基础算法选讲

徐沐杰

南京大学

2023 年 7 月 7 日

目录

目录

一点数学

- $\in$：属于，$[n] = \{1, 2, 3, \cdots, n\}$，$\sum$：求和。

一点数学

- $\in$：属于，$[n] = \{1, 2, 3, \cdots, n\}$，$\sum$：求和。
- $\sum_{i \in [n]} a_i = a_1 + \cdots + a_n, \sum_{i=l}^{r} = a_l + \cdots + a_r$。

一点数学

- $\in$：属于，$[n] = \{1, 2, 3, \cdots, n\}$，$\sum$：求和。
- $\sum_{i \in [n]} a_i = a_1 + \cdots + a_n, \sum_{i=l}^{r} = a_l + \cdots + a_r$。
- $\forall$：对于任意的，$\exists$：存在一个。

一点数学

- $\in$：属于，$[n] = \{1, 2, 3, \cdots, n\}$，$\sum$：求和。
- $\sum_{i \in [n]} a_i = a_1 + \cdots + a_n, \sum_{i=l}^{r} = a_l + \cdots + a_r$。
- $\forall$：对于任意的，$\exists$：存在一个。
- $\log_a b = x \leftrightarrow a^x = b$。

一点数学

- $\in$：属于，$[n] = \{1, 2, 3, \cdots, n\}$，$\sum$：求和。
- $\sum_{i \in [n]} a_i = a_1 + \cdots + a_n$，$\sum_{i=l}^{r} = a_l + \cdots + a_r$。
- $\forall$：对于任意的，$\exists$：存在一个。
- $\log_a b = x \leftrightarrow a^x = b$。
- $f(n) = O(n \log n)$ 的意思是，

$$\exists c,\ f(n) \leq c \cdot n \log_2(n)$$

一点数学

- $\in$：属于，$[n] = \{1, 2, 3, \cdots, n\}$，$\sum$：求和。
- $\sum_{i \in [n]} a_i = a_1 + \cdots + a_n$, $\sum_{i=l}^{r} = a_l + \cdots + a_r$。
- $\forall$：对于任意的，$\exists$：存在一个。
- $\log_a b = x \leftrightarrow a^x = b$。
- $f(n) = O(n \log n)$ 的意思是，

$$\exists c,\ f(n) \leq c \cdot n \log_2(n)$$

- 一般地，$f(n) = O(g(n))$ 的意思是，

$$\exists c,\ f(n) \leq c \cdot g(n)$$

一点数学

- $\in$：属于，$[n] = \{1, 2, 3, \cdots, n\}$，$\sum$：求和。
- $\sum_{i \in [n]} a_i = a_1 + \cdots + a_n$, $\sum_{i=l}^{r} = a_l + \cdots + a_r$。
- $\forall$：对于任意的，$\exists$：存在一个。
- $\log_a b = x \leftrightarrow a^x = b$。
- $f(n) = O(n \log n)$ 的意思是，

$$\exists c,\ f(n) \leq c \cdot n \log_2(n)$$

- 一般地，$f(n) = O(g(n))$ 的意思是，

$$\exists c,\ f(n) \leq c \cdot g(n)$$

- 后文将使用 O（渐进复杂度上界）来表示时间、空间复杂度。

STL 容器

- 大部分时候，算法竞赛选手只需要会写 C with Class，较少接触 Cpp 高级特性。

STL 容器

- 大部分时候，算法竞赛选手只需要会写 C with Class，较少接触 Cpp 高级特性。
- 虽然但是，Cpp 提供的 STL 模板库接口是不用白不用的。

STL 容器

- 大部分时候，算法竞赛选手只需要会写 C with Class，较少接触 Cpp 高级特性。
- 虽然但是，Cpp 提供的 STL 模板库接口是不用白不用的。
- ~~如果细讲 Cpp 面向对象特性一个小时估计都讲不完~~，所以这里只简单介绍 STL 容器的使用。

STL 容器

- 大部分时候，算法竞赛选手只需要会写 C with Class，较少接触 Cpp 高级特性。
- 虽然但是，Cpp 提供的 STL 模板库接口是不用白不用的。
- ~~如果细讲 Cpp 面向对象特性一个小时估计都讲不完，所以这里只简单介绍 STL 容器的使用。~~
- STL 容器一般都支持多种类型，以模板的形式存在，使用是如 `vector<int> xx` 等。

STL 容器

- 大部分时候，算法竞赛选手只需要会写 C with Class，较少接触 Cpp 高级特性。
- 虽然但是，Cpp 提供的 STL 模板库接口是不用白不用的。
- ~~如果细讲 Cpp 面向对象特性一个小时估计都讲不完~~，所以这里只简单介绍 STL 容器的使用。
- STL 容器一般都支持多种类型，以模板的形式存在，使用是如 `vector<int> xx` 等。
- 如果没有显式的指明所占空间，STL 容器的空间一般都是动态分配的，用多少占多少。

STL 容器

- 大部分时候，算法竞赛选手只需要会写 C with Class，较少接触 Cpp 高级特性。
- 虽然但是，Cpp 提供的 STL 模板库接口是不用白不用的。
- ~~如果细讲 Cpp 面向对象特性一个小时估计都讲不完~~，所以这里只简单介绍 STL 容器的使用。
- STL 容器一般都支持多种类型，以模板的形式存在，使用是如 `vector<int> xx` 等。
- 如果没有显式的指明所占空间，STL 容器的空间一般都是动态分配的，用多少占多少。
- `xx.size()`，`xx.empty()`，`xx.begin()`，`xx.end()` 等，这些一般是 STL 容器通用的方法，意义如名字。后两个是迭代器，我们马上就会讲到。

STL 容器

- 大部分时候，算法竞赛选手只需要会写 C with Class，较少接触 Cpp 高级特性。
- 虽然但是，Cpp 提供的 STL 模板库接口是不用白不用的。
- ~~如果细讲 Cpp 面向对象特性一个小时估计都讲不完~~，所以这里只简单介绍 STL 容器的使用。
- STL 容器一般都支持多种类型，以模板的形式存在，使用是如 `vector<int> xx` 等。
- 如果没有显式的指明所占空间，STL 容器的空间一般都是动态分配的，用多少占多少。
- `xx.size()`，`xx.empty()`，`xx.begin()`，`xx.end()` 等，这些一般是 STL 容器通用的方法，意义如名字。后两个是迭代器，我们马上就会讲到。
- STL 还有很多实用的算法，但时间所限这里不多赘述。这里推荐参考 OI Wiki 的相关内容 。

vector 和迭代器

- `vector` 可能是大部分人第一个接触的 STL 容器，它的名字直译为向量，但实际上表现的像个「动态数组」，它支持随机访问和在末尾添加元素。

vector 和迭代器

- `vector` 可能是大部分人第一个接触的 STL 容器，它的名字直译为向量，但实际上表现的像个「动态数组」，它支持随机访问和在末尾添加元素。
- 同时，STL 容器提供了类似 C 中指针的迭代器来提供容器的遍历功能。

vector 和迭代器

- `vector` 可能是大部分人第一个接触的 STL 容器，它的名字直译为向量，但实际上表现的像个「动态数组」，它支持随机访问和在末尾添加元素。
- 同时，STL 容器提供了类似 C 中指针的迭代器来提供容器的遍历功能。
- 迭代器也可以作为 STL 通用算法的参数（如 `sort`）和返回值（如 `lower_bound`）。

vector 和迭代器

- `vector` 可能是大部分人第一个接触的 STL 容器，它的名字直译为向量，但实际上表现的像个「动态数组」，它支持随机访问和在末尾添加元素。
- 同时，STL 容器提供了类似 C 中指针的迭代器来提供容器的遍历功能。
- 迭代器也可以作为 STL 通用算法的参数（如 `sort`）和返回值（如 `lower_bound`）。
- 顺口一提，STL 有序容器一般要求模板类型具有 < 重载运算符，或者在模板参数中给定排序谓词（重载了 () 的伪函数），推荐使用重载运算符的方法。

vector 和迭代器

```cpp
//end() 表示序列的「末尾的后继」
vector<int> vec = {6, 9, 3, 2, 1};
sort(vec.begin(), vec.end());
auto it = vec.begin();
vec.push_back(8);
//迭代器
for (; it != vec.end(); it++)
  cout << *it;
//直接下标访问
for (int i = 0; i < vec.size(); i++)
  cout << vec[i];
//CXX11
for (auto x : vec) cout << x;
```

set 与 unordered_set

- 维护一个集合，支持插入、删除、查找元素。

set 与 unordered_set

- 维护一个集合，支持插入、删除、查找元素。
- 其中有序的 `set` 可以动态的维护一个有序的序列（实质上是红黑树），并支持使用 `lower_bound` 等查找元素。

set 与 unordered_set

- 维护一个集合，支持插入、删除、查找元素。
- 其中有序的 `set` 可以动态的维护一个有序的序列（实质上是红黑树），并支持使用 `lower_bound` 等查找元素。
- `set` 单次操作的代价是 $O(\log n)$，`unordered_set` 单次操作的代价均摊 $O(1)$。

set 与 unordered_set

- 维护一个集合，支持插入、删除、查找元素。
- 其中有序的 `set` 可以动态的维护一个有序的序列（实质上是红黑树），并支持使用 `lower_bound` 等查找元素。
- `set` 单次操作的代价是 $O(\log n)$，`unordered_set` 单次操作的代价均摊 $O(1)$。

```
1  set<int> s;
2  s.insert(5);
3  s.insert(8);
4  auto it = s.find(5);
5  assert(it != s.end());   //5 在集合中存在
6  assert(*it == 5);        //集合迭代器可以得到元素值
7  s.erase(it);             //可以通过迭代器或值删除
8  s.erase(8);
```

map 与 unordered_map

- “可以再装一个值”的 `set`。

map 与 unordered_map

- “可以再装一个值”的 `set`。
- 其中有序的 `map` 可以动态的维护一个有序的键-值对（实质上是红黑树），并支持使用 `[]` 运算符根据键查找元素。

map 与 unordered_map

- “可以再装一个值”的 `set`。
- 其中有序的 `map` 可以动态的维护一个有序的键-值对（实质上是红黑树），并支持使用 `[]` 运算符根据键查找元素。
- `map` 单次操作的代价是 $O(\log n)$，`unordered_map` 单次操作的代价均摊 $O(1)$。

map 与 unordered_map

- “可以再装一个值”的 set。
- 其中有序的 map 可以动态的维护一个有序的键-值对（实质上是红黑树），并支持使用 [] 运算符根据键查找元素。
- map 单次操作的代价是 $O(\log n)$，unordered_map 单次操作的代价均摊 $O(1)$。

```
map<string, int> m;
//两种方法均可创建元素
m.insert(make_pair("Bardi", 8));
m["Litre"] = 9;
//两种方法均可访问元素
assert(m["Bardi"] == 8);
assert(m.find("Litre")->second == 9);
//两种方法均可删除元素
m.erase(m.find("Litre"));
m.erase("Bardi");
```

queue 与 priority_queue

- `queue` 维护一个先进先出的队列，`priority_queue` 维护一个优先队列（队头的是优先级最高的元素），本质是二叉大根堆。

queue 与 priority_queue

- `queue` 维护一个先进先出的队列，`priority_queue` 维护一个优先队列（队头的是优先级最高的元素），本质是二叉大根堆。
- `priority_queue` 单次操作的代价是 $O(\log n)$。

queue 与 priority_queue

- queue 维护一个先进先出的队列，priority_queue 维护一个优先队列（队头的是优先级最高的元素），本质是二叉大根堆。
- priority_queue 单次操作的代价是 $O(\log n)$。

```
priority_queue<int> pq;          //优先队列
queue<float> q;                  //FIFO 队列
q.push(5.9); q.push(3.8);        //往队尾加数
q.pop();                         //弹出队头
assert(q.front() == 3.8);        //访问队头
pq.push(4); pq.push(7);
assert(pq.top() == 7);           //默认访问优先级最高
pq.pop();                        //弹出的是 7
```

bitset

- 简单理解：一个很快的压位布尔数组。

bitset

- 简单理解：一个很快的压位布尔数组。
- 原理：把 64 位布尔值压进了一个加长整型数中。

bitset

- 简单理解：一个很快的压位布尔数组。
- 原理：把 64 位布尔值压进了一个加长整型数中。
- 应用举例：埃氏筛法。

bitset

- 简单理解：一个很快的压位布尔数组。
- 原理：把 64 位布尔值压进了一个加长整型数中。
- 应用举例：埃氏筛法。

```
1  bitset<N> vis;
2  void Prime(int n) {
3    vis.set(); //全设为 1
4    //像操作布尔数组那样操作即可
5    vis[0] = vis[1] = 0;
6    for (int i = 2; i * i <= n; i++)
7      if (vis[i])
8        for (int j = i * i; j <= n; j += i)
9          vis[j] = 0;
10 }
```

目录

离散化：概念和实现

- 对于 n 个可能很大的数，赋给 $[len]$ 中的值，其中 len 是这些数中不同数的数目，且仍然保留原有的大小关系。

离散化：概念和实现

- 对于 n 个可能很大的数，赋给 $[len]$ 中的值，其中 len 是这些数中不同数的数目，且仍然保留原有的大小关系。
- 一般是用在要求将某个值很大的属性用数组下标进行索引的情景(同时需要保留大小关系)。

离散化：概念和实现

- 对于 n 个可能很大的数，赋给 $[len]$ 中的值，其中 len 是这些数中不同数的数目，且仍然保留原有的大小关系。
- 一般是用在要求将某个值很大的属性用数组下标进行索引的情景(同时需要保留大小关系)。
- 只适用于离线情形，在线的话可以使用 `map` 进行索引，如果对顺序没有要求，也可以使用 `unordered_map`。

离散化：概念和实现

- 对于 n 个可能很大的数，赋给 $[len]$ 中的值，其中 len 是这些数中不同数的数目，且仍然保留原有的大小关系。
- 一般是用在要求将某个值很大的属性用数组下标进行索引的情景 (同时需要保留大小关系)。
- 只适用于离线情形，在线的话可以使用 `map` 进行索引，如果对顺序没有要求，也可以使用 `unordered_map`。

```
1  // a 为初始数组，下标范围为 [1, n]
2  // len 为离散化后数组的有效长度
3  sort(a + 1, a + 1 + n);
4  //离散化整个数组，并求出离散化后不同数的个数。
5  len = unique(a + 1, a + n + 1) - a - 1;
6  // b 是初始数组的一份拷贝
7  for (int i = 1; i <= n; i++)
8    b[i] = lower_bound(a + 1, a + len + 1, b[i]) - a;
```

目录

前缀和：引子

例 (Presum-0)

有长度为 n 的整数序列 a 和 q 个询问，每次给出一个区间 $[l, r]$，要求分别给出区间内所有数的平均数和方差。

前缀和：引子

例 (Presum-0)

有长度为 n 的整数序列 a 和 q 个询问，每次给出一个区间 $[l, r]$，要求分别给出区间内所有数的平均数和方差。

解 (Presum-0)

- 我们知道方差就是平均数的平方减去平方的平均数，所以只需分别求 $\sum_{i=l}^{r} a_i$ 和 $\sum_{i=l}^{r} a_i^2$ 的值就好。我们发现这完全可以递推，也就是

$$S_i = \sum_{j \in [i]} a_i \to S_i = S_{i-1} + a_i \to \sum_{i=l}^{r} a_i = S_r - S_{l-1}$$

前缀和：引子

例 (Presum-0)

有长度为 n 的整数序列 a 和 q 个询问，每次给出一个区间 $[l, r]$，要求分别给出区间内所有数的平均数和方差。

解 (Presum-0)

- 我们知道方差就是平均数的平方减去平方的平均数，所以只需分别求 $\sum_{i=l}^{r} a_i$ 和 $\sum_{i=l}^{r} a_i^2$ 的值就好。我们发现这完全可以递推，也就是

$$S_i = \sum_{j \in [i]} a_i \to S_i = S_{i-1} + a_i \to \sum_{i=l}^{r} a_i = S_r - S_{l-1}$$

- 求平方和同理。

前缀和：引子

例 (Presum-0)

有长度为 n 的整数序列 a 和 q 个询问，每次给出一个区间 $[l, r]$，要求分别给出区间内所有数的平均数和方差。

解 (Presum-0)

- 我们知道方差就是平均数的平方减去平方的平均数，所以只需分别求 $\sum_{i=l}^{r} a_i$ 和 $\sum_{i=l}^{r} a_i^2$ 的值就好。我们发现这完全可以递推，也就是

$$S_i = \sum_{j \in [i]} a_i \to S_i = S_{i-1} + a_i \to \sum_{i=l}^{r} a_i = S_r - S_{l-1}$$

- 求平方和同理。
- 这就是**前缀和**，可以被用来维护有逆运算下的区间信息。

前缀和：推广

- 二维前缀和：如果已知 $(1,1)-(i,j)$ 正方形的和 $S_{i,j}$，那么有 $\sum_{i=l_1}^{r_1}\sum_{j=l_2}^{r_2} a_{ij} = S_{r_1,r_2} - S_{l_1-1,r_2} - S_{r_1,l_2-1} + S_{l_1-1,l_2-1}$

前缀和：推广

- 二维前缀和：如果已知 $(1,1)-(i,j)$ 正方形的和 $S_{i,j}$，那么有 $\sum_{i=l_1}^{r_1}\sum_{j=l_2}^{r_2} a_{ij} = S_{r_1,r_2} - S_{l_1-1,r_2} - S_{r_1,l_2-1} + S_{l_1-1,l_2-1}$
- 树上前缀和：如果已知 l, r 的最近公共祖先 x，那么我们可以通过维护每个节点到根的前缀和 S_i 来计算 l, r 路径上的和，即，$S_l + S_r - S_x + a_x$。

前缀和：推广

- 二维前缀和：如果已知 $(1,1)-(i,j)$ 正方形的和 $S_{i,j}$，那么有 $\sum_{i=l_1}^{r_1}\sum_{j=l_2}^{r_2} a_{ij} = S_{r_1,r_2} - S_{l_1-1,r_2} - S_{r_1,l_2-1} + S_{l_1-1,l_2-1}$
- 树上前缀和：如果已知 l, r 的最近公共祖先 x，那么我们可以通过维护每个节点到根的前缀和 S_i 来计算 l, r 路径上的和，即，$S_l + S_r - S_x + a_x$。
- 结合 DFS 序（暂时超纲），可以计算子树前缀和。

前缀和：推广

例 (Presum-1)

有长度为 n 的整数序列 a。找到其中 $n-1$ 个数，使得它们的最大公约数最大。

前缀和：推广

例 (Presum-1)

有长度为 n 的整数序列 a。找到其中 $n-1$ 个数，使得它们的最大公约数最大。

解 (Presum-1)

考虑到一定只有一个数没被选上，所以我们选择做一次前缀 gcd，一次后缀 gcd，任何 $n-1$ 个数一定是一个前缀拼上一个后缀，计算 n 次 gcd 就能算出所有 $n-1$ 个数的组合的 gcd 了。

前缀和：推广

例 (Presum-1)

有长度为 n 的整数序列 a。找到其中 $n-1$ 个数，使得它们的最大公约数最大。

解 (Presum-1)

考虑到一定只有一个数没被选上，所以我们选择做一次前缀 gcd，一次后缀 gcd，任何 $n-1$ 个数一定是一个前缀拼上一个后缀，计算 n 次 gcd 就能算出所有 $n-1$ 个数的组合的 gcd 了。

可以看出，前缀和可以被推广到大部分可结合的运算上，但当此运算没有逆时，维护区间信息困难了不少。

差分：引子

例 (Presum-2)

有长度为 n 的整数序列 a 和 q 个操作，每次给出一个区间 $[l, r]$ 和一个操作数 k，要求将区间中所有数加上 k，最后输出操作完后的序列。

差分：引子

例 (Presum-2)

有长度为 n 的整数序列 a 和 q 个操作，每次给出一个区间 $[l, r]$ 和一个操作数 k，要求将区间中所有数加上 k，最后输出操作完后的序列。

解 (Presum-2)

我们定义新序列 b，

$$b_i = \begin{cases} a_i - a_{i-1} & i > 1 \\ a_1 & i = 1 \end{cases}$$

容易发现 $a_i = \sum_{j \in [i]} b_j$，这就是说，**差分**是前缀和的逆。

差分：引子

例 (Presum-2)

有长度为 n 的整数序列 a 和 q 个操作，每次给出一个区间 $[l, r]$ 和一个操作数 k，要求将区间中所有数加上 k，最后输出操作完后的序列。

解 (Presum-2)

我们定义新序列 b，

$$b_i = \begin{cases} a_i - a_{i-1} & i > 1 \\ a_1 & i = 1 \end{cases}$$

容易发现 $a_i = \sum_{j \in [i]} b_j$，这就是说，**差分**是前缀和的逆。
此时，对于 $[l, r]$ 的区间加操作相当于把 b_l 加上 k，b_{r+1}（如果存在）减去 k，因为最后只统一查询一次，所以重新前缀和恢复序列即可。

差分：来道小学奥数

例 (Presum-3)

给一个长度为 n 的 01 串和一个整数 k，每次你可以翻转连续 k 位，请问至少需要多少次把 01 串变成全零？

差分：来道小学奥数

例 (Presum-3)

给一个长度为 n 的 01 串和一个整数 k，每次你可以翻转连续 k 位，请问至少需要多少次把 01 串变成全零？

解 (Presum-3)

- 差分同样可以用在非加法的运算上，只要该运算有逆。

差分：来道小学奥数

例 (Presum-3)

给一个长度为 n 的 01 串和一个整数 k，每次你可以翻转连续 k 位，请问至少需要多少次把 01 串变成全零？

解 (Presum-3)

- 差分同样可以用在非加法的运算上，只要该运算有逆。
- 这个例子想说明差分可以方便地能够转化序列问题。

差分：来道小学奥数

例 (Presum-3)

给一个长度为 n 的 01 串和一个整数 k，每次你可以翻转连续 k 位，请问至少需要多少次把 01 串变成全零？

解 (Presum-3)

- 差分同样可以用在非加法的运算上，只要该运算有逆。
- 这个例子想说明差分可以方便地能够转化序列问题。
- 序列全零意味着差分也全零，但是序列翻转在差分中意味着翻转距离为 k 的两个数。

差分：来道小学奥数

例 (Presum-3)

给一个长度为 n 的 01 串和一个整数 k，每次你可以翻转连续 k 位，请问至少需要多少次把 01 串变成全零？

解 (Presum-3)

- 差分同样可以用在非加法的运算上，只要该运算有逆。
- 这个例子想说明差分可以方便地能够转化序列问题。
- 序列全零意味着差分也全零，但是序列翻转在差分中意味着翻转距离为 k 的两个数。
- 接下来只要按照 $\bmod k$ 将各位分组，组内 1 的个数为奇数的显然无解。此后我们注意到，翻转相邻的两个点相当于在前缀和中翻转左边的那一位。

差分：来道小学奥数

例 (Presum-3)

给一个长度为 n 的 01 串和一个整数 k，每次你可以翻转连续 k 位，请问至少需要多少次把 01 串变成全零？

解 (Presum-3)

- 差分同样可以用在非加法的运算上，只要该运算有逆。
- 这个例子想说明差分可以方便地能够转化序列问题。
- 序列全零意味着差分也全零，但是序列翻转在差分中意味着翻转距离为 k 的两个数。
- 接下来只要按照 $\bmod k$ 将各位分组，组内 1 的个数为奇数的显然无解。此后我们注意到，翻转相邻的两个点相当于在前缀和中翻转左边的那一位。
- 于是对整组进行前缀和，算出 1 的个数即是该组的贡献。

差分：来道小学奥数

例 (Presum-3)

给一个长度为 n 的 01 串和一个整数 k，每次你可以翻转连续 k 位，请问至少需要多少次把 01 串变成全零？

解 (Presum-3)

- 差分同样可以用在非加法的运算上，只要该运算有逆。
- 这个例子想说明差分可以方便地能够转化序列问题。
- 序列全零意味着差分也全零，但是序列翻转在差分中意味着翻转距离为 k 的两个数。
- 接下来只要按照 $\bmod k$ 将各位分组，组内 1 的个数为奇数的显然无解。此后我们注意到，翻转相邻的两个点相当于在前缀和中翻转左边的那一位。
- 于是对整组进行前缀和，算出 1 的个数即是该组的贡献。
- 注：需要在原序列最后加一个 0，防止差分右端点超限。

目录

二分和二分答案

例 (Binary-0)

输入一个长度为 n 的序列 a。给出 q 个询问，每次给出一个数 u，问 a 中最大的小于等于 u 的数是多少。

二分和二分答案

例 (Binary-0)

输入一个长度为 n 的序列 a。给出 q 个询问，每次给出一个数 u，问 a 中最大的小于等于 u 的数是多少。

解 (Binary-0)

- 放个我自己用的二分模板。

二分和二分答案

例 (Binary-0)

输入一个长度为 n 的序列 a。给出 q 个询问，每次给出一个数 u，问 a 中最大的小于等于 u 的数是多少。

解 (Binary-0)

- 放个我自己用的二分模板。

```
long long l = 0,r = 5e16,ans = -1;
while (l <= r) {
  long long mid = (l + r) >> 1;
  if(check(mid)){
    ans = mid;
    l = mid + 1;
  }
  else r = mid - 1;
}
```

二分和二分答案

- 可以看到只要满足 $check(mid) \to \forall v < mid, check(v)$ 的最大化问题，都可以使用二分答案解决。

二分和二分答案

- 可以看到只要满足 $check(mid) \to \forall v < mid, check(v)$ 的最大化问题，都可以使用二分答案解决。
- 最小化问题当然也一样，反过来而已。我们称这种性质为问题的 **单调性**。

二分和二分答案

- 可以看到只要满足 $check(mid) \to \forall v < mid, check(v)$ 的最大化问题，都可以使用二分答案解决。
- 最小化问题当然也一样，反过来而已。我们称这种性质为问题的 **单调性**。
- 在实际解决问题时，如果很难直接确定问题的优化策略，不妨观察一下问题的可行解是否满足单调性。

二分和二分答案

- 可以看到只要满足 $check(mid) \to \forall v < mid, check(v)$ 的最大化问题，都可以使用二分答案解决。
- 最小化问题当然也一样，反过来而已。我们称这种性质为问题的 **单调性**。
- 在实际解决问题时，如果很难直接确定问题的优化策略，不妨观察一下问题的可行解是否满足单调性。
- 二分的想法将在后面频繁地出现，尤其是倍增小节。

三分

例 (Binary-1)

输入一个长度为 n 的序列 a。保证序列 a 先增后减，求序列 a 的最小值。

三分

例 (Binary-1)

输入一个长度为 n 的序列 a。保证序列 a 先增后减，求序列 a 的最小值。

解 (Binary-1)

- 乍一看非常简单，隐式差分后数列肯定单调。在其上找到最后一个小于等于 0 的数不就是答案了？转化为上一题。

三分

例 (Binary-1)

输入一个长度为 n 的序列 a。保证序列 a 先增后减，求序列 a 的最小值。

解 (Binary-1)

- 乍一看非常简单，隐式差分后数列肯定单调。在其上找到最后一个小于等于 0 的数不就是答案了？转化为上一题。
- 你先别急。

三分

例 (Binary-1)

输入一个长度为 n 的序列 a。保证序列 a 先增后减，求序列 a 的最小值。

解 (Binary-1)

- 乍一看非常简单，隐式差分后数列肯定单调。在其上找到最后一个小于等于 0 的数不就是答案了？转化为上一题。
- 你先别急。

例 (Binary-2)

给定一个在 $[0, n]$ 上有定义且先增后减的连续函数 $f(x)$，求序列 $f(x)$ 在 $[0, n]$ 上的最小值（允许一定误差）。

三分

例 (Binary-1)

输入一个长度为 n 的序列 a。保证序列 a 先增后减，求序列 a 的最小值。

解 (Binary-1)

- 乍一看非常简单，隐式差分后数列肯定单调。在其上找到最后一个小于等于 0 的数不就是答案了？转化为上一题。
- 你先别急。

例 (Binary-2)

给定一个在 $[0, n]$ 上有定义且先增后减的连续函数 $f(x)$，求序列 $f(x)$ 在 $[0, n]$ 上的最小值（允许一定误差）。

- 阁下又该如何应对？每隔 ϵ 来个差分？这就是 **三分**。

三分

例 (Binary-2)

给定一个在 $[0, n]$ 上有定义且先增后减的连续函数 $f(x)$，求序列 $f(x)$ 在 $[0, n]$ 上的最小值（允许一定误差）。

三分

例 (Binary-2)

给定一个在 $[0, n]$ 上有定义且先增后减的连续函数 $f(x)$，求序列 $f(x)$ 在 $[0, n]$ 上的最小值（允许一定误差）。

- 取区间中点后，计算 $f(mid + \frac{\epsilon}{2}), f(mid - \frac{\epsilon}{2})$，于是：

三分

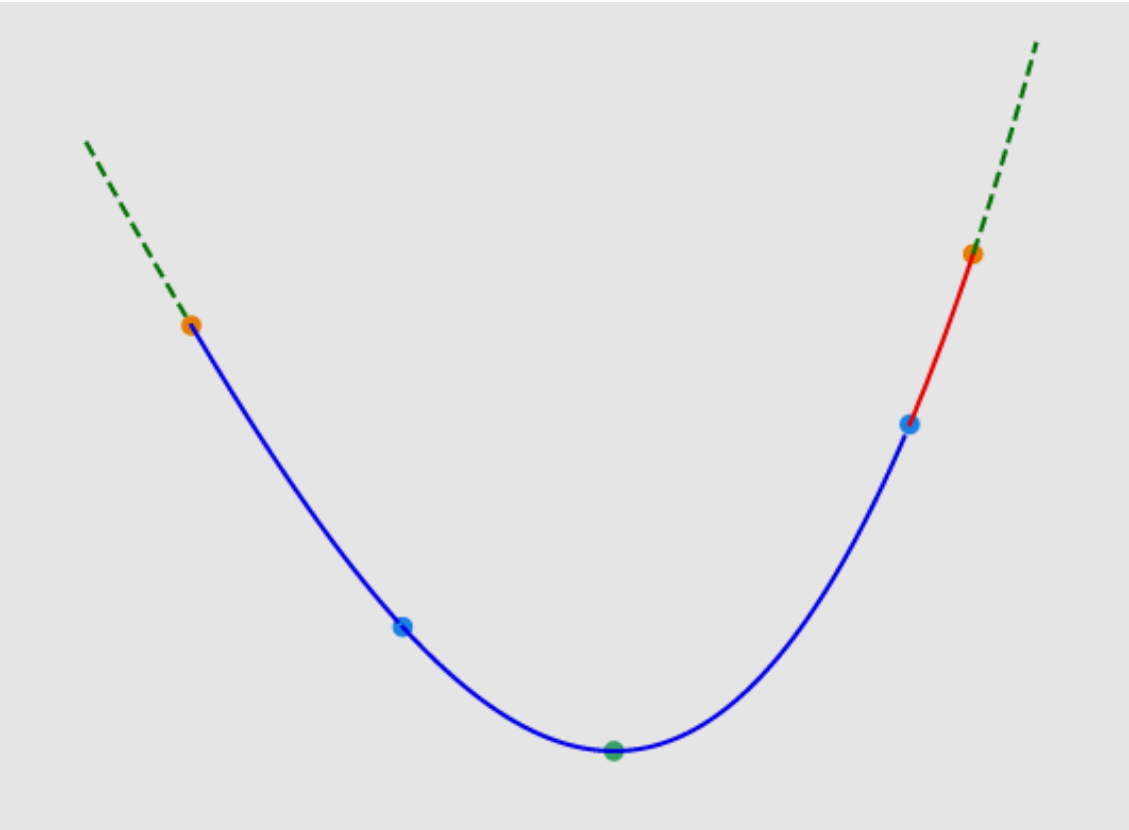
例 (Binary-2)

给定一个在 $[0, n]$ 上有定义且先增后减的连续函数 $f(x)$，求序列 $f(x)$ 在 $[0, n]$ 上的最小值（允许一定误差）。

- 取区间中点后，计算 $f(mid + \frac{\epsilon}{2}), f(mid - \frac{\epsilon}{2})$，于是：
- $f(mid + \frac{\epsilon}{2}) = f(mid - \frac{\epsilon}{2})$，答案在 $[mid - \epsilon, mid + \epsilon]$。
- $f(mid + \frac{\epsilon}{2}) < f(mid - \frac{\epsilon}{2})$，答案在 $[mid - \epsilon, r]$。
- $f(mid + \frac{\epsilon}{2}) > f(mid - \frac{\epsilon}{2})$，答案在 $[l, mid + \epsilon]$。

三分

- 取区间中点后，计算 $f(mid+\frac{\epsilon}{2}), f(mid-\frac{\epsilon}{2})$，于是：
- $f(mid+\frac{\epsilon}{2}) = f(mid-\frac{\epsilon}{2})$，答案在 $[mid-\epsilon, mid+\epsilon]$。
- $f(mid+\frac{\epsilon}{2}) < f(mid-\frac{\epsilon}{2})$，答案在 $[mid-\epsilon, r]$。
- $f(mid+\frac{\epsilon}{2}) > f(mid-\frac{\epsilon}{2})$，答案在 $[l, mid+\epsilon]$。



分数规划

例 (Binary-3)

给 n 个二元组 (s_i, p_i)，选取其中一个子集 K 使得 $\frac{\sum_{i\in K} s_i}{\sum_{i\in K} p_i}$ 最大。

分数规划

例 (Binary-3)

给 n 个二元组 (s_i, p_i)，选取其中一个子集 K 使得 $\frac{\sum_{i\in K} s_i}{\sum_{i\in K} p_i}$ 最大。

解 (Binary-3)

- 容易发现答案是满足单调性的。
- 考虑一个解 k 是否可行，只需有方案使得 $\sum_{i\in K} s_i \geq k\sum_{i\in K} p_i \to \sum_{i\in K}(s_i - kp_i) \geq 0$。
- 而这就是一个贪心问题了。

目录

倍增：引入

例 (Double-0)

给出两个数 a, n，计算 a^n。

倍增：引入

例 (Double-0)

给出两个数 a, n，计算 a^n。

解 (Double-0)

- 这就是所谓的快速幂问题。

倍增：引入

例 (Double-0)

给出两个数 a, n，计算 a^n。

解 (Double-0)

- 这就是所谓的快速幂问题。
- 我们知道一个数 n 总可以表示为 $O(\log n)$ 个 2 的次幂之和，具体的分解方法就是可以它的二进制表示获得。

倍增：引入

例 (Double-0)

给出两个数 a, n，计算 a^n。

解 (Double-0)

- 这就是所谓的快速幂问题。
- 我们知道一个数 n 总可以表示为 $O(\log n)$ 个 2 的次幂之和，具体的分解方法就是可以它的二进制表示获得。
- 对于任何可以结合的变化过程，要快速算出 n 步相同步骤之后的结果，都可以先计算出所有 2 的次幂步后的变化步骤，这样 n 步也可以在 $O(\log n)$ 次计算中完成。

```
1  int res = 1, a, n;
2  for (; n; a = a * a, n >>= 1) {
3    if (n & 1) res = res * a;
4  }
```

倍增：二分

例 (Double-1)

给出两个数 a, n，计算 $\lfloor \log_a n \rfloor$。
假设 n 非常大，但大整数乘法时间仍然设为 $O(1)$。

倍增：二分

例 (Double-1)

给出两个数 a, n，计算 $\lfloor \log_a n \rfloor$。
假设 n 非常大，但大整数乘法时间仍然设为 $O(1)$。

解 (Double-1)

- 答案可能很大，但是显然满足单调性质。

倍增：二分

例 (Double-1)

给出两个数 a, n，计算 $\lfloor \log_a n \rfloor$。
假设 n 非常大，但大整数乘法时间仍然设为 $O(1)$。

解 (Double-1)

- 答案可能很大，但是显然满足单调性质。
- 考虑二分答案 k，只需判断是否有 $a^k \leq n$。

倍增：二分

例 (Double-1)

给出两个数 a, n，计算 $\lfloor \log_a n \rfloor$。
假设 n 非常大，但大整数乘法时间仍然设为 $O(1)$。

解 (Double-1)

- 答案可能很大，但是显然满足单调性质。
- 考虑二分答案 k，只需判断是否有 $a^k \leq n$。
- 但即使用快速幂也不能在常数时间内计算出 a^k。

倍增：二分

例 (Double-1)

给出两个数 a, n，计算 $\lfloor \log_a n \rfloor$。
假设 n 非常大，但大整数乘法时间仍然设为 $O(1)$。

解 (Double-1)

- 答案可能很大，但是显然满足单调性质。
- 考虑二分答案 k，只需判断是否有 $a^k \leq n$。
- 但即使用快速幂也不能在常数时间内计算出 a^k。
- 考虑将二分点设为二的整数次幂而不是区间中点，仍然有 $O()$

倍增：二分

例 (Double-1)

给出两个数 a, n，计算 $\lfloor \log_a n \rfloor$。
假设 n 非常大，但大整数乘法时间仍然设为 $O(1)$。

倍增：二分

例 (Double-1)

给出两个数 a, n，计算 $\lfloor \log_a n \rfloor$。
假设 n 非常大，但大整数乘法时间仍然设为 $O(1)$。

```
1  //假设答案最多 2^maxpower - 1 大小
2  const int maxpower = 26;
3  //存有 a 的 2 次幂 次方的结果
4  //power_a[i] = a^(2^i)
5  int power_a[maxpower];
6  int now = 1, n, ans = 0;
7  for (int i = maxpower - 1; i >= 0; i--) {
8    // 二分点满足条件就到右区间
9    if (now * power_a[i] <= n)
10     now *= power_a[i], ans += (1 << i);
11 }
```

倍增：序列数据结构

例 (Double-2)

给出一个长度为 n 的环和一个常数 k，每次会从第 i 个点跳到第 $(i+k) \bmod n+1$ 个点，总共跳了 m 次。每个点都有一个权值，记为 a_i，求 m 次跳跃的起点的权值之和。

倍增：序列数据结构

例 (Double-2)

给出一个长度为 n 的环和一个常数 k，每次会从第 i 个点跳到第 $(i+k) \bmod n+1$ 个点，总共跳了 m 次。每个点都有一个权值，记为 a_i，求 m 次跳跃的起点的权值之和。

解 (Double-2)

- 可以对序列中第 i 点维护跳 2^j 步会跳到的点 $go_{i,j}$，以及权值和 $sum_{i,j}$，明显有递推关系 $go_{i,j} = go_{go_{i,j-1},j-1}$，$sum_{i,j} = sum_{i,j-1} + sum_{go_{i,j-1},j-1}$。

倍增：序列数据结构

例 (Double-2)

给出一个长度为 n 的环和一个常数 k，每次会从第 i 个点跳到第 $(i+k) \bmod n+1$ 个点，总共跳了 m 次。每个点都有一个权值，记为 a_i，求 m 次跳跃的起点的权值之和。

解 (Double-2)

- 可以对序列中第 i 点维护跳 2^j 步会跳到的点 $go_{i,j}$，以及权值和 $sum_{i,j}$，明显有递推关系 $go_{i,j} = go_{go_{i,j-1},j-1}$，$sum_{i,j} = sum_{i,j-1} + sum_{go_{i,j-1},j-1}$。
- 使用类似的方法，我们可以用这种序列结构维护很多前缀和因为没有逆而不能维护的区间信息，比如 **ST 表**。

倍增：序列数据结构

例 (Double-2)

给出一个长度为 n 的环和一个常数 k，每次会从第 i 个点跳到第 $(i+k) \bmod n+1$ 个点，总共跳了 m 次。每个点都有一个权值，记为 a_i，求 m 次跳跃的起点的权值之和。

解 (Double-2)

- 可以对序列中第 i 点维护跳 2^j 步会跳到的点 $go_{i,j}$，以及权值和 $sum_{i,j}$，明显有递推关系 $go_{i,j} = go_{go_{i,j-1},j-1}$，$sum_{i,j} = sum_{i,j-1} + sum_{go_{i,j-1},j-1}$。
- 使用类似的方法，我们可以用这种序列结构维护很多前缀和因为没有逆而不能维护的区间信息，比如 **ST 表**。
- ST 表其实是明天课程内容的一部分，但因为它和倍增的紧密联系以及实现过程的简易程度，我认为放在这个专题讲是很适合的。

倍增：树形数据结构

例 (Double-3)

给出一个树，每次询问给出两个点，要求求出这两个点的最近公共祖先。

倍增：树形数据结构

例 (Double-3)

给出一个树，每次询问给出两个点，要求求出这两个点的最近公共祖先。

解 (Double-3)

- 这是所谓 LCA 问题。

倍增：树形数据结构

例 (Double-3)

给出一个树，每次询问给出两个点，要求求出这两个点的最近公共祖先。

解 (Double-3)

- 这是所谓 *LCA* 问题。
- 令 $ances_{i,j}$ 表示第 i 个点的 2^j 级祖先，用同样方法递推即可。

倍增：树形数据结构

例 (Double-3)

给出一个树，每次询问给出两个点，要求求出这两个点的最近公共祖先。

解 (Double-3)

- 这是所谓 *LCA* 问题。
- 令 $ances_{i,j}$ 表示第 i 个点的 2^j 级祖先，用同样方法递推即可。
- 查询时，由于答案满足单调性，可使用倍增二分的方法计算。

倍增：树形数据结构

例 (Double-3)

给出一个树，每次询问给出两个点，要求求出这两个点的最近公共祖先。

解 (Double-3)

- 这是所谓 *LCA* 问题。
- 令 $ances_{i,j}$ 表示第 i 个点的 2^j 级祖先，用同样方法递推即可。
- 查询时，由于答案满足单调性，可使用倍增二分的方法计算。
- 预处理的复杂度是每个点 $O(\log n)$，单次询问复杂度亦然。

倍增：区间无逆信息维护

例 (Double-4)

有长度为 n 的 $k \times k$ 矩阵序列 a 和 q 个询问，每次给出一个区间 $[l, r]$，要求计算区间内所有矩阵的乘积。

倍增：区间无逆信息维护

例 (Double-4)

有长度为 n 的 $k \times k$ 矩阵序列 a 和 q 个询问，每次给出一个区间 $[l, r]$，要求计算区间内所有矩阵的乘积。

解 (Double-4)

- 关于矩阵乘法：可能没有逆，但满足结合律（幺半群）。单次复杂度 $O(k^3)$。

倍增：区间无逆信息维护

例 (Double-4)

有长度为 n 的 $k \times k$ 矩阵序列 a 和 q 个询问，每次给出一个区间 $[l, r]$，要求计算区间内所有矩阵的乘积。

解 (Double-4)

- 关于矩阵乘法：可能没有逆，但满足结合律（幺半群）。单次复杂度 $O(k^3)$。
- 令 $sum_{i,j}$ 表示 $[i, i + 2^j)$ 的积，用倍增二分的方法计算区间积（固定左端点，每次二分查找右端点的同时维护信息）。

倍增：区间无逆信息维护

例 (Double-4)

有长度为 n 的 $k \times k$ 矩阵序列 a 和 q 个询问，每次给出一个区间 $[l, r]$，要求计算区间内所有矩阵的乘积。

解 (Double-4)

- 关于矩阵乘法：可能没有逆，但满足结合律（幺半群）。单次复杂度 $O(k^3)$。
- 令 $sum_{i,j}$ 表示 $[i, i + 2^j)$ 的积，用倍增二分的方法计算区间积（固定左端点，每次二分查找右端点的同时维护信息）。
- 复杂度 $O(k^3(n + q) \log n)$。

倍增：区间可重无逆信息维护

例 (Double-5)

有长度为 n 的数列 a 和 q 个询问，每次给出一个区间 $[l, r]$，要求计算区间内所有数的最大值。

倍增：区间可重无逆信息维护

例 (Double-5)

有长度为 n 的数列 a 和 q 个询问，每次给出一个区间 $[l, r]$，要求计算区间内所有数的最大值。

解 (Double-5)

- 这是所谓 *RMQ* 问题。
  RMQ 问题相当经典，本次课程中会多次出现。

倍增：区间可重无逆信息维护

例 (Double-5)

有长度为 n 的数列 a 和 q 个询问，每次给出一个区间 $[l, r]$，要求计算区间内所有数的最大值。

解 (Double-5)

- 这是所谓 *RMQ* 问题。
  RMQ 问题相当经典，本次课程中会多次出现。
- 令 $premax_{i,j}$ 表示 $[i, i + 2^j)$ 的最大值，显然可以轻易递推。

倍增：区间可重无逆信息维护

例 (Double-5)

有长度为 n 的数列 a 和 q 个询问，每次给出一个区间 $[l, r]$，要求计算区间内所有数的最大值。

解 (Double-5)

- 这是所谓 *RMQ* 问题。
  RMQ 问题相当经典，本次课程中会多次出现。
- 令 $premax_{i,j}$ 表示 $[i, i + 2^j)$ 的最大值，显然可以轻易递推。
- 但此处查询可以 $O(1)$，因为最大值具有**可重性质**，即 $\max\{a_i, \cdots, a_i\} = \max\{a_i, \cdots\}$。

倍增：区间可重无逆信息维护

例 (Double-5)

有长度为 n 的数列 a 和 q 个询问，每次给出一个区间 $[l, r]$，要求计算区间内所有数的最大值。

解 (Double-5)

- 这是所谓 *RMQ* 问题。
  RMQ 问题相当经典，本次课程中会多次出现。
- 令 $premax_{i,j}$ 表示 $[i, i+2^j)$ 的最大值，显然可以轻易递推。
- 但此处查询可以 $O(1)$，因为最大值具有**可重性质**，即 $\max\{a_i, \cdots, a_i\} = \max\{a_i, \cdots\}$。
- 这意味着，可以从左右端点两端出发取一个 i，使得 $l \le r - 2^i + 1 \le l + 2^i - 1 \le r \to i = \lfloor \log_2(r - l + 1) \rfloor$，此对数可以 $O(n)$ 预处理。

倍增：区间可重无逆信息维护

例 (Double-5)

有长度为 n 的数列 a 和 q 个询问，每次给出一个区间 $[l, r]$，要求计算区间内所有数的最大值。

解 (Double-5)

- 这是所谓 *RMQ* 问题。
  RMQ 问题相当经典，本次课程中会多次出现。
- 令 $premax_{i,j}$ 表示 $[i, i+2^j)$ 的最大值，显然可以轻易递推。
- 但此处查询可以 $O(1)$，因为最大值具有**可重性质**，即 $\max\{a_i, \cdots, a_i\} = \max\{a_i, \cdots\}$。
- 这意味着，可以从左右端点两端出发取一个 i，使得 $l \le r - 2^i + 1 \le l + 2^i - 1 \le r \to i = \lfloor \log_2(r - l + 1) \rfloor$，此对数可以 $O(n)$ 预处理。
- 复杂度 $O(n \log n + q)$，这个数据结构就是 ***ST* 表**。

倍增：区间可重无逆信息维护

例 (Double-6)

有长度为 n 的数列 a 和 q 个询问，找到最大的区间 $[l, r]$，使得区间内的数的最大公约数不为 1。

倍增：区间可重无逆信息维护

例 (Double-6)

有长度为 n 的数列 a 和 q 个询问，找到最大的区间 $[l, r]$，使得区间内的数的最大公约数不为 1。

- 留作习题。

倍增：区间可重无逆信息维护

例 (Double-6)

有长度为 n 的数列 a 和 q 个询问，找到最大的区间 $[l, r]$，使得区间内的数的最大公约数不为 1。

- 留作习题。
- 提示：不讨论数论做法。这里最大公约数（gcd）满足可重性质，即 $gcd(a_i, \cdots, a_i) = gcd(a_i, \cdots)$。

倍增：区间可重无逆信息维护

例 (Double-6)

有长度为 n 的数列 a 和 q 个询问，找到最大的区间 $[l, r]$，使得区间内的数的最大公约数不为 1。

- 留作习题。
- 提示：不讨论数论做法。这里最大公约数（gcd）满足可重性质，即 $gcd(a_i, \cdots, a_i) = gcd(a_i, \cdots)$。
- 还是提示：结合倍增二分法。

目录

搜索：引子

例 (Search-0)

有三个容量为 a, b, c 升的杯子，最初只有第三个杯子里有 c 升水，杯子间可以互相倾倒，但由于杯子没有刻度，所以只能倒到某个杯子空或者某个杯子满为止，问操作任意次后任意杯子能达到的最接近 d 升的 $d' \leq d$ 升水是多少。$\max\{a, b, c\} \leq 200$。

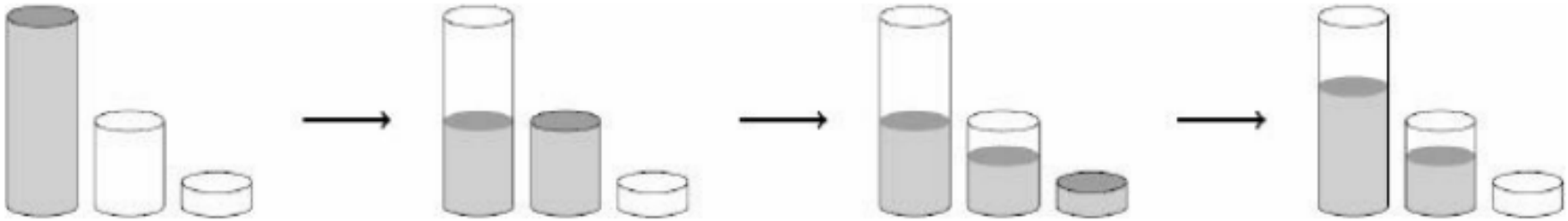


图7-15　倒水问题：一种方法是(6,0,0)→(3,3,0)→(3,2,1)→(4,2,0)

搜索：引子

例 (Search-0)

有三个容量为 a, b, c 升的杯子，最初只有第三个杯子里有 c 升水，杯子间可以互相倾倒，但由于杯子没有刻度，所以只能倒到某个杯子空或者某个杯子满为止，问操作任意次后任意杯子能达到的最接近 d 升的 $d' \leq d$ 升水是多少。$\max\{a, b, c\} \leq 200$。

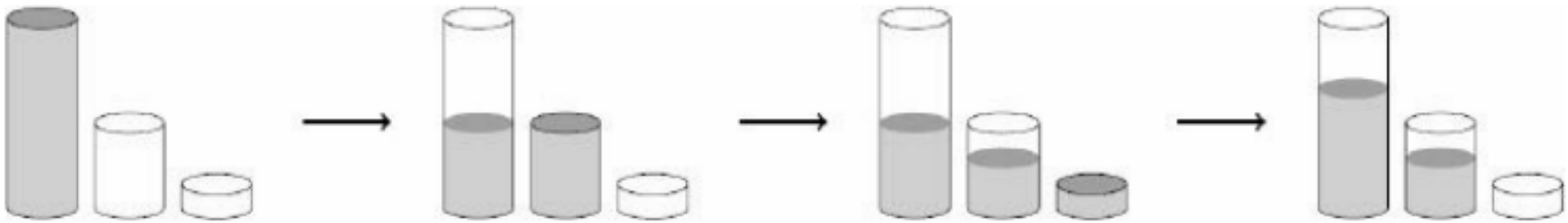


图7-15　倒水问题：一种方法是(6,0,0)→(3,3,0)→(3,2,1)→(4,2,0)

解 (Search-0)

用三个数（杯子里的水量）就可以完整表示问题的状态，由于和为定值，所以实际上只需要两个数。状态数仅 $201 * 201 = 40401$ 种。只需要枚举状态就可解决问题。(如何枚举?)

搜索：状态图

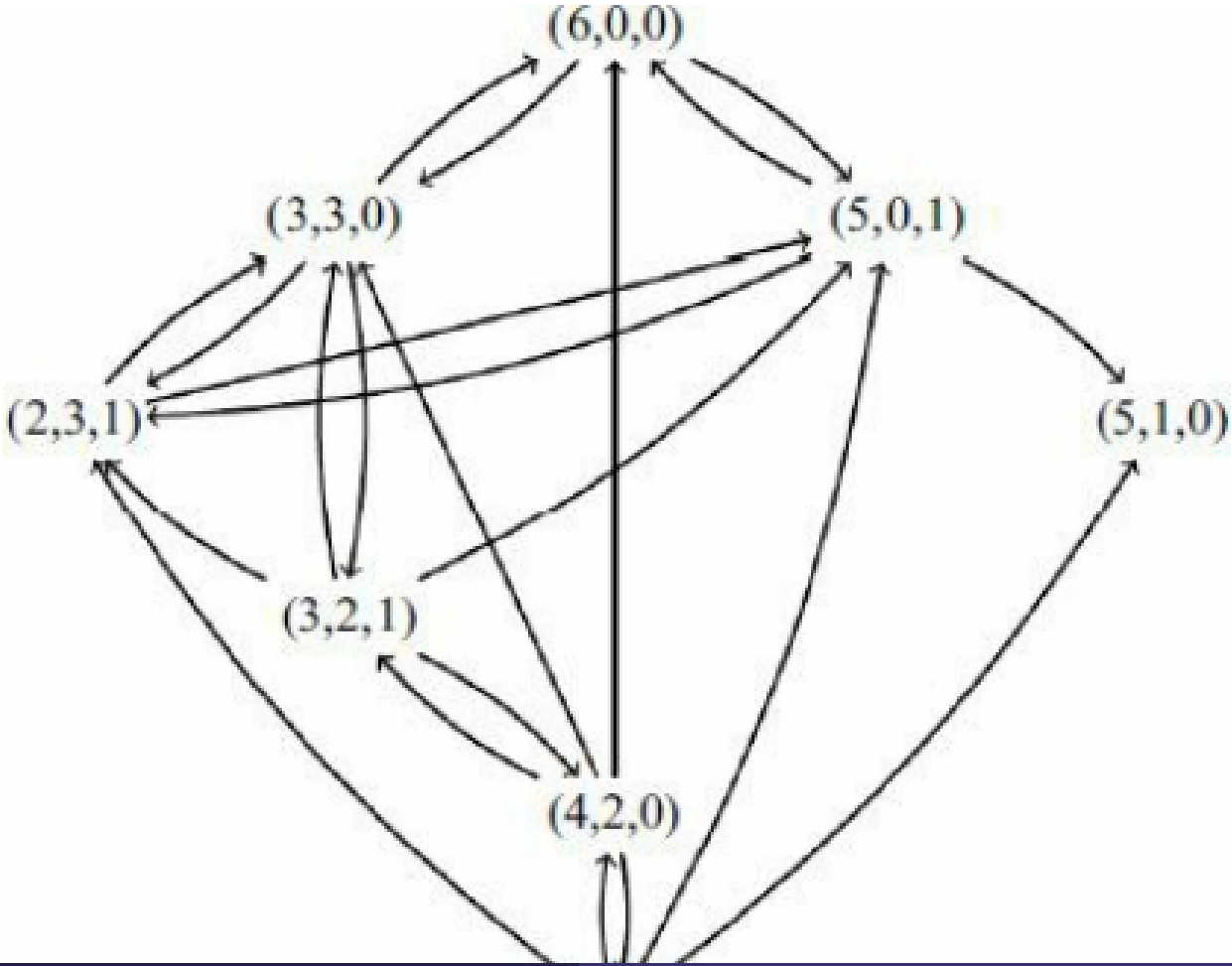
- 如果已知问题现在的状态，可以计算出“后继状态”，如果已知状态之间的转移关系，可以得到“状态图”。

搜索：状态图

- 如果已知问题现在的状态，可以计算出“后继状态”，如果已知状态之间的转移关系，可以得到“状态图”。
- 搜索的本质：遍历状态图。

搜索：状态图

- 如果已知问题现在的状态，可以计算出“后继状态”，如果已知状态之间的转移关系，可以得到“状态图”。
- 搜索的本质：遍历状态图。



搜索：Breath First or Depth First?

- 遍历状态图的策略：深度优先（DFS）和广度优先（BFS）。

搜索：Breath First or Depth First?

- 遍历状态图的策略：深度优先（DFS）和广度优先（BFS）。
- 深度优先：总是立即从遍历到的后继状态开始搜索，**递归**搜索完该状态后才考虑下一个后继状态。
- 广度优先：总是优先搜索最早被遍历到的状态，所以遍历后继状态时将它们放到**先进先出队列**中。

搜索：Breath First or Depth First?

```cpp
void bfs(type start) {
  queue<type> q;
  q.push(start);
  while (!q.empty()) {
    auto now = q.front(); q.pop();
    for (auto next : now.next())
      q.push(next);
  }
}
```

```cpp
void dfs(type now) {
  for (auto next : now.next())
    dfs(next);
}
```

搜索：记忆化

- 没有记忆的 BFS 和 DFS 会反复经过同一个状态，如果状态图上有环，DFS 甚至会陷入死循环。

搜索：记忆化

- 没有记忆的 BFS 和 DFS 会反复经过同一个状态，如果状态图上有环，DFS 甚至会陷入死循环。
- 如果状态数比较少，可以开全局数组记录下状态是否被访问过，**甚至是状态的答案**。

搜索：记忆化

- 没有记忆的 BFS 和 DFS 会反复经过同一个状态，如果状态图上有环，DFS 甚至会陷入死循环。
- 如果状态数比较少，可以开全局数组记录下状态是否被访问过，**甚至是状态的答案。**

例 (Search-1)

求出斐波那契数列第 x 项。

搜索：记忆化

- 没有记忆的 BFS 和 DFS 会反复经过同一个状态，如果状态图上有环，DFS 甚至会陷入死循环。
- 如果状态数比较少，可以开全局数组记录下状态是否被访问过，**甚至是状态的答案。**

例 (Search-1)

求出斐波那契数列第 x 项。

```
1  long long fib[35] = {1, 1, 2};
2  long long getfib(int x) {
3    // fib 数组也隐式存储了访问标记。
4    if (fib[x]) return fib[x];
5    return fib[x] = getfib(x-1) + getfib(x-2);
6  }
```

搜索：本质是一种暴力

各种局限

- **在状态少的情况下非常快**，因为可以开各种记忆化数组。

搜索：本质是一种暴力

各种局限

- **在状态少的情况下非常快**，因为可以开各种记忆化数组。
- 状态多 $\rightarrow$ 不能记忆化 $+$ 不止一个后继 $\rightarrow$ 指数级复杂度！

搜索：本质是一种暴力

各种局限

- **在状态少的情况下非常快**，因为可以开各种记忆化数组。
- 状态多 $\rightarrow$ 不能记忆化 $+$ 不止一个后继 $\rightarrow$ 指数级复杂度！
- 有环 $+$ 状态多 $\rightarrow$ 只能用迭代加深搜索（限制深度的 DFS）。

搜索：本质是一种暴力

各种局限

- **在状态少的情况下非常快**，因为可以开各种记忆化数组。
- 状态多 $\rightarrow$ 不能记忆化 $+$ 不止一个后继 $\rightarrow$ 指数级复杂度！
- 有环 $+$ 状态多 $\rightarrow$ 只能用迭代加深搜索（限制深度的 DFS）。
- 要实现最优搜索 $\rightarrow$ 只能 BFS 或迭代加深搜索。

搜索：本质是一种暴力

各种局限

- **在状态少的情况下非常快**，因为可以开各种记忆化数组。
- 状态多 $\rightarrow$ 不能记忆化 $+$ 不止一个后继 $\rightarrow$ 指数级复杂度！
- 有环 $+$ 状态多 $\rightarrow$ 只能用迭代加深搜索（限制深度的 DFS）。
- 要实现最优搜索 $\rightarrow$ 只能 BFS 或迭代加深搜索。

乱搞之路

- 下面主要讨论对指数级复杂度搜索的几种可能优化。

搜索：本质是一种暴力

各种局限

- **在状态少的情况下非常快**，因为可以开各种记忆化数组。
- 状态多 $\rightarrow$ 不能记忆化 $+$ 不止一个后继 $\rightarrow$ 指数级复杂度！
- 有环 $+$ 状态多 $\rightarrow$ 只能用迭代加深搜索（限制深度的 DFS）。
- 要实现最优搜索 $\rightarrow$ 只能 BFS 或迭代加深搜索。

乱搞之路

- 下面主要讨论对指数级复杂度搜索的几种可能优化。
- 可被优化问题的特征：数据范围看起来“就差一点”。

搜索：本质是一种暴力

各种局限

- **在状态少的情况下非常快**，因为可以开各种记忆化数组。
- 状态多 $\rightarrow$ 不能记忆化 $+$ 不止一个后继 $\rightarrow$ 指数级复杂度！
- 有环 $+$ 状态多 $\rightarrow$ 只能用迭代加深搜索（限制深度的 DFS）。
- 要实现最优搜索 $\rightarrow$ 只能 BFS 或迭代加深搜索。

乱搞之路

- 下面主要讨论对指数级复杂度搜索的几种可能优化。
- 可被优化问题的特征：数据范围看起来“就差一点”。
- 优化的手法：状态简化，花式剪枝，双向搜索，IDA*，...

搜索：八数码

例 (Search-2)

3×3 的棋盘上有一个空格和八个有标号棋子，棋子可以移动到四连通的空格上，问最少需要多少步达到目标状态。

搜索：八数码

例 (Search-2)

3×3 的棋盘上有一个空格和八个有标号棋子，棋子可以移动到四连通的空格上，问最少需要多少步达到目标状态。

- 我们发现状态不是很好编码。如果直接把九个数写下来需要开很大的 `vis` 数组，这导致空间超限。

搜索：八数码

例 (Search-2)

3×3 的棋盘上有一个空格和八个有标号棋子，棋子可以移动到四连通的空格上，问最少需要多少步达到目标状态。

- 我们发现状态不是很好编码。如果直接把九个数写下来需要开很大的 `vis` 数组，这导致空间超限。
- 柳暗花明：可以使用康托展开来编码排列。

搜索：八数码

例 (Search-2)

3×3 的棋盘上有一个空格和八个有标号棋子，棋子可以移动到四连通的空格上，问最少需要多少步达到目标状态。

- 我们发现状态不是很好编码。如果直接把九个数写下来需要开很大的 `vis` 数组，这导致空间超限。
- 柳暗花明：可以使用康托展开来编码排列。
- 或者直接使用 `unordered_map` 来保存状态访问情况。

搜索：八数码

例 (Search-2)

3×3 的棋盘上有一个空格和八个有标号棋子，棋子可以移动到四连通的空格上，问最少需要多少步达到目标状态。

- 我们发现状态不是很好编码。如果直接把九个数写下来需要开很大的 `vis` 数组，这导致空间超限。
- 柳暗花明：可以使用康托展开来编码排列。
- 或者直接使用 `unordered_map` 来保存状态访问情况。
- 当结束状态已知时可以使用双向 BFS：从结束状态和出发状态同时开始遍历状态图，当遇到相同的状态就停止搜索。

搜索：八数码

例 (Search-2)

3×3 的棋盘上有一个空格和八个有标号棋子，棋子可以移动到四连通的空格上，问最少需要多少步达到目标状态。

- 我们发现状态不是很好编码。如果直接把九个数写下来需要开很大的 `vis` 数组，这导致空间超限。
- 柳暗花明：可以使用康托展开来编码排列。
- 或者直接使用 `unordered_map` 来保存状态访问情况。
- 当结束状态已知时可以使用双向 BFS：从结束状态和出发状态同时开始遍历状态图，当遇到相同的状态就停止搜索。假设答案为 k，搜索 n 层需要的复杂度为 $O(f(n))$，则可以把复杂度从 $O(f(n))$ 变为 $O(f(\frac{n}{2}))$，由于搜索的复杂度函数一般都是指数函数，所以后者会从复杂度等级上优于前者。

搜索：迭代加深搜索

- BFS 需要开辟一个队列来存储本层/下一层的带搜索节点，这个队列很可能会占用过大的空间。

搜索：迭代加深搜索

- BFS 需要开辟一个队列来存储本层/下一层的带搜索节点，这个队列很可能会占用过大的空间。
- 迭代加深搜索使用多次限制深度的 DFS 来模拟 BFS。

搜索：迭代加深搜索

- BFS 需要开辟一个队列来存储本层/下一层的带搜索节点，这个队列很可能会占用过大的空间。
- 迭代加深搜索使用多次限制深度的 DFS 来模拟 BFS。
- 具体来说，迭代加深搜索从 1 开始，每次都把搜索深度增加 1，直到某次搜索搜到了想要的解为止。

搜索：迭代加深搜索

- BFS 需要开辟一个队列来存储本层/下一层的带搜索节点，这个队列很可能会占用过大的空间。
- 迭代加深搜索使用多次限制深度的 DFS 来模拟 BFS。
- 具体来说，迭代加深搜索从 1 开始，每次都把搜索深度增加 1，直到某次搜索搜到了想要的解为止。
- 这种方法以一定的时间代价换取了空间代价。

搜索：迭代加深搜索

- BFS 需要开辟一个队列来存储本层/下一层的带搜索节点，这个队列很可能会占用过大的空间。
- 迭代加深搜索使用多次限制深度的 DFS 来模拟 BFS。
- 具体来说，迭代加深搜索从 1 开始，每次都把搜索深度增加 1，直到某次搜索搜到了想要的解为止。
- 这种方法以一定的时间代价换取了空间代价。
- 假设搜索树的下一层的规模是上一层的两倍，时间代价仅翻了一倍，就把空间代价从 $O(2^d)$ 变成了 $O(d)$，d 为深度。

搜索：迭代加深搜索

- BFS 需要开辟一个队列来存储本层/下一层的带搜索节点，这个队列很可能会占用过大的空间。
- 迭代加深搜索使用多次限制深度的 DFS 来模拟 BFS。
- 具体来说，迭代加深搜索从 1 开始，每次都把搜索深度增加 1，直到某次搜索搜到了想要的解为止。
- 这种方法以一定的时间代价换取了空间代价。
- 假设搜索树的下一层的规模是上一层的两倍，时间代价仅翻了一倍，就把空间代价从 $O(2^d)$ 变成了 $O(d)$，d 为深度。

```
1  //step: 最大深度
2  bool dfs(int step); //返回是否找到解
3  int ans = 0;
4  while (true)
5    if (dfs(++ans)) break;
```

搜索：IDA*

- IDA* = ID（迭代加深搜索）+ A*

搜索：IDA*

- IDA* = ID（迭代加深搜索）+ A*
- 给出到终点的距离估计函数 h，假设目前在 x 状态，已经搜索深度为 d，深度上限为 t，当 $h(x) + d > t$ 时结束搜索。

搜索：IDA*

- IDA* = ID（迭代加深搜索）+ A*
- 给出到终点的距离估计函数 h，假设目前在 x 状态，已经搜索深度为 d，深度上限为 t，当 $h(x) + d > t$ 时结束搜索。
- 为了保证正确性，h 必须是**乐观的**（不能高估剩余距离）。

搜索：IDA*

- IDA* = ID（迭代加深搜索）+ A*
- 给出到终点的距离估计函数 h，假设目前在 x 状态，已经搜索深度为 d，深度上限为 t，当 $h(x) + d > t$ 时结束搜索。
- 为了保证正确性，h 必须是**乐观的**（不能高估剩余距离）。

```
bool dfs(int x, int y, int ttl) {
  if (abs(x - target_x) + abs(y - target_y) > ttl)
    return false;
  if (x == target_x && y == target_y) return true;
  for (int i = 0; i < 4; i++) {
    if (!legal(x + dx[i], y + dy[i])) continue;
    if (vis[x + dx[i]][y + dy[i]]) continue;
    if (dfs(x + dx[i], y + dy[i], ttl - 1))
      return true;
  }
}
```

搜索：骑士精神

例 (Search-3)

给你一个初始有各 12 个白骑士和黑骑士的 5×5 的棋盘，要求走最少的「日」字步达到目标状态（如下图）。

搜索：骑士精神

解 (Search-3)

- 状态太多了，且要求最优搜索。

搜索：骑士精神

解 (Search-3)

- 状态太多了，且要求最优搜索。
- 可以立即应用 *IDA**，估价函数设为不在正确位置的骑士的数量 (显然是非常乐观的)。

搜索：骑士精神

解 (Search-3)

- 状态太多了，且要求最优搜索。
- 可以立即应用 *IDA**，估价函数设为不在正确位置的骑士的数量 (显然是非常乐观的)。
- 实际应用时注意特判不走单步的回头路。

搜索：骑士精神

解 (Search-3)

- 状态太多了，且要求最优搜索。
- 可以立即应用 *IDA**，估价函数设为不在正确位置的骑士的数量 (显然是非常乐观的)。
- 实际应用时注意特判不走单步的回头路。
- 也可以使用双向搜索。
  (此时搜索树深度降低到允许使用 map 记录状态的量级)

搜索：中间相遇

例 (Search-4)

给定一个数 u 和另外 k 个数，给出一个最小的前述 k 个数的子集 A，使得 $\mathrm{xor}_{i \in A}(i) = u$。保证 $k \leq 40$。

搜索：中间相遇

例 (Search-4)

给定一个数 u 和另外 k 个数，给出一个最小的前述 k 个数的子集 A，使得 $\mathrm{xor}_{i \in A}(i) = u$。保证 $k \leq 40$。

- 采用一种叫做中间相遇的方法。

搜索：中间相遇

例 (Search-4)

给定一个数 u 和另外 k 个数，给出一个最小的前述 k 个数的子集 A，使得 $\mathrm{xor}_{i\in A}(i) = u$。保证 $k \le 40$。

- 采用一种叫做中间相遇的方法。
- 我们各搜索一半数的所有取值可能性的结果，全部存在以**异或和**为索引，最小步数为键值的 `map` 中。(这里也可以先找出所有的异或和后再离散化来避免使用 `map`，但没必要。)

搜索：中间相遇

例 (Search-4)

给定一个数 u 和另外 k 个数，给出一个最小的前述 k 个数的子集 A，使得 $\mathrm{xor}_{i\in A}(i) = u$。保证 $k \le 40$。

- 采用一种叫做中间相遇的方法。
- 我们各搜索一半数的所有取值可能性的结果，全部存在以**异或和**为索引，最小步数为键值的 `map` 中。(这里也可以先找出所有的异或和后再离散化来避免使用 `map`，但没必要。)
- 对于得到的两个 `map` 中的状态，我们发现它们两两组合能够合法当且仅当它们的异或和等于 u。

搜索：中间相遇

例 (Search-4)

给定一个数 u 和另外 k 个数，给出一个最小的前述 k 个数的子集 A，使得 $\mathrm{xor}_{i\in A}(i) = u$。保证 $k \le 40$。

- 采用一种叫做中间相遇的方法。
- 我们各搜索一半数的所有取值可能性的结果，全部存在以**异或和**为索引，最小步数为键值的 `map` 中。(这里也可以先找出所有的异或和后再离散化来避免使用 `map`，但没必要。)
- 对于得到的两个 `map` 中的状态，我们发现它们两两组合能够合法当且仅当它们的异或和等于 u。
- 于是只需要遍历某半边的状态，对于异或和是 d 的状态，直接查另一半的键值为 d xor u 的最小步骤数即可。

搜索：中间相遇

例 (Search-4)

给定一个数 u 和另外 k 个数，给出一个最小的前述 k 个数的子集 A，使得 $\mathrm{xor}_{i\in A}(i) = u$。保证 $k \le 40$。

- 采用一种叫做中间相遇的方法。
- 我们各搜索一半数的所有取值可能性的结果，全部存在以**异或和**为索引，最小步数为键值的 `map` 中。(这里也可以先找出所有的异或和后再离散化来避免使用 `map`，但没必要。)
- 对于得到的两个 `map` 中的状态，我们发现它们两两组合能够合法当且仅当它们的异或和等于 u。
- 于是只需要遍历某半边的状态，对于异或和是 d 的状态，直接查另一半的键值为 d xor u 的最小步骤数即可。
- 复杂度 $O(2^{\frac{k}{2}})$。

搜索：Max-Min 对抗搜索

例 (Search-5)

给出一个零和、确定、完全信息的二人博弈所有最终局面的收益，是否有一方存在必不败策略？请给出这个策略。

搜索：Max-Min 对抗搜索

例 (Search-5)

给出一个零和、确定、完全信息的二人博弈所有最终局面的收益，是否有一方存在必不败策略？请给出这个策略。

定理 (策梅洛定理)

任何有限、确定、完全信息、零和（否则没有输赢之分）的二人博弈中，至少有一方有必不败策略。

搜索：Max-Min 对抗搜索

例 (Search-5)

给出一个零和、确定、完全信息的二人博弈所有最终局面的收益，是否有一方存在必不败策略？请给出这个策略。

定理 (策梅洛定理)

任何有限、确定、完全信息、零和（否则没有输赢之分）的二人博弈中，至少有一方有必不败策略。

- 假设收益以先手的形式给出。

搜索：Max-Min 对抗搜索

例 (Search-5)

给出一个零和、确定、完全信息的二人博弈所有最终局面的收益，是否有一方存在必不败策略？请给出这个策略。

定理 (策梅洛定理)

任何有限、确定、完全信息、零和（否则没有输赢之分）的二人博弈中，至少有一方有必不败策略。

- 假设收益以先手的形式给出。
- 先从起始局面搜到所有的最终局面，从最终局面上溯时，如果是先手的层，先手总会取出其中收益最大的作为后继局面，而如果是后手的层，其总会取出最小的。

搜索：Max-Min 对抗搜索

例 (Search-5)

给出一个零和、确定、完全信息的二人博弈所有最终局面的收益，是否有一方存在必不败策略？请给出这个策略。

定理 (策梅洛定理)

任何有限、确定、完全信息、零和（否则没有输赢之分）的二人博弈中，至少有一方有必不败策略。

- 假设收益以先手的形式给出。
- 先从起始局面搜到所有的最终局面，从最终局面上溯时，如果是先手的层，先手总会取出其中收益最大的作为后继局面，而如果是后手的层，其总会取出最小的。
- 依次层层上溯，最终可以计算出初始局面在双方选取最优策略时的收益，若此时某方不败，则该方有必不败的策略。

搜索：Max-Min 对抗搜索

例 (Search-5)

给出一个零和、确定、完全信息的二人博弈所有最终局面的收益，是否有一方存在必不败策略？请给出这个策略。

搜索：Max-Min 对抗搜索

例 (Search-5)

给出一个零和、确定、完全信息的二人博弈所有最终局面的收益，是否有一方存在必不败策略？请给出这个策略。

- 状态太多了，怎么办？

搜索：Max-Min 对抗搜索

例 (Search-5)

给出一个零和、确定、完全信息的二人博弈所有最终局面的收益，是否有一方存在必不败策略？请给出这个策略。

- 状态太多了，怎么办？
- 考虑使用估值函数，在搜到一定深度时启发式地估计局面收益并停止搜索。

搜索：Max-Min 对抗搜索

例 (Search-5)

给出一个零和、确定、完全信息的二人博弈所有最终局面的收益，是否有一方存在必不败策略？请给出这个策略。

- 状态太多了，怎么办？
- 考虑使用估值函数，在搜到一定深度时启发式地估计局面收益并停止搜索。
- 然后运用迭代加深搜索卡时间。

搜索：Alpha-Beta 剪枝

例 (Search-5)

给出一个零和、确定、完全信息的二人博弈所有最终局面的收益，是否有一方存在必不败策略？请给出这个策略。

搜索：Alpha-Beta 剪枝

例 (Search-5)

给出一个零和、确定、完全信息的二人博弈所有最终局面的收益，是否有一方存在必不败策略？请给出这个策略。

- 还是太慢了。

搜索：Alpha-Beta 剪枝

例 (Search-5)

给出一个零和、确定、完全信息的二人博弈所有最终局面的收益，是否有一方存在必不败策略？请给出这个策略。

- 还是太慢了。
- 考虑**剪枝**，搜索到一个局面时，维护父节点的最优值 β 和目前节点的最优值 α。

搜索：Alpha-Beta 剪枝

例 (Search-5)

给出一个零和、确定、完全信息的二人博弈所有最终局面的收益，是否有一方存在必不败策略？请给出这个策略。

- 还是太慢了。
- 考虑**剪枝**，搜索到一个局面时，维护父节点的最优值 β 和目前节点的最优值 α。
- 当目前节点发现一个不如父节点目前最优值优的值时，由于目前节点和父节点优化目标相反，目前节点一定不会被父节点考虑到了，所以可以结束目前节点的搜索。

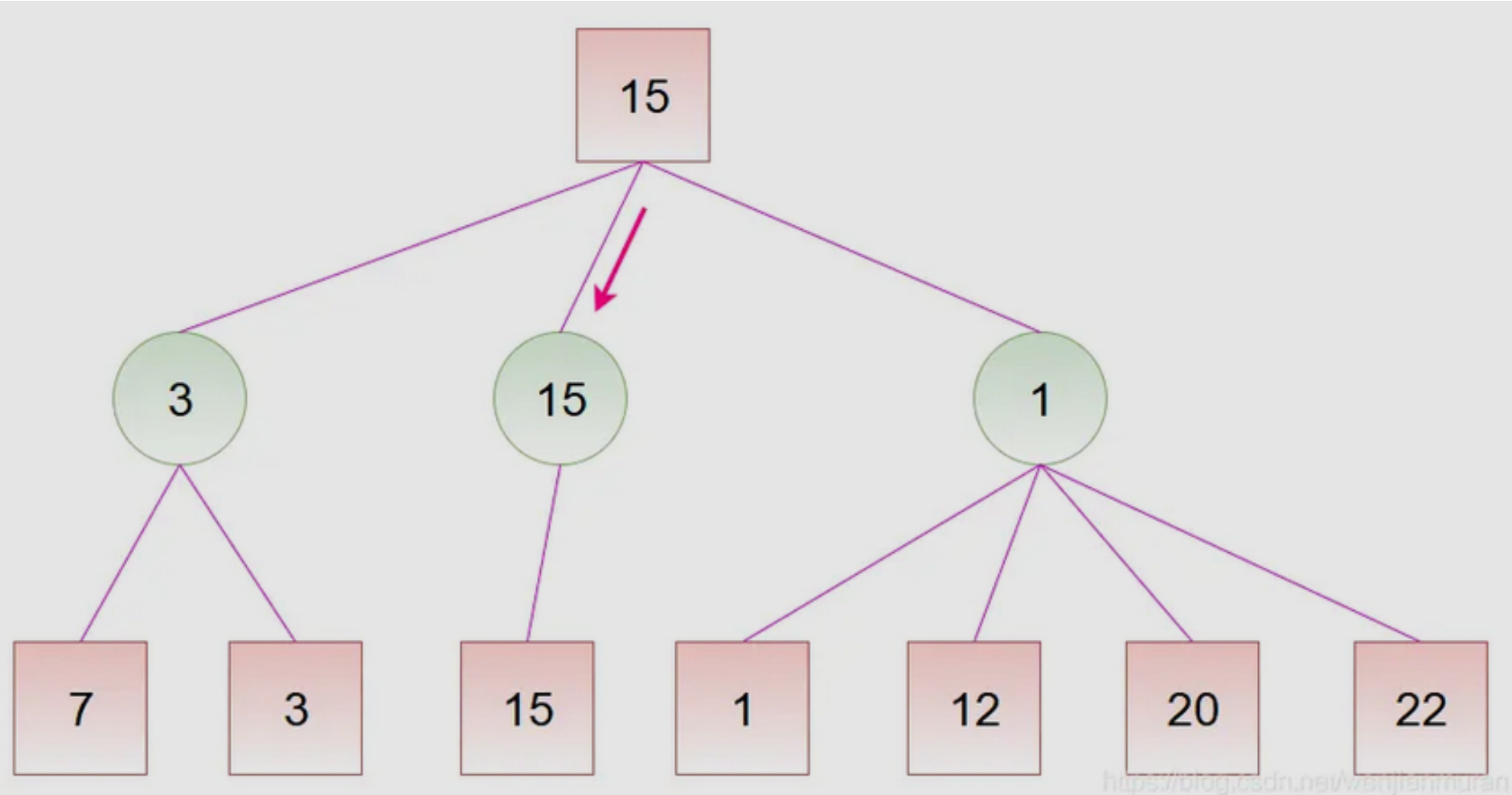
搜索：Alpha-Beta 剪枝

例 (Search-5)

给出一个零和、确定、完全信息的二人博弈所有最终局面的收益，是否有一方存在必不败策略？请给出这个策略。

- 还是太慢了。
- 考虑**剪枝**，搜索到一个局面时，维护父节点的最优值 β 和目前节点的最优值 α。
- 当目前节点发现一个不如父节点目前最优值优的值时，由于目前节点和父节点优化目标相反，目前节点一定不会被父节点考虑到了，所以可以结束目前节点的搜索。
- 这就是 Alpha-Beta 剪枝。

实例展示



目录

分治：引入

例 (DivideConquer-0)

输入 n 个整数，从小到大输出它们。

分治：引入

例 (DivideConquer-0)

输入 n 个整数，从小到大输出它们。

解 (DivideConquer-0)

- 大家一定学过各种基于分治的排序算法。
  其中最为经典的要数快速排序和归并排序。

```
void qsort(int l, int r) {
  if (l >= r) return ;
  //把大的和小的分开
  int mid = split();
  qsort(l, mid);
  qsort(mid + 1, r);
}
```

```
1  void msort(int l, int r) {
2    if (l >= r) return ;
3    msort(l, mid);
4    msort(mid + 1, r);
5    //归并两半的有序数组
6    merge();
7  }
```

分治：概念

- 容易发现，分治的流程包括这几步：

分治：概念

- 容易发现，分治的流程包括这几步：
  1. 把父问题划分成若干规模更小的子问题。
  2. 递归求解各个子问题。
  3. 合并子问题的解，解决父问题。

分治：概念

- 容易发现，分治的流程包括这几步：
  1. 把父问题划分成若干规模更小的子问题。
  2. 递归求解各个子问题。
  3. 合并子问题的解，解决父问题。
- 于是分治算法的复杂度可以写成 $f(n) = af(\frac{n}{b}) + \Theta(n^k \log^p n)$。

分治：概念

- 容易发现，分治的流程包括这几步：
  1. 把父问题划分成若干规模更小的子问题。
  2. 递归求解各个子问题。
  3. 合并子问题的解，解决父问题。
- 于是分治算法的复杂度可以写成 $f(n) = af(\frac{n}{b}) + \Theta(n^k \log^p n)$。
- 其中 a 是分成的子问题的个数，b 是子问题的规模缩减的倍数（缩减到 b 分之一）。

分治：概念

- 容易发现，分治的流程包括这几步：
  1. 把父问题划分成若干规模更小的子问题。
  2. 递归求解各个子问题。
  3. 合并子问题的解，解决父问题。
- 于是分治算法的复杂度可以写成 $f(n) = af(\frac{n}{b}) + \Theta(n^k \log^p n)$。
- 其中 a 是分成的子问题的个数，b 是子问题的规模缩减的倍数（缩减到 b 分之一）。
- 在排序问题中，$a = b = 2, k = 1, p = 0$，有 $f(n) = O(n \log n)$。

分治：概念

- 容易发现，分治的流程包括这几步：
  1. 把父问题划分成若干规模更小的子问题。
  2. 递归求解各个子问题。
  3. 合并子问题的解，解决父问题。
- 于是分治算法的复杂度可以写成 $f(n) = af(\frac{n}{b}) + \Theta(n^k \log^p n)$。
- 其中 a 是分成的子问题的个数，b 是子问题的规模缩减的倍数（缩减到 b 分之一）。
- 在排序问题中，$a = b = 2, k = 1, p = 0$，有 $f(n) = O(n \log n)$。
- 我还参加 OI 的时候，初赛还会考分治算法的复杂度计算……

分治：概念

- 容易发现，分治的流程包括这几步：
  1. 把父问题划分成若干规模更小的子问题。
  2. 递归求解各个子问题。
  3. 合并子问题的解，解决父问题。
- 于是分治算法的复杂度可以写成 $f(n) = af(\frac{n}{b}) + \Theta(n^k \log^p n)$。
- 其中 a 是分成的子问题的个数，b 是子问题的规模缩减的倍数（缩减到 b 分之一）。
- 在排序问题中，$a = b = 2, k = 1, p = 0$，有 $f(n) = O(n \log n)$。
- 我还参加 OI 的时候，初赛还会考分治算法的复杂度计算……
- 我们一般使用主定理计算分治算法的复杂度。

分治：主定理

定理

当 $f(n) = af(\frac{n}{b}) + \Theta(n^k \log^p n)$ 时，

$$f(n) = \begin{cases} \Theta(n^k \log^p n & k > \log_b a) \\ \Theta(n^k \log^{p+1} n & k = \log_b a) \\ \Theta(n^{\log_b a} & k < \log_b a) \end{cases}$$

分治：主定理

定理

当 $f(n) = af(\frac{n}{b}) + \Theta(n^k \log^p n)$ 时，

$$f(n) = \begin{cases} \Theta(n^k \log^p n & k > \log_b a) \\ \Theta(n^k \log^{p+1} n & k = \log_b a) \\ \Theta(n^{\log_b a} & k < \log_b a) \end{cases}$$

证明.

仅针对 $k \neq \log_b a$ 的情况给出简要说明。

分治：主定理

定理

当 $f(n) = af(\frac{n}{b}) + \Theta(n^k \log^p n)$ 时，

$$f(n) = \begin{cases} \Theta(n^k \log^p n & k > \log_b a) \\ \Theta(n^k \log^{p+1} n & k = \log_b a) \\ \Theta(n^{\log_b a} & k < \log_b a) \end{cases}$$

证明.

仅针对 $k \neq \log_b a$ 的情况给出简要说明。
考虑到分治在最底层处理的子问题总数是 $\Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$ 个，因为 $\frac{a}{b^k} \neq 1$，所以各层复杂度变化是单调的，只需比较第一层 $\Theta(n^k \log^p n)$ 和最底层 $\Theta(n^{\log_b a})$ 的复杂度，哪个大取哪个，于是就有上述结论。 □

分治：最近点对

例 (DivideConquer-1)

平面上有 n 个点，求出相距最近的两个点的距离。

分治：最近点对

例 (DivideConquer-1)

平面上有 n 个点，求出相距最近的两个点的距离。

解 (DivideConquer-1)

- 先按照 x 坐标排序，每次按照 x 坐标中位数将平面一分为二，分别求出两部分内部的最近点对，并同时对 y 坐标进行归并排序，假设两部分较近的最近点对间距离为 d。

分治：最近点对

例 (DivideConquer-1)

平面上有 n 个点，求出相距最近的两个点的距离。

解 (DivideConquer-1)

- 先按照 x 坐标排序，每次按照 x 坐标中位数将平面一分为二，分别求出两部分内部的最近点对，并同时对 y 坐标进行归并排序，假设两部分较近的最近点对间距离为 d。
- 对 y 坐标归并排序完成后，考虑两部分之间的最近点对，显然只需考虑分割线左右 d 以内的点。

分治：最近点对

例 (DivideConquer-1)

平面上有 n 个点，求出相距最近的两个点的距离。

解 (DivideConquer-1)

- 先按照 x 坐标排序，每次按照 x 坐标中位数将平面一分为二，分别求出两部分内部的最近点对，并同时对 y 坐标进行归并排序，假设两部分较近的最近点对间距离为 d。
- 对 y 坐标归并排序完成后，考虑两部分之间的最近点对，显然只需考虑分割线左右 d 以内的点。
- 不失一般性，对于每个点只找 y 坐标小于它且和它距离不超过 d 的最近点。

分治：最近点对

例 (DivideConquer-1)

平面上有 n 个点，求出相距最近的两个点的距离。

解 (DivideConquer-1)

- 先按照 x 坐标排序，每次按照 x 坐标中位数将平面一分为二，分别求出两部分内部的最近点对，并同时对 y 坐标进行归并排序，假设两部分较近的最近点对间距离为 d。
- 对 y 坐标归并排序完成后，考虑两部分之间的最近点对，显然只需考虑分割线左右 d 以内的点。
- 不失一般性，对于每个点只找 y 坐标小于它且和它距离不超过 d 的最近点。
- 这样的点最多只有七个，因为一个 $d \times d$ 的方格内不可能容下超过四个两两距离大于等于 d 的点。

分治：最近点对

例 (DivideConquer-1)

平面上有 n 个点，求出相距最近的两个点的距离。

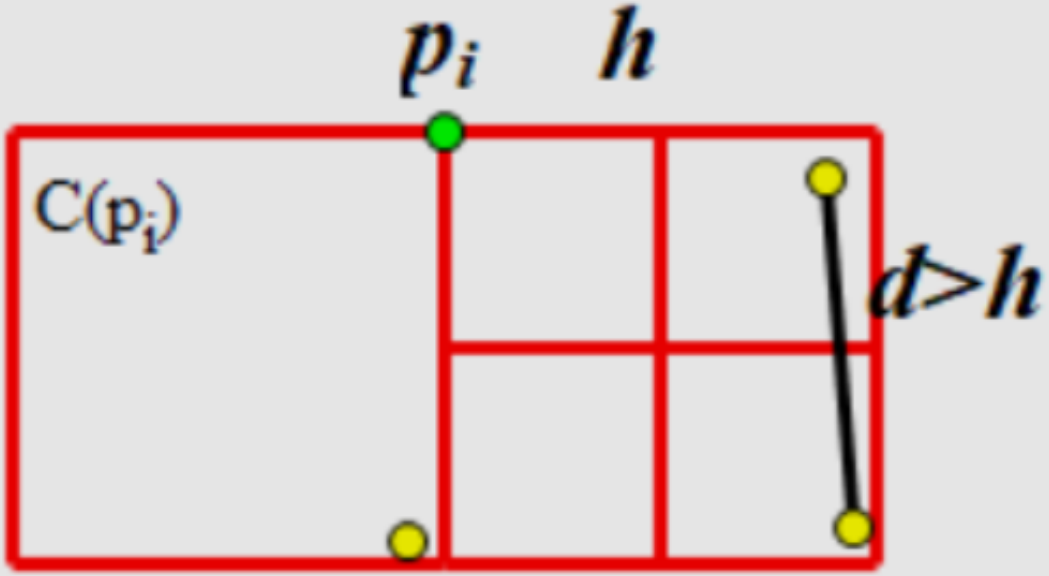
分治：最近点对

例 (DivideConquer-1)

平面上有 n 个点，求出相距最近的两个点的距离。

解 (DivideConquer-1)

- 于是合并子问题解的复杂度是 $O(n)$ 的，总复杂度为 $O(n \log n)$。



分治：二维偏序

例 (DivideConquer-2)

输入一个长度为 n 的序列 a。
求满足 $a_i > a_j$ 且 $i < j$ 的 (i, j) 的个数。

分治：二维偏序

例 (DivideConquer-2)

输入一个长度为 n 的序列 a。
求满足 $a_i > a_j$ 且 $i < j$ 的 (i, j) 的个数。

解 (DivideConquer-2)

- 递归计算左右两半区间的答案。考虑左右区间间的逆序对。

分治：二维偏序

例 (DivideConquer-2)

输入一个长度为 n 的序列 a。
求满足 $a_i > a_j$ 且 $i < j$ 的 (i, j) 的个数。

解 (DivideConquer-2)

- 递归计算左右两半区间的答案。考虑左右区间间的逆序对。
- 显然可以只统计左区间的每个数能匹配上右区间数的数量。

分治：二维偏序

例 (DivideConquer-2)

输入一个长度为 n 的序列 a。
求满足 $a_i > a_j$ 且 $i < j$ 的 (i, j) 的个数。

解 (DivideConquer-2)

- 递归计算左右两半区间的答案。考虑左右区间间的逆序对。
- 显然可以只统计左区间的每个数能匹配上右区间数的数量。
- 此即对于每个左区间数计算右区间中比自己小的数的数量。

分治：二维偏序

例 (DivideConquer-2)

输入一个长度为 n 的序列 a。
求满足 $a_i > a_j$ 且 $i < j$ 的 (i, j) 的个数。

解 (DivideConquer-2)

- 递归计算左右两半区间的答案。考虑左右区间间的逆序对。
- 显然可以只统计左区间的每个数能匹配上右区间数的数量。
- 此即对于每个左区间数计算右区间中比自己小的数的数量。
- 归并时维护归并数组里已经放入的右区间数的数量，当有左区间数被放入时，将答案加上这个数即可。

分治：二维偏序

例 (DivideConquer-2)

输入一个长度为 n 的序列 a。
求满足 $a_i > a_j$ 且 $i < j$ 的 (i, j) 的个数。

解 (DivideConquer-2)

- 递归计算左右两半区间的答案。考虑左右区间间的逆序对。
- 显然可以只统计左区间的每个数能匹配上右区间数的数量。
- 此即对于每个左区间数计算右区间中比自己小的数的数量。
- 归并时维护归并数组里已经放入的右区间数的数量，当有左区间数被放入时，将答案加上这个数即可。
- 我们将这种通过分治维护物品对间“影响”的思想称作 *CDQ* 分治。这种思想在离线处理区间问题、*DP* 优化上都有应用。但由于时间所限，这里不再赘述。

分治：三维偏序

例 (DivideConquer-3)

输入一个长度为 n 的二元组序列 a, b。
求满足 $a_i > a_j$ 且 $b_i > b_j$ 且 $i < j$ 的 (i, j) 的个数。

分治：三维偏序

例 (DivideConquer-3)

输入一个长度为 n 的二元组序列 a, b。
求满足 $a_i > a_j$ 且 $b_i > b_j$ 且 $i < j$ 的 (i, j) 的个数。

- 这是一道 CDQ 分治的例题，鉴于课堂容量仅留做选做题。可以查阅 OI Wiki 相关词条。

目录

贪心：概念

- 贪心选择：对于任何子问题，直接选取局部最优解（最优化本次选择的收益）来得到下一子问题。

贪心：概念

- 贪心选择：对于任何子问题，直接选取局部最优解（最优化本次选择的收益）来得到下一子问题。
- 如果某一问题的全局最优解可以通过不断进行贪心选择来构造，那么称这种问题满足贪心选择性质。

贪心：概念

- 贪心选择：对于任何子问题，直接选取局部最优解（最优化本次选择的收益）来得到下一子问题。
- 如果某一问题的全局最优解可以通过不断进行贪心选择来构造，那么称这种问题满足贪心选择性质。
- 可以看出，实现贪心算法是简单的（往往只需要一次排序），问题在于发现并证明~~（也许并不需要证明）~~问题的贪心选择性质。

贪心：概念

- 贪心选择：对于任何子问题，直接选取局部最优解（最优化本次选择的收益）来得到下一子问题。
- 如果某一问题的全局最优解可以通过不断进行贪心选择来构造，那么称这种问题满足贪心选择性质。
- 可以看出，实现贪心算法是简单的（往往只需要一次排序），问题在于发现并证明~~（也许并不需要证明）~~问题的贪心选择性质。
- 下面我们通过一个例子来理解贪心算法。

贪心：引入

例 (Greedy-0)

给定 n 个产品，第 i 个产品的价值是 u_i，还有 m 个产商，第 i 个产商的代价为 v_i。任何一个产商都可以被指定制造任何一种产品，但是一个产商只能制造一种产品。问代价最小而制造的产品总值最高的方案。

贪心：引入

例 (Greedy-0)

给定 n 个产品，第 i 个产品的价值是 u_i，还有 m 个产商，第 i 个产商的代价为 v_i。任何一个产商都可以被指定制造任何一种产品，但是一个产商只能制造一种产品。问代价最小而制造的产品总值最高的方案。

解 (Greedy-0)

贪心选择：每次指定最便宜的可用产商生产最昂贵的产品。

贪心：引入

例 (Greedy-0)

给定 n 个产品，第 i 个产品的价值是 u_i，还有 m 个产商，第 i 个产商的代价为 v_i。任何一个产商都可以被指定制造任何一种产品，但是一个产商只能制造一种产品。问代价最小而制造的产品总值最高的方案。

解 (Greedy-0)

贪心选择：每次指定最便宜的可用产商生产最昂贵的产品。

- 这个问题的贪心选择性质是显然的。

贪心：引入

例 (Greedy-0)

给定 n 个产品，第 i 个产品的价值是 u_i，还有 m 个产商，第 i 个产商的代价为 v_i。任何一个产商都可以被指定制造任何一种产品，但是一个产商只能制造一种产品。问代价最小而制造的产品总值最高的方案。

解 (Greedy-0)

贪心选择：每次指定最便宜的可用产商生产最昂贵的产品。

- 这个问题的贪心选择性质是显然的。
- 接下来我们通过一些经典的贪心问题来学习如何发现并证明问题的贪心选择性质。

贪心：活动安排问题

例 (Greedy-1)

有 n 个活动，其中第 i 个活动的举办时间是 $[b_i, e_i]$，给出一个选取活动的方案，使得选取的活动最多且它们不相互冲突。(不冲突指的是，对于任意两个活动 $i \neq j$，有 $b_i \geq e_j$ 或 $b_j \geq e_i$)

贪心：活动安排问题

例 (Greedy-1)

有 n 个活动，其中第 i 个活动的举办时间是 $[b_i, e_i]$，给出一个选取活动的方案，使得选取的活动最多且它们不相互冲突。(不冲突指的是，对于任意两个活动 $i \neq j$，有 $b_i \geq e_j$ 或 $b_j \geq e_i$)

- 在这个问题中，什么是对于当前局面“最贪心”的选择呢?

贪心：活动安排问题

例 (Greedy-1)

有 n 个活动，其中第 i 个活动的举办时间是 $[b_i, e_i]$，给出一个选取活动的方案，使得选取的活动最多且它们不相互冲突。(不冲突指的是，对于任意两个活动 $i \neq j$，有 $b_i \geq e_j$ 或 $b_j \geq e_i$)

- 在这个问题中，什么是对于当前局面“最贪心”的选择呢?
- 对于我们来说，由于活动并没有价值的区分，收益最大的活动是那些「不容易产生冲突的活动」。

贪心：活动安排问题

例 (Greedy-1)

有 n 个活动，其中第 i 个活动的举办时间是 $[b_i, e_i]$，给出一个选取活动的方案，使得选取的活动最多且它们不相互冲突。(不冲突指的是，对于任意两个活动 $i \neq j$，有 $b_i \geq e_j$ 或 $b_j \geq e_i$)

- 在这个问题中，什么是对于当前局面“最贪心”的选择呢?
- 对于我们来说，由于活动并没有价值的区分，收益最大的活动是那些「不容易产生冲突的活动」。
- 于是考虑我们选取的最早活动。对于后续局面来说，此最早活动的结束时间越早，则可供后续活动选择的空间越大。

贪心：活动安排问题

例 (Greedy-1)

有 n 个活动，其中第 i 个活动的举办时间是 $[b_i, e_i]$，给出一个选取活动的方案，使得选取的活动最多且它们不相互冲突。(不冲突指的是，对于任意两个活动 $i \neq j$，有 $b_i \geq e_j$ 或 $b_j \geq e_i$)

- 在这个问题中，什么是对于当前局面“最贪心”的选择呢?
- 对于我们来说，由于活动并没有价值的区分，收益最大的活动是那些「不容易产生冲突的活动」。
- 于是考虑我们选取的最早活动。对于后续局面来说，此最早活动的结束时间越早，则可供后续活动选择的空间越大。

解 (Greedy-1)

贪心选择：每次指定结束时间最早的活动。

贪心：活动安排问题

- 怎么证明问题的贪心选择性质？

贪心：活动安排问题

- 怎么证明问题的贪心选择性质？
- 假设出一个更优的方案，并找出此最优方案与贪心方案第一次不同的局部选择，证明将此选择换为贪心选择可以得到一个至少不会更差的全局解。

贪心：活动安排问题

- 怎么证明问题的贪心选择性质？
- 假设出一个更优的方案，并找出此最优方案与贪心方案第一次不同的局部选择，证明将此选择换为贪心选择可以得到一个至少不会更差的全局解。
- 这种方法称为**剪切-粘贴法**，下面用这个方法证明活动安排问题的贪心选择性质。

贪心：活动安排问题

- 怎么证明问题的贪心选择性质？
- 假设出一个更优的方案，并找出此最优方案与贪心方案第一次不同的局部选择，证明将此选择换为贪心选择可以得到一个至少不会更差的全局解。
- 这种方法称为**剪切-粘贴法**，下面用这个方法证明活动安排问题的贪心选择性质。

证 (Greedy-1)

假设最优解是 $(\mathbf{b}', \mathbf{e}')$，找出该方案有而贪心方案 $(\mathbf{b}, \mathbf{e})$ 没有的第一个活动，设为第 i 个活动。易知 $e_i \leq e'_i$。那么将该方案的活动换为贪心方案的第 i 个活动。

贪心：活动安排问题

- 怎么证明问题的贪心选择性质？
- 假设出一个更优的方案，并找出此最优方案与贪心方案第一次不同的局部选择，证明将此选择换为贪心选择可以得到一个至少不会更差的全局解。
- 这种方法称为**剪切-粘贴法**，下面用这个方法证明活动安排问题的贪心选择性质。

证 (Greedy-1)

假设最优解是 $(\mathbf{b}', \mathbf{e}')$，找出该方案有而贪心方案 $(\mathbf{b}, \mathbf{e})$ 没有的第一个活动，设为第 i 个活动。易知 $e_i \leq e'_i$。那么将该方案的活动换为贪心方案的第 i 个活动。

因为 $\forall j < i, e_j = e'_j, b_j = b'_j \to \forall j < i, b_i \geq e_j = e'_j$（前面不冲突），且有 $\forall j > i, b'_j \geq e'_i \geq e_i$（后面不冲突）。

贪心：活动安排问题

- 怎么证明问题的贪心选择性质？
- 假设出一个更优的方案，并找出此最优方案与贪心方案第一次不同的局部选择，证明将此选择换为贪心选择可以得到一个至少不会更差的全局解。
- 这种方法称为**剪切-粘贴法**，下面用这个方法证明活动安排问题的贪心选择性质。

证 (Greedy-1)

假设最优解是 $(\mathbf{b}', \mathbf{e}')$，找出该方案有而贪心方案 $(\mathbf{b}, \mathbf{e})$ 没有的第一个活动，设为第 i 个活动。易知 $e_i \leq e'_i$。那么将该方案的活动换为贪心方案的第 i 个活动。

因为 $\forall j < i, e_j = e'_j, b_j = b'_j \to \forall j < i, b_i \geq e_j = e'_j$（前面不冲突），且有 $\forall j > i, b'_j \geq e'_i \geq e_i$（后面不冲突）。

所以此处使用贪心选择不会使解变差。

贪心：活动安排问题

- 怎么证明问题的贪心选择性质？
- 假设出一个更优的方案，并找出此最优方案与贪心方案第一次不同的局部选择，证明将此选择换为贪心选择可以得到一个至少不会更差的全局解。
- 这种方法称为**剪切-粘贴法**，下面用这个方法证明活动安排问题的贪心选择性质。

证 (Greedy-1)

假设最优解是 $(\mathbf{b}', \mathbf{e}')$，找出该方案有而贪心方案 $(\mathbf{b}, \mathbf{e})$ 没有的第一个活动，设为第 i 个活动。易知 $e_i \leq e'_i$。那么将该方案的活动换为贪心方案的第 i 个活动。

因为 $\forall j < i, e_j = e'_j, b_j = b'_j \to \forall j < i, b_i \geq e_j = e'_j$（前面不冲突），且有 $\forall j > i, b'_j \geq e'_i \geq e_i$（后面不冲突）。

所以此处使用贪心选择不会使解变差。

用归纳法，可以证明贪心算法得到的是全局最优解。

贪心：国王游戏

例 (Greedy-2)

有 n 个人，每个人左右手各有一个数，称为 a, b，对于每个人，他可以得到的收益是前面所有人左手的数乘积除以自己右手上的数，问怎么给这些人排序，使得他们中收益最大的人最小。

贪心：国王游戏

例 (Greedy-2)

有 n 个人，每个人左右手各有一个数，称为 a, b，对于每个人，他可以得到的收益是前面所有人左手的数乘积除以自己右手上的数，问怎么给这些人排序，使得他们中收益最大的人最小。

- 直觉告诉我们左手/右手数尽可能小者应当排在前面。

贪心：国王游戏

例 (Greedy-2)

有 n 个人，每个人左右手各有一个数，称为 a, b，对于每个人，他可以得到的收益是前面所有人左手的数乘积除以自己右手上的数，问怎么给这些人排序，使得他们中收益最大的人最小。

- 直觉告诉我们左手/右手数尽可能小者应当排在前面。
- 考虑任意相邻的两个人，他们之前的人对收益的影响可以约去，设为 A。

贪心：国王游戏

例 (Greedy-2)

有 n 个人，每个人左右手各有一个数，称为 a, b，对于每个人，他可以得到的收益是前面所有人左手的数乘积除以自己右手上的数，问怎么给这些人排序，使得他们中收益最大的人最小。

- 直觉告诉我们左手/右手数尽可能小者应当排在前面。
- 考虑任意相邻的两个人，他们之前的人对收益的影响可以约去，设为 A。
- 设两人手中数分别为 a_1, b_1 和 a_2, b_2，有 $a_1 b_1 < a_2 b_2$。

贪心：国王游戏

例 (Greedy-2)

有 n 个人，每个人左右手各有一个数，称为 a, b，对于每个人，他可以得到的收益是前面所有人左手的数乘积除以自己右手上的数，问怎么给这些人排序，使得他们中收益最大的人最小。

- 直觉告诉我们左手/右手数尽可能小者应当排在前面。
- 考虑任意相邻的两个人，他们之前的人对收益的影响可以约去，设为 A。
- 设两人手中数分别为 a_1, b_1 和 a_2, b_2，有 $a_1 b_1 < a_2 b_2$。
- 1 排前面时，两人的最大值是 $\max\{\frac{A}{b_1}, \frac{Aa_1}{b_2}\}$，反之最大值是 $\max\{\frac{A}{b_2}, \frac{Aa_2}{b_1}\}$，易知 $\frac{Aa_1}{b_2} < \frac{Aa_2}{b_1}$，且 $\frac{A}{b_1} \leq \frac{Aa_2}{b_1}$，所以 $\max\{\frac{A}{b_1}, \frac{Aa_1}{b_2}\} \leq \max\{\frac{A}{b_2}, \frac{Aa_2}{b_1}\}$。

贪心：国王游戏

例 (Greedy-2)

有 n 个人，每个人左右手各有一个数，称为 a, b，对于每个人，他可以得到的收益是前面所有人左手的数乘积除以自己右手上的数，问怎么给这些人排序，使得他们中收益最大的人最小。

- 直觉告诉我们左手/右手数尽可能小者应当排在前面。
- 考虑任意相邻的两个人，他们之前的人对收益的影响可以约去，设为 A。
- 设两人手中数分别为 a_1, b_1 和 a_2, b_2，有 $a_1 b_1 < a_2 b_2$。
- 1 排前面时，两人的最大值是 $\max\{\frac{A}{b_1}, \frac{Aa_1}{b_2}\}$，反之最大值是 $\max\{\frac{A}{b_2}, \frac{Aa_2}{b_1}\}$，易知 $\frac{Aa_1}{b_2} < \frac{Aa_2}{b_1}$，且 $\frac{A}{b_1} \leq \frac{Aa_2}{b_1}$，所以 $\max\{\frac{A}{b_1}, \frac{Aa_1}{b_2}\} \leq \max\{\frac{A}{b_2}, \frac{Aa_2}{b_1}\}$。
- 交换二人顺序不会影响其他人的收益。于是，由归纳法可以证明没有任何方案比按照 ab 排序的序列的解来的优。

贪心：国王游戏

例 (Greedy-2)

有 n 个人，每个人左右手各有一个数，称为 a, b，对于每个人，他可以得到的收益是前面所有人左手的数乘积除以自己右手上的数，问怎么给这些人排序，使得他们中收益最大的人最小。

贪心：国王游戏

例 (Greedy-2)

有 n 个人，每个人左右手各有一个数，称为 a, b，对于每个人，他可以得到的收益是前面所有人左手的数乘积除以自己右手上的数，问怎么给这些人排序，使得他们中收益最大的人最小。

- 可以看出，除了剪切-粘贴法以外，上述方法也有效证明了问题的贪心选择性质。

贪心：国王游戏

例 (Greedy-2)

有 n 个人，每个人左右手各有一个数，称为 a, b，对于每个人，他可以得到的收益是前面所有人左手的数乘积除以自己右手上的数，问怎么给这些人排序，使得他们中收益最大的人最小。

- 可以看出，除了剪切-粘贴法以外，上述方法也有效证明了问题的贪心选择性质。
- 具体来说，这个方法在解空间上定义了和收益的全序关系一致的偏序关系。在这个偏序关系下的极小元就是问题的局部最优解。

贪心：国王游戏

例 (Greedy-2)

有 n 个人，每个人左右手各有一个数，称为 a, b，对于每个人，他可以得到的收益是前面所有人左手的数乘积除以自己右手上的数，问怎么给这些人排序，使得他们中收益最大的人最小。

- 可以看出，除了剪切-粘贴法以外，上述方法也有效证明了问题的贪心选择性质。
- 具体来说，这个方法在解空间上定义了和收益的全序关系一致的偏序关系。在这个偏序关系下的极小元就是问题的局部最优解。
- 进一步地，如果可以证明该极小元唯一，那么可以证明问题的局部最优解就是全局最优解。

贪心：国王游戏

例 (Greedy-2)

有 n 个人，每个人左右手各有一个数，称为 a, b，对于每个人，他可以得到的收益是前面所有人左手的数乘积除以自己右手上的数，问怎么给这些人排序，使得他们中收益最大的人最小。

- 可以看出，除了剪切-粘贴法以外，上述方法也有效证明了问题的贪心选择性质。
- 具体来说，这个方法在解空间上定义了和收益的全序关系一致的偏序关系。在这个偏序关系下的极小元就是问题的局部最优解。
- 进一步地，如果可以证明该极小元唯一，那么可以证明问题的局部最优解就是全局最优解。

解 (Greedy-2)

贪心选择：按照 ab 对所有人排序。

贪心：最优编码

例 (Greedy-3)

有 n 个数，第 i 个数是 a_i，要求为每一个数指派一个 01 编码，且没有任何一个编码是另外一个编码的前缀，假设第 i 个数的编码长度为 len_i，并最小化 $\sum_{i\in[n]} a_i \cdot len_i$。

贪心：最优编码

例 (Greedy-3)

有 n 个数，第 i 个数是 a_i，要求为每一个数指派一个 01 编码，且没有任何一个编码是另外一个编码的前缀，假设第 i 个数的编码长度为 len_i，并最小化 $\sum_{i\in[n]} a_i \cdot len_i$。

- 这题虽然是经典题，但是留作思考题。

贪心：最优编码

例 (Greedy-3)

有 n 个数，第 i 个数是 a_i，要求为每一个数指派一个 01 编码，且没有任何一个编码是另外一个编码的前缀，假设第 i 个数的编码长度为 len_i，并最小化 $\sum_{i\in[n]} a_i \cdot len_i$。

- 这题虽然是经典题，但是留作思考题。
- 提示：在任意局面下，为最小的两个数指派具有相同长度的，且**只有最后一位不同**的编码总是不会更劣的。

贪心：最优编码

例 (Greedy-3)

有 n 个数，第 i 个数是 a_i，要求为每一个数指派一个 01 编码，且没有任何一个编码是另外一个编码的前缀，假设第 i 个数的编码长度为 len_i，并最小化 $\sum_{i\in[n]} a_i \cdot len_i$。

- 这题虽然是经典题，但是留作思考题。
- 提示：在任意局面下，为最小的两个数指派具有相同长度的，且**只有最后一位不同**的编码总是不会更劣的。
- 又是提示：除贪心选择外，还需考虑选择的后继局面建模。

贪心：种树

例 (Greedy-4)

有 n 个数，第 i 个数是 a_i，选择至多 k 个两两不相邻的数，使得选择数的值总和最大。

贪心：种树

例 (Greedy-4)

有 n 个数，第 i 个数是 a_i，选择至多 k 个两两不相邻的数，使得选择数的值总和最大。

- 留作习题。

贪心：种树

例 (Greedy-4)

有 n 个数，第 i 个数是 a_i，选择至多 k 个两两不相邻的数，使得选择数的值总和最大。

- 留作习题。
- 提示：可以使用优先队列进行贪心选择。

贪心：种树

例 (Greedy-4)

有 n 个数，第 i 个数是 a_i，选择至多 k 个两两不相邻的数，使得选择数的值总和最大。

- 留作习题。
- 提示：可以使用优先队列进行贪心选择。
- 又是提示：除贪心选择外，还需考虑选择的后继局面建模。

贪心：种树

例 (Greedy-4)

有 n 个数，第 i 个数是 a_i，选择至多 k 个两两不相邻的数，使得选择数的值总和最大。

- 留作习题。
- 提示：可以使用优先队列进行贪心选择。
- 又是提示：除贪心选择外，还需考虑选择的后继局面建模。
- 鼓励：大胆猜想，不要拘泥于证明。

目录

参考资料

- STL 容器：OI Wiki 相关词条。

参考资料

- STL 容器：OI Wiki 相关词条。
- Again，推荐使用 OI Wiki 进行本次课提到的各种算法的自学，尤其是**掌握算法模板实现**。这是一节区区几小时的课**无法**教会你的，需要课下**自学**。
  ~~（但是上面也偶发小错，比如 Splay，二月份好像才被改好……）~~

关于练习题

- 时间所限（无论是你们的还是我的），只有四道题。

关于练习题

- 时间所限（无论是你们的还是我的），只有四道题。
- 重点：ST 表，搜索，前缀和，二分答案。
  因为主要是练习目的，我把考点直接写在了题面里。

关于练习题

- 时间所限（无论是你们的还是我的），只有四道题。
- 重点：ST 表，搜索，前缀和，二分答案。
  因为主要是练习目的，我把考点直接写在了题面里。
- 四题应该都是普及组第三题难度吧（?）毕竟出题要求说按一等奖水平看待大家，怕太水了大家做着没意思。

关于练习题

- 时间所限（无论是你们的还是我的），只有四道题。
- 重点：ST 表，搜索，前缀和，二分答案。
  因为主要是练习目的，我把考点直接写在了题面里。
- 四题应该都是普及组第三题难度吧（?）毕竟出题要求说按一等奖水平看待大家，怕太水了大家做着没意思。
- 部分分还是给的很多的，不要轻易放弃。

关于练习题

- 时间所限（无论是你们的还是我的），只有四道题。
- 重点：ST 表，搜索，前缀和，二分答案。
  因为主要是练习目的，我把考点直接写在了题面里。
- 四题应该都是普及组第三题难度吧（?）毕竟出题要求说按一等奖水平看待大家，怕太水了大家做着没意思。
- 部分分还是给的很多的，不要轻易放弃。
- 部分题目没有唯一解，如果没记住模板可以想想别的做法。

关于练习题

- 时间所限（无论是你们的还是我的），只有四道题。
- 重点：ST 表，搜索，前缀和，二分答案。
  因为主要是练习目的，我把考点直接写在了题面里。
- 四题应该都是普及组第三题难度吧（?）毕竟出题要求说按一等奖水平看待大家，怕太水了大家做着没意思。
- 部分分还是给的很多的，不要轻易放弃。
- 部分题目没有唯一解，如果没记住模板可以想想别的做法。
- 数据出的比较放水，要是大家卡过了还请轻 D。

关于练习题

- 时间所限（无论是你们的还是我的），只有四道题。
- 重点：ST 表，搜索，前缀和，二分答案。
  因为主要是练习目的，我把考点直接写在了题面里。
- 四题应该都是普及组第三题难度吧（?）毕竟出题要求说按一等奖水平看待大家，怕太水了大家做着没意思。
- 部分分还是给的很多的，不要轻易放弃。
- 部分题目没有唯一解，如果没记住模板可以想想别的做法。
- 数据出的比较放水，要是大家卡过了还请轻 D。
- 谢谢大家！祝大家午餐愉快，下午做题顺利。