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;
using vi=vector<ll>;
using vvi=vector<vi>;
using mli=unordered_map<ll,list<ll>>;
using mlc=unordered_map<ll,list<char>>;

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};

void bfs(ll n, mli &dist,mlc &ch,unordered_set<ll> &ss) {
  queue<ll> q;
  for (auto u : ss) {
    dist[u].push_front(u);
    q.push(u);
  }
  while (!q.empty()) {
    ll u = q.front(); q.pop();
    if(u==n) return;
    for (auto v : ss) {
      if ((u%v==0) && dist.find(u/v) == dist.end()) {
        dist[u/v] = dist[u];
        dist[u/v].push_back(v);
        ch[u/v] = ch[u];
        if (!ch[u].empty()) {
          ch[u/v].push_front('(');
          ch[u/v].push_back(')');
        }
        ch[u/v].push_back('/');
        q.push(u/v);
      }

      if (dist.find(u*v) == dist.end()) {
```

```
        dist[u*v] = dist[u];
        dist[u*v].push_back(v);
        ch[u*v] = ch[u];
        if (!ch[u].empty()) {
          ch[u*v].push_front('(');
          ch[u*v].push_back(')');
        }
        ch[u*v].push_back('*');
        q.push(u*v);
      }
      if (dist.find(u+v) == dist.end()) {
        dist[u+v] = dist[u];
        dist[u+v].push_back(v);
        ch[u+v] = ch[u];
        ch[u+v].push_back('+');
        q.push(u+v);
      }
      if (dist.find(u-v) == dist.end()) {
        dist[u-v] = dist[u];
        dist[u-v].push_back(v);
        ch[u-v] = ch[u];
        ch[u-v].push_back('-');
        q.push(u-v);
      }
    }
  }
}

bool cases() {
  ll n,p;
  cin >> n >> p;

  mli dist;
  mlc ch;

  unordered_set<ll> ss = s;
  ss.erase(p);

  bfs(n,dist,ch,ss);

  while (!ch[n].empty()) {
    if(ch[n].front() == '(') {
      cout << ch[n].front(); ch[n].pop_front();
      continue;
    }
    cout << dist[n].front(); dist[n].pop_front();
    if(ch[n].front() == ')') {
      cout << ch[n].front(); ch[n].pop_front();
      if (!ch[n].empty()) {
```

```
        cout << ch[n].front(); ch[n].pop_front();
      }
    }
    else {
      cout << ch[n].front(); ch[n].pop_front();
    }
  }

  if (!dist[n].empty())
    cout << dist[n].front();

  cout << '\n';

  return true;
}

int main() {
  while(cases());

  return 0;
}
```

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.