Olimpíada Brasileira de Informática 2017

Nível Sênior – Fase 2: *Upsolving*

Prof. Edson Alves – UnB/FGA

2020

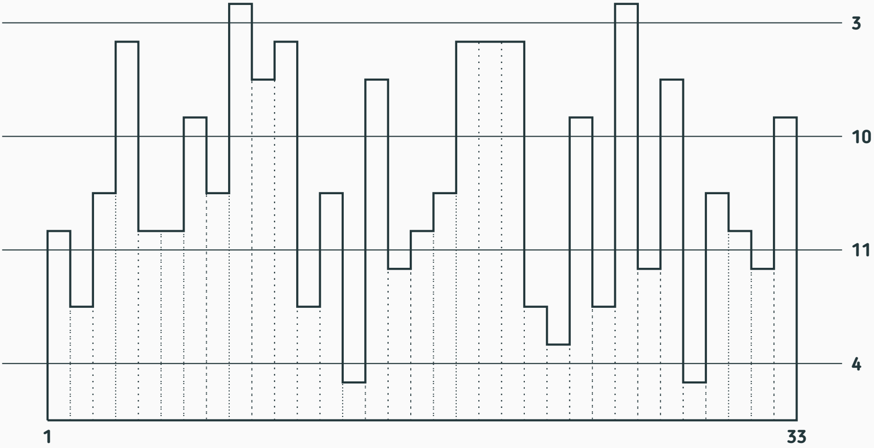
Sumário

Cortando o Papel

Problema

Uma folha de papel é composta de uma sequência de retângulos com diferentes alturas mas com larguras fixas, tal que as bases dos retângulos estão assentadas em uma linha horizontal. A figura ilustra uma folha exemplo com 33 retângulos. Nós gostaríamos de fazer um único corte horizontal, com a ajuda de um estilete e uma régua, que maximize o número resultante de pedaços separados pelo corte. A figura mostra quatro diferentes cortes que resultariam, respectivamente, em 4, 11, 10 e 3 pedaços.

Problema



Entrada

A primeira linha da entrada contém um inteiro N, representando o número de retângulos na folha de papel. A segunda linha contém N inteiros A_i, $1 \leq i \leq N$, representando a sequência de alturas dos retângulos.

Saída

Seu programa deve imprimir uma linha contendo um inteiro representando o número máximo de pedaços possível, com um único corte horizontal.

Entrada e saída

Restrições

- $2 \leq N \leq 10^5$
- $1 \leq A_i \leq 10^9$, para $1 \leq i \leq N$

Informações sobre a pontuação

- Em um conjunto de casos de teste somando 40 pontos, $N \leq 1000$

Exemplo de entradas e saídas

Entrada	Saída
5 6 1 2 4 1 3 3 4 3 6 4 5 2 2 4 1 3 5 5	7

Exemplo de entradas e saídas

Entrada

```
7 10
1 2 5
3 1 32
1 4 3
2 3 4
2 6 20
6 3 1
6 4 9
6 5 6
3 7 18
5 7 2
```

Saída

```
18
```

Solução

- Uma parte importante do problema consiste em identificar o número de pedaços resultantes de um corte na altura x
- Conforme pode ser observado na figura, um pedaço consiste em uma série de retângulos contíguos cujas alturas são todas maiores do que x
- Além disso, há o pedaço formado por todos os retângulos, ou partes de retângulo, que ficaram abaixo de x
- É possível usar a técnica de dois ponteiros para identificar os pedaços resultantes de um corte em x

Rotina que computa os pedaços para um corte em x

```cpp
int pieces(double x, int N, const vector<int>& hs)
{
    auto L = 0, res = 0;

    while (L < N)
    {
        auto R = L + 1;

        if (hs[L] > x)
        {
            while (R < N and hs[R] > x)
                ++R;

            ++res;
        }

        L = R;
    }

    return res + 1;
}
```

Solução

- A rotina $pieces(x)$, que computa o número de pedaços resultantes de um corte em x, tem complexidade $O(N)$
- Uma solução para o problema consiste em computar o valor máximo de $pieces(x)$ para $x \in [1, M]$, onde $M = \max\{h_1, h_2, \ldots, h_N\}$
- Tal solução tem complexidade $O(NM)$, e como $N \leq 10^5$ e $M \leq 10^9$, tal abordagem resulta em um veredito TLE

Solução TLE $O(MN)$

```cpp
#include <bits/stdc++.h>

using namespace std;

int pieces(double x, int N, const vector<int>& hs)
{
    auto L = 0, res = 0;

    while (L < N)
    {
        if (hs[L] <= x)
        {
            ++L;
            continue;
        }

        auto R = L + 1;

        while (R < N and hs[R] > x)
            ++R;

```

Solução TLE $O(MN)$

```
        ++res;
        L = R;
    }

    return res + 1;
}

int solve(int N, const vector<int>& hs)
{
    int M = *max_element(hs.begin(), hs.end()), ans = 2;

    for (int i = 1; i <= M; ++i)
        ans = max(ans, pieces(i, N, hs));

   return ans;
}

int main()
{
    ios::sync_with_stdio(false);

```

Solução TLE $O(MN)$

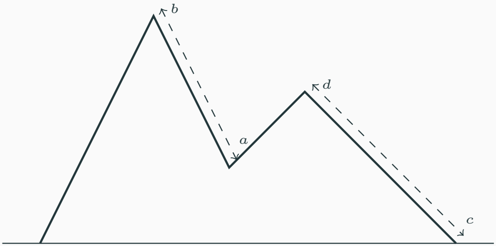
```cpp
    int N;
    cin >> N;

    vector<int> hs(N);

    for (int i = 0; i < N; ++i)
        cin >> hs[i];

    auto ans = solve(N, hs);

    cout << ans << '\n';

    return 0;
}
```

Solução

- Por meio de uma observação cuidadosa dos cortes para valores consecutivos de x leva a conclusão que os valores de $pieces(x)$ só se modificam nos valores que correspondem às alturas dos retângulos
- Isto reduz a quantidade máxima de alturas a serem verificadas de 10^9 para 10^5
- Embora o veredito continue sendo TLE, esta redução permite resolver com sucesso o conjunto de casos de testes que correspondem a 40 pontos
- Para somar todos os 100 pontos, é preciso uma abordagem distinta que identifique, de forma eficiente, os valores de x que levam aos cortes com o número máximo de pedaços

Solução

- Considere a figura abaixo



- Um corte com $x \in [a, b)$ divide o primeiro pico em duas partes
- Já um corte com $x \in [c, d)$ divide o segundo pico em duas partes

Solução

- Agora, se $x \in [a, b) \cap [c, d)$, ambos serão divididos
- Considerando que o i-ésimo pico pode ser caracterizado por um ponto de máximo local b_i e um ponto de mínimo local a_i, o problema passa a ser determinar a maior interseção possível entre todos os intervalos $[a_i, b_i)$
- Para identificar de forma mais simples e eficiente os pontos de máximo e mínimo locais, a entrada pode ser comprimida, eliminando-se alturas iguais e consecutivas:

```
auto it = unique(hs.begin(), hs.end());

N = (int) distance(hs.begin(), it);
hs.resize(N);
```

Solução

- Após a compressão, há um ponto de máximo local em h_i se $h_{i-1} < h_i$ e $h_{i+1} < h_i$
- De forma análoga, há um ponco de mínimo local em h_i se $h_{i-1} > h_i$ e $hi + 1 > h_i$
- Por fim, a maior interseção possível entre todos os intervalos $[a_i, b_i)$ pode ser determinada por meio de um algoritmo de *line sweep*
- Para cada intervalo a_i deve ser criados dois eventos: um evento de abertura em a_i e um e fechamento em b_i
- Os eventos devem ser ordenados pelo ponto de ocorrência
- Em caso de empate, os eventos de fechamento devem vir antes dos de abertura

Solução

- Os eventos devem ser processados em ordem
- Inicialmente não há nenhum intervalo aberto
- Cada abertura incrementa em uma unidade o número de intervalos abertos
- Os eventos de fechamento reduzem o número de intervalos abertos em uma unidade
- A solução do problema é o maior número de intervalos abertos registrado durante o processamento dos eventos
- Assim, como a compressão tem complexidade $O(N)$ e a ordenação dos eventos $O(N \log N)$, a solução tem complexidade $O(N \log N)$

Solução $O(N \log N)$

```cpp
#include <bits/stdc++.h>

using namespace std;
using ii = pair<int, int>;

int solve(int N, vector<int>& hs)
{
    // Compressão dos adjacentes iguais
    auto it = unique(hs.begin(), hs.end());

    N = (int) distance(hs.begin(), it);
    hs.resize(N);

    // Identificação dos pontos críticos
    vector<int> cs;

    for (int i = 1; i <= N - 2; ++i)
        if ((hs[i - 1] < hs[i] and hs[i] > hs[i + 1]) ||
            (hs[i - 1] > hs[i] and hs[i] < hs[i + 1]))
                cs.push_back(hs[i]);

```

Solução $O(N \log N)$

```cpp
    // Finaliza com um mínimo
    if (cs.size() % 2)
        cs.push_back(0);

    // Geração dos eventos para o sweep line
    vector<ii> es;
    int OPEN = 1, CLOSE = -1, ans = 1, open = 0;

    for (size_t i = 0; i < cs.size(); i += 2)
    {
        es.push_back(ii(cs[i + 1], OPEN));
        es.push_back(ii(cs[i], CLOSE));
    }

    sort(es.begin(), es.end());

    for (auto e : es)
    {
        open += e.second;
        ans = max(ans, open);
    }
```

Solução $O(N \log N)$

```
    return ans + 1;
}

int main()
{
    ios::sync_with_stdio(false);

    int N;
    cin >> N;

    vector<int> hs(N + 2, 0);

    for (int i = 1; i <= N; ++i)
        cin >> hs[i];

    cout << solve(N, hs) << '\n';

    return 0;
}
```

Referências

1. Dario e Xerxes
2. Frete
3. Mapa
4. Cortando o Papel
5. URI 1828 – Bazinga!