1. Versión 1

La primera versión consiste en realizar una búsqueda en anchura estándar sobre el árbol de soluciones. Es decir, se recorre cada posible solución, pero recorriendo todas las posibilidades de longitud n antes de pasar a las soluciones de longitud $n+1$. Así nos aseguramos de que pasamos por la solución óptima antes que cualquier otra.

El código del que partimos es el siguiente:

```
#include <iostream>
#include <climits>
#include <unordered_set>
#include <unordered_map>
#include <vector>
#include <list>
#include <queue>

using namespace std;
using ll = long long int;
using mli=unordered_map<ll,list<ll>>;
using mlc=unordered_map<ll,list<char>>;

// Conjunto de primos utilizables .
const unordered_set<ll> s =
    {1,2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97};

// Para poder switchear e iterar sobre las operaciones
// las convierto tanto en enumerado como en vector.
// Cuidado, redundancia .
enum ops {DIV,MUL,SUM,RES};
const vector<ops> opsvec = {DIV,MUL,SUM,RES};

// Del enum a caracter .
char tochar(ops o) {
  switch(o) {
    case DIV : return '/';
    case MUL : return '*';
    case SUM : return '+';
    case RES : return '-';
    default  : return '0';
  }
}

// Realizar la operacion entre u y v dada por o.
ll op(int u, int v, ops o) {
  switch(o) {
    case DIV : return u/v;
    case MUL : return u*v;
    case SUM : return u+v;
    case RES : return u-v;
```

```
    default    : return 0;
  }
}

// Busqueda en anchura.
// n es el destino, ss son los primos utilizables menos el restringido,
// dist y ch guardan, para cada entero por el que pasamos, el estado
// en el que se encuentran la cadena de numeros y operaciones, respectivamente.
void bfs(ll n, mli &dist,mlc &ch,unordered_set<ll> &ss) {
  queue<ll> q; // La cola para guardar los nodos de cada nivel.
  for (auto u : ss) {// Todos los primos permitidos tienen como cadena ellos mismos.
    dist[u].push_front(u);
    q.push(u);
  }
  while (!q.empty()) {
    ll u = q.front(); q.pop();
    for (auto v : ss) { // Itero sobre los primos
      for (auto o : opsvec) { // y sobre cada operacion.
        if(o==DIV && (u%v!=0)) continue; // Para dividir debe dar resultado entero.
        ll w = op(u,v,o);
        if (!dist[w].empty()) { // Si aun no se ha pasado por w
          dist[w] = dist[u]; // La nueva cadena es la anterior
          dist[w].push_back(v); // mas el nuevo primo.
          ch[w] = ch[u]; // La cadena de operaciones es la anterior
          if ((o==DIV || o==MUL) && !ch[u].empty()) {
            ch[w].push_front('('); // mas parentesis si son * o /
            ch[w].push_back(')');
          }
          ch[w].push_back(tochar(o)); // mas la nueva operacion.
          if(w==n) return; // Se comprueba si hemos llegado al resultado.
          q.push(w); // Se guarda el nodo en la cola.
        }
      }
    }
  }
}

bool caso() {
  ll n,p; // n es el numero destino, p el primo restringido.
  cin >> n >> p;

  mli dist;
  mlc ch;

  unordered_set<ll> ss = s;
  ss.erase(p); // Se elimina de s el primo restringido.

  bfs(n,dist,ch,ss);
```

```
  // Salida por pantalla.
  while (!ch[n].empty()) {
    if(ch[n].front() == '(') {
      cout << ch[n].front(); ch[n].pop_front();
      continue; // Si el char es ( solo se saca eso.
    }
    // Se saca el numero.
    cout << dist[n].front(); dist[n].pop_front();

    char next = ch[n].front();
    cout << next; ch[n].pop_front();
    if(next == ')' && !ch[n].empty()) { // Si el char anterior era un parentesis
      cout << ch[n].front(); ch[n].pop_front(); // quizas haya otra operacion detras.
    }
  }

  if (!dist[n].empty()) // Quizas quede un numero detras.
    cout << dist[n].front();

  cout << '\n';

  return true;
}

int main() {
  while(caso());
}
```

1.1. Restricciones

1. Está en C++.
2. A partir de 10 000 tarda demasiado.
3. No se tienen en cuenta el orden de los operandos.
4. Tiene más paréntesis de la cuenta.

1.2. A hacer

1. Ver si ayuda optimizar factorizando.
2. Ver cuál sería la estructura óptima (x*y+-z).
3. Ver si la división toma partido o no.