表，并且调到实际块也不一定删除的话，那么跳表次数将不再与删除次数同阶。

实现块间合并的最简单方法是在块内同样也挂一个链表，形成「大链表的数据是小链表」的结构，这样两个块可以 O(1) 合并他们自身构成的链表。但是这个结构比较难写难调。

也可以把每个块内的元素扔到 set 里，采用「启发式合并」的方法进行合并。可以做到 $O(n \log^2 n)$ 的复杂度，但是码量会有显著减少

启发式合并

考虑有 n 个集合，集合都只支持 O(T) 插入单个元素，这就是说两个集合合并时只能枚举其中一个集合的元素，逐个插入另一个。

现在给出 q 条合并指令，每次要求合并两个集合，如何给出一个比较优秀的合并算法？

启发式合并

在每次合并时，暴力枚举较小的集合中的元素，插入到大集合里。

时间复杂度?

考虑一个元素，如果它某次被暴力枚举到了，那么它属于一个较小的集合，这就是说，在它本次被插入到另一个集合中后，它所属的集合大小至少会翻倍。

一个大小为 1 的集合，翻倍多少次大小会变成 n？ $\log n$ 次。

这就是说，每个元素至多会被暴力枚举 $\log n$ 次。于是总复杂度为 $O(Tn \log n)$

[CSP-J2021]小熊的果篮

回到本题，我们启发式合并两个 set，每次把较小的 set 直接 insert 到更大的一个即可。

题外话：本题中的「块」如果大小恒为 O(sqrt n)，则我们实现的数据结构本质上是块状链表。

```cpp
int main() {
  int n;
  std::cin >> n;
  auto head = new Node(), tail = head;
  for (int i = 1, pre = -1, x; i <= n; ++i) {
    std::cin >> x;
    if (x != pre) {
      tail = tail->nxt = new Node();
      pre = x;
    }
    tail->s.insert(i);
  }
  for (auto u = head->nxt; u != nullptr; u = head->nxt) {
    for (auto v = head->nxt; v != nullptr; v = v->nxt) {
      std::cout << *(v->s.begin()) << ' ';
      v->s.erase(v->s.begin());
    }
    for (auto pre = head; u != nullptr; u = u->nxt) if (u->s.empty()) {
      auto v = u->nxt; pre->nxt = v;
      if (v && pre != head) {
        if (v->s.size() > pre->s.size()) {
          pre->s.swap(v->s);
        }
        for (auto &&w : v->s) {
          pre->s.insert(w);
        }
        pre->nxt = v->nxt;
        u = v;
      }
    } else {
      pre = u;
    }
    std::cout << '\n';
  }
}
```

```cpp
struct Node {
  Node *nxt;
  std::set<int> s;

  Node() : nxt(nullptr) {}
};
```

heap

最简单的支持最值查询的非线性数据结构

heap

Heap，即「堆」，是一个树形数据结构。他支持查询一个集合中的最小值、删除该最小值、插入一个任意值。单次操作的时间复杂度均为 O(log n)，其中 n 为堆中元素个数。

显然堆在功能上被 std::set 吊起来打，但是其优势在于无可比拟的小常数。

考虑到我们是 junior-based 课程，受到考纲的限制，我们将不介绍 heap 的原理、结构、实现，而是只介绍优先队列 priority_queue，将其作为黑箱使用。

std::priority_queue

操作	功能
std::priority_queue<int> Q	定义一个存放 int 的大根堆，即支持查询集合最大值
Q.push(x)	将 x 加入堆中
Q.top()	获得堆里的最大值
Q.pop()	删除堆的最大值
Q.size()	元素个数
Q.empty()	是否为空
std::priority_queue<int, std::vector<int>, std::greater<int> > Q	定义一个存放 int 的小根堆。

需要头文件 <queue>

关于最后一个操作和对自定义类型、其他类型的比较，我们放在晚上的课程中。

std::priority_queue 的底层容器（默认）是 basic_string 而不是 deque，前者也是 vector 的底层容器，所以其常数（相对 deque）非常小。

[P3378]【模板】堆

给定一个初始为空的数列，要求：

1 x：插入一个整数 x

2：求数列最小值

3：删除数列最小值（有多个时只删一个）

洛谷

```cpp
#include <queue>
#include <iostream>
#include <functional>

int main() {
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);
  std::priority_queue<int, std::vector<int>, std::greater<int> > Q;
  int n; std::cin >> n;
  for (int op, x; n; --n) {
    std::cin >> op;
    if (op == 1) {
      std::cin >> x; Q.push(x);
    } else if (op == 2) {
      std::cout << Q.top() << '\n';
    } else {
      Q.pop();
    }
  }
}
```

P1631 序列合并

给定两个长度为 n 的数列 a,b。在这两个数列中各取一个数，求他们的和，共能得到 n^2 个和（来自不同的位置算不同的和）。现在求这 n^2 个和中最小的 n 个。

n = 1e5

P1631 序列合并

考虑把 a,b 都从小到大排序，于是我们的和有如下关系：

a[1]+b[1] <= a[1]+b[2] <= a[1]+b[3] <= …… <= a[1] + b[n]

a[2]+b[1] <= a[2]+b[2] <= a[2]+b[3] <= …… <= a[2] + b[n]

a[3]+b[1] <= a[3]+b[2] <= a[3]+b[3] <= …… <= a[3] + b[n]

……

a[n]+b[1] <= a[n]+b[2] <= a[n]+b[3] <= …… <= a[n] + b[n]

我们用一个小根堆维护所有的 a[i]+b[1]，这些数里最小的显然是所有和里最小的。

每次取出堆顶。假设取出的是 a[i]+b[j]，则把 a[i]+b[j+1] 加入堆。则显然取出的是当前没选的里最小的和。

时间复杂度 O(n log n)。

P1090 合并果子

给你 n 堆果子，有不同的重量。

每次可以选两堆合成一堆，代价为这两堆的重量之和。

求最小的代价把所有的果子合并成一堆。

（注意这里的堆是物理意义上的堆，不是数据结构）

n = 1e5

3 1 2 9

15

P1090 合并果子

做法是贪心，每次取当前最小的两堆果子进行合并即可。本质上是构建了 Huffman 树。

对 Huffman 树正确性的证明很繁琐，有兴趣的同学可以自行搜索，我们只考虑如何进行这个贪心。

把初始时每一堆果子的重量都扔进小根堆里，每次取两次堆顶并弹出，然后把重量之和压入即可。

P6033 合并傻子加强版

同样的问题，但是 n = 1e7，重量小于 1e5。

P6033 合并傻子加强版

首先可以计数排序（桶排）把果子堆们从小到大排序。

考虑我们接下来生成的果子堆，越靠后生成的则大小越大，也就是他们天然也满足从小到小的顺序。

开两个队列，A 从小到大表示还没被合并过的初始果子堆，B 从小到大表示还没被合并的新生成的果子堆。

每次新生成果子堆就往 B 后面 push 就可以了

取最小的两堆只需要找到这两个队列的前两个元素（共四个），取他们中最小的俩就可以了。

时间复杂度 O(n + a)。

单调数据结构

这部分是选讲内容。我们视时间进行介绍

求数列所有后缀最大值的位置

给定一个数列 a 初始为空，有 n 次操作，每次首先在 a 后面添加一个数，然后请你求出当前 a 所有后缀最大值的位置的个数。也即有多少个 i 满足：对所有的 j > i，都有 a[i] > a[j]。

9 7 6 8 5 4 1 3 2

9 7 6 8 5 4 1 3 2 5

要求复杂度线性。

原题里要求的是求这些位置的按位异或和，没什么本质区别。

求数列所有后缀最大值的位置

考虑用一个栈按顺序维护当前数列的后缀最大值的位置（下标）。初始栈是空的。

注意到这个栈里下标对应的元素是单调递减的。

当新加入一个数时，新加入的数显然是当前数列的后缀最大值。考虑原有的后缀最大值：如果这个最大值大于新数，则它不受影响，否则它将不再是后缀最大值，需要把它在栈里删除。

因为栈里下标对应的元素是单调的，所以我们删除的其实是栈顶一侧的一个连续区间。可以一直比较当前栈顶和新数，前者较小则删除，直到前者较大。

因为下标对应的元素是单调的，所以这个栈被称为单调栈。

我们本质上维护的是数列所有前缀的后缀最值，这是单调栈的功能。

```cpp
#include <vector>
#include <array>
#include <iostream>

int n, ans;
std::vector<int> stk;
std::array<unsigned long long, 1000005> a;

int main() {
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);
  std::cin >> n;
  for (int i = 1; i <= n; ++i) {
    std::cin >> a.at(i);
    while (stk.size() && (a.at(i) >= a.at(stk.back()))) {
      ans ^= stk.back();
      stk.pop_back();
    }
    stk.push_back(i);
    ans ^= i;
    std::cout << ans << '\n';
  }
}
```

P6510 奶牛排队

给定一个数列 a，找到一个最长的区间 [l, r]，满足 a[l] 是 a[l],a[l+1],…,a[r] 中最小的，a[r] 是 a[l],a[l+1],…,a[r] 中最大的。

最大最小均为严格最大最小。

N = 1e5。

P6510 奶牛排队

区间问题的一个常见思路就是枚举一个端点，考察另一个端点的情况。

考虑枚举右端点 r。则左端点 l 一定是 a[1],a[2],…,a[r] 这个前缀的一个后缀最小值点。

r 必须是这段区间内最大的，所以 l 必须大于当前倒数第二个后缀最大值点（r 是倒数第一个）。

用单调栈维护后缀最大最小值，在最大值栈上找到倒数第二个后缀最大值点，然后在最小值栈上二分出大于该点的最小的 l，即使对当前 r 最长的区间。复杂度 O(n log n)。

```cpp
#include <cstdio>
#include <algorithm>

const int maxn = 100005;

int n, ans, tx, tn;
int a[maxn], sx[maxn], sn[maxn];

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; ++i) scanf("%d", a + i);
  for (int i = 1; i <= n; ++i) {
    while (tn && a[sn[tn]] >= a[i]) --tn;
    while (tx && a[sx[tx]] < a[i]) --tx;
    int k = std::upper_bound(sn + 1, sn + 1 + tn, sx[tx]) - sn;
    if (k != (tn + 1)) {
      ans = std::max(ans, i - sn[k] + 1);
    }
    sn[++tn] = i;
    sx[++tx] = i;
  }
  printf("%d\n", ans);
  return 0;
}
```

求区间后缀最大值

给定一个数列 a，对于 a 中每个长度为 k 的子区间，求这个子区间里的构成的数列的后缀最大值个数。

要求线性。

求区间后缀最大值

考虑之前单调栈求前缀的后缀最大值做法。

注意到从栈底（注意不是栈顶）一侧删除元素其实不会破坏整个容器的性质。

为了能从栈底一侧删除元素，我们把 stack 改为双端队列。

然后扫描时判一下队列头是否超出区间，如果超出就 pop_front 即可。

P2627 修剪草坪

给定一个长度为 n 的非负整数列 a，要求选择 a 的一个子列，使得子列和最大，且这个子列没有连续 k 个数在原数列中是连续的。

N = 1e5

P2627 修剪草坪

容易设计一个 O(n^2) 的 dp：设 f_i 表示考虑了前 i 个数的最优答案。则 [i – k, i - 1] 中一定有一个是不能选的。考虑枚举这个断点 j:

$$f_i = \max\{f_{j-1} + s_i - s_j\}$$

这里 s 是数列前缀和。

注意到 s_i 与 j 无关，我们将之提到外面

$$f_i = s_i + \max_{j=i-k}^{i-1}\{f_j - s_j\}$$

如果令 $g_i = f_i - s_i$，则我们转移的第二项就是求 [i-k,i-1] 这个区间内 g 的最大值。

我们已经会维护区间所有的后缀最大值了。整个区间的最大值显然就是最前面的一个后缀最大值。

所以用单调队列维护这个转移即可。

说明

对于今天所讲授的内容，请以掌握 vector queue stack list 四个容器为主，其余内容（包括堆和单调数据结构的理论）可以以后再学。

今天晚上有第二课堂。