资料

kmp 与 z函数

前缀函数

```
vector<int> prefix_function(string s) {
  int n = (int)s.length();
  vector<int> pi(n);
  for (int i = 1; i < n; i++) {
    int j = pi[i - 1];
    while (j > 0 && s[i] != s[j]) j = pi[j - 1];
    if (s[i] == s[j]) j++;
    pi[i] = j;
  }
  return pi;
}
```

kmp

```
vector<int> find_occurrences(string text, string pattern) {
  string cur = pattern + '#' + text;
  int sz1 = text.size(), sz2 = pattern.size();
  vector<int> v;
  vector<int> lps = prefix_function(cur);
  for (int i = sz2 + 1; i <= sz1 + sz2; i++) {
    if (lps[i] == sz2) v.push_back(i - 2 * sz2);
  }
  return v;
}
```

对于一个长度为 n 的字符串 s，定义函数 z[i] 表示 s 和 s[i,n-1]（即以 s[i] 开头的后缀）的最长公共前缀（LCP）的长度，则 z 被称为 s 的 Z 函数。特别地，z[0] = 0。

```
vector<int> z_function(string s) {
  int n = (int)s.length();
  vector<int> z(n);
  for (int i = 1, l = 0, r = 0; i < n; ++i) {
    if (i <= r && z[i - l] < r - i + 1) {
      z[i] = z[i - l];
    } else {
      z[i] = max(0, r - i + 1);
      while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
    }
    if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
  }
  return z;
}
```

exGcd

```
void exGcd(int a, int b, int& x, int& y) {
    if (!b) { x = 1, y = 0; }
    else exGcd(b, a % b, y, x), y -= a / b * x;
}
```

组合数学

```
i64 qpow(i64 x, i64 p) {
    i64 ret = 1;
    while (p) {
        if (p & 1)ret = ret * x % mod;
        p >>= 1;
        x = x * x % mod;
    }
    return ret;
}

#define inv(x) qpow(x,mod-2)

std::vector<int> fact(1, 1);
std::vector<int> inv_fact(1, 1);

auto get_fact(int x, bool inv = 0) {
    while ((int)fact.size() < x + 1) {
        fact.push_back(1ll * fact.back() * fact.size() % mod);
        inv_fact.push_back(inv(fact.back()));
    }
    return (inv ? inv_fact[x] : fact[x]);
}

auto get_inv_fact(int x) { return get_fact(x, 1); }

i64 C(int n, int k) {
    if (k<0 || k>n)return 0;
    return 1ll * get_fact(n) * get_inv_fact(k) % mod * get_inv_fact(n - k) % mod;
}

i64 A(int n, int k) {
    return 1ll * get_fact(n) * get_inv_fact(n - k) % mod;
}

i64 F(int n) { return get_fact(n); }
```

马拉车

```
vector<int> d1(n);
for (int i = 0, l = 0, r = -1; i < n; i++) {
  int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
  while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) {
    k++;
  }
  d1[i] = k--;
  if (i + k > r) {
    l = i - k;
    r = i + k;
  }
}

vector<int> d2(n);
for (int i = 0, l = 0, r = -1; i < n; i++) {
  int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
  while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) {
    k++;
  }
  d2[i] = k--;
  if (i + k > r) {
    l = i - k - 1;
    r = i + k;
  }
}
```

凸包

定义
凸多边形
凸多边形是指所有内角大小都在$[0, \pi]$范围内的 简单多边形。

凸包
在平面上能包含所有给定点的最小凸多边形叫做凸包。

其定义为：对于给定集合 X，所有包含 X 的凸集的交集 S 被称为 X 的 凸包。

实际上可以理解为用一个橡皮筋包含住所有给定点的形态。

凸包用最小的周长围住了给定的所有点。如果一个凹多边形围住了所有的点，它的周长一定不是最小，如下图。根据三角不等式，凸多边形在周长上一定是最优的。

Andrew 算法求凸包

常用的求法有 Graham 扫描法和 Andrew 算法，这里主要介绍 Andrew 算法。

性质

该算法的时间复杂度为 O($n \log n$)，其中 n 为待求凸包点集的大小，复杂度的瓶颈在于对所有点坐标的双关键字排序。

过程

首先把所有点以横坐标为第一关键字，纵坐标为第二关键字排序。

显然排序后最小的元素和最大的元素一定在凸包上。而且因为是凸多边形，我们如果从一个点出发逆时针走，轨迹总是「左拐」的，一旦出现右拐，就说明这一段不在凸包上。因此我们可以用一个单调栈来维护上下凸壳。

因为从左向右看，上下凸壳所旋转的方向不同，为了让单调栈起作用，我们首先 升序枚举 求出下凸壳，然后 降序 求出上凸壳。

求凸壳时，一旦发现即将进栈的点（P）和栈顶的两个点（S_1, S_2，其中 S_1 为栈顶）行进的方向向右旋转，即叉积小于 0：

$\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} < 0$，则弹出栈顶，回到上一步，继续检测，直到

$\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} \geq 0$ 或者栈内仅剩一个元素为止。

通常情况下不需要保留位于凸包边上的点，因此上面一段中

$\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} < 0$ 这个条件中的「<」可以视情况改为 $\leq$，同时后面一个条件应改为 >。

实现

代码实现

```
// stk[] 是整型，存的是下标
// p[] 存储向量或点
tp = 0;                          // 初始化栈
std::sort(p + 1, p + 1 + n);  // 对点进行排序
stk[++tp] = 1;
// 栈内添加第一个元素，且不更新 used，使得 1 在最后封闭凸包时也对单调栈更新
for (int i = 2; i <= n; ++i) {
  while (tp >= 2  // 下一行 * 操作符被重载为叉积
         && (p[stk[tp]] - p[stk[tp - 1]]) * (p[i] - p[stk[tp]]) <= 0)
    used[stk[tp--]] = 0;
  used[i] = 1;  // used 表示在凸壳上
  stk[++tp] = i;
}
int tmp = tp;  // tmp 表示下凸壳大小
for (int i = n - 1; i > 0; --i)
  if (!used[i]) {
    // ↓求上凸壳时不影响下凸壳
    while (tp > tmp && (p[stk[tp]] - p[stk[tp - 1]]) * (p[i] - p[stk[tp]]) <= 0)
      used[stk[tp--]] = 0;
    used[i] = 1;
    stk[++tp] = i;
  }
for (int i = 1; i <= tp; ++i)  // 复制到新数组中去
  h[i] = p[stk[i]];
int ans = tp - 1;
```

根据上面的代码，最后凸包上有 ans 个元素（额外存储了 1 号点，因此 h 数组中有 $ans+1$ 个元素），并且按逆时针方向排序。周长就是所有相邻点的距离和。