Prüfer code. Cayley formula. The number of ways to make a graph connected

In this article we consider the so-called **Prüfer code** , which is a way to uniquely encode labeled tree by a sequence of numbers.

With the help of codes Priifer demonstrates proof **of Cayley's formula** (specifying the number of spanning trees in a complete graph), and the solution of the problem of the number of ways to add to the given graph edges to turn it into a coherent.

Note . We will not consider trees consisting of a single vertex - is a special case in which many statements degenerate.

Prüfer code

Prüfer code - a way bijective coding labeled trees with $n$vertices by a sequence $n-2$of integers in the interval $[1;n]$. In other words, the code Priifer - is **a bijection** between all spanning trees of a complete graph and numerical sequences.

Although use Prüfer code for storing and manipulating trees impractical due to the specificity of representation codes Priifer find application in solving combinatorial problems.

Author - Heinz Prüfer (Heinz Prüfer) - suggested this code in 1918 as proof of Cayley's formula (see below).

Building code Priifer for that tree

Prüfer code is constructed as follows. Will $n-2$fold prodelyvat procedure: choose a tree leaf with the lowest number, remove it from the tree, and add the code Priifer node number that was associated with this list. Eventually there will be only a tree $2$tops, and this algorithm is completed (the number of vertices is explicitly recorded in the code).

Thus, for a given code Priifer tree - a sequence of $n-2$numbers where each number - the number of the vertex associated with the smallest at the time sheet - ieis a number in the segment $[1;n]$.

Algorithm for computing Prüfer code is easy to implement with the asymptotic behavior $O(n \log n)$, simply maintaining a data structure for extracting a minimum (for example, $\mathrm{set} <>$or $\mathrm{priority_queue} <>$in the language of C + +), containing a list of all current leaves:

```
const int MAXN = ...;
int n;
vector<int> g[MAXN];
int degree[MAXN];
bool killed[MAXN];

vector<int> prufer_code() {
     set<int> leaves;
     for (int i=0; i<n; ++i) {
          degree[i] = (int) g[i].size();
          if (degree[i] == 1)
               leaves.insert (i);
          killed[i] = false;
     }

     vector<int> result (n-2);
     for (int iter=0; iter<n-2; ++iter) {
          int leaf = *leaves.begin();
          leaves.erase (leaves.begin());
          killed[leaf] = true;

          int v;
          for (size_t i=0; i<g[leaf].size(); ++i)
               if (!killed[g[leaf][i]])
                    v = g[leaf][i];

          result[iter] = v;
          if (--degree[v] == 1)
               leaves.insert (v);
     }
     return result;
}
```

However, the construction of Prüfer code can be implemented in linear time and that is described in the next section.

Building code Priifer for that tree in linear time

We present here a simple algorithm that asymptotics $O(n)$.

The essence of the algorithm is to keep **moving the pointer** ptr , which will always be to move only in the direction of increasing numbers of vertices.

At first glance, this is not possible, because in the process of building code Priifer rooms leaves can both increase and **decrease** . But it is easy to notice that the reduction occurs only in one case: if you remove the current code sheet his ancestor has a smaller number (this will be the ancestor of the minimum sheet and removed from the tree on the very next step Prüfer code). Thus, reduction of cases can be treated in time $O(1)$, and does not interfere with the construction of the algorithm **linear asymptotic behavior** :

```
const int MAXN = ...;
int n;
vector<int> g[MAXN];
int parent[MAXN], degree[MAXN];

void dfs (int v) {
	for (size_t i=0; i<g[v].size(); ++i) {
		int to = g[v][i];
		if (to != parent[v]) {
			parent[to] = v;
			dfs (to);
		}
	}
}

vector<int> prufer_code() {
	parent[n-1] = -1;
	dfs (n-1);

	int ptr = -1;
	for (int i=0; i<n; ++i) {
		degree[i] = (int) g[i].size();
		if (degree[i] == 1 && ptr == -1)
			ptr = i;
	}

	vector<int> result;
	int leaf = ptr;
	for (int iter=0; iter<n-2; ++iter) {
		int next = parent[leaf];
		result.push_back (next);
		--degree[next];
		if (degree[next] == 1 && next < ptr)
```

```
                leaf = next;
            else {
                ++ptr;
                while (ptr<n && degree[ptr] != 1)
                    ++ptr;
                leaf = ptr;
            }
        }
        return result;
}
```

Comment on this code. The main function here - $\text{prufer_code}()$which returns Priifer wood specified in global variables n(number of vertices) and g(adjacency lists specifying the graph). Initially, we find for each vertex its ancestor $\text{parent}[i]$- ie that ancestor which this vertex will have the time of disposal of wood (all we can find in advance, using the fact that the maximum peak $n-1$never removed from the tree). Also, we find for each vertex its degree $\text{degree}[i]$. Variable ptr- is moving the pointer ("candidate" for a minimum sheet), which varies only always upward. Variable leaf- is a current sheet with a minimum number. Thus, each iteration Prüfer code is to add leafin response, as well as making sure there was not $\text{parent}[\text{leaf}]$less than the current candidate ptr: if it turned out less, we simply assign $\text{leaf} = \text{parent}[\text{leaf}]$, and otherwise - move the pointer $\text{ptr}$to the next sheet.

As can be seen easily by the code, the asymptotic behavior of the algorithm is really $O(n)$a pointer $\text{ptr}$to undergo a $O(n)$change, and all other parts of the algorithm obviously operate in linear time.

Some properties of codes Priifer

- Upon completion of construction of Prüfer code in the tree remain unremoved two vertices.
- One of them will surely be the vertex with the maximum number - $n-1$and that's about another vertex nothing definite can be said.
- Each vertex occurs in code Priifer certain number of times equal to its power minus one.
- This is easily understood if we note that the vertex is removed from the tree at a time when its degree is equal to one - ie at this point all edges adjacent to it, but one, has been removed. (For the two remaining vertices after building code this statement is also true.)

Recovering tree by its code Priifer

To restore the tree enough to see from the previous paragraph that the degrees of all vertices in the target tree, we already know (and can calculate and save from an array $degree[]$). Consequently, we can find all the leaves, and, accordingly, the smallest number plate - which was removed in the first step. This sheet was connected to the top, the number of which is recorded in the first cell Prüfer code.

Thus, we find the first edge, remote Priifer code. Add this edge back, then reduce power $degree[]$at both ends of the ribs.

We will repeat this operation until you have reviewed all the code Priifer: search with minimal vertex $degree = 1$, connect it with another top Prüfer code, decrease $degree[]$at both ends.

In the end we are left with only two vertices $degree = 1$- those peaks that algorithm Priifer left unremoved. Connect them an edge.

The algorithm is complete, the desired tree is built.

Implement this algorithm is easily during $O(n \log n)$: supporting data structure used to extract a minimum (for example, $\mathrm{set} <>$or $\mathrm{priority_queue} <>$in C + +) numbers of all vertices with $degree = 1$, and removing from it every time at least.

Here are the appropriate implementation (where the function $prufer_decode()$returns a list of the edges of the required tree):

```
vector < pair<int,int> > prufer_decode (const vector<int> &
prufer_code) {
     int n = (int) prufer_code.size() + 2;
     vector<int> degree (n, 1);
     for (int i=0; i<n-2; ++i)
          ++degree[prufer_code[i]];

     set<int> leaves;
     for (int i=0; i<n; ++i)
          if (degree[i] == 1)
               leaves.insert (i);

     vector < pair<int,int> > result;
     for (int i=0; i<n-2; ++i) {
          int leaf = *leaves.begin();
```

```
            leaves.erase (leaves.begin());

            int v = prufer_code[i];
            result.push_back (make_pair (leaf, v));
            if (--degree[v] == 1)
                 leaves.insert (v);
      }
      result.push_back (make_pair (*leaves.begin(), *--
leaves.end()));
      return result;
}
```

Recovering tree code Priifer in linear time

For the algorithm with linear asymptotics can apply the same method that was used to obtain a linear algorithm for computing Prüfer code.

In fact, in order to find the lowest-numbered sheet optionally start the data structure to retrieve the minimum. Instead, you'll notice that after we find and treat the current sheet, it adds to the consideration of only one new vertex. Consequently, we can get one with a moving pointer variable, over a minimum current list:

```
vector < pair<int,int> > prufer_decode_linear (const vector<int>
& prufer_code) {
      int n = (int) prufer_code.size() + 2;
      vector<int> degree (n, 1);
      for (int i=0; i<n-2; ++i)
            ++degree[prufer_code[i]];

      int ptr = 0;
      while (ptr < n && degree[ptr] != 1)
            ++ptr;
      int leaf = ptr;

      vector < pair<int,int> > result;
      for (int i=0; i<n-2; ++i) {
            int v = prufer_code[i];
            result.push_back (make_pair (leaf, v));

            --degree[leaf];
            if (--degree[v] == 1 && v < ptr)
                 leaf = v;
            else {
                ++ptr;
                while (ptr < n && degree[ptr] != 1)
                     ++ptr;
                leaf = ptr;
           }
      }
      for (int v=0; v<n-1; ++v)
           if (degree[v] == 1)
                result.push_back (make_pair (v, n-1));
      return result;
}
```

One correspondence between trees and Prüfer codes

On the one hand, for every tree there is exactly one Prüfer code, the corresponding (this follows from the definition of Prüfer code).

On the other hand, the correctness of the tree reconstruction algorithm code Priifer that any code Prufer (ie sequence of $n-2$numbers where each number lies in the interval $[1;n]$) corresponds to a tree.

Thus, all the trees and all the codes Priifer form **one correspondence** .

Cayley formula

Cayley formula states that **the number of spanning trees in full labeled graph** of n vertices is equal to:

n^{n-2}.

There are many **proofs** of this formula, but the proof using codes Priifer clearly and constructively.

In fact, any set of $n-2$numbers from the interval $[1;n]$corresponds uniquely to a tree of $n$nodes. Unique codes Prufer n^{n-2}. As in the case of a complete graph of $n$vertices as the core fits any tree, then the number of spanning trees of power n^{n-2}, as required.

The number of ways to make a graph connected

Power Prufer codes is that they allow you to get a more general formula than formula Cayley.

Thus, a graph of $n$vertices and $m$edges, let k- the number of connected components in this graph. Required to find the number of ways to add $k-1$an edge to the graph became connected (obviously $k-1$rib - the minimum number of edges to make the graph connected).

Derive ready formula for solving this problem.

We denote $s_1, \ldots, s_k$the sizes of connected components of the graph. Since adding edges connected components inside smoking, it turns out that the problem is very similar to the number of search spanning trees in a complete graph of $k$vertices: but the difference here is that each node has its own "weight" s_i: each edge is adjacent to the i second vertex, multiplies the answer to s_i.

Thus, for counting the number of ways is important what degree are all $k$vertices in the backbone. To obtain equations for the problem must be summed over all possible answers powers.

Let $d_1, \ldots, d_k$- degrees of vertices in the backbone. Sum of the vertex degrees is equal to twice the number of edges, so:

$$\sum_{i=1}^{k} d_i = 2k-2.$$

If iI-vertex has degree d_i, then it enters the code Priifer $d_i - 1$times. Prufer Code for the tree of $k$vertices has a length $k-2$. Number of ways to choose a set of $k-2$ numbers, where the number $i$appears exactly $d_i - 1$once, power **multinomial coefficient** (by analogy with the binomial coefficient)

$$\binom{k-2}{d_1-1,\ d_2-1,\ \ldots\ ,d_k-1} = \frac{(k-2)!}{(d_1-1)!\,(d_2-1)!\ \ldots\ (d_k-1)!}.$$

Given the fact that each edge adjacent to the $i$second vertex multiplies response to s_i obtain the response, with the proviso that power peaks are $d_1, \ldots, d_k$equal to:

$$s_1^{d_1} \cdot s_2^{d_2} \cdot \ldots \cdot s_k^{d_k} \cdot \frac{(k-2)!}{(d_1-1)!\,(d_2-1)!\ \ldots\ (d_k-1)!}.$$

To answer this problem we must sum over all possible valid formula sets $\{d_i\}_{i=1}^{i=k}$:

$$\sum_{\substack{d_i \geq 1, \\ \sum_{i=1}^{k} d_i = 2k-2}} s_1^{d_1} \cdot s_2^{d_2} \cdot \ldots \cdot s_k^{d_k} \cdot \frac{(k-2)!}{(d_1-1)!\,(d_2-1)!\ \ldots\ (d_k-1)!}.$$

To minimize this formula we use the definition of multinomial coefficient:

$$(x_1 + \ldots x_m)^p = \sum_{\substack{c_i \geq 0, \\ \sum_{i=1}^m c_i = p}} x_1^{c_1} \cdot x_2^{c_2} \cdot \ldots \cdot x_m^{c_m} \cdot \binom{m}{c_1,\ c_2,\ \ldots\ ,c_k}.$$

Comparing this with the previous formula, we find that if we introduce the notation $e_i = d_i - 1$:

$$\sum_{\substack{e_i \geq 0, \\ \sum_{i=1}^k e_i = k-2}} s_1^{e_1+1} \cdot s_2^{e_2+1} \cdot \ldots \cdot s_k^{e_k+1} \cdot \frac{(k-2)!}{e_1!\ e_2!\ \ldots\ e_k!},$$

after folding **answer to the problem** is:

$$s_1 \cdot s_2 \cdot \ldots \cdot s_k \cdot (s_1 + s_2 + \ldots + s_k)^{k-2} = s_1 \cdot s_2 \cdot \ldots \cdot s_k \cdot n^{k-2}.$$

(This formula is true and $k = 1$, although formally the proof it should not.)

Problem in online judges

Problem in online judges, which use codes Priifer:

- UVA # 10843 **"Anne's game"** [Difficulty: Easy]
- TIMUS # 1069 **"Code Priifer"** [Difficulty: Easy]
- Codeforces 110D **"Clues"** [Difficulty: Medium]
- TopCoder SRM 460 **"TheCitiesAndRoadsDivTwo"** [Difficulty: Medium]

In this article we consider the so-called **Prüfer code** , which is a way to uniquely encode labeled tree by a sequence of numbers.

With the help of codes Priifer demonstrates proof **of Cayley's formula** (specifying the number of spanning trees in a complete graph), and the solution of the problem of the number of ways to add to the given graph edges to turn it into a coherent.

Note . We will not consider trees consisting of a single vertex - is a special case in which many statements degenerate.

Prüfer code

Prüfer code - a way bijective coding labeled trees with $n$vertices by a sequence $n-2$of integers in the interval $[1; n]$. In other words, the code Priifer - is **a bijection** between all spanning trees of a complete graph and numerical sequences.

Although use Prüfer code for storing and manipulating trees impractical due to the specificity of representation codes Priifer find application in solving combinatorial problems.

Author - Heinz Prüfer (Heinz Prüfer) - suggested this code in 1918 as proof of Cayley's formula (see below).

Building code Priifer for that tree

Prüfer code is constructed as follows. Will $n-2$fold prodelyvat procedure: choose a tree leaf with the lowest number, remove it from the tree, and add the code Priifer node number that was associated with this list. Eventually there will be only a tree $2$tops, and this algorithm is completed (the number of vertices is explicitly recorded in the code).

Thus, for a given code Priifer tree - a sequence of $n-2$numbers where each number - the number of the vertex associated with the smallest at the time sheet - ieis a number in the segment $[1;n]$.

Algorithm for computing Prüfer code is easy to implement with the asymptotic behavior $O(n\log n)$, simply maintaining a data structure for extracting a minimum (for example, $\mathrm{set} <>$or $\mathrm{priority_queue} <>$in the language of C + +), containing a list of all current leaves:

```
const int MAXN = ...;
int n;
vector<int> g[MAXN];
int degree[MAXN];
bool killed[MAXN];

vector<int> prufer_code() {
     set<int> leaves;
     for (int i=0; i<n; ++i) {
          degree[i] = (int) g[i].size();
          if (degree[i] == 1)
               leaves.insert (i);
          killed[i] = false;
     }

     vector<int> result (n-2);
     for (int iter=0; iter<n-2; ++iter) {
          int leaf = *leaves.begin();
          leaves.erase (leaves.begin());
          killed[leaf] = true;

          int v;
          for (size_t i=0; i<g[leaf].size(); ++i)
```

```
                    if (!killed[g[leaf][i]])
                        v = g[leaf][i];

                result[iter] = v;
                if (--degree[v] == 1)
                    leaves.insert (v);
        }
        return result;
}
```

However, the construction of Prüfer code can be implemented in linear time and that is described in the next section.

Building code Priifer for that tree in linear time

We present here a simple algorithm that asymptotics $O(n)$.

The essence of the algorithm is to keep **moving the pointer** ptr , which will always be to move only in the direction of increasing numbers of vertices.

At first glance, this is not possible, because in the process of building code Priifer rooms leaves can both increase and **decrease** . But it is easy to notice that the reduction occurs only in one case: if you remove the current code sheet his ancestor has a smaller number (this will be the ancestor of the minimum sheet and removed from the tree on the very next step Prüfer code). Thus, reduction of cases can be treated in time $O(1)$, and does not interfere with the construction of the algorithm **linear asymptotic behavior** :

```
const int MAXN = ...;
int n;
vector<int> g[MAXN];
int parent[MAXN], degree[MAXN];

void dfs (int v) {
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i];
        if (to != parent[v]) {
            parent[to] = v;
            dfs (to);
        }
    }
}

```

```
vector<int> prufer_code() {
     parent[n-1] = -1;
     dfs (n-1);

     int ptr = -1;
     for (int i=0; i<n; ++i) {
          degree[i] = (int) g[i].size();
          if (degree[i] == 1 && ptr == -1)
               ptr = i;
     }

     vector<int> result;
     int leaf = ptr;
     for (int iter=0; iter<n-2; ++iter) {
          int next = parent[leaf];
          result.push_back (next);
          --degree[next];
          if (degree[next] == 1 && next < ptr)
               leaf = next;
          else {
               ++ptr;
               while (ptr<n && degree[ptr] != 1)
                    ++ptr;
               leaf = ptr;
          }
     }
     return result;
}
```

Comment on this code. The main function here - $\mathrm{prufer_code}()$which returns Priifer wood specified in global variables n(number of vertices) and g(adjacency lists specifying the graph). Initially, we find for each vertex its ancestor $\mathrm{parent}[i]$- ie that ancestor which this vertex will have the time of disposal of wood (all we can find in advance, using the fact that the maximum peak $n-1$never removed from the tree). Also, we find for each vertex its degree $\mathrm{degree}[i]$. Variable ptr- is moving the pointer ("candidate" for a minimum sheet), which varies only always upward. Variable leaf- is a current sheet with a minimum number. Thus, each iteration Prüfer code is to add leafin response, as well as making sure there was not $\mathrm{parent}[\mathrm{leaf}]$less than the current candidate ptr: if it turned out less, we simply assign $\mathrm{leaf} = \mathrm{parent}[\mathrm{leaf}]$, and otherwise - move the pointer $\mathrm{ptr}$to the next sheet.

As can be seen easily by the code, the asymptotic behavior of the algorithm is really $O(n)$a pointer $\mathrm{ptr}$to undergo a $O(n)$change, and all other parts of the algorithm obviously operate in linear time.

Some properties of codes Priifer

- Upon completion of construction of Prüfer code in the tree remain unremoved two vertices.
- One of them will surely be the vertex with the maximum number - $n-1$and that's about another vertex nothing definite can be said.
- Each vertex occurs in code Priifer certain number of times equal to its power minus one.
- This is easily understood if we note that the vertex is removed from the tree at a time when its degree is equal to one - ie at this point all edges adjacent to it, but one, has been removed. (For the two remaining vertices after building code this statement is also true.)

Recovering tree by its code Priifer

To restore the tree enough to see from the previous paragraph that the degrees of all vertices in the target tree, we already know (and can calculate and save from an array $degree[]$). Consequently, we can find all the leaves, and, accordingly, the smallest number plate - which was removed in the first step. This sheet was connected to the top, the number of which is recorded in the first cell Prüfer code.

Thus, we find the first edge, remote Priifer code. Add this edge back, then reduce power $degree[]$at both ends of the ribs.

We will repeat this operation until you have reviewed all the code Priifer: search with minimal vertex $degree = 1$, connect it with another top Prüfer code, decrease $degree[]$at both ends.

In the end we are left with only two vertices $degree = 1$- those peaks that algorithm Priifer left unremoved. Connect them an edge.

The algorithm is complete, the desired tree is built.

Implement this algorithm is easily during $O(n \log n)$: supporting data structure used to extract a minimum (for example, $\mathrm{set} <>$or $\mathrm{priority_queue} <>$in C + +) numbers of all vertices with $degree = 1$, and removing from it every time at least.

Here are the appropriate implementation (where the function $prufer_decode()$returns a list of the edges of the required tree):

```
vector < pair<int,int> > prufer_decode (const vector<int> &
prufer_code) {
     int n = (int) prufer_code.size() + 2;
     vector<int> degree (n, 1);
     for (int i=0; i<n-2; ++i)
          ++degree[prufer_code[i]];

     set<int> leaves;
     for (int i=0; i<n; ++i)
          if (degree[i] == 1)
               leaves.insert (i);

     vector < pair<int,int> > result;
     for (int i=0; i<n-2; ++i) {
          int leaf = *leaves.begin();
          leaves.erase (leaves.begin());

          int v = prufer_code[i];
          result.push_back (make_pair (leaf, v));
          if (--degree[v] == 1)
               leaves.insert (v);
     }
     result.push_back (make_pair (*leaves.begin(), *--
leaves.end()));
     return result;
}
```

Recovering tree code Priifer in linear time

For the algorithm with linear asymptotics can apply the same method that was used to obtain a linear algorithm for computing Prüfer code.

In fact, in order to find the lowest-numbered sheet optionally start the data structure to retrieve the minimum. Instead, you'll notice that after we find and treat the current sheet, it adds to the consideration of only one new vertex. Consequently, we can get one with a moving pointer variable, over a minimum current list:

```
vector < pair<int,int> > prufer_decode_linear (const vector<int>
& prufer_code) {
     int n = (int) prufer_code.size() + 2;
```

```
	vector<int> degree (n, 1);
	for (int i=0; i<n-2; ++i)
		++degree[prufer_code[i]];

	int ptr = 0;
	while (ptr < n && degree[ptr] != 1)
		++ptr;
	int leaf = ptr;

	vector < pair<int,int> > result;
	for (int i=0; i<n-2; ++i) {
		int v = prufer_code[i];
		result.push_back (make_pair (leaf, v));

		--degree[leaf];
		if (--degree[v] == 1 && v < ptr)
			leaf = v;
		else {
			++ptr;
			while (ptr < n && degree[ptr] != 1)
				++ptr;
			leaf = ptr;
		}
	}
	for (int v=0; v<n-1; ++v)
		if (degree[v] == 1)
			result.push_back (make_pair (v, n-1));
	return result;
}
```

One correspondence between trees and Prüfer codes

On the one hand, for every tree there is exactly one Prüfer code, the corresponding (this follows from the definition of Prüfer code).

On the other hand, the correctness of the tree reconstruction algorithm code Priifer that any code Prufer (ie sequence of $n-2$numbers where each number lies in the interval $[1;n]$) corresponds to a tree.

Thus, all the trees and all the codes Priifer form **one correspondence** .

Cayley formula

Cayley formula states that **the number of spanning trees in full labeled graph** of n vertices is equal to:

n^{n-2}.

There are many **proofs** of this formula, but the proof using codes Priifer clearly and constructively.

In fact, any set of $n-2$numbers from the interval $[1;n]$corresponds uniquely to a tree of $n$nodes. Unique codes Prufer n^{n-2}. As in the case of a complete graph of $n$vertices as the core fits any tree, then the number of spanning trees of power n^{n-2}, as required.

The number of ways to make a graph connected

Power Prufer codes is that they allow you to get a more general formula than formula Cayley.

Thus, a graph of $n$vertices and $m$edges, let k- the number of connected components in this graph. Required to find the number of ways to add $k-1$an edge to the graph became connected (obviously $k-1$rib - the minimum number of edges to make the graph connected).

Derive ready formula for solving this problem.

We denote $s_1, \ldots, s_k$the sizes of connected components of the graph. Since adding edges connected components inside smoking, it turns out that the problem is very similar to the number of search spanning trees in a complete graph of $k$vertices: but the difference here is that each node has its own "weight" s_i: each edge is adjacent to the i second vertex, multiplies the answer to s_i.

Thus, for counting the number of ways is important what degree are all $k$vertices in the backbone. To obtain equations for the problem must be summed over all possible answers powers.

Let $d_1, \ldots, d_k$- degrees of vertices in the backbone. Sum of the vertex degrees is equal to twice the number of edges, so:

$$\sum_{i=1}^{k} d_i = 2k - 2.$$

If iI-vertex has degree d_i, then it enters the code Priifer $d_i - 1$times. Prufer Code for the tree of $k$vertices has a length $k-2$. Number of ways to choose a set of $k-2$ numbers, where the number $i$appears exactly $d_i - 1$once, power **multinomial coefficient** (by analogy with the binomial coefficient)

$$\binom{k-2}{d_1-1,\ d_2-1,\ \ldots\ ,d_k-1} = \frac{(k-2)!}{(d_1-1)!\,(d_2-1)!\ \ldots\ (d_k-1)!}.$$

Given the fact that each edge adjacent to the $i$second vertex multiplies response to s_i obtain the response, with the proviso that power peaks are $d_1, \ldots, d_k$equal to:

$$s_1^{d_1} \cdot s_2^{d_2} \cdot \ldots \cdot s_k^{d_k} \cdot \frac{(k-2)!}{(d_1-1)!\,(d_2-1)!\ \ldots\ (d_k-1)!}.$$

To answer this problem we must sum over all possible valid formula sets $\{d_i\}_{i=1}^{i=k}$:

$$\sum_{\substack{d_i \geq 1, \\ \sum_{i=1}^k d_i = 2k-2}} s_1^{d_1} \cdot s_2^{d_2} \cdot \ldots \cdot s_k^{d_k} \cdot \frac{(k-2)!}{(d_1-1)!\,(d_2-1)!\ \ldots\ (d_k-1)!}.$$

To minimize this formula we use the definition of multinomial coefficient:

$$(x_1 + \ldots x_m)^p = \sum_{\substack{c_i \geq 0, \\ \sum_{i=1}^m c_i = p}} x_1^{c_1} \cdot x_2^{c_2} \cdot \ldots \cdot x_m^{c_m} \cdot \binom{m}{c_1,\ c_2,\ \ldots\ ,c_k}.$$

Comparing this with the previous formula, we find that if we introduce the notation $e_i = d_i - 1$:

$$\sum_{\substack{e_i \geq 0, \\ \sum_{i=1}^k e_i = k-2}} s_1^{e_1+1} \cdot s_2^{e_2+1} \cdot \ldots \cdot s_k^{e_k+1} \cdot \frac{(k-2)!}{e_1!\ e_2!\ \ldots\ e_k!},$$

after folding **answer to the problem** is:

$$s_1 \cdot s_2 \cdot \ldots \cdot s_k \cdot (s_1 + s_2 + \ldots + s_k)^{k-2} = s_1 \cdot s_2 \cdot \ldots \cdot s_k \cdot n^{k-2}.$$

(This formula is true and $k = 1$, although formally the proof it should not.)

Problem in online judges

Problem in online judges, which use codes Priifer:

- UVA # 10843 **"Anne's game"** [Difficulty: Easy]
- TIMUS # 1069 **"Code Priifer"** [Difficulty: Easy]
- Codeforces 110D **"Clues"** [Difficulty: Medium]
- TopCoder SRM 460 **"TheCitiesAndRoadsDivTwo"** [Difficulty: Medium]