readme

`Datamaker.h` 是 @Arahc （别点了，没有链接）整合的一个用于辅助制造测试数据的头文件。它的存在让出题造数据这一部分变得不那么困难。

下面将会列出本头文件的所有功能。阅读此文档全部或部分内容后（请注意，无论如何，请务必阅读**内置函数**一栏的内容），您就可以更轻松地制造需要的题目数据了。

在后文中，对于所有类型为任意数的参数或返回值类型，用 `anytyp` 代替，对于所有类型为任意容器的参数或返回值类型，用 `anycon` 代替，对于所有类型为任意指针的参数或返回值类型，用 `anyptr` 代替。

当前版本适配 windows（64 位）和 linux 系统。

当使用本头文件时，请不要再使用 C++ 头文件内置的 `srand`、`rand`、`random_shuffle`、`mt_19937` 等随机数工具。而是使用本头文件提供的随机数工具。否则可能会出现一些错误。

除了内置函数外，所有函数均封装在其对应类中。例如，使用 `rnd` 生成器（默认随机数生成器），调用 `rand(1,100)` 生成 $[1,100]$ 中的随机数时，**需要写成** `rnd.rand(1,100)` 而不是直接写成 `rand(1,100)`。

请注意，使用本头文件时，请勿定义如下变量或函数名：

```
registerGen, startTest, autoGenerate, rnd, cst, Consts_t, Random_t, Tree, Graph, Edge, Vector, Polygon
```

内置函数

registerGen

函数 `registerGen` 的类型为 `void registerGen (int argc, char *argv[])`，**您应当在主函数的第一条语句中调用此函数**。以此来初始化随机种子。

随机种子与您命令行的参数有关。例如，当您的生成数据的可执行文件名为 `run` 时，运行 `./run ababab` 即可将随机种子设为字符串 `ababab`（的 Hash 值）。无论何时运行，无论运行的机器系统，只要生成数据的代码和随机种子相同，得到的数据就是一样的。

startTest

函数 `startTest` 的类型为 `FILE* startTest (string filename)`。它可以将您在此之后的输出重定向至文件 `filename` 中。该函数将返回文件操作的结构体 `FILE*`。

autoGenerate

函数 `autoGenerate` 的类型为 `void autoGenerate (string proname, vector<int> subtasks, void (*Generate)(int,int), string stdname="", string valname="")`。它是批量生成数据的利器。下面将会一一为您介绍各个参数的含义。

- `proname`：题目英文名。生成数据的文件名格式为 **`proname`X-Y.in/out**，其中 X 为 subtask 编号，Y 表示这是该 subtask 的第几个数据。特别地，当只有一个 subtask 时（一般意味着本题不开设子任务），生成的文件名格式为 **`proname`Y.in/out**。
- `subtasks`：一个 `vector` 类型，其大小为本题的子任务数量，各个元素按顺序依次表示子任务的测试点数。做为参数传入的 `vector` 下标从 0 开始，但是数据文件名和 `Generate` 函数传入的参数将会从 1 开始。

- `Generate`：一个 `void` 类型的函数，传入两个 `int` 类型的参数，分别表示该测试点所在的子任务编号和它是这个子任务的第几个测试点。**您需要自行实现这个函数**来生成输入数据。您只需将数据输出到标准输出即可，请勿自行重定向输出到文件中。
- `stdname`：std 的**可执行文件名**，请注意，您的 std 中不应该使用文件输入输出，应当采用标准输入输出。若不传入，则不会执行 std，同时不会自动生成 `.out` 文件。
- `valname`：validator（检验数据是否合法的校验器）的**可执行文件名**。该程序需要在数据不合法时返回非零值，合法时返回 0。同样地，validator 应当采用标准输入输出。若不传入，则不会执行数据校验器。

当 std 返回非零值（或 RE）或 validator 返回非零值

该函数运行的流程是：对于每个数据，它将自动将文件重定向这个数据点上，并调用 `Generate` 函数，随后运行 validator 校验数据（若未传入次参数则跳过这一步），若数据合法则运行 std 生成输出数据。

随机数（Random_t）

生成随机数有关的函数，即随机数生成器，封装入了一个类 `Random_t` 中。并且我们已经为您定义好了一个随机数生成器 `rnd`。请注意，除非您知道您正在做什么，并且如何操作，请不要自行定义其它随机数生成器。

rand

函数 `rand` 的类型为 `anytyp rand (anytyp from, anytyp to, int t=0)`。当 $t=0$ 时，该函数将会返回一个位于 $[from,to]$ 区间中的随机数字，返回值类型与传入参数类型相同。特别地，当传入类型为浮点数时，返回的数字将会在 $[from,to)$ 之间随机而非 $[from,to]$。

当 $t\neq 0$ 时，此时返回值如下：

$$\text{rand}(from,to,t)=\begin{cases}\max\{\text{rand}(from,to,0),\text{rand}(\text{from},\text{to},\text{t}-1)\} & t>0\\ \min\{\text{rand}(from,to,0),\text{rand}(\text{from},\text{to},\text{t}+1)\} & t<0\end{cases}$$

any

函数 `any` 的类型为 `anytyp any (anycon, int t=0)`。该函数将会在容器内随机选择一个数返回。容器可以是 `string`、`vecotr`、`set`、`map`……等等。

函数 `any` 也可以传入第二个参数 t，含义和 `rand` 的第三个参数类似。使用这个偏移量会使在给定容器中返回的数字在容器内的位置更靠近 `begin()` 或 `end()`。

shuffle

函数 `shuffle` 的类型为 `void shuffle (anyptr from, anyptr last)`。它将会随机打乱 $[from,last)$ 内的数字。

permutation

函数 `permutation` 的类型为 `vector<anytyp> permutation (anytyp first, anytyp last)`。其参数应当为**非** `short,unsigned short` **的非负整数**。它会返回一个 `vector`，其中存储的为 $[first,last]$ 的数字中的一个随机排列。

distinct

函数 `distinct` 的类型为 `vector<anytyp> distinct (int size, anytyp first, anytyp last)`。其参数应当为**非** `short,unsigned short` **的非负整数**，且 $last-first+1\geq size$。它会返回一个 `vector`，其中存储了均匀随机生成的 $size$ 个在区间 $[first,last]$ 中的互不相同的数字。`vector` 中的数字是无序的。

partition

函数 `partition` 的类型为 `vector<anytyp> partition (int size, anytyp sum, anytyp minnum)`。其参数应当为**非** `short,unsigned short` **的非负整数**，且 $sum \geq size \cdot minnum$。它会返回一个 `vector`，其中存储了均匀随机生成的一种将 sum 拆分成 $size$ 个 $\geq minnum$ 的非负整数的和的拆分方案。`vector` 中的数字是无序的。

边（Edge）

为了方便构建树和图，我们封装了一条边的情况。并给予了其一些基本的操作。因为单独需要定义边的问题并不常见，这里直接给出源码，大家可自行查看。我们认为一条边为 $u \to v$ 的边权为 w 的边。

```cpp
template <typename T = int>
struct Edge
{
    int u, v;
    T w;
    Edge(int a = 0, int b = 0, T c = T(0))
    {
        u = a, v = b;
        w = c;
    }
    bool operator==(Edge<T> b) const
    {
        Edge a = *this;
        return (a.u == b.u && a.v == b.v && a.w == b.w);
    }
    bool operator<(Edge<T> b) const
    {
        Edge a = *this;
        if (a.w == b.w)
            return a.u == b.u ? a.v < b.v : a.u < b.u;
        return a.w < b.w;
    }
    void reverse()
    {
        u ^= v ^= u ^= v;
    }
};
```

树（Tree）

您可以通过 `Tree<typename> tr(n);` 来定义一个边权的类型为 `typename`（可以不填，即 `Tree<> tr(n)`，默认类型为 `int`）的，点和边的下标均从 0 开始的大小为 n 的树（若不传入 n，则 n 默认为 1），记为 `tr`。本类内所有的随机均由默认随机数生成器 `rnd` 生成。

值得注意的是，本类型并没有内设点权。如果您需要给树上每个节点维护点权的话，请另开数组维护每个节点的点权。

当您构造了一棵树 A 时，您可以使用 `A[i]` 来访问 A 的第 i 条边（下标从 0 开始）。一条边的类型为 `Edge` 类型。

补充：一些题目可能对于边权有特殊限制。此时不建议您使用该类型内的边权系统维护。而是自行在生成完树的形态后修改边权。下一部分的“图”也一样。

size

函数 `size` 的类型为 `size_t size ()`，该函数返回树的大小。

`size_t` 类型的本质是 `unsigned long int`，使用 printf 的时候为 `%ld`。STL 容器（如 `vector` 等）中返回容器大小的返回值也是这个类型。

shuffle

函数 `shuffle` 的类型为 `vector<int> shuffle (bool fl=1,bool op=1)`，它将打乱当前边的顺序，若 $fl=1$，则同时会打乱结点的编号。若 $op=0$，每条边将有 0.5 的概率交换其存储的 u,v 节点（适合无向树）。若 $op=1$，则不会交换一条边链接的两点（用于有向树）。该函数的返回值为一个长度为 n 的 `vector<int> pos`，表示打乱节点编号前的 i 号节点对应打乱编号后的 pos_i 号节点（若 $op=0$，则返回的 `vector` 就是 $0,1,2,\cdots,n-1$）。

random1

函数 `random1` 的类型为 `void random1 (int t1=0, anytp (*Weight)() = []{return T(0);}))`。它将会将这棵树设置为一个随机树。其中以 0 为根，$fa_i=\operatorname{rand}(0,i-1,t_1)$。边权生成函数为您自设的函数 `Weight`（不传参，函数名可改），默认为 0。其中每条边的存储方式都是由 i 指向 fa_i 的。

random2

函数 `random2` 的类型为 `void random2 (double l=0, double r=1, anytp (*Weight)() = []{return T(0);})`。它将会生成一个随机树。其中以 0 为根，$fa_i=\operatorname{rand}(\lfloor l(i-1)\rfloor,\lceil r(i-1)\rceil)$。边权生成函数为您自设的函数 `Weight`（不传参，函数名可改），默认为 0。其中每条边的存储方式都是由 i 指向 fa_i 的。

random3

函数 `random3` 的类型为 `void random3 (int K=1, anytp (*Weight)() = []{return T(0);})`。它将会生成一个随机树。其中以 0 为根，$fa_i=\operatorname{rand}(\max\{0,i-K\},i-1)$。边权生成函数为您自设的函数 `Weight`（不传参，函数名可改），默认为 0。。其中每条边的存储方式都是由 i 指向 fa_i 的。

chain

函数 `chain` 的类型为 `void chain (int rt=0, anytp (*Weight)() = []{return T(0);})`，它将会生成一条以 `rt` 为一端，`(n+rt-1)%n` 为另一端的链。其中第 i 条边由 $(rt+i)\bmod n$ 指向 $(rt+i+1)\bmod n$。边权生成函数为您自设的函数 `Weight`（不传参，函数名可改），默认为 0。

star/flower

函数 `star` 和函数 `flower` 的类型为 `void star/flower (int rt=0, anytp (*Weight)() = []{return T(0);})`，它将会生成一个 `rt` 为中心的星形图（菊花图）。其中每条边都由 rt 出发指向其叶子节点。边权生成函数为您自设的函数 `Weight`（不传参，函数名可改），默认为 0。

该函数之所以有两个命名，是因为这样的图（存在一个点的度为树的大小减一）在国外被称为 star（星形图），在中国被称为菊花图。

limchild

函数 `limchild` 的类型为 `void limchild(int rt=0, int K=2, anytp (*Weight)() = []{return T(0);})`，它将会生成一个以 `rt` 为根的随机 K 叉树。边权生成函数为您自设的函数 `Weight`（不传参，函数名可改），默认为 0。

树的生成方式为：维护一个可重集，初始时插入 K 个 rt，随后对于每个 i，随机一个数作为父亲，并删除一个这个数字，然后在集合中插入 K 个 i。

因此该函数的时间复杂度和空间复杂度均为 nK 的。

complete

函数 `complete` 的类型为 `void complete (int rt=0, int K=2, anytp (*Weight)() = []{return T(0);})`，它将会生成一个 rt 为根的完全 K 叉树。边权生成函数为您自设的函数 `Weight`（不传参，函数名可改），默认为 0。树的生成方式为：对于每个节点 i，设置一条由 $\left\lfloor\frac{i}{K}\right\rfloor$ 连向 i 的边，最后交换 0 节点和 rt 节点。

silkworm

函数 `silkworm` 的类型为 `void silkworm (int rt=0, anytyp L=0, anytyp R=0, int t=0`，它将会生成一个以 rt 为一段的毛毛虫。具体生成方式为：用编号为 $[0,\lceil\frac{n}{2}\rceil)$ 的点生成一个长为 $\lceil\frac{n}{2}\rceil$ 的链，对于编号不少于 $\lceil\frac{n}{2}\rceil$ 的点，将 i 号点向 $i-\lceil\frac{n}{2}\rceil$ 号点连边。边权生成函数为您自设的函数 `Weight`（不传参，函数名可改），默认为 0。最后交换 0 节点和 rt 节点。

redirect

函数 `redirect` 的类型为 `void redirect (int rt=0, bool op=1)`，它将会统一修改树上所有边的方向。以 rt 为根，若 $op=1$ 则由深度小的点指向深度大的点；若 $op=0$ 则由深度大的点指向深度小的点。

`redirect` 的实现原理即为对原树进行一次遍历，时间复杂度为 $\mathcal{O}(n)$ 的。

addpoint

函数 `addpoint` 的类型为 `void addpoint (int x=0, anytyp w=0)`，它将在这棵树内追加一个叶子，这个叶子与 x 相连，边权为 w。

merge

函数 `merge` 的类型为 `void merge (Tree B, int u=0, int v=0, anytyp w=0)`，调用 `A.merge(B, ...)`，它将会合并两棵树 A,B。其中新树的随机生成器与 A 相同。合并的方式为添加一条边权为 w 的边，链接 A 树的节点 u 和 B 树的节点 v。合并后的树将会储存在 A 中。合并后 A 树内的节点编号不变，B 树内节点编号整体加上 $|A|$。

`merge` 的时间复杂度为 $\mathcal{O}(|B|)$ 的。

expand

函数 `expand` 的类型为 `void expand (Tree B, int con1=0, int con2=0)`，调用 `A.expand(B, ...)`，它将会把 A 树中的每个节点替换为一棵 B 树。对每个点进行替换的时候，对于该点连出去的边，这条边在替换后会变为由 B 树上对应的 con_1 节点出发；对于连向该点的边，这条边在替换后会变为连向 B 树上对应的 con_2 节点。合并后，A 树的节点编号会乘上 $|B|$，每个点替换后得出的节点编号依次替补。

con_1 和 con_2 可以传入 -1，此时 A 树上的每个点对应在 B 树上的点将会是随机的。

`expand` 的时间复杂度为 $\mathcal{O}(|A|\,|B|)$ 的。

图（Graph）

您可以通过 `Graph<typename> G(n,m);` 来定义一个边权的类型为 `typename`（可以不填，即 `Graph<> G(n,m)`，默认类型为 `int`）的，点和边的下标均从 0 开始的，n 点 m 边的图（若不传入 n,m，则默认 $n=1,m=0$），记为 `G`。它在内部实现的时候，均以有向图的形式实现。强烈建议不要强行将每条边的反边加入到图中使其变为无向图，因为大部分函数针对有向图和无向图无论是实现还是具体含义都是一样的。

同样地，本类内所有随机均由默认随机树生成器 `rnd` 生成。

当您构造了一个图 G 时，您可以使用 `G[i]` 来访问 G 的第 i 条边（下标从 0 开始）。一条边的类型为 `Edge` 类型。

除非特殊说明，所有的随机生成的图均默认为**无重边自环**的简单图。如果需要重边自环，请在最后手动添加。**手动添加的重边、自环和反边可能会影响本类中部分函数的实现正确性。**

请注意：请确保您真的需要使用图类型而非只需在树类型上做一些简单的修改。因为树类型的功能比图类型丰富得多。

在使用后文里大部分的构造函数时，请保证边数 m 符合要求。

size

函数 `size` 的类型为 `pair<size_t,size_t> size ()`，该函数返回的第一个数为该图的点数，第二个数为该图的边数。

shuffle

函数 `shuffle` 的类型为 `vector<int> shuffle (bool fl=1, bool op=1)`，它将打乱当前边的顺序，若 $fl=1$，则同时会打乱结点的编号。若 $op=0$，每条边将有 0.5 的概率交换其存储的 u,v 节点（适合无向图）。若 $op=1$，则不会交换一条边连接的两点（用于有向图）。该函数的返回值为一个长度为 n 的 `vector<int> pos`，表示打乱节点编号前的 i 号节点对应打乱编号后的 pos_i 号节点（$fl=0$ 则返回的 `vector` 就是 $0,1,2,\cdots,n-1$）。

random1

函数 `random1` 的类型为 `void random1 (bool op=0, anytp (*Weight)() = []{return T(0);})`，它将会将这个图设置为一个随机图（**不保证连通**）。若 $op=0$，则不会出现反向边（即 $u\to v$ 和 $v\to u$ 不会同时出现），适用于有向图；若 $op=1$，则允许出现反向边。边权生成函数为您自设的函数 `Weight`（不传参，函数名可改），默认为 0。

random2

函数 `random2` 的类型为 `void random2 (bool op=0, anytp (*Weight)() = []{return T(0);})`，它将会将这个图设置为一个随机图（**保证连通**）。若 $op=0$，则不会出现反向边（即 $u\to v$ 和 $v\to u$ 不会同时出现），适用于有向图；若 $op=1$，则允许出现反向边。边权生成函数为您自设的函数 `Weight`（不传参，函数名可改），默认为 0。

random1 和 random2 的唯一区别在于，前者不保证图连通，后者保证连通。

DAG

函数 `DAG` 的类型为 `void DAG (vector<int> vec={-1}, anytp (*Weight)() = []{return T(0);})`，它将会将这个图设置为一个**保证弱连通**的 DAG。其中 vec 为一个大小为 n 的 `vector`，表示这个 DAG 的一种拓扑序（下标从 0 开始），默认为随机拓扑序。边权生成函数为您自设的函数 `Weight`（不传参，函数名可改），默认为 0。

若拓扑序无要求（随机生成），`vector` 内请传入一个整数 -1。若传入的 `vector` 大小不为 n，则 n 将会自动设置为其大小。

complete

函数 `complete` 的类型为 `void complete (anytp (*Weight)() = []{return T(0);})`，它将会将这个图设置为一个完全图。若当前图的 $m\neq\frac{n(n-1)}{2}$，则 m 将会自动设定为 $\frac{n(n-1)}{2}$。边权生成函数为您自设的函数 `Weight`（不传参，函数名可改），默认为 0。

若您需要的是一个有向完全图，请自行给每条边添加反边。否则您得到的将会是一个竞赛图，每条边的方向是随机的。

limdegree1

函数 `limdegree1` 的类型为 `void limdegree1 (vector<int> vec, bool op=1, anytp (*Weight)() = []{return T(0);})`。它会生成一个**不保证连通**的简单有向图。若 $op=1$，则点 i 的**出度**为 vec_i；否则点 i 的**入度**为 vec_i。边权生成函数为您自设的函数 `Weight`（不传参，函数名可改），默认为 0。

受随机数影响，其复杂度期望为 $m\log n$。最优 $\mathcal{O}(m)$ 最坏 $\mathcal{O}(\max\{n^2,m\})$。设置时，$n,m$ 会自动调整为给定 `vector` 的大小和合法边数。

请注意：可能同时出现 $u\to v$ 和 $v\to u$ 的边。

limdegree2

函数 `limdegree3` 的类型为 `void limdegree3 (vector<int> vec, anytp (*Weight)() = []{return T(0);})`。它会生成一个**不保证连通**的简单无向图，其中点 i 的度为 vec_i。边权生成函数为您自设的函数 `Weight`（不传参，函数名可改），默认为 0。

复杂度严格 $m\log n$。设置时，n,m 会自动调整为给定 `vector` 的大小和合法边数。

circle

函数 `circle` 的类型为 `void circle (anytp (*Weight)() = []{return T(0);})`，它将会将这个图设置为一个环。若当前图的 $m\neq n$，则 m 将会自动设定为 m。边权生成函数为您自设的函数 `Weight`（不传参，函数名可改），默认为 0。第 i 条边由 i 连向 $(i+1)\bmod n$。

当 $n=1$ 时，调用 `circle` 函数并不会生成一个自环。且会把 m 设置为 0。

bipartite

函数 `bipartite` 的类型为 `void bipartite (int k, anytp (*Weight)() = []{return T(0);})`，它将会将这个图设置为一个随机二分图（**不保证连通**）。其中二分图的左部为 $[0,k)$ 编号内的点，右部为 $[k,n)$ 编号内的点。边的方向均为左部连接到右部。边权生成函数为您自设的函数 `Weight`（不传参，函数名可改），默认为 0。

hackspfa

函数 `hackspfa` 的类型为 `void hackspfa (anytyp L=0, anytyp R=0)`，它将会将这个图设置为一个具备卡 SPFA 性质的，源点为 0 节点的**连通图**。边权在 $[L,R]$ 范围内。

使用本函数时，请保证 $2n\leq m\leq 2.5n$（推荐 $m=2n$）。否则效果会大打折扣。

另外，调用本函数结束后，请不要按顺序直接输出此时的边，而是调用一次 `shuffle` 函数，否则会几乎没有效果。

PS：本质就是网格图。

redirect

函数 `redirect` 的类型为 `void redirect (int rt=0, bool op=1)`，它将会将这个图的边进行重定向。。具体地，它将会从 `rt` 出发对图进行 bfs，在 bfs 过程中对边按 bfs 的方向（$op=1$）或反方向（$op=0$）重定向。它可以将一个任意形态的连通图转化为 DAG。

您需要保证本图连通。否则它将只会对 rt 所在的连通块进行边重定向。

merge

函数 `merge` 的类型为 `void mereg (Graph B, vector<Edge> edg1={}, vector<Edge> edg2={})`，调用 `A.merge(B,edg1,edg2)`，它将会把 A,B 两个图合并。两个图之间用 `edg1,edg2` 内的边链接（可以不传该参数，此时则不会添加其它边），`edg1` 中一条边的 `u,v,w` 表示一条从 A 图内的点 u 连向 B 图内的点 v 的，边权为 w 的边；`edg2` 中一条边的 `u,v,w` 表示一条从 B 图内的点 u 连向 A 图内的点 v 的，边权为 w 的边。

添加完成后，B 图内的点编号整体加上 $|A|$。

`merge` 的复杂度是 $\mathcal{O}(B.m+|edg1|+|edg2|)$ 的。

我们还准备了 `void merge (Tree B, vector<Edge> edg1={}, vector<Edge> edg2={})`，它可以将一图与一棵合并。工作原理基本相同。

向量（Vector）

为了方便构建多边形类，我们封装了向量的情况。并给予了其一些基本的操作。因为单独需要定义向量的问题并不常见，这里直接给出源码，大家可自行查看。

```cpp
template <typename T = int>
struct Vector
{
    T x, y;
    Vector(T X = T(0), T Y = T(0)) : x(X), y(Y) {}
    ~Vector() {}

    bool operator==(Vector b)
    {
        return x == b.x && y == b.y;
    }
    bool operator<(Vector b)
    {
        return atan2(y, x) < atan2(b.y, b.x);
    }
};
```

多边形（Polygon）

您可以通过 `Polygon<typename> pol(n);` 来定义一个点的坐标的类型为 `typename`（可以不填，即 `Tree<> tr(n)`，默认类型为 `int`）的，点的下标从 0 开始的大小为 n 的多边形（若不传入 n，则 n 默认为 0），记为 `pol`。初始条件下每个点都是原点。本类内所有的随机均由默认随机数生成器 `rnd` 生成。

您也可以通过 `Polygon<typename> pol(vector<Vector<typename>> vec);` 来直接用顶点序列 `vec` 生成一个多边形。

当您构造了一个多边形 P 后，您可以使用 `P[i]` 来访问 P 的第 i 个点（下标从 0 开始）。一个点的类型为 `Vector` 类型。

您可能更希望使用 double 类的多边形，但需要注意的是这样的随机点无法保证坐标控制在一定小数位数之内（坐标小数位数可能很长）。因此请尽可能地使用整数类型避免精度问题。

size

函数 `size` 的类型为 `size_t size()`，返回当前多边形的点数。

random

函数 `random` 的类型为 `void random (anytyp L=0, anytyp R=0)`。它会将这个多边形设置为一个随机多边形，每个点的横坐标和纵坐标分别互不相同，且均在 $[L,R]$ 内。

生成的多边形不会出现重复的点，不会自交，可能出现三线共点，点按**顺时针**排序。本函数允许的时间复杂度为 $n\log n$。

convex

函数 `convex` 的类型为 `void convex (anytyp L=0, anytyp R=0)`。它会将这个多边形设置为一个随机凸包，凸包的每个点横坐标和纵坐标分别互不相同，且均在 $[L,R]$ 内。

生成的凸包顶点中不会出现重复的点，可能出现三点共线，点按**顺时针**排序。本函数运行的时间复杂度为 $n\log n$。

内置常量

内置常量封装在一个结构体 `Consts_t` 中。我们已经为您定义好了一个 `Consts_t` 类型的结构体常量 `cst`。您可以借此直接使用提供的常量。例如，您可以使用 `cst._1e9` 来获取常量 10^9，避免手动输入 9 个零可能会带来的输错等问题。

所有的常量如下：

常量名	类型	含义
`Number`	string	所有的数字，即 `0123456789`
`SmallLetter`	string	所有的小写字母，按字母表顺序排列
`CapitalLetter`	string	所有的大写字母，按字母表顺序排列
`Letter`	string	所有的英文字母，格式形如 `AaBbCcDdEe...`
`Character`	string	ASCII 为 $[33, 126]$ 的字符，即所有可见字符
`CharaSpace`	string	ASCII 为 $[32, 126]$ 的字符，即所有可见字符和空格
`Pi`	double	圆周率
`E`	double	自然底数
`_1e4`	int	10^4
`_1e5`	int	10^5
`_1e6`	int	10^6
`_1e7`	int	10^7
`_1e8`	int	10^8
`_1e9`	int	10^9
`_1e10`	long long	10^{10}
`_1e11`	long long	10^{11}
`_1e12`	long long	10^{12}
`_1e13`	long long	10^{13}
`_1e14`	long long	10^{14}
`_1e15`	long long	10^{15}
`_1e16`	long long	10^{16}
`_1e17`	long long	10^{17}
`_1e18`	long long	10^{18}
`HackUnMap`	int	107897。插入其 n 个倍数到 `unordered_map` 中可将复杂度卡至 n^2