

1.4 动态规划

树形DP

典例

题目大意

解法

树上背包 DP

矩阵快速幂优化 DP

单调队列优化 DP

典例 POJ3017

多重背包

二进制拆分法

单调队列优化

斜率优化 DP

决策单调性

四边形不等式

一维决策单调性

二维决策单调性

GarsiaWachs 算法

整除分块优化 DP

典例 HDU7217

题目大意

解法

凸优化 DP

1.4 计算几何

基础操作

欧拉公式

Pick 定理

凸包

闵可夫斯基和

1.4 动态规划

- 通过发掘转移时**最优解的必要条件**往往能使抽象的转移条件变得简单。
- 当问题有多维的限制时，**将其中一维排序**往往能简化状态设计和转移。

树形DP

典例

题目大意

- 给一棵 n 个点的带边权树，要求找出 k 个点 A_1, A_2, \dots, A_k ，使得 $\sum_{i=1}^{k-1} \text{dist}(A_i, A_{i+1})$ 最小。
- $n, k \leq 3000$

解法

- 设 $f_{u,j}$ 表示从点 u 出发，在 u 的子树中经过 j 个点最后回到点 u 的最小距离和。
- 设 $g_{u,j}$ 表示从点 u 出发，在 u 的子树中经过 j 个点最后停在任意一点（也相当于从 u 的子树中任意一点出发，经过 j 个点最后回到点 u ）的最小距离和。

- 设 $h_{u,j}$ 表示在 u 的子树中从任意一点出发，经过 j 个点并保证经过点 u ，最后停在任意一点的最小距离和。
- 显然 f, g, h 都是由 u 的子节点 v 转移过来，则我们可以得到如下转移（为了描述方便，设已经处理完的 u 的子节点的子树结点集合为 A ，当前处理的子节点 v 的子树结点集合为 B ，用 \rightarrow 表示每种转移所对应树上走的方案，转移前令 $f' = f, g' = g, h' = h$ ）：

- 对于 $f_{u,j}$

$$f'_{u,j+k} = \min\{f_{v,k} + f_{u,j} + 2\text{dist}(u, v)\} \quad u \rightarrow A \rightarrow u \rightarrow B \rightarrow u$$

- 对于 $g_{u,j}$

$$g'_{u,j+k} = \min\{g_{v,k} + f_{u,j} + \text{dist}(u, v)\} \quad u \rightarrow A \rightarrow u \rightarrow B$$

$$g'_{u,j+k} = \min\{f_{v,k} + g_{u,j} + 2\text{dist}(u, v)\} \quad u \rightarrow B \rightarrow u \rightarrow A$$

- 对于 $h_{u,j}$

$$h'_{u,j+k} = \min\{f_{v,k} + h_{u,j} + 2\text{dist}(u, v)\} \quad A \rightarrow u \rightarrow B \rightarrow u \rightarrow A$$

$$h'_{u,j+k} = \min\{h_{v,k} + f_{u,j} + 2\text{dist}(u, v)\} \quad B \rightarrow u \rightarrow A \rightarrow u \rightarrow B$$

$$h'_{u,j+k} = \min\{g_{v,k} + g_{u,j} + \text{dist}(u, v)\} \quad A \rightarrow u \rightarrow B$$

- 时间复杂度 $\mathcal{O}(n^2)$ 。

树上背包 DP

- **问题** 给一棵树，要求设计一个动态规划算法，求出以每个点为根大小不超过 k 的连通块个数。
- **算法** 设 $f_{x,i}$ 表示以 x 为根大小为 i 的连通块个数，当合并一个子结点 y ，采用如下的转移方式：

```

1  for (int i = 0; i <= sze[x] + sze[y]; ++i)
2      g[i] = 0;
3  for (int i = 0; i <= min(sze[x], k); ++i)
4      for (int j = 0; j <= min(sze[y], k); ++j)
5          g[i + j] += f[x][i] * f[y][j];
6  sze[x] += sze[y];
7  for (int i = 0; i <= sze[x]; ++i)
8      f[x][i] = g[i];

```

- **引理** 当 $k = n$ 时，该算法的时间复杂度为 $\mathcal{O}(n^2)$ 。

证明 考虑由树上结点生成的 n^2 个点对，将枚举次数看作是点对的选择，每个点对只会在其 LCA 处被恰好统计一次。

- **定理** 该算法的复杂度为 $\mathcal{O}(nk)$ 。

证明 记合并过程 $\text{sze}_x \leq k$ 则称此时的 x 为小子树，否则为大树。考虑分几种情况讨论：

1. 若大树与大树合并，显然大树的个数不会超过 $\frac{n}{k}$ 个，故总的合并复杂度不会超过 $k^2 \times \frac{n}{k} = nk$ 。
2. 若小子树与大树合并，小子树合并后必然成为大树，因此合并的小子树必然互不相交，总合并复杂度不超过 nk 。
3. 若小子树与小子树合并，以整体的眼光来看，假设这些合并过程最终产生了大小为 x_1, x_2, \dots, x_m 的 m 棵子树，由 **引理**，分析最劣情况的复杂度可以看作是求解以下问题：

$$\begin{aligned} & \text{maximize } \sum_{i=1}^m x_i^2 \\ & \text{subject to } x_i \leq k \quad \forall 1 \leq i \leq m \\ & \quad \sum_{i=1}^m x_i = n \end{aligned}$$

容易通过调整法证明，将每个 x_i 取得尽可能接近 k 可使上式取得最大值，因此上限同样为 nk 。

矩阵快速幂优化 DP

- 以下面的顺序计算矩阵乘法，访问内存连续，效率最高。

```

1      friend inline matrix operator * (const matrix &a, const matrix &b)
2      {
3          matrix c;
4          c.clear(a.gn, b.gm);
5          for (int i = 0; i < a.gn; ++i)
6              for (int k = 0; k < a.gm; ++k)
7              {
8                  int s = a.g[i][k];
9                  for (int j = 0; j < b.gm; ++j)
10                     c.g[i][j] = (1ll * s * b.g[k][j] + c.g[i][j]) % mod;
11              }
12          return c;
13      }

```

- 定义广义矩阵乘法 $C_{i,j} = \bigoplus_{k=1}^n A_{i,k} \otimes B_{k,j}$ 只需满足 \otimes 具有结合律， \oplus 具有交换律，且 \otimes 对 \oplus 具有分配律，矩阵乘法就存在结合律。

证明 假设有矩阵 $D = ABC$ ，需证明 $D = A(BC)$ ，即

$$\begin{aligned} D_{i,j} &= \bigoplus_{l=1}^n (AB)_{i,l} \otimes C_{l,j} \\ &= \bigoplus_{l=1}^n \left(\bigoplus_{k=1}^n A_{i,k} \otimes B_{k,l} \right) \otimes C_{l,j} \\ &= \bigoplus_{l=1}^n \bigoplus_{k=1}^n A_{i,k} \otimes B_{k,l} \otimes C_{l,j} \\ &= \bigoplus_{k=1}^n \bigoplus_{l=1}^n A_{i,k} \otimes B_{k,l} \otimes C_{l,j} \\ &= \bigoplus_{k=1}^n \bigoplus_{l=1}^n A_{i,k} \otimes (B_{k,l} \otimes C_{l,j}) \\ &= \bigoplus_{k=1}^n A_{i,k} \otimes \left(\bigoplus_{l=1}^n B_{k,l} \otimes C_{l,j} \right) \\ &= \bigoplus_{k=1}^n A_{i,k} \otimes (BC)_{k,j} \end{aligned}$$

- 可通过以下几种方式优化矩阵乘法的复杂度：
 - 矩阵中有效（如非零等）的元素很少，只维护有效的元素。
 - 矩阵的实际意义使得其有一些特殊的性质，部分元素是完全重复的（如 $C_{i,j} = C_{i+k,j+k}$ 等），则重复的元素不需维护。

单调队列优化 DP

- 单调队列/栈上结点也可以记录信息，如前缀和等，或在插入或删除的过程中用数据结构维护。

典例 POJ3017

- 设 f_i 表示将前 i 个数分成若干段、满足每段所有数的和不超过 M 时各段最大值之和的最小值。不难得到转移：

$$f_i = \min_{0 \leq j < i \text{ 且 } \sum_{k=j+1}^i a_k \leq M} \{f_j + \max_{j+1 \leq k \leq i} \{a_k\}\}$$

- 因为在此题的限制条件中 f_i 是单调不降的，容易证明，最优解需满足以下两个条件之一：
 - $a_j = \max_{j \leq k \leq i} \{a_k\}$
 - $\sum_{k=j}^i a_k > M$
- 第一个条件可以维护一个决策点 j 单调递增、数值 a_j 单调递减的队列，并用数据结构维护最优转移，这里使用的是懒惰删除的二叉堆，第二个条件只需要维护单个指针单独转移即可。

```
1  l = 0, ql = 1, qr = 0;
2  for (int i = 1; i <= n; ++i)
3  {
4      while (l < i && sum[i] - sum[l] > M)
5          ++l;
6      while (ql <= qr && sum[i] - sum[que[ql]] > M)
7          val[que[ql++]] = -1;
8      while (ql <= qr && a[que[qr]] <= a[i])
9          val[que[--qr]] = -1;
10     if (ql <= qr)
11         q.push(point(val[que[qr]] = f[que[qr]] + a[i], que[qr]));
12     que[++qr] = i;
13     f[i] = f[l] + a[que[ql]];
14     while (!q.empty() && val[q.top().t] != q.top().s)
15         q.pop();
16     if (!q.empty())
17         ckmin(f[i], q.top().s);
18 }
```

多重背包

- 给定 n 种物品，其中第 i 种物品的体积为 v_i ，价值为 w_i ，有 c_i 个。
- 求能够放入容积 m 的背包的物品的最大价值总和。

二进制拆分法

- 求出 $\sum_{k=0}^p 2^k \leq c_i$ 最大的 p ，令 $r_i = c_i - \sum_{k=0}^p 2^k$ 。
- 将数量为 c_i 的第 i 种物品拆成 $p + 2$ 种物品，其体积分别为：
$$2^0 v_i, 2^1 v_i, \dots, 2^p v_i, r_i v_i$$
- 时间复杂度 $\mathcal{O}(nm \log c)$ ，其中 $c = \max_{1 \leq i \leq n} \{c_i\}$ 。

单调队列优化

- 将容积那一维按照模 v_i 的余数分类，则可利用单调队列优化。
- 时间复杂度 $\mathcal{O}(nm)$ 。

```
1   for (int i = 1; i <= n; ++i)
2       for (int j = 0; j < v[i]; ++j)
3       {
4           cm = 0;
5           for (int k = j; k <= m; k += v[i])
6               cur[++cm] = k;
7           int r = cm - 1, ql = 1, qr = 0;
8           for (int k = cm; k >= 2; --k)
9           {
10              while (ql <= qr && que[ql] >= k)
11                  ++ql;
12              while (r && k - r <= c[i])
13              {
14                  while (ql <= qr && f[cur[que[qr]]] - que[qr] * w[i]
15                      <= f[cur[r]] - r * w[i]) --qr;
16                  que[++qr] = r--;
17              }
18              CkMax(f[cur[k]], f[cur[que[ql]]] + (k - que[ql]) * w[i]);
19          }
20      }
```

斜率优化 DP

- 形如下式的转移宜采用斜率优化：

$$f_i = \max / \min \{f_j + A(i)B(j) + C(i) + D(j)\}$$

其中 A, B, C, D 分别为含对应变量的多项式， A, B 中的最高次数为一次。

- 去除 \max / \min 的限制，移项得到：

$$-A(i)B(j) + f_i - C(i) = f_j + D(j)$$

将每个可能的决策 $(B(j), f_j + D(j))$ 视作平面上的一点，最优决策即用固定斜率 $-A(i)$ 的直线去截这些点，最大化/最小化截距，因而只需要维护这些点的上凸壳/下凸壳。

- 维护的方式视具体情况而定：
 - 若 $A(i), B(j)$ 均单调，用单调队列/单调栈维护即可。
 - 若 $A(i), B(j)$ 其中一个单调，将单调的那个变量视作决策点（若只有 $A(i)$ 单调则需要倒着转移），求最优决策时在上凸壳上二分即可。
 - 若 $A(i), B(j)$ 均不单调，则需要用平衡树维护凸壳或离线后 CDQ 分治。
- 尽量不要用实数判斜率，同时要注意**横坐标相同**的情况要特殊处理，视题目的要求只保留纵坐标最大/最小的点即可。
- 以下为平衡树 Splay 维护下凸壳的模板，利用了平衡树上二分和 Splay 操作，减少了部分常数。

```
1   namespace Hull
2   {
3       const int N = 1e5 + 5;
4       int rt;
5       int fa[N], lc[N], rc[N];
6       int suf[N], pre[N];
7       ll valx[N], valy[N];
```

```

8
9 inline void Init(int x, ll vx, ll vy)
10 {
11     valx[x] = vx;
12     valy[x] = vy;
13 }
14
15 inline bool Slope1(int x, int y, int z)
16 {
17     if (!x || !y || !z)
18         return true;
19     return (valy[y] - valy[x]) * (valx[z] - valx[y])
20         <= (valy[z] - valy[y]) * (valx[y] - valx[x]);
21 }
22
23 inline bool Slope2(int x, int y, ll z)
24 {
25     if (!x || !y)
26         return true;
27     return (valy[y] - valy[x]) <= z * (valx[y] - valx[x]);
28 }
29
30 inline void Rotate(int x)
31 {
32     int y = fa[x], z = fa[y];
33     bool flag = lc[y] == x;
34     int b = flag ? rc[x] : lc[x];
35     fa[x] = z, fa[y] = x;
36     b ? fa[b] = y : 0;
37     z ? (lc[z] == y ? lc[z] : rc[z]) = x : 0;
38     flag ? (rc[x] = y, lc[y] = b) : (lc[x] = y, rc[y] = b);
39 }
40
41 inline bool whichSide(int x)
42 {
43     return rc[fa[x]] == x;
44 }
45
46 inline void Splay(int x, int tar)
47 {
48     while (fa[x] != tar)
49     {
50         if (fa[fa[x]] != tar)
51             Rotate(whichSide(fa[x]) == whichSide(x) ? fa[x] : x);
52         Rotate(x);
53     }
54     !tar ? rt = x : 0;
55 }
56
57 inline int findLeft(int x, int y)
58 {
59     int res = x;
60     while (x)
61     {
62         if (Slope1(pre[x], x, y))
63             res = x, x = rc[x];
64         else
65             x = lc[x];

```

```

66     }
67     return res;
68 }
69
70 inline int findRight(int x, int y)
71 {
72     int res = x;
73     while (x)
74     {
75         if (Slope1(y, x, suf[x]))
76             res = x, x = lc[x];
77         else
78             x = rc[x];
79     }
80     return res;
81 }
82
83 inline void Clear(int &x)
84 {
85     if (!x)
86         return ;
87     clear(lc[x]);
88     clear(rc[x]);
89     fa[x] = pre[x] = suf[x] = 0;
90     x = 0;
91 }
92
93 inline void Insert(int id)
94 {
95     int x = rt, y = 0, dir;
96     while (x)
97     {
98         y = x;
99         if (valx[id] < valx[x])
100             x = lc[x], dir = 0;
101         else
102             x = rc[x], dir = 1;
103     }
104     fa[x = id] = y;
105     if (y) (dir ? rc[y] : lc[y]) = x;
106     Splay(x, 0);
107     if (lc[x])
108     {
109         int z = findLeft(lc[x], x);
110         Splay(z, x);
111         clear(rc[z]);
112         suf[z] = x;
113         pre[x] = z;
114     }
115     if (rc[x])
116     {
117         int z = findRight(rc[x], x);
118         Splay(z, x);
119         clear(lc[z]);
120         pre[z] = x;
121         suf[x] = z;
122     }
123     if (!Slope1(pre[x], x, suf[x]))

```

```

124         {
125             rt = lc[x];
126             rc[rt] = rc[x];
127             fa[rc[x]] = rt;
128             fa[rt] = 0;
129             lc[x] = rc[x] = fa[x] = 0;
130             suf[rt] = rc[rt];
131             pre[rc[rt]] = rt;
132         }
133     }
134
135     inline int Query(int z)
136     {
137         int x = rt, res = 0;
138         while (x)
139         {
140             if (Slope2(pre[x], x, z))
141                 res = x, x = rc[x];
142             else
143                 x = lc[x];
144         }
145         return Splay(res, 0), res;
146     }
147 }

```

- 还可以换一种表示形式：

$$f_i = \max / \min \{B(j)A(i) + D(j) + f_j\} + C(i)$$

- \max / \min 内部的式子可以看作是若干条以 $B(j)$ 为斜率、 $D(j) + f_j$ 为截距的直线，即可用李超线段树维护，相较于平衡树的写法要简单很多，代码见数据结构部分。
- 若采用动态开点的写法，还可支持线段树合并，因而还可支持子树形式的转移。

决策单调性

四边形不等式

- $w(x, y)$ 为定义在整数集合上的二元函数，对于定义域上的任意整数 a, b, c, d ，其中 $a \leq b \leq c \leq d$ ，都有 $w(a, d) + w(b, c) \geq w(a, c) + w(b, d)$ ，称为函数 w 满足四边形不等式。

一维决策单调性

- 在状态转移方程 $f_i = \min_{0 \leq j < i} \{f_j + w(j, i)\}$ 中，若函数 w 满足四边形不等式，则 f 具有决策单调性。
- 用队列维护若干个三元组 (l, r, j) ，表示 $[l, r]$ 内的最优决策均为 j 。
- 在队尾插入时：
 - 若比前一个三元组整个区间内的决策都优，则直接合并，继续检查前一个三元组。
 - 若比前一个三元组整个区间内的决策都不优，则直接插入队尾。
 - 否则在区间上二分找到分界点，修改两个区间的分界后插入队尾。

二维决策单调性

- 在状态转移方程 $f_{i,j} = \min_{i \leq k < j} \{f_{i,k} + f_{k+1,j} + w(i,j)\}$ 中, 若下面三个条件成立:
 - 可规定 $f_{i,i} = w(i,i) = 0$ 。
 - w 满足四边形不等式。
 - 对于任意的 $a \leq b \leq c \leq d$, 有 $w(a,d) \geq w(b,c)$ 。
- 则 f 也满足四边形不等式, 设 $p_{i,j}$ 表示令 $f_{i,j}$ 取到最小值的 k 值, 则恒有 $p_{i,j-1} \leq p_{i,j} \leq p_{i+1,j}$ 。
- 因此我们在转移时只需枚举 $[p_{i,j-1}, p_{i+1,j}]$ 内的 k , 对于长度为 $L+1$ 的区间, 总枚举量为:

$$(p_{2,L+1} - p_{1,L}) + (p_{3,L+2} - p_{2,L+1}) + \cdots + (p_{n-L+1,n} - p_{n-L,n-1}) = p_{n-L+1,n} - p_{1,L}$$

- 因此总的时间复杂度为 $\mathcal{O}(\sum_{i=1}^{n-1} (p_{n-i+1,n} - p_{1,i})) = \mathcal{O}(n^2)$ 。
- 形如 $f_{i,j} = \max_{i \leq k \leq j} / \min_{i \leq k \leq j} \{f_{i,k} + f_{k,j} + C\}, C \in \mathbb{R}$ 也可以套用上述流程优化。

GarsiaWachs 算法

- 上述二维决策单调性能解决的石子合并问题有一种专门的非动态规划算法。
- 设第 i 堆石子的数目为 a_i , 令 $a_0 = a_{n+1} = +\infty$, 具体步骤如下:
 - 找到满足 $a_{k-1} < a_{k+1}$ 最大的 k 。
 - 找到满足 $a_j > a_{k-1} + a_k$ 且 $j < k$ 的最大的 j 。
 - 删除 a_{k-1}, a_k , 在 a_j 后插入 $a_k + a_{k-1}$ 。
 - 重复上述过程直至石子被合并为一堆。
- 空间复杂度 $\mathcal{O}(n)$, 时间复杂度 $\mathcal{O}(n^2)$, 可用平衡树优化至 $\mathcal{O}(n \log n)$ 。

证明 待补充。

```
1  read(n);
2  v.push_back(Maxn);
3  for (int i = 1, x; i <= n; ++i)
4  {
5      read(x);
6      v.push_back(x);
7  }
8  v.push_back(Maxn);
9  while (n > 1)
10 {
11     int j, k;
12     for (k = 1; k <= n; ++k)
13         if (v[k-1] < v[k+1])
14             break;
15     for (j = k-1; j >= 0; --j)
16         if (v[j] > v[k-1] + v[k])
17             break;
18     int sum = v[k-1] + v[k];
19     v.erase(v.begin() + k-1);
20     v.erase(v.begin() + k-1);
21     v.insert(v.begin() + j+1, sum);
22     ans += sum;
23     --n;
24 }
25 printf("%d\n", ans);
```

整除分块优化 DP

典例 [HDU7217](#)

题目大意

- 对长度不超过 n 且序列中最大值不超过 m 的序列 a 取模计数，设序列长度为 l ，则 $\forall 1 \leq i < l, a_i | a_{i+1}$ 。
- $n, m \leq 10^9$ 。

解法

- 若序列 a 中各元素互不相同，则序列 a 的长度不会超过 $1 + \lfloor \log_2 m \rfloor$ ，因而只需统计各元素互不相同的序列 a 的方案，再通过乘组合数插回去即可，设当前长度为 t ，则插回长度不超过 n 的序列的方案数为 $\sum_{i=t}^n \binom{i-1}{t-1} = \binom{n}{t}$ 。
- 设 $f_{t,x}$ 表示从大到小填数，序列已经填了 t 个数字，目前填的最小的数字最大不超过 x 的方案数，有转移：

$$\forall 2 \leq d \leq x, f_{t+1, \lfloor \frac{x}{d} \rfloor} \leftarrow f_{t,x}$$

- 初始时 $f_{1,m} = 1$ ，最后答案为 $\sum \left(f_{t,x} \times x \times \binom{n}{t} \right)$ 。
- 可通过整除分块优化，注意到所有转移到的 x 都可以表示为 $\lfloor \frac{m}{d} \rfloor$ 的形式，总共只有 $2\sqrt{m}$ 个状态，类似 Min_25 筛将 $\lfloor \frac{m}{d} \rfloor$ 按照是否大于 \sqrt{m} 分类即可进行离散化编号，在转移过程中统计答案即可，时间复杂度为 $\mathcal{O}(m^{\frac{3}{4}})$ ，分析与杜教筛的一致。

凸优化 DP

- 一般这类题目会要求选出恰好 K 件物品的最优代价 $f(K)$ ，但 K 较大不能记录在 DP 状态中，且可证明答案函数关于 K 是凸函数。
- 由费用流模型中关于流量的费用函数是凸的，如能设计出一个理论上的费用流模型，即可证明答案函数是凸函数。
- 以最大化代价为例，考虑二分恰好能在答案函数上截到点 $(K, f(K))$ 的直线斜率 C ，就将问题转化为选择每件物品时有额外的代价 $-C$ ，要最大化总代价，设最优化代价时对应的物品件数为 $g(C)$ ，而二分的结果为满足 $g(C) \geq K$ 的最大的 C （记为 C_0 ）。最后答案即为端点为 $(g(C_0 + 1), f(g(C_0 + 1)))$ 和 $(g(C_0), f(g(C_0)))$ 的线段在 $x = K$ 处的取值。

1.4 计算几何

基础操作

- 判断点 A 在线段 PQ 上 即判断 $|AP| + |AQ| = |PQ|$ 。
- 向量的旋转 将向量 $\vec{a} = (x, y)$ 逆时针旋转 θ 得到向量 $\vec{b} = (x \cos \theta - y \sin \theta, x \sin \theta + y \cos \theta)$ ，由三角函数的和角公式可证。
- 曼哈顿距离与切比雪夫距离
 - 二维曼哈顿距离为 $|x_1 - x_2| + |y_1 - y_2|$ ，切比雪夫距离为 $\max\{|x_1 - x_2|, |y_1 - y_2|\}$
 - (x, y) 坐标下的曼哈顿距离，等同于 $(x + y, x - y)$ 坐标下的切比雪夫距离。
 - (x, y) 坐标下的切比雪夫距离，等同于 $(\frac{x+y}{2}, \frac{x-y}{2})$ 坐标下的曼哈顿距离。

欧拉公式

- 在任何一个规则球面地图上，定义 R 为区域个数， V 为顶点个数， E 为边界个数，则恒有：

$$R + V - E = 2$$

- 推论** 凸正多面体（柏拉图立体）有且仅有 5 种。


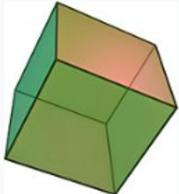

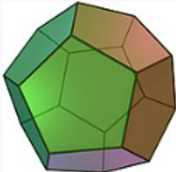
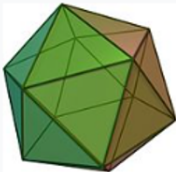
证明 取凸正多面体一个顶点，设其向外有 $n(n \geq 3)$ 条棱，再取凸正多面体一个面，设其为正 $m(m \geq 3)$ 边形。则

$$\begin{aligned} nV &= 2E \Leftrightarrow V = \frac{2E}{n} \\ mR &= 2E \Leftrightarrow R = \frac{2E}{m} \end{aligned}$$

带入欧拉公式，得

$$\frac{1}{m} + \frac{1}{n} = \frac{1}{E} + \frac{1}{2}$$

注意到 m, n 不能同时大于 3，且其中一个等于 3 时另一个不能超过 5。
符合条件的 m, n 的解只有五组，如下图所示。

				
Tetrahedron {3, 3}	Cube {4, 3}	Octahedron {3, 4}	Dodecahedron {5, 3}	Icosahedron {3, 5}

- 设棱长为 a ，五种凸正多面体的各项数据如下（棱切球即过各边中点的球）。

	正四面体	立方体	正八面体	正十二面体	正二十面体
R	4	6	8	12	20
V	4	8	6	20	12
E	6	12	12	30	30
内切球半径	$\frac{\sqrt{6}}{12}a$	$\frac{1}{2}a$	$\frac{\sqrt{6}}{6}a$	$\frac{1}{2}\sqrt{\frac{5}{2} + \frac{11}{10}\sqrt{5}}a$	$\frac{3\sqrt{3}+\sqrt{15}}{12}a$
外接球半径	$\frac{\sqrt{6}}{4}a$	$\frac{\sqrt{3}}{2}a$	$\frac{\sqrt{2}}{2}a$	$\frac{\sqrt{3}}{4}(1 + \sqrt{5})a$	$\frac{\sqrt{10+2\sqrt{5}}}{4}a$
棱切球半径	$\frac{\sqrt{2}}{4}a$	$\frac{\sqrt{2}}{2}a$	$\frac{1}{2}a$	$\frac{3+\sqrt{5}}{4}a$	$\frac{1+\sqrt{5}}{4}a$
表面积	$\sqrt{3}a^2$	$6a^2$	$2\sqrt{3}a^2$	$3\sqrt{25 + 10\sqrt{5}}a^2$	$5\sqrt{3}a^2$
体积	$\frac{\sqrt{2}}{12}a^3$	a^3	$\frac{\sqrt{2}}{3}a^3$	$\frac{15+7\sqrt{5}}{4}a^3$	$\frac{15+5\sqrt{5}}{12}a^3$

Pick 定理

- 定义** 对于一个所有顶点均为整点的简单多边形，定义 S 为这个多边形的面积， i 为严格在这个多边形内部的格点数， b 为在这个多边形边上的格点数，则三者满足关系式 $S = i + \frac{b}{2} - 1$ 。

证明 考虑证明 **引理1** 和 **引理3**，则 Pick 定理显然成立。

- **引理1** 若两个只有一条公共边的多边形满足 Pick 定理，则将两个多边形去掉公共边，合并成的一个多边形也满足 Pick 定理。

证明 设合并的两个多边形为 P, Q ，它们的公共边上的格点数（不包括端点）为 c 。

$$\begin{aligned} S &= S_P + S_Q \\ &= i_P + i_Q + \frac{b_P + b_Q}{2} - 2 \\ &= i - c + \frac{b + 2c + 2}{2} - 2 \\ &= i + \frac{b}{2} - 1 \end{aligned}$$

- **引理2** 有两个只有一条公共边的多边形，若将这两个多边形去掉公共边，合并成的一个多边形满足 Pick 定理，且这两个多边形中有一个满足 Pick 定理，则另一个多边形也满足 Pick 定理。

证明 用类似 **引理1** 的方法即可。

- **引理3** 任意一个三角形都满足 Pick 定理。

证明

1. 已知面积为 1 的正方形满足 Pick 定理，由 **引理1** 得任意大小的矩形都满足 Pick 定理。
2. 将一个矩形拆分为两个全等的直角三角形，证明任意一个直角三角形都满足 Pick 定理。

证明 设两个直角三角形公共边上的格点数（不包括端点）为 c ，原矩形为 R ，拆分出的直角三角形为 T 。

$$\begin{aligned} S_T &= \frac{S_R}{2} \\ &= \frac{i_R}{2} + \frac{b_R}{4} - \frac{1}{2} \\ &= \frac{2i_T + c}{2} + \frac{2b_T - 2c - 2}{4} - \frac{1}{2} \\ &= i_T + \frac{b_T}{2} - 1 \end{aligned}$$

3. 任意一个三角形显然可以由一个矩形拆去不多于3个的直角三角形得到，由 **引理2** 可知得证。

凸包

- 常见的求法为 Graham 扫描法，先找出最左下角的点，将其余点按照相对于该点的极角排序（实现时不需要真的求出极角，只需要通过叉积判断即可），然后维护一个栈，将点按照极角序加入该栈，同样通过叉积判断，若不满足凸包的形态则弹栈，最后即能求出凸包。

闵可夫斯基和

- 定义点的加法为对应坐标相加，则点集 A 和点集 B 的闵可夫斯基和

$$\{x + y | x \in A, y \in B\}$$

- 定义凸集 C （即轮廓是凸包的区域）

$$\forall x, y \in C, \forall \lambda \in [0, 1], \lambda x + (1 - \lambda)y \in C$$

- 容易根据定义证明两个凸集的闵可夫斯基和仍然是凸集，因此两个点集的闵可夫斯基和的凸包可通过两个凸包的闵可夫斯基和得到。
- 具体求解相当于将一个凸包绕着另一个凸包的轮廓转一圈，可通过双指针实现。

```

1  typedef long long ll;
2  const int N = 4e5 + 5;
3  int top;
4
5  struct point
6  {
7      ll x, y;
8
9      point() {}
10     point(ll x, ll y):
11         x(x), y(y) {}
12
13     inline void scan()
14     {
15         int _x, _y;
16         cin >> _x >> _y;
17         x = _x, y = _y;
18     }
19
20     inline bool operator < (const point &a) const
21     {
22         return x < a.x || x == a.x && y < a.y;
23     }
24
25     inline ll dist() const
26     {
27         return x * x + y * y;
28     }
29
30     inline point operator + (const point &a) const
31     {
32         return point(x + a.x, y + a.y);
33     }
34
35     inline point operator - (const point &a) const
36     {
37         return point(x - a.x, y - a.y);
38     }
39
40     inline ll operator * (const point &a) const
41     {
42         return x * a.y - y * a.x;
43     }
44 }stk[N];
45
46 inline bool cmp(const point &x, const point &y)
47 {
48     ll del = x * y;
49     if (del == 0)
50         return x.dist() < y.dist();
51     else
52         return del > 0;
53 }
54
55 struct hull
56 {
57     point p[N];

```

```

58     int n;
59
60     inline void normal() // Graham 扫描法求凸包
61     {
62         int id = 1;
63         for (int i = 2; i <= n; ++i)
64             if (p[i] < p[id])
65                 id = i;
66         if (id != 1)
67             std::swap(p[id], p[1]);
68         for (int i = n; i >= 2; --i)
69             p[i] = p[i] - p[1];
70         std::sort(p + 2, p + n + 1, cmp);
71
72         stk[top = 1] = point(0, 0);
73         for (int i = 2; i <= n; ++i)
74         {
75             while (top > 1 && (p[i] - stk[top - 1]) * (stk[top] - stk[top -
1]) >= 0) --top;
76             stk[++top] = p[i];
77         }
78         n = top;
79         for (int i = 2; i <= top; ++i)
80             p[i] = stk[i] + p[1];
81     }
82
83     inline void scan(int _n)
84     {
85         n = _n;
86         for (int i = 1; i <= n; ++i)
87             p[i].scan();
88         normal();
89     }
90
91     inline hull operator + (hull a)
92     {
93         hull f;
94         f.p[f.n = 1] = p[1] + a.p[1];
95         int i = 1, j = 1;
96         while (i <= n || j <= a.n)
97         {
98             point tx = p[(i - 1) % n + 1] + a.p[j % a.n + 1],
99                 ty = p[i % n + 1] + a.p[(j - 1) % a.n + 1];
100             if ((tx - f.p[f.n]) * (ty - f.p[f.n]) >= 0)
101                 ++j, f.p[++f.n] = tx;
102             else
103                 ++i, f.p[++f.n] = ty;
104         }
105         for (int i = 2; i <= f.n; ++i)
106             f.p[i] = f.p[i] - f.p[1];
107         stk[top = 1] = point(0, 0);
108         for (int i = 2; i <= f.n; ++i)
109         {
110             while (top > 1 && (f.p[i] - stk[top - 1]) * (stk[top] - stk[top
- 1]) >= 0) --top;
111             stk[++top] = f.p[i];
112         }
113         f.n = top;

```

```

114         for (int i = 2; i <= top; ++i)
115             f.p[i] = stk[i] + f.p[1];
116         --f.n; //最后一个点一定和第一个点相同
117         return f;
118     }
119 };

```

- 若是求上/下凸壳，写法会有些不同。

```

1  struct hull
2  {    //求一个上凸壳，使得其余点均在该凸壳下方
3      point p[N];
4      int n;
5
6      inline void normal()
7      {
8          top = 0;
9          for (int i = 1; i <= n; ++i)
10             {
11                 while (top > 1 && (p[i] - stk[top - 1]) * (stk[top] - stk[top -
12                     1]) <= 0)
13                     --top;
14                 stk[++top] = p[i];
15             }
16             n = top;
17             for (int i = 1; i <= top; ++i)
18                 p[i] = stk[i];
19         }
20
21         inline hull operator + (hull a)
22         {
23             hull f;
24             f.p[f.n = 1] = p[1] + a.p[1];
25             int i = 1, j = 1;
26             while (i < n && j < a.n)
27             {
28                 point tx = p[i] + a.p[j + 1],
29                     ty = p[i + 1] + a.p[j];
30                 if ((tx - f.p[f.n]) * (ty - f.p[f.n]) <= 0)
31                     ++j, f.p[++f.n] = tx;
32                 else
33                     ++i, f.p[++f.n] = ty;
34             }
35             while (i < n)
36                 f.p[++f.n] = p[++i] + a.p[j];
37             while (j < a.n)
38                 f.p[++f.n] = p[i] + a.p[++j];
39             f.normal();
40             return f;
41         }
42     };

```

