Apuntes OIFem II Nivel 3

Programación dinámica avanzada- CSES Problem Set

Array Description

Teoría

- Programación dinámica top-down
- Operaciones con módulo

Solución

Para este problema, construiremos una solución $O(nm)$ (aunque en la práctica será mucho más rápida en la mayoría de casos). La clave será empezar por la última posición del array e ir *delegando* el trabajo al índice anterior hasta llegar al principio, haciendo recursión y guardando resultados intermedios.

```cpp
#include <iostream>
#include <vector>
using namespace std;
typedef long long int ll;

int n, m;
int mod = 1e9 + 7;
vector<int> nums;
vector<vector<int>> DP;

// ways(i, j) = formas de rellenar nums[0..i] donde nums[i] = j
int ways(int index, int valueAtIndex) {
    // el valor no está en el rango aceptado
    if (valueAtIndex < 1 || valueAtIndex > m)
        return 0;
    // ya hemos calculado este resultado
    if (DP[index][valueAtIndex] != -1)
        return DP[index][valueAtIndex];
    // la posición de nums ya tiene valor y no es el que buscamos
    if (nums[index] != 0 && nums[index] != valueAtIndex)
        return DP[index][valueAtIndex] = 0;
    // hemos llegado al principio de nums
    if (index == 0)
        return DP[index][valueAtIndex] = 1;
    // la posición anterior puede ser la actual - 1, la actual o la actual + 1
```

```cpp
    return DP[index][valueAtIndex] = ((ways(index-1, valueAtIndex-1) + ways(index-1,
    ↪  valueAtIndex)) % mod + ways(index-1, valueAtIndex+1)) % mod;
}

int solve() {
    DP.assign(n, vector<int>(m+1, -1));
    int ans = 0;
    // retornamos la suma de todas las formas de terminar nums con un número del rango
    for (int i = 1; i <= m; i++) {
        ans += ways(n-1, i);
        ans %= mod;
    }
    return ans;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    cin >> n >> m;
    nums = vector<int>(n);
    for (int i = 0; i < n; i++)
        cin >> nums[i];
    cout << solve() << '\n';
    return 0;
}
```

Rectangle Cutting

Teoría

- Programación dinámica top-down

Solución

Dado un rectángulo con lados de longitud a y b, tenemos dos opciones: $a = b$, por lo que el rectángulo es un cuadrado y no hace falta que hagamos cortes; o $a \neq b$, en cuyo caso probamos todos los cortes posibles y escogemos el mejor.

```cpp
#include <iostream>
#include <vector>
using namespace std;

vector<vector<int>> DP;

// cuttings(a, b) = mínima cantidad de cortes para un rectángunuevoLado con lados a y b
int cuttings(int a, int b) {
    if (DP[a][b] != -1)
        return DP[a][b];
    if (a == b)
        return DP[a][b] = 0;
    int ans = 1e9;
    // Cortamos el lado a en dos lados:
    for (int nuevoLado = 1; nuevoLado < a; nuevoLado++)
        ans = min(ans, 1 + cuttings(nuevoLado, b) + cuttings(a-nuevoLado, b));
    // Cortamos el lado b en dos lados:
    for (int nuevoLado = 1; nuevoLado < b; nuevoLado++)
        ans = min(ans, 1 + cuttings(nuevoLado, a) + cuttings(b-nuevoLado, a));
    return DP[a][b] = DP[b][a] = ans;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    DP.assign(501, vector<int>(501, -1));
    int a, b;
    cin >> a >> b;
    cout << cuttings(a, b);
    return 0;
}
```

Money Sums

Teoría

- Programación dinámica bottom-up

Solución

Este problema nos sirvió como ejemplo para ver que podemos girar el orden del bucle interno para evitar usar dos dimensiones de memoria en la DP.

Para resolverlo, iteramos sobre todas las monedas, guardando en todo momento las sumas alcanzables con las monedas anteriores y actualizándolas cada vez que añadimos una moneda si esta nos desbloquea posibilidades.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void solve() {
    int N;
    cin >> N;
    vector<int> C(N);
    int sum = 0;
    for (int i = 0; i < N; i++) {
        cin >> C[i];
        sum += C[i];
    }

    // DP[s] = true si es posible sumar s con las monedas que hay
    vector<bool> DP(sum + 1, 0);
    DP[0] = true;
    for (int i = 0; i < N; i++)
        for (int j = sum; j >= C[i]; j--)
                DP[j] = DP[j] || DP[j - C[i]];

    cout << count(DP.begin(), DP.end(), true)-1 << '\n';
    for (int i = 1; i <= sum; i++)
        if (DP[i])
            cout << i << " ";
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    solve();
    return 0;
}
```

Removal Game

Teoría

- Minimax
- Programación dinámica top-down

Solución

Usamos minimax para encontrar la estrategia óptima, al ser un juego de suma cero.

```cpp
#include <iostream>
#include <vector>
using namespace std;
typedef long long int ll;

int N;
vector<vector<ll>> DP;
vector<int> nums;

// puntuación máxima alcanzable empezando a jugar con el segmento [l,r]
ll maxPos(int l, int r) {
    if (l > r)
        return 0;
    if (DP[l][r] != -1)
        return DP[l][r];
    if (l == r)
        return DP[l][r] = nums[l];
    // minimax
    return DP[l][r] = max(min(maxPos(l+2, r), maxPos(l+1, r-1)) + nums[l], min(maxPos(l+1,
    ↪  r-1), maxPos(l, r-2)) + nums[r]);
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    cin >> N;
    DP.assign(N, vector<ll>(N, -1));
    nums = vector<int>(N);
    for (int i = 0; i < N; i++)
        cin >> nums[i];
    cout << maxPos(0, N-1);
    return 0;
}
```

Two Sets II

Teoría

- Programación dinámica bottom-up

Solución

Modificamos el código de Money Sums para guardar el número de sumas en lugar de si son posibles o no. El único extra interesante de este problema es que tenemos que hacer módulo del resultado dividido entre 2. Para evitar tener un resultado incorrecto en el caso en el que el resultado esté entre $M = 10^9 + 7$ y $2M$, durante el cálculo iremos haciendo módulo con $2M$ y solo haremos módulo M al final del todo.

```cpp
#include <iostream>
#include <vector>
using namespace std;
typedef long long int ll;

ll solve(int n) {
    if ((n * (n + 1)) % 4 != 0)
        return 0;

    int sumaBuscada = n * (n + 1) / 4;
    vector<ll> DP(sumaBuscada + 1, 0); // DP[i] = formas de sumar i con números diferentes entre 1 y n
    ll mod = (1e9 + 7) * 2;
    DP[0] = 1;
    for (int i = 1; i <= n; i++) {
        // iteramos en orden descendente para evitar repetir i en la suma
        for (int j = sumaBuscada; j >= i; j--) {
            DP[j] += DP[j - i];
            DP[j] %= mod;
        }
    }
    return (DP[sumaBuscada] / 2) % (mod / 2);
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    int n;
    cin >> n;
    cout << solve(n);
    return 0;
}
```

Increasing Subsequence

Teoría

- Algoritmo $O(n \log n)$ para LIS

Solución

Usamos el algoritmo de Longest Increasing Subsequence que funciona en $O(n \log n)$. Aquí está explicado.

```cpp
#include <iostream>
#include <vector>
using namespace std;

int N;
vector<int> nums, last;

// encuentra el mayor índice i tal que last[i] < A
int findPos(int A) {
    int lo = 0, hi = (int) last.size() - 1, mid, ret = 0;
    while (lo <= hi) {
        mid = lo + (hi - lo) / 2;
        if (last[mid] == A)
            return mid;
        else if (last[mid] < A) {
            ret = mid;
            lo = mid + 1;
        } else
            hi = mid - 1;
    }
    return ret + 1;
}

int solve() {
    last.push_back(nums[0]);
    int len = 1; // la longitud de last
    // vamos iterando sobre los elementos de nums, añadiéndolos a la secuencia activa que
    ↪ corresponda
    for (int i = 1; i < N; i++) {
        if (nums[i] < last[0]) {
            last[0] = nums[i];
        } else if (nums[i] > last[len - 1]) {
            last.push_back(nums[i]);
            len++;
        } else if (nums[i] != last[len - 1]) {
            int pos = findPos(nums[i]);
            last[pos] = nums[i];
        }
    }
    return len;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    cin >> N;
```

```
46      nums = vector<int>(N);
47      for (int i = 0; i < N; i++)
48          cin >> nums[i];
49      cout << solve();
50      return 0;
51  }
```

Elevator Rides

Teoría

- Operaciones con bits
- Programación dinámica con máscaras de bits

Solución

Representaremos grupos de personas usando máscaras de bits.

Guardaremos en todo momento para cada máscara el mínimo número de ascensores que hemos encontrado y, partiendo de este mínimo número de subidas y bajadas, el mínimo peso que puede colocarse en la última de estas. Usando una variante de la versión de Dijkstra aprendida el año pasado, vamos encontrando mejoras a los valores de cada máscara y compartiendo estas con sus máscaras vecinas hasta haber explorado ya todas las mejoras.

```
#include <iostream>
#include <vector>
#include <queue>
#include <set>
using namespace std;
typedef long long int ll;

vector<ll> weights;
int n;
ll maxWeight;

int solve() {
    vector<int> minRides(1<<n, 100);
    minRides[0] = 1;
    vector<ll> lastRideWeight(1<<n);
    lastRideWeight[0] = 0;
    queue<int> Q;
    Q.push(0);
    while(!Q.empty()) {
        int mask = Q.front();
        Q.pop();
        for (int i = 0; i < n; i++) {
            if (!(mask & (1<<i))) {
                int newMask = mask | (1<<i);
                if (lastRideWeight[mask] + weights[i] <= maxWeight) {
                    if (minRides[newMask] > minRides[mask] ||
                        (minRides[newMask] == minRides[mask] && lastRideWeight[newMask] >
                        ↪  lastRideWeight[mask] + weights[i])) {
                        minRides[newMask] = minRides[mask];
                        lastRideWeight[newMask] = lastRideWeight[mask] + weights[i];
                        Q.push(newMask);
                    }
                } else if (minRides[newMask] > minRides[mask]+1 ||
                        (minRides[newMask] == minRides[mask]+1 && lastRideWeight[newMask] >
                        ↪  weights[i])) {
                    minRides[newMask] = minRides[mask]+1;
                    lastRideWeight[newMask] = weights[i];
                    Q.push(newMask);
                }
```

```
            }
        }
    }
    return minRides[(1<<n)-1];
}

int main() {
    cin >> n >> maxWeight;
    weights = vector<ll>(n);
    for (int i = 0; i < n; i++)
        cin >> weights[i];
    cout << solve() << '\n';
    return 0;
}
```