Paradigmas de Resolução de Problemas

Programação Dinâmica – Definição: Exercícios Resolvidos

Prof. Edson Alves - UnB/FGA

2020

Sumário

OJ 10465 – Homer Simpson

Problema

Homer Simpson, a very smart guy, likes eating Krusty-burgers. It takes Homer m minutes to eat a Krusty-burger. However, there's a new type of burger in Apu's Kwik-e-Mart. Homer likes those too. It takes him n minutes to eat one of these burgers. Given t minutes, you have to find out the maximum number of burgers Homer can eat without wasting any time. If he must waste time, he can have beer.

Entrada e saída

Input

Input consists of several test cases. Each test case consists of three integers m, n, t $(0 < m, n, t < 10000)$. Input is terminated by EOF.

Output

For each test case, print in a single line the maximum number of burgers Homer can eat without having beer. If homer must have beer, then also print the time he gets for drinking, separated by a single space. It is preferable that Homer drinks as little beer as possible.

Exemplo de entradas e saídas

Sample Input	Sample Output
3 5 54	18
3 5 55	17

Solução $O(T)$

- Observe que o problema consiste em minimizar o consumo de cerveja e, em segundo lugar, maximizar o número de hambúrger
- O problema pode ser caracterizado por um único parâmetro: o recurso disponível t
- Para cada subproblema $p(t)$ a solução é caracterizada por um par de valores (b, h): o número mínimo de cervejas e o máximo de hambúrgeres a serem consumidos
- Para manter a ordenação das soluções ótimas e usar a função `max()` do C++, o valor de b será representado pelo seu simétrico

Solução $O(T)$

- O caso base ocorre quando $t = 0$: neste caso, $p(0) = (0, 0)$
- Há 3 transições possíveis para o estado $p(t)$:
    1. gastar todo o recurso com cervejas
    2. comprar uma cerveja por m
    3. comprar uma cerveja por n
- Como há $O(T)$ estados distintos, e as transições tem custo $O(1)$, uma solução baseada em programação dinâmica tem complexidade $O(T)$
- A complexidade em memória também é $O(T)$

Solução $O(T)$

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

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

const int MAX { 10010 };

ii st[MAX];

ii dp(int t, int N, int M)
{
    if (t == 0)
        return { 0, 0 };

    if (st[t] != ii(-1, -1))
        return st[t];

    auto res = ii(-t, 0);

    if (t >= N)
```

Solução $O(T)$

```cpp
    {
        auto [beer, sol] = dp(t - N, N, M);
        res = max(res, ii(beer, sol + 1));
    }

    if (t >= M)
    {
        auto [beer, sol] = dp(t - M, N, M);
        res = max(res, ii(beer, sol + 1));
    }

    st[t] = res;
    return res;
}

ii solve(int N, int M, int T)
{
    memset(st, -1, sizeof st);
    return dp(T, N, M);
}

```

Solução $O(T)$

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

    int M, N, T;

    while (cin >> M >> N >> T)
    {
        auto [beer, ans] = solve(M, N, T);

        cout << ans;

        if (beer)
            cout << ' ' << -beer;

        cout << '\n';
    }

    return 0;
}
```

SPOJ SQRBR – Square Brackets

Problema

You are given:

- a positive integer n,
- an integer k, $1 \leq k \leq n$,
- an increasing sequence of k integers $0 < s_1 < s_2 < \ldots < s_k \leq 2n$.

What is the number of proper bracket expressions of length $2n$ with opening brackets appearing in positions $s_1, s_2, \ldots, s_k$?

Problema

Illustration

Several proper bracket expressions:

```
[[]][[[]][]]
[[[][]]][][[]]
```

An improper bracket expression:

```
[[[][]]][]][[]]
```

There is exactly one proper expression of length 8 with opening brackets in positions 2, 5 and 7.

Problema

Task

Write a program which for each data set from a sequence of several data sets:

- reads integers n, k and an increasing sequence of k integers from input,
- computes the number of proper bracket expressions of length $2n$ with opening brackets appearing at positions $s_1, s_2, \ldots, s_k$,
- writes the result to output.

Input

The first line of the input file contains one integer d, $1 \leq d \leq 10$, which is the number of data sets. The data sets follow. Each data set occupies two lines of the input file. The first line contains two integers n and k separated by single space, $1 \leq n \leq 19, 1 \leq k \leq n$. The second line contains an increasing sequence of k integers from the interval $[1; 2n]$ separated by single spaces.

Output

The i-th line of output should contain one integer – the number of proper bracket expressions of length $2n$ with opening brackets appearing at positions $s_1, s_2, \ldots, s_k$.

Exemplo de entradas e saídas

Sample Input

```
5
1 1
1
1 1
2
2 1
1
3 1
2
4 2
5 7
```

Sample Output

```
1
0
2
3
2
```

Solução $O(N^2)$

- Uma solução de força bruta listaria todas as 2^{2n} strings s_i formadas pelos caracteres '`[`' e '`]`' e, para cada i, identificaria se s_i é uma sequência válida e, em caso afirmativo, se os caracteres das posições indicadas na entrada são iguais a '`[`'
- A verificação da validade de s_i é feita em $O(N)$, de modo que tal solução teria complexidade $O(N2^{2N})$, o que levaria a um veredito TLE
- Uma forma de reduzir esta complexidade é construir as sequências válidas iterativamente e não recalcular a validade de uma mesma subsequência múltiplas vezes
- Isto pode ser feito por meio de programação dinâmica

Solução $O(N^2)$

- Seja s uma string de tamanho $2n$ tal que $s_j =$'[' para toda posição j indicada na entrada
- Defina $c(i, open)$ como o número de sequências válidas que podem ser formadas a partir do sufixo $s[i, 2n]$ sem modificar os caracteres s_j pré-definidos, considerando que restaram $open$ caracteres '[' sem o ']' correspondente no prefixo $s[1, (i-1)]$
- Uma vez que os sufixos $s[m, 2n]$ são vazios se $m > 2n$, então $c(m, open) = 0$ se $m > 2n$
- Atenção, porém, ao caso $m = 2n + 1$: ele trata o primeiro sufixo vazio, e indica que todos os caracteres anteriores foram definidos
- Assim, se $open = 0$, a sequência definida anteriormente é valida, de modo que $c(2n + 1, 0) = 1$
- Os demais permanecem iguais a zero, se $m > 2n$

Solução $O(N^2)$

- Para $1 \leq i \leq 2n$, há duas transições possíveis:
  1. adicionar mais um símbolo '`[`' na string
  2. adicionar um símbolo '`]`' na string, se $open > 0$ e se a posição não estiver pré-definida
- A primeira transição corresponde a

$$c(i, open) = c(i+1, open+1)$$

- Esta primeira transição sempre pode ser feita
- A segunda transição nem sempre pode ser feita, devido as restrições apresentadas
- Caso $open > 0$ e i não seja um dos índices pré-definidos com '`[`', então

$$c(i, open) = c(i+1, open+1) + c(i+1, open-1)$$

- Como há $O(N^2)$ estados e as transições são feitas em $O(1)$, a solução tem complexidade $O(N^2)$

Solução $O(N \log N)$

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

using namespace std;

const int MAX { 25 };

int st[2*MAX][2*MAX];

int dp(int i, int open, int N, const set<int>& xs)
{
    if (i == N)
        return open ? 0 : 1;

    if (st[i][open] != -1)
        return st[i][open];

    auto res = dp(i + 1, open + 1, N, xs);

    if (xs.count(i) == 0 and open)
        res += dp(i + 1, open - 1, N, xs);

```

Solução $O(N \log N)$

```
    st[i][open] = res;
    return res;
}

int solve(int N, const set<int>& xs)
{
    memset(st, -1, sizeof st);

    return dp(0, 0, 2*N, xs);
}

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

    int T;
    cin >> T;

    while (T--)
    {
        int N, K;
```

Solução $O(N \log N)$

```
        cin >> N >> K;

        set<int> xs;

        for (int i = 0; i < K; ++i)
        {
            int x;
            cin >> x;

            xs.insert(x - 1);
        }

        auto ans = solve(N, xs);

        cout << ans << '\n';
    }

    return 0;
}
```

Referências

1. OJ 10465 – Homer Simpson
2. SPOJ SQRBR - Square Brackets