ABC 137 解説

DEGwer, drafear, evima, gazelle, IH19980412, potetisensei, yokozuna57

2019 年 8 月 10 日

For International Readers: English editorial starts on page 9.

A: +-x

指示通り $A+B$, $A-B$, $A\times B$ の中で最大の数を出力すればいいです。C++ による実装例を以下に示します。

Listing 1　C++ による実装例

```cpp
#include<bits/stdc++.h>
using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    cout << max({a + b, a - b, a * b}) << endl;
    return 0;
}
```

B: One Clue

(原案: DEGwer, 準備・解説: evima)

黒い石のうち最も左にある (座標が最も小さい) ものの座標を L とすると、K 個の黒い石の座標は $L, L+1, \ldots, L+K-1$ と書けます。

座標 X の石が黒であることから、L は $X-K+1, X-K+2, \ldots, X$ のいずれかです。よって、黒い石が置かれている可能性のある座標は $X-K+1, X-K+2, \ldots, X+K-1$ の $2K-1$ 個です。

あとは純粋なプログラミングの課題で、Python での実装例を二つ掲載することをもって解説に代えさせていただきます。[*1] 他の言語でもほぼ同等な実装が可能なはずです。

```
K, X = map(int, input().split())
for i in range(X - K + 1, X + K):
    if i < X + K - 1:
        print(i, end=' ')
    else:
        print(i)
```

```
K, X = map(int, input().split())
print(' '.join(map(str, range(X - K + 1, X + K))))
```

[*1] 実は、一つ目の実装例で 3, 5, 6 行目を削除して、最後の座標の直後でも改行ではなく空白を出力しても正解と判定されます。AtCoder に限らず、競技プログラミングのジャッジは「空白系の文字」の扱いに関して寛容なことが多いです。ただし、ターミナルに出力が表示される際の見やすさを考えてもできるだけ出力の末尾では改行されることを勧めます

C: Green Bin

(原案: yokozuna57, 準備・解説: evima)

まず、二つの文字列 s,t が与えられた際に s が t のアナグラムであるかを判定する方法には次の二つがあります。

1. 26 種のアルファベットそれぞれが s と t にそれぞれ何回出現するか数え、どのアルファベットについても出現回数が同じか確認する。
2. 両方の文字列をソートし、一致するか確認する。

今回の問題では最大で 10 万個の文字列が与えられ、すべてのペア (最大で 50 億個程度) について上の方法をそのまま実行すると 2 秒という時間制限を過ぎてしまうでしょう。しかし、上の方法で文字列から得る「エッセンス」を元に文字列をグループに分けることで計算を効率化できます。入力例 3 を用いて、方法 2 を元に効率化した計算の例を以下に示します。

1. $s_1 =$ `abaaaaaaaa` をソートして `aaaaaaaaab` を得る。
2. $s_2 =$ `oneplustwo` をソートして `elnoopstuw` を得る。
3. $s_3 =$ `aaaaaaaaba` をソートして `aaaaaaaaab` を得る。この結果はすでに 1 回現れているので、答えに 1 を足す。
4. $s_4 =$ `twoplusone` をソートして `elnoopstuw` を得る。この結果はすでに 1 回現れているので、答えに 1 を足す。
5. $s_5 =$ `aaaaaaaaba` をソートして `aaaaaaaaab` を得る。この結果はすでに 2 回現れているので、答えに 2 を足す。

「この結果はすでに 1 回現れているので」などの部分は、ハッシュテーブル (C++ の undered_map, Java の HashMap, Python の dictionary など) を用いて実装すれば線形時間で動作します。他に、 N 個の文字列から得られる N 個の「エッセンス」をソートするという解法も考えられますが、こちらは言語によっては実行が間に合わないかもしれません。

D: Summer Vacation

夏休みにお金を稼いで沖縄旅行に行きたいという気持ちで解くと解きやすいかもしれません。M 日後までに報酬を得る必要があるので M 日後から i $(1 \leq i \leq M)$ 日前には $A_j \leq i$ となる日雇いアルバイトしか選ぶことができません。こういった問題は制約の厳しい方から見ていくと見通しが良くなることが多いです。実際、この問題は後ろ (M 日後から 1 日前) から見ていくと、まず $A_j \leq 1$ となる j の中で B_j が最大のもの (j_1 とする) を選び、次に $A_j \leq 2$ かつ $j \neq j_1$ となる j の中で B_j が最大のものを選び、…としていくのが最適です[*2]。

次に、実装を考えます。したい操作は

- 候補の追加
- 候補の中から最大のものを取り出す

であって、それぞれ $O(N), O(M)$ 回行います。優先度付きキュー (Priority Queue)[*3]を用いると、これらを高速に処理することができます。

これを実装すると、$O(M + NlogN)$ の時間計算量で解くことができます。

別解

少し高度な解き方です。一言で言うと、実行可能解の集合はマトロイドなので、貪欲法で解くことができます。また、最小費用流の動作もこの貪欲法と同じです。

まず、次の 3 つの公理を満たす有限[*4]集合族[*5] I のことをマトロイドといいます。

1. 空集合は I の要素である。
2. 集合 X が I の要素であるとき、X のどんな部分集合 Y も I の要素である。
3. 大きさの異なる集合 X, Y $(|X| < |Y|)$ が I の要素であるとき、集合 Y に属するが集合 X に属さない要素 v であって、集合 X に加えても I の要素となるような v が存在する。

今回の問題に当てはめると、次のようになります。I を実行可能解[*6]の集合と考えます。

1. 空集合 (どの日雇いアルバイトも選ばない) は I の要素 (すなわち、実行可能解) である。
2. 日雇いアルバイト $\{i_1, i_2, ..., i_k\}$ の報酬が間に合うようにうまく割り当てられるとき、ここからいくつかやらないことにした $\{i'_1, i'_2, ..., i'_l\} \subseteq \{i_1, i_2, ..., i_k\}$ についても報酬が間に合うようにうまく割り当てることができる。
3. $\{i_1, i_2, ..., i_k\}$ も $\{j_1, j_2, ..., j_l\}$ も報酬が間に合うようにうまく割り当てられ、$k < l$ であるとき、

[*2] これはこの問題を最小費用流に帰着させて示すことができます。しかし、最小費用流で解こうとすると $O(M(N+M)log(N+M))$ となり間に合いません。

[*3] 優先度付きキューの仕組みの説明はここでは省略しますが、優先度付きキューに追加された要素数を X とすると、$O(logX)$ で要素の追加、最大値の削除を、$O(1)$ で最大値の取得が行えます。C++ では標準 C++ ライブラリ (STL) で提供されています。

[*4] 要素数が有限の集合であるということ。

[*5] 各要素が集合であるような集合。集合の集合。

[*6] 条件を満たすような選び方。最適解とは限らない。今回の場合だと、いつやるかをうまく割り当てると選んだ全ての日雇いアルバイトができる (報酬が間に合う) ような、選ぶ日雇いアルバイトの番号の集合。

$\{i_1, i_2, ..., i_k, j_x\}$ がうまく割り当てられるような x が存在する。

証明は省きますが、この 3 つは満たされるので、今回の問題の実行可能解の集合もマトロイドです。さて、マトロイドであれば何が嬉しいのでしょうか。集合族 I がマトロイドであって、I の要素である集合の要素となりうるもの x に対して重み $w(x)$ が定まっているものとします。今回の場合ですと、各日雇いアルバイトに対して報酬といった重みが定まっています。I の要素 X の重みを、その集合の各要素の重みの和 $\sum_{x \in X} w(x)$ と定義します。どの y についても $w(y) \geq 0$ であるとき、I の要素の重みの最大値を貪欲法で求めることができます。

マトロイド I に対する貪欲法とは次のようなものです。

1. S を空集合とします (後に、S に要素を順番に追加していき、最適解を構成します)。マトロイドの公理 1 より、$S \in I$ です。
2. 重みの大きい順に解の要素として選べるもの (今回の場合だと日雇いアルバイト) をソートし、$x_1, x_2, ..., x_N$ とします。
3. $y = x_1, x_2, ..., x_N$ と順に以下を行う。
   (a) S に y を加えても I の要素である ($S \cup \{y\} \in I$ である) とき、S に y を加える。
4. S を出力する (S が最適解である)。

上のアルゴリズムの中でマトロイドによって異なる部分は「S に y を加えても I の要素である」ことを判定する部分です。「S に y を加えても I の要素である」を今回の問題について言い換えると、「日雇いアルバイト番号の集合 $S \cup \{y\} = \{i_1, i_2, ..., i_k\}$ が実行可能解であるか、すなわちうまく割り当てて全ての報酬を M 日後までに得られるか」です。これを愚直に判定すると全体として $O(N^2)$ となり間に合いません。少し考えてみると、$i_1, i_2, ..., i_k$ をうまく割り当てられることと、好きな順に次のように割り当てられることが同値であることがわかります。

1. 好きな順に割り当てる。今回割り当てるものを x とする。
2. M 日後から x 日前に割り当てようとする。できなければ (すなわち既に他の日雇いアルバイトが割り当てられていれば) M 日後から $x-1$ 日前に割り当てようとする。できなければ M 日後から $x-2$ 日前に割り当てようとする。これを初日まで繰り返しても割り当てられないとき、割り当てに失敗する。

すなわち、上の手順 2 は、M 日後から x 日前よりも以前であって、まだどの日雇いアルバイトも割り当てられていない日のうち最も遅い日が高速に求まれば高速に判定できます。これは、区間の最大値を求められ、一点更新が可能なセグメント木を使えば、S に対する割り当ての情報を保持しながら y に対して $O(logM)$ で判定できるため、ソートも合わせて全体として $O(NlogN + NlogM)$ で解くことができます。

上の判定を区間の最大値を求められるセグメント木の代わりに区間和を求められるセグメント木と二分探索を組み合わせて高速化することや、このような上手い割り当て方を考察せずとも区間加算、区間最小値、一点挿入が高速にできる平衡二分木を使って解くこともできます。

別解の冒頭でも述べましたが、最小費用流に帰着させたときの動作を考えると、マトロイドの解法と同じ動作になっているため、これを高速化する方針でも解くことができます。

E: Coins Respawn

(原案: IH19980412, 準備・解説: evima)

ゲーム終了時に $T \times P$ 枚のコインを支払うのではなく、辺を通るたびに P 枚のコインを支払う (その結果道中でコインの所持枚数が負になることも許容する) ことにすれば、時間の概念は不要になります。求めたいものは、各辺 i の重みを $C_i - P$ としたときの頂点 1 から頂点 N に至るパスの最大の重みです。各辺の重みをさらに -1 倍すると、これは最短路問題そのものになります。

最短路問題を解く有名なアルゴリズムにダイクストラ法 (Wikipedia の記事へのリンク) がありますが、今回は負の重みを持つ辺があるため使えません。ワーシャル–フロイド法 (Wikipedia の記事へのリンク) は負の重みを持つ辺に対応しますが、頂点数を V として $\mathrm{O}(V^3)$ の計算量を要しこれも今回用いるには厳しいでしょう。

今回用いるべきアルゴリズムは ベルマン–フォード法 (Wikipedia の記事へのリンク) です。このアルゴリズムは負の重みを持つ辺に対応し、頂点数を V、辺数を E として計算量 $\mathrm{O}(VE)$ で動作します。これを上で定めたグラフにほぼそのまま適用すれば問題が解けます。ただし、入力例 3 にあるような頂点 1 から頂点 N への移動と無関係な負閉路に反応しないように注意する必要があります。~~その最も簡単な方法は、アルゴリズムにおける「辺の緩和」を追加で V 回行って「頂点 N までの最短距離と思われる距離」が変化しないか確かめることでしょう。~~ (8/11 0:08 JST 修正: この記述は誤りでした)

F: Polynomial Construction

(原案: potetisensei, 準備・解説: evima)

以下、整数 i $(1 \leq i \leq p-1)$ に対し、$ij \equiv 1 \pmod{p}$ であるような整数 j $(1 \leq j \leq p-1)$ を i の逆数と呼びます。p が素数のとき、このような整数は必ず存在し、かつ一つに定まります (詳しくは後述します)。

式 $f(i) \equiv a_i \pmod{p}$ に $i = 0, 1, \ldots, p-1$ を実際に代入することで $b_0, b_1, \ldots, b_{p-1}$ に関する n 本の方程式が得られ、これをあたかも実数の連立一次方程式を解くかのように (ただし割り算を行おうとする際は代わりに割る数の逆数を掛ける) 掃き出し法を用いて解けば $\mathrm{O}(p^3)$ 時間で (唯一の) 解が求まりますが、これでは間に合いません。解をより直接的に構成する必要があります。

天下り的になってしまいますが、鍵を握るのはフェルマーの小定理 (Wikipedia の記事へのリンク): 「a が p の倍数でない整数のとき $a^{p-1} \equiv 1 \pmod{p}$」です。この先を読む前に、この定理を元にして条件を満たす多項式を得る方法を考案されることを勧めます。

(次のページへ続く)

多項式の構成方法を述べます。フェルマーの小定理より、整数 j $(0 \leq j \leq p-1)$ に対して値 $1-(x-j)^{p-1}$ は $x=j$ のときのみ 1、それ以外のとき 0 となります。この値を $a_j = 1$ であるようなすべての j に対して足し合わせれば所望の多項式が得られます。

あとは与えられた j に対して $(x-j)^{p-1}$ の展開を $\mathrm{O}(p)$ 時間で行えれば $\mathrm{O}(p^2)$ 時間で解が求まります。これには二項係数 $\binom{p-1}{i}$ の計算が必要であり、この計算は $0!, 1!, \ldots, (p-1)!$ の逆数を事前に求めておけば $(p-1)!$ に $i!$ の逆数と $(p-1-i)!$ の逆数を掛けることで行えます。

最後に、与えられた整数に対する逆数の求め方と存在性に関して述べます (今回は逆数を全探索しても間に合いますが)。再びフェルマーの小定理を用いると整数 i $(1 \leq i \leq p-1)$ の逆数は i^{p-2} を p で割った余りとして求められ、これは繰り返し二乗法を用いて高速に計算できます。

ABC 137 Editorial

DEGwer, drafear, evima, gazelle, IH19980412, potetisensei, yokozuna57

August 10, 2019

A: +-x

Implement the instructions given in the problem statement, just output the maximum of $A + B$, $A - B$, $A \times B$. A sample implementation in C++ is shown below.

Listing 1 An implementation example in C++

```
#include<bits/stdc++.h>
using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    cout << max({a + b, a - b, a * b}) << endl;
    return 0;
}
```

B: One Clue

(Original writer: DEGwer, preparation and editorial: evaim)

Let the leftmost black stone (the stone with the least coordinate) be L, then the coordinates of K black stones will be $L, L+1, \ldots, L+K-1$.

Since the stone at coordinate X is black, L is one of $X-K+1, X-K+2, \ldots, X$. Therefore, possible coordinates of black stone are those $2K-1$ values: $X-K+1, X-K+2, \ldots, X+K-1$.

The remaining task is a pure programming task, so we'll show you two implementation examples in Python instead of explanation. [*1] Similar implementation should be possible in other languages.

```python
K, X = map(int, input().split())
for i in range(X - K + 1, X + K):
    if i < X + K - 1:
        print(i, end=' ')
    else:
        print(i)
```

```python
K, X = map(int, input().split())
print(' '.join(map(str, range(X - K + 1, X + K))))
```

[*1] In fact, you can remove the third, fifth and sixth lines from the first implementation, and output a whitespace instead of newline after the last coordinate. Not only AtCoder, but many competitive programming's judge is tolerant to "whitespace-like letters" in many cases. Though, thinking about the visibility shown in terminals, we recommend you print new line at the end of the output

C: Green Bin

(Original writer: yokozuna57, preparation and editorial: evima)

First, given two strings s, t, there are the following two ways to check if s is an anagram of t:

1. For 26 kinds of alphabets, check if how many of them appears in s and t, and check if those alphabets appears same the same time.
2. Sort both strings and check if it corresponds.

This time there are at most 100 million strings, so if you try to check for all pairs (at most about 5 billions), it will not finish in time limit. The following is the optimized version of the second way, using the third input example:

1. Sort $s_1 =$ `abaaaaaaaa` and get `aaaaaaaaab` .
2. Sort $s_2 =$ `oneplustwo` and get `elnoopstuw` .
3. Sort $s_3 =$ `aaaaaaaaba` and get `aaaaaaaaab` . It has been appeared 1 time before, so add 1 to the answer.
4. Sort $s_4 =$ `twoplusone` and get `elnoopstuw` . It has been appeared 1 time before, so add 1 to the answer.
5. Sort $s_5 =$ `aaaaaaaaba` and get `aaaaaaaaab` . It has been appeared 2 time before, so add 2 to the answer.

The step of calculating "it has been appeared 1 times" and so on can be implemented with hash tables (C++'s unordered_map, Java's HashMap, Python's dictionary etc.) so that it works in a linear time. Otherwise, you can sort N "essences" obtained from N strings, but it may not fit the time limit, depending on programming languages.

D: Summer Vacation

If you dream of earning enough money to take a trip for Okinawa, this problem may be solved easily (Okinawa is one of the most famous tourist spot in Japan). You have to receive the rewards no later than M days later, so in the day i ($1 \leq i \leq M$) days before the day M days after today, you can only choose one-off job such that $A_j \leq i$. This kind of problem often becomes clearer if you examine the harder constrains first. Actually, if you check them back to forth (starting from the day 1 day after the day M days after today), you can choose j such that B_j is maximum among the js such that $A_j \leq 1$ (let it be j_1), then choose j such that B_j is maximum among the js such that $A_j \leq 2$ and $j \neq j_1$, ... and so on, you can find the optimal answer[*2].

Next, let's think about the implementation. The operations needed are

- to add a candidate
- pop the maximum value from the candidates

and each of them are done $O(N), O(M)$ times. If you use priority queue[*3], you can perform those operations fast.

By implementing it, you can solve this problem in a total of $O(M + NlogN)$ time.

Another Solution

Here is a little advanced solution. Briefly, the set of possible selection is a matroid, so it can be solved greedily. Also, the behavior of min-cost flow algorithm is the same.

First, matroid is a finite[*4] family of set[*5] I with the following axioms:

1. An empty set is an element of I.
2. When a set X is an element of I, any subset Y of X is an element of I.
3. If a pair of set X, Y with different number of elements ($|X| < |Y|$) are both contained in I, then there exist an element v which is in Y but not in X, such that after v is added to X it is still an element of I.

This time, it's like as follows. Let I be the set of possible selection[*6].

1. An empty set (choosing none of the jobs) is an element of I (in other words, it's a possible selection).

[*2] this can be proved by rephrasing this problem into minimum-cost flow problem. However, solving it as an minimum-cost flow problem takes a total of $O(M(N+M)log(N+M))$ time, so it won't fit in time limit.

[*3] We will not explain the structure of priority queue here. It can operate addition of element and removal of maximum element in $O(logX)$ time, and find maximum value in $O(1)$ time. In C++, it is provided in the standard C++ library (STL).

[*4] That means that the set contains finite number of elements.

[*5] A family of set is a set such that its element is a set, or a set of set.

[*6] A maximum selection is a selection of jobs, possibly not an optimal answer. This time, a set of indices such that there exists an assignment of those job into what day to work, such that all the rewards is paid in time.

2. If a selection of $\{i_1, i_2, ..., i_k\}$-th job can be assigned into workdays properly so that all the rewards is paid in time, if you decided not to do some of them and decided to choose $\{i'_1, i'_2, ..., i'_l\} \subseteq \{i_1, i_2, ..., i_k\}$, there still exists a valid assignment of workdays.
3. If both selection $\{i_1, i_2, ..., i_k\}$ and $\{j_1, j_2, ..., j_l\}$ has a valid assignment and $k < l$, then there exist x such that $\{i_1, i_2, ..., i_k, j_x\}$ has valid assignment.

We will not show the proof here, but these 3 conditions holds. so the set of possible selection for this problem is a matroid. Then what's a good point in matroid? Let's assume that a family of sets I is a matroid, and cost $w(x)$ is determined for any x that can be element of a set in I. This time, for each one-off job, "reward" is determined, that can be treated as its cost. Let define the cost of X in I as the sum of costs of each element of the set, $\sum_{x \in X} w(x)$. If $w(y) \geq 0$ for all y, you can find the maximum cost of elements in I greedil.

A greedy algorithm for a matroid I is as follows:

1. Let S be an empty set (in which elements are to be added one by one later to construct an optimal solution). According to the first axiom of matroid, $S \in I$ holds.
2. Sort the possible elements of solution (that is, the day-off jobs this time) and let it be $x_1, x_2, ..., x_N$.
3. For $y = x_1, x_2, ..., x_N$, do the following:
   (a) If even after y is added to S, it is still an element of I (that is $S \cup \{y\} \in I$), then add y to S.
4. Output S (S is an optimal answer).

In the algorithm above, the steps that depend on matroid is the judgement step of "check if even after y is added to S it's still in I." If you rephrase "check if even after y is added to S it's still in I" for this problem, it is "whether a set of indices of one-day off jobs $S \cup \{y\} = \{i_1, i_2, ..., i_k\}$ is a possible selection or not, in other words, all the rewards can be obtained until M days after today." If you judge this naively, it takes a total of $O(N^2)$ and won't finish in time. Thinking a little more carefully, it appears that the two conditions, whether $i_1, i_2, ..., i_k$ can be assigned to workdays properly, and whether these can be assigned in an arbitrary order as follows, is equivalent.

1. Assign them randomly. Let the current assignment be x.
2. Try to assign it to the day x days after the day M days after today. If not possible (that is, if there exist some other day-off jobs), try to assign it to the day $x - 1$ days after the day M days after today. If not possible (that is, if there exist some other day-off jobs), try to assign it to the day $x - 2$ days after the day M days after today. After checking until the first day, if it was not possible to assign to all the days, then it fails to assign.

Therefore, the second procedure above can be judged rapidly if such value can be found rapidly: the latest day that is before the day x days after the day M days after today, such that none of the jobs is assigned to. If you use a segment tree that provides maximum value of segments and update of one element, it can be judged in $O(logM)$ for y while saving the assignment information of S. So, it can be solved in a total of $O(NlogN + NlogM)$ time.

You can make the judge faster by using a segment tree that provides segment sum combined with

binary search instead of a segment tree that provides minimum of segments, or solve this problem without considering proper assignments, instead using a self-balancing binary tree that provides segment sums, segment minimums, and one-element updates.

As we stated in the beginning of this section, if you consider the behavior when the problem is regarded as a min-cost flow, the behavior is same to that of matroid, so you can solve the problem by making it faster.

E: Coins Respawn

(Original writer: IH19980412, preparation and editorial: evima)

Instead of paying $T \times P$ coins at the end of the game, if you decide to pay P coins every time you traverse an edge (and allow having negative number of coins), you don't have to think about the time order of events. What we want to find is the maximum cost of path from vertex 1 to vertex N, where cost of each edge i is set to $C_i - P$. If you multiply the cost by -1, the problem becomes a shortest-path problem.

One of the famous algorithms to find shortest path is Dijkstra's algorithm (Link to Wikipedia article), but this time there exist edges with negative cost, so it cannot be applied. Floyd – Warshall algorithm (Link to Wikipedia supports edges with negative costs, but it needs a total of $O(V^3)$ time where V is number of vertex, so it is a little bit difficult to finish in time.

The algorithm that should be used this time is Bellman – Ford algorithm (Link to Wikipedia article). This algorithm supports edges with negative costs and works in $O(VE)$ time where V is number of vertices and E is number of edges. If you apply almost directly to the graph mentioned above, you can solve the problem. Note that it should not misdetect negative cycle that is not concerned with the traverse from vertex 1 to vertex N. ~~The easiest way to achieve this is doing "relaxation" V more times and check if "apparent minimum cost to vertex N" does not change.~~ (8/11 0:08 JST corrected: this statement was wrong.)

F: Polynomial Construction

(Original writer: potetisensei, preparation and editorial: evima)

For an integer i ($1 \leq i \leq p-1$), we call an integer j ($1 \leq j \leq p-1$) such that $ij \equiv 1 \pmod{p}$ as "inverse of i." When p is prime, there always exists such an integer, and it's always unique (which will be explained later).

By assigning $i = 0, 1, \ldots, p-1$ to the equations $f(i) \equiv a_i \pmod{p}$, we will obtain n simultaneous equations, and with row reduction just like solving normal linear equations series (except that instead of dividing with some integer, multiplying its inverse), the (unique) solution can be found in a total of $O(p^3)$ time, but it is not enough. You have to find a solution more directly.

It's a bit kind of sudden, but the key is Fermat's little theorem (Link to Wikipedia article: "if a is an integer that is not a multiple of p, then $a^{p-1} \equiv 1 \pmod{p}$." Before reading further, we recommend you consider the ways to obtain a polynomial that meets the conditions.

(Continue to next page)

Here is a way of construction. By Fermat's little theorem, for any integer j $(0 \leq j \leq p-1)$, the value of $1-(x-j)^{p-1}$ is 1 only if $x = j$, and 0 otherwise. If you sum them up for all j, that will be the desired answer.

The remaining task is to expand $(x-j)^{p-1}$ in $\mathrm{O}(p)$ time, in which case the answer can be found in a total of $\mathrm{O}(p^2)$ time. For this task you need to calculate binominal coefficients $\binom{p-1}{i}$, and if you precalculate inverses of $0!, 1!, \ldots, (p-1)!$, those can be calculated by multiplying $(p-1!$ by inverse of $i!$ and inverse of $(p-1-i)!$.

Finally, we will explain how to obtain the inverse of a given number, and its existence (this time though, brute forcing inverses will meet the time). Applying Fermat's little theorem again, the inverse of an integer i $(1 \leq i \leq p-1)$ can be found as a remainder of i^{p-2} divided by p, and this can be calculated by "exponentiation by squaring."