Tutorial: Adelaide

Guilherme Ramos

A resposta é trivial, basta repetir a entrada e acrescentar as duas mensagens indicadas (que são sempre as mesmas).

Tutorial: Descompressão FoR

Daniel Saad Nogueira Nunes

A solução decorre diretamente da descrição do problema. Para cada bloco de inteiros, você deve ler o inteiro de referência e a quantidade de bits necessários para representar os inteiros comprimidos. Em seguida, você deve ler os inteiros comprimidos, descompactá-los e adicionar o inteiro de referência para obter os inteiros originais.

Uma forma de ler os inteiros comprimidos, dada uma posição i qualquer do vetor de bits, é usando o seguinte algoritmo.

Algoritmo 1: UNPACK(data, i, width)

Input: vetor de bytes *data*, posição i e quantidade de bits *width* a serem lidos
Output: inteiro descompactado

```
1  idx_byte ← ⌊i/8⌋
2  idx_bit ← i mod 8
3  result ← 0
4  if idx_bit + width ≤ 8 then
5      result ← (data[idx_byte] >> idx_bit) & ((1 << width) − 1)
6  else
7      bits_in_first_byte ← 8 − idx_bit
8      result ← (data[idx_byte] >> idx_bit) & ((1 << bits_in_first_byte) − 1)
9      result ← result | ((data[idx_byte + 1] & ((1 << (width − bits_in_first_byte)) − 1)) <<
           bits_in_first_byte)
10 return result
```

Tutorial: Figurinhas da Copa

Daniel Saad Nogueira Nunes

Uma forma gulosa de resolver esse problema é sempre adicionar a uma variável acumuladora a diferença $p[i+1]-p[i]$ quando $p[i+1] > p[i]$. Isso é realizado em tempo $\Theta(n)$. O algoritmo abaixo ilustra o explicado.

Algoritmo 1: MAX-PROFIT(p, n)

```
1 sum ← 0
2 for i ← 0 to n − 2 do
3 |  sum ← sum + max(0, p[i + 1] − p[i])
4 return sum
```

O único cuidado é que a soma pode ser bem grande, então é necessário utilizar um tipo de dado que suporte inteiros grandes, como `uint64_t` em `C` ou `C++`.

Tutorial: Genomas Revertidos

Daniel Saad Nogueira Nunes

Esse problema pode ser reduzido ao problema do menor caminho em um grafo onde os vértices são as permutações e existe uma aresta de um vértice u a um vértice v com peso 1 sempre que a permutação de v possa ser obtida da permutação de u com uma operação de reversão sobre índices i e j.

Contudo, o espaço de busca é muito grande, pois temos $n!$ permutações possíveis e, para testar todas as operações de reversão sobre uma permutação, gasta-se tempo $\Theta(n^2)$.

Precisamos melhorar a nossa busca em largura.

Otimização 1: A*

Em vez de uma simples busca em largura, podemos usar o algoritmo A* para encontrar o menor caminho. Aqui usaremos a heurística de *breakpoints*, ou ponto de quebras. Uma permutação π tem um ponto de quebra entre os elementos $\pi[i]$ e $\pi[i+1]$ se $|\pi[i+1] - \pi[i]| \neq 1$. A metade do número de pontos de quebra em uma permutação é um limite inferior para o número de operações de reversão necessárias para ordenar a permutação, visto que uma reversão elimina, no máximo, 2 pontos de quebra. Portanto, usaremos a metade do número de pontos de quebra como nossa heurística. Além disso, precisamos aplicar a heurística sobre a permutação π' aumentada, que acrescenta os elementos 0 e $n+1$ ao início e ao final da permutação, respectivamente. Por exemplo, se $\pi = \langle 3, 1, 2 \rangle$, $\pi' = \langle 0, 3, 1, 2, 4 \rangle$. Isso é necessário para que possamos contar os pontos de quebra entre o início e o fim da permutação.

Otimização 2: codificação da permutação

Para armazenar quais permutações já foram calculadas, usaremos uma tabela de *hash*. Contudo, para otimizar o tempo, não utilizaremos o vetor de permutação como chave da tabela. Podemos calcular uma bijeção de qualquer permutação para um número inteiro entre 0 e $n! - 1$ e reverter o processo sempre que desejarmos. Uma forma de fazer isso é utilizando a codificação de Lehmer e a base fatorial.

Tutorial: Horda de Gols

Jeremias Moreira Gomes

Para determinar a equipe vencedora, não é necessário armazenar toda a sequência de eventos. Basta manter, para cada equipe:

- sua pontuação final;
- o instante de sua última alteração de pontuação.

Para isto, basta manter dusa tabelas associativas (hash maps) e ao processar um evento da equipe s no instante t com valor g:

- somamos g à pontuação de s;
- atualizamos o último instante de s para o maior valor já observado.

Após processar todos os eventos, percorremos todas as equipes encontradas.
A equipe vencedora será aquela que:

1. possui a maior pontuação final;
2. em caso de empate, possui o menor instante de última alteração.

Assim, a leitura dos eventos realiza E atualizações e ao final percorremos T equipes registradas, onde a complexidade fica:

$$O(E + T)$$

Tutorial: Kamzabek, o Kazakh

Alberto Neto

For a permutation p, define the array $a(p)$ by a_i = the number of elements to the right of p_i that are smaller than p_i. (Lehmer code)

Then the following facts hold:

- $0 \le a_i \le n-1-i$,
- the sum of all a_i is equal to the number of inversions,
- there is a bijection between a and p,
- if for two permutations x and y we have $x \le y$ lexicographically, then $a(x) \le a(y)$.

So the problem becomes counting arrays a such that $L \le a \le R, \quad 0 \le a_i \le n-1-i, \quad \sum a_i = k$.

We compute $F(S)$ — number of valid arrays $a \le S$ — and answer is $F(R) - F(L^-)$.

To compute $F(S)$, scan from left to right. At position i, assume prefix is equal to S, and try to make the first smaller choice here.

Let:

- pref = sum of fixed prefix,
- rem $= k -$ pref,
- $\ell = n-1-i$ (remaining length).

If we set $a_i = t < S_i$, then the suffix can be any valid array of length ℓ with sum rem $- t$.

Define Ways(ℓ, s) = number of valid suffixes of length ℓ with sum s.

Then for this position we add all such possibilities.

How to compute Ways**:**

- If $\ell \le k$, use standard DP over inversion sequences.
- If $\ell > k$, observe that only the last k positions are constrained; the first $\ell - k$ positions behave like free nonnegative variables.

So we split:

- free part (size $\ell - k$) $\to$ simple combinations (stars and bars),
- tail (size k) $\to$ DP.

To answer queries efficiently, we precompute prefix sums: $H(\ell, r) = \sum_{s=0}^{r} \text{Ways}(\ell, s)$, so each step becomes a single range query on H.

A key observation for complexity: although we scan all n positions, we only need to evaluate $H(\ell, r)$ when $S_i > 0$ (otherwise there is no smaller choice). Each such step increases pref by at least 1, and once pref $> k$ we stop. Therefore this happens at most k times, so all expensive work is $O(k^2)$ overall.

Finally, if we never go smaller and $\sum S_i = k$, we add 1.

Complexity:

- DP: $O(k^2)$,
- processing permutation: $O(n \log n)$,
- suffix queries: overall $O(k^2)$.

Total: $O(n \log n + k^2)$.

Tutorial: Lobotomia Mental

Jeremias Moreira Gomes

A principal observação do problema é perceber que a representação em "six seven" corresponde exatamente à representação binária de um número, onde "six" representa o dígito 1 e "seven" representa o dígito 0.

Para resolver o problema, basta ordenar os números utilizando como critério a quantidade de dígitos "1" na representação binária. Caso haja empate, o critério fica sendo o valor do número em si. A complexidade da solução é dada pelo custo de ordenação, que é:

$$O(N \log N)$$