Soluciones OIFem II Nivel 2

Entrenamiento 3 - Programación dinámica

Minimizing Coins

Solución

Definimos $dp[i]$ como el mínimo número de monedas para formar el valor i. Sabemos que $dp[0] = 0$ ya que no se necesitan monedas para formar el 0. A partir de aquí usamos la siguiente transición: Si estamos calculando $dp[i]$, probamos qué pasaría usando alguna de las monedas. Por ejemplo, una moneda de valor m, entonces usamos 1 moneda + las necesarias para formar $i - m$, que la forma óptima ya habrá sido calculada en un paso anterior y guardada en $dp[i - m]$. Por lo que $dp[i] = \min_{m \text{ moneda}} \Big\{1 + dp[i - m]\Big\}$

Código

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

using vi = vector<int>;
using vvi = vector<vi>;

const int INF = 1e9;
vi c, dp;

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int n, x;
    cin >> n >> x;
    c = vi(n);
    for (int i = 0; i < n; ++i) cin >> c[i];

    dp = vi(x+1, INF); // dp[i] : mínimo número de monedas para formar i
    dp[0] = 0;
    for (int i = 1; i <= x; ++i) {
        for (int m : c) {
            if (m <= i) dp[i] = min(dp[i], 1+dp[i-m]);
        }
    }
    if (dp[x] == INF) cout << -1 << endl;
    else cout << dp[x] << endl;
}
```

Balance beam (2)

Solución

Visto en clase.

Código

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

using vi = vector<int>;
using vvi = vector<vi>;


int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    const int M = 1e8+7;

    int m, n;
    cin >> m >> n;
    vi l(n);
    for (int i = 0; i < n; ++i) cin >> l[i];

    // dp[i][j] : se puede conseguir llegar a la posicion
    //            j haciendo los i primeros saltos? (0/1)
    vvi dp(n+1, vi(m+1, 0));

    // dp2[i][j] : formas de conseguir llegar a la posicion
    //             j haciendo los i primeros saltos
    vvi dp2(n+1, vi(m+1, 0));

    dp[0][m/2] = 1;
    dp2[0][m/2] = 1;
    for (int i = 1; i <= n; ++i) {
        int salto = l[i-1];
        for (int j = 0; j <= m; ++j) {
            if (j-salto >= 0 and dp[i-1][j-salto]) {
                dp[i][j] = 1;
                dp2[i][j] += dp2[i-1][j-salto];
                dp2[i][j] %= M;
            }
            if (j+salto <= m and dp[i-1][j+salto]) {
                dp[i][j] = 1;
                dp2[i][j] += dp2[i-1][j+salto];
                dp2[i][j] %= M;
            }
        }
    }

    for (int j = 0; j <= m; ++j) {
        if (dp[n][j]) {
            cout << j - m/2 << " " << dp2[n][j] << endl;
        }
    }
}
```

Suma de productes

Solución

Visto en clase. La idea más importante es que como solo podemos multiplicar como mucho 3 números consecutivos, vamos probando en cada paso las 3 opciones basándonos en las soluciones óptimas anteriormente calculadas.

Código

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

using vl = vector<long long>;

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int n;
    while (cin >> n) {
        vl v(n);
        for (int i = 0; i < n; ++i) cin >> v[i];

        // dp[i] : máximo usando los i primeros números
        vl dp(n+1);
        dp[1] = v[0];
        for (int i = 2; i <= n; ++i) {
            dp[i] = dp[i-1] + v[i-1];
            dp[i] = max(dp[i], dp[i-2] + v[i-2]*v[i-1]);
            if (i >= 3) {
                dp[i] = max(dp[i], dp[i-3] + v[i-3]*v[i-2]*v[i-1]);
            }
        }
        cout << dp[n] << endl;
    }
}
```

Min Cost Climbing Stairs

Solución

Tenemos escalones del 0 al $n-1$ con sus respectivos costes. El objetivo es alcanzar el escalón n-ésimo (que nos hemos inventado). Definimos $dp[i]$ como el coste mínimo para alcanzar el i-ésimo escalón. Solo podemos haber llegado desde alguno de los dos escalones anteriores, y el coste desde otro escalón es = coste para llegar a ese escalón ($dp[j]$) + el coste para pasar de ese al actual ($cost[j]$).
Con esto vemos que $dp[i] = \min(dp[i-1] + cost[i-1],\ dp[i-2] + cost[i-2])$.

Código

```
int minCostClimbingStairs(vector<int>& cost) {
    int n = cost.size();
    vector<int> dp(n+1);
    dp[0] = 0;
    dp[1] = 0;
    for (int i = 2; i <= n; ++i) {
        dp[i] = min(dp[i-1] + cost[i-1], dp[i-2] + cost[i-2]);
    }
    return dp[n];
}
```

Divisor Game

Solución

Definimos $\text{win}(k)$ como la función que nos dice si un número es ganador ($\text{win}(k) = 1$) o perdedor ($\text{win}(k) = 0$) en este juego.
Es fácil que si hay algún divisor d de k tal que $\text{win}(k-d) = 0$, entonces $\text{win}(k) = 1$ ya que desde el número k mandamos a nuestro oponente a la posición $k-d$, y como es una posición perdedora el oponente pierde y nosotros ganamos.
Una posición será perdedora entonces si no podemos llevar al oponente a ninguna posición perdedora, es decir, si para todo divisor d de k, $\text{win}(k-d) = 1$.
Esta solución recursiva tendría coste exponencial implementada directamente, pero podemos implementarla y luego añadir memorización/DP, por lo que una vez calculado $\text{win}(k)$ ya no lo recalculamos más sino que devolvemos el valor guardado. Haciendo esto tenemos únicamente n posibles valores de k, y por tanto, el coste será $\mathcal{O}(n \times \{\text{calcular un win}(k)\})$. Como tenemos que simplemente probar los divisores y podemos hacer esto en $\mathcal{O}(\sqrt{n})$, el coste final es $\mathcal{O}(n\sqrt{n})$.

Código

```cpp
vector<int> dp;

int win(int k) {
    if (k == 0) return 0;

    int& ans = dp[k];
    if (ans != -1) return ans;

    ans = 0;
    for (int x = 1; x*x <= k; ++x) {
        if (k % x == 0) {
            if (x < k and not win(k-x)) ans = 1; // d = x
            if (k/x < k and not win(k-k/x)) ans = 1; // d = k/x
        }
    }
    return ans;
}

bool divisorGame(int n) {
    dp = vector<int>(n+1, -1);
    return win(n);
}
```

2 X 1

Solución

Es óptimo separar los platos en dos grupos, de forma que el precio total de un grupo sea lo más cercano al otro. Esto a su vez es equivalente a encontrar el subconjunto de los platos más cercano a la mitad del precio de todos los platos juntos (ya que minimizará la diferencia entre el conjunto y los restantes).
Definimos $dp[i]$ como 1 si se puede conseguir un conjunto de suma i y 0 si no se puede.
Inicialmente consideraremos que solo hay un plato y calcularemos todos los $dp[i]$.
Luego añadiremos el segundo plato recalculando los $dp[i]$, y así sucesivamente.
Cuando podemos conseguir una cierta suma j, (menor que total/2), significa que podríamos conseguir un precio de total-j. Por lo que iremos probando todos los j posibles y nos quedamos con el que nos de menor precio (que será el más cercano a la mitad).

Código

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

using vi = vector<int>;

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int n, k;
    while (cin >> n >> k) {
        int sum = 0;
        vi a(k);
        for (int i = 0; i < k; ++i) {
            cin >> a[i];
            sum += a[i];
        }

        if (n == 1) {
            cout << sum << endl;
        }
        else {
            int hsum = sum / 2;

            vi dp(hsum+1, false);
            dp[0] = true;

            int ans = sum;

            for (int i = 0; i < k; ++i) {
                for (int j = hsum; j >= a[i]; --j) {
                    if (dp[j-a[i]]) {
                        dp[j] = true;
                        ans = min(ans, sum - j);
                    }
                }
            }

            cout << ans << endl;
        }
    }
}
```

Omgillas

Solución

El valor máximo que nos pueden pedir es 29×20. En otro caso la respuesta es 0. Definimos $dp[k][i]$ como el número de formas de obtener el valor i usando k de las monedas disponibles. Para calcularlo primero asumimos que no tenemos monedas, entonces $dp[0][0] = 1$ ya que podemos conseguir el valor 0 con 0 monedas de una forma (sin usar monedas) y el resto todo a 0. A partir de aquí iremos actualizando dp añadiendo nuevas monedas que se pueden usar. Si añadimos una moneda nueva, entonces tenemos tantas formas como había sin esta moneda más las formas habiéndola usado $dp'[k][i] = dp[k][i] + dp[k-1][i-coin]$ (teniendo en cuenta que $dp[k-1][i-coin]$ se haya calculado teniendo en cuenta que ya se puede usar la moneda nueva, ya que se pueden usar múltiples veces y por eso es importante el orden de los bucles en este problema).
Es importante ver que para todos los casos la DP es la misma por lo que es mejor calcular para todos los valores antes de empezar y luego simplemente imprimir valores de la tabla.
El coste entonces sería $\mathcal{O}(\text{monedas} \times \text{valor máximo} \times C)$, que según el enunciado serían como mucho 10, 20×29 y 20, respectivamente.

Código

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

using vi = vector<int>;
using vvi = vector<vi>;

int main() {
    const vi coins = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
    const int C = 20, Val = 29*20;
    vvi dp(C+1, vi(Val+1, 0));
    dp[0][0] = 1;
    for (int coin : coins) {
        for (int k = 1; k <= C; ++k) {
            for (int i = 0; i <= Val; ++i) {
                if (i-coin >= 0) dp[k][i] += dp[k-1][i-coin];
            }
        }
    }

    int n, c, v;
    cin >> n;
    for (int t = 0; t < n; ++t) {
        cin >> c >> v;
        if (v > Val) cout << 0 << endl;
        else cout << dp[c][v] << endl;
    }
}
```

Counting cool words

Solución

Definimos $dp1[n][v]$ como el número de palabras de tamaño n usando v vocales que **acaban en vocal** y $dp2[n][v]$ como el número palabras de tamaño n usando v vocales que **acaban en consonante**.
Si tenemos cualquier palabra de longitud $n-1$ y $v-1$ vocales (que acabe tanto en vocal como consonante) y le añadimos una vocal, tenemos 5 formas de crear una palabra de longitud n con v vocales que acaba en vocal, por lo que $dp1[n][v] = 5 \times (dp1[n-1][v-1] + dp2[n-1][v-1])$.
De manera similar, y como no pueden haber dos consonantes seguidas, la única forma de crear una palabra de tamaño n con v vocales que acabe en consonante es coger una palabra de tamaño $n-1$ y v vocales para añadirle una consonante al final, y como tenemos 21 opciones para esta consonante, tenemos que $dp2[n][v] = 21 \times (dp1[n-1][v])$.

Código

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

using ll = long long;
using vl = vector<ll>;
using vvl = vector<vl>;

const int MAXN = 15;
const int V = 5; // numero de vocales
const int C = 21; // numero de consonantes

// dp1[n][v]: palabras de tamaño n usando v vocales que acaban en vocal
vvl dp1(MAXN+1, vl(MAXN+1));
// dp2[n][v]: palabras de tamaño n usando v vocales que acaban en consonante
vvl dp2(MAXN+1, vl(MAXN+1));

int main() {
    dp1[1][1] = V;
    dp2[1][0] = C;
    for (int n = 2; n <= MAXN; ++n) {
        for (int v = 1; v <= n; ++v) {
            dp1[n][v] = V*(dp1[n-1][v-1] + dp2[n-1][v-1]);
            dp2[n][v] = C*(dp1[n-1][v]);
        }
    }

    int n, v;
    while (cin >> n >> v) {
        cout << dp1[n][v] + dp2[n][v] << endl;
    }
}
```

Chained pawns

Solución

En un tablero de r filas y c columnas, podemos definir $dp[k][i][j]$ como el número de formas de resolver el problema usando exactamente k peones, con el último peón en la posición (i, j).
El número de formas de resolver el problema con k peones acabando en la posición (i, j) se puede calcular recursivamente ya que esa solución proviene de una solución que usa un peón menos y acaba arriba a la izquierda $(i-1, j-1)$ o arriba a la derecha $(i-1, j+1)$, es decir,
$dp[k][i][j] = dp[k-1][i-1][j-1] + dp[k-1][i-1][j+1]$.
La solución final es la suma de $dp[r][r-1][j]$ para todo j ya que usamos un peón por fila y podemos acabar en cualquier posición de la última fila.

Para este problema resulta más eficiente precalcular las soluciones a todos los posibles inputs y así no recalculamos dos veces para un mismo input. Además usamos siempre el mismo vector dp en lugar de crear uno nuevo ya que así ahorramos tiempo de ejecución (el tiempo está bastante ajustado en jutge).

Código

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

using ll = long long;
using vl = vector<ll>;
using vvl = vector<vl>;
using vvvl = vector<vvl>;

int main() {
    vvl sol(51, vl(51));
    vvvl dp(51, vvl(50, vl(50)));

    for (int r = 1; r <= 50; ++r) {
        for (int c = 1; c <= 50; ++c) {
            // resolvemos para todos los posibles inputs (r, c)
            // y almacenamos las soluciones en sol[r][c]
            for (int i = 0; i < r; ++i)
                for (int j = 0; j < c; ++j)
                    dp[1][i][j] = 1; // caso base, 1 peon en cualquier posición
            for (int k = 2; k <= r; ++k) {
                for (int i = 1; i < r; ++i) {
                    for (int j = 0; j < c; ++j) {
                        dp[k][i][j] = 0; // reseteamos valores previos a 0
                        if (j-1 >= 0) dp[k][i][j] += dp[k-1][i-1][j-1];
                        if (j+1 <  c) dp[k][i][j] += dp[k-1][i-1][j+1];
                    }
                }
            }
            for (int j = 0; j < c; ++j)
                sol[r][c] += dp[r][r-1][j];
        }
    }

    int r, c;
    while (cin >> r >> c) {
        cout << sol[r][c] << endl;
    }
}
```

Firefighters and grannies (2)

Solución

Definimos $f(i, g)$ como el máximo número de ancianas que podemos salvar con g grupos de bomberos en las escuelas $0, 1, \ldots, i$. Vemos que solo hay dos opciones: pones un grupo en la escuela i o no ponerlo, y resolver recursivamente.
Obervamos que si ponemos un grupo de bomberos en la posición i entonces nunca lo ponemos en la posición $i-1$, ya que poniéndolo en la $i-2$ siempre sería mejor (como cubren la mitad de los adyacentes, entre i y $i-2$ ya cubren completamente $i-1$).
Vigilamos los casos base y aplicamos DP a la función f para conseguir un coste $\mathcal{O}(ng)$ por caso.

Remarcar que en el código he añadido dos escuelas "fantasma" con 0 personas, una al principio y otra al final, para simplificar la cantidad de casos que hay que considerar.

Código

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

using vi = vector<int>;
using vvi = vector<vi>;

vi v;
vvi dp;

int f(int i, int g) {
    if (g == 0) return 0;
    if (i < 0) return 0;
    if (i == 0) return v[0] + v[1]/2;
    int& ans = dp[i][g];
    if (ans == -1)
        ans = max(f(i-1, g), f(i-2, g-1) + v[i-1]/2 + v[i] + v[i+1]/2);
    return ans;
}

int main() {
    int g, n;
    while (cin >> g >> n) {
        v = vi(n+2, 0);
        for (int i = 0; i < n; ++i) cin >> v[1+i];
        dp = vvi(n+1, vi(g+1, -1));
        cout << f(n, g) << endl;
    }
}
```

Writing parentheses

Solución

Si n es impar no hay solución ya que necesitamos la misma cantidad de paréntesis abiertos que cerrados para que sea una secuencia correcta. Cuando n es par, la solución es el número de Catalan $n/2$ (es decir, $C_{n/2}$ sabiendo que $C_k = C_0C_{k-1} + C_1C_{k-2} + \cdots + C_{k-1}C_0$ y $C_0 = 1$).

Esto se puede explicar definiendo A_n como el número de secuencias correctas con n paréntesis abiertos y cerrados. Nos fijamos en que toda secuencia correcta se puede ver como: $(X)Y$ donde X e Y son a su vez secuencias correctas.
Por ejemplo, para `(())((()))` tenemos $X =$ `()`, $Y =$ `((()))`.
Vemos que para `((()))` tenemos $X' =$ `(())`, $Y' =$ nada.
y como caso base, para `()` tenemos $X'' =$ nada, $Y'' =$ nada.
Es decir, toda secuencia se puede ir descomponiendo recursivamente de esta forma, por lo que todas se pueden generar mediante el proceso inverso, cogiendo dos correctas y colocando la primera entre paréntesis.
Si las secuencias originales tienen tamaño $2n$ y $2m$, nos dan una de tamaño $2n + 2 + 2m$, usando justamente $n + m + 1$ paréntesis abiertos y cerrados.
Vamos a calcular A_n suponiendo que ya sabemos $A_1, \ldots, A_{n-1}$:
Primero, podemos formar $()Y$ siendo Y una secuencia correcta que use $n - 1$ abiertos y cerrados, por lo que tenemos A_{n-1} formas de elegir Y.
Otra opción es $(X)Y$ siendo X una secuencia correcta que use 1 abierto y cerrado, y Y una secuencia correcta que use $n - 2$ abiertos y cerrados, por lo que tenemos $A_1 \times A_{n-2}$ opciones.
Otra opción es $(X)Y$ siendo X una secuencia correcta que use 2 abiertos y cerrados, y Y una secuencia correcta que use $n - 3$ abiertos y cerrados, por lo que tenemos $A_2 \times A_{n-3}$ opciones.
Etc...
Así vemos que $A_n = A_{n-1} + A_1A_{n-2} + \cdots + A_{n-2}A_1 + A_{n-1} + A_{n-1}$. Vemos que si definimos $A_0 = 1$ (esto es algo arbitario ya que el caso $n = 0$ no tiene mucho sentido) podemos expresarlo astutamente como $A_n = A_0A_{n-1} + A_1A_{n-2} + \cdots + A_{n-2}A_1 + A_{n-1} + A_{n-1}A_0$, lo que demuestra que $A_n = C_n$, ya que tienen la misma expresión y el mismo caso base.

Código

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

using ll = long long;
using vl = vector<ll>;

int main() {
    vl c(34, 0);
    c[0] = c[1] = 1;
    for (int i = 2; i < 34; ++i)
        for (int j = 0; j <= i-1; ++j)
            c[i] += c[j] * c[i-1-j];
    int n;
    while (cin >> n) {
        if (n%2 == 1)
            cout << 0 << endl;
        else
            cout << c[n/2] << endl;
    }
}
```

Bags and boxes

Solución

Definimos $C[i][j] = \binom{i}{j}$ (el número de formas de elegir j elementos de un conjunto de i). Inicialmente tenemos n objetos distintos. Para las y cajas tenemos $C[n][y]$ formas de colocar y objetos en ellas.
Para cada una de estas formas nos quedan $n - y$ objetos que debemos repartir en las x bolsas de forma que todas tengan al menos un objeto.
Si definimos $S[i][j]$ como el número de formas de repartir i objetos en j bolsas, tenemos entonces que la solución al problema es: $C[n][y] \times S[n-y][x]$.

El problema ahora consiste en calcular tanto $C[i][j]$ como $S[i][j]$.

Para $C[i][j]$ podemos observar que podemos poner o no el último elemento, si lo ponemos nos quedan $i-1$ y tenemos que coger otros $j-1$ ($C[i-1][j-1]$ formas), si no lo ponemos nos quedan $i-1$ elementos y tenemos que coger j ($C[i-1][j]$ formas). Por lo que $C[i][j] = C[i-1][j-1] + C[i-1][j]$.
Los casos base son coger todos o nada, que solo hay una forma ($C[i][0] = C[i][i] = 1$ para toda i).

Para $S[i][j]$ podemos observar que el último elemento va a ir en una bolsa él solo, o si lo añadimos a una bolsa de algún otro elemento.
Si va solo, lo ponemos en una bolsa él solo y nos quedan $i-1$ elementos para colocar en $j-1$ bolsas (hay $S[i-1][j-1]$ formas).
Si no va solo, resolvemos el problema con $i-1$ elementos y j bolsas ($S[i-1][j-1]$ formas), y luego lo añadimos a una de las j bolsas, por lo que tendríamos $S[i-1][j-1] \times j$ formas de hacerlo.
Sumando las distintas formas, nos queda que $S[i][j] = S[i-1][j] \times j + S[i-1][j-1]$.
Los casos base se pueden establecer de diferentes formas, la más simple es considerar que $S[0][0] = 1$, como que la única forma de colocar 0 elementos en 0 bolsas es una (no hacer nada, una forma de pensar casos base muy habitual y que suele ser útil).

Por si alguna está interesada: los números $C[i][j] = \binom{i}{j}$ se conocen como coeficientes binomiales, y los números $S[i][j]$ se conocen como números de Stirling de segunda especie.

Código

```cpp
#include <bits/stdc++.h>
using namespace std;  using vl = vector<long long>;  using vvl = vector<vl>;

int main() {
    vvl C(26, vl(26)), S(26, vl(26));
    for (int i = 0; i <= 25; ++i)
        C[i][0] = C[i][i] = 1;
    for (int i = 2; i <= 25; ++i)
        for (int j = 1; j < i; ++j)
            C[i][j] = C[i-1][j-1] + C[i-1][j];
    S[0][0] = 1;
    for (int i = 1; i <= 25; ++i)
        for (int j = 1; j <= i; ++j)
            S[i][j] = S[i-1][j] * j + S[i-1][j-1];
    int n, x, y;
    while (cin >> n >> x >> y) {
        if (n < x + y) cout << 0 << endl;
        else cout << C[n][y] * S[n-y][x] << endl;
    }
}
```

Multiplicity

Solución

Si definimos $dp[i][j]$ como las formas de hacer subsecuencias de tamaño j usando los primeros i elementos, podemos ver que cuando $a[i-1]$ no es divisible entre j solo podemos usar los elementos anteriores y $dp[i][j] = dp[i-1][j]$. Cuando $a[i-1]$ si que es divisible entre j, entonces también podemos cogerlo y añadirlo a cualquier subsecuencia previa de tamaño $j-1$, por lo que nos queda que $dp[i][j] = dp[i-1][j] + (dp[i-1][j-1]$ solo si $a[i-1]\%j == 0)$. Esto requiere de tiempo $\mathcal{O}(n^2)$.
Se puede mejorar esta solución teniendo en cuenta dos cosas:

- Para calcular $dp[i][\dots]$ siempre usamos valores de $dp[i-1][\dots]$, por lo que podríamos usar un vector de una sola dimensión $dp[\dots]$ que se va actualizando: $dp[j] = dp[j] + (dp[j-1]$ si $a[i-1]\%j == 0)$.
- Solo se modifican los elementos cuando $a[i-1]\%j == 0$, por lo que solo tenemos que iterar por j's que sean divisores de $a[i-1]$, y podemos pasar por los divisores de $a[i-1]$ en $\mathcal{O}(\sqrt{a[i-1]})$.

Con todo esto, podemos hacer una solución con tiempo $\mathcal{O}(n\sqrt{\max a})$, que teniendo en cuenta las restricciones del problema es un tiempo aceptable. Finalmente, ya que $dp[j]$ es la cantidad de subsecuencias de tamaño j, el resultado es la suma para toda j de 1 a n .

Código

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

int main() {
    int n;
    cin >> n;
    vi a(n);
    for (int i = 0; i < n; ++i)
        cin >> a[i];

    const int MOD = 1e9+7;
    vi dp(n+1);
    dp[0] = 1;
    for (int i = 1; i <= n; ++i) {
        set<int> divisores;
        for (int k = 1; k*k <= a[i-1]; ++k) {
            if (a[i-1] % k == 0) {
                divisores.insert(k);
                divisores.insert(a[i-1]/k);
            }
        }
        // iteramos de grandes a pequeños para no pisar los resultados
        for (auto it = divisores.rbegin(); it != divisores.rend(); ++it) {
            int j = *it;
            if (j <= n) dp[j] = (dp[j] + dp[j-1]) % MOD;
        }
    }
    int ans = 0;
    for (int j = 1; j <= n; ++j)
        ans = (ans + dp[j]) % MOD;
    cout << ans << endl;
}
```

Counting trees

Solución

Notación:

- $\lceil x/2 \rceil$ denota el redondeo hacia arriba de x/2, y en código se calcula así: `(x+1)/2`
- $i|n$ suele denotar que i es un divisor de n

Definimos un árbol especial como aquel que cumple:

- Todos los hijos de un nodo tienen el mismo tamaño.
- Los tamaños de todos los subárboles son números primos.
- Todos los subárboles son simétricos.

Definimos $f(n)$ como el número de árboles especiales de n nodos.
Como caso base tenemos que $f(1) = f(2) = 1$. Vamos a suponer que sabemos calcular $f(k)$ cuando $k < n$ para intentar calcular $f(n)$ recursivamente.
Si tenemos n nodos, tenemos uno que es la raíz, y nos quedan $n-1$ nodos que distribuir entre los hijos. Como los tamaños tienen que ser iguales, solo podemos colocar un número de hijos que divida $n-1$ para poder repartir en grupos del mismo tamaño. Además este tamaño tiene que ser un número primo (lo mejor es al principio del programa usar la criba de Eratóstenes hasta la n máxima y crear una tabla que nos indique que números son o no primos).

Digamos que hacemos n/i grupos de tamaño i (i divisor de n). Entonces cada grupo tiene que formar un subárbol especular de i nodos, y sabemos que esto se puede hacer de $f(i)$ formas.
La condición de simetría impone que un lado decidos como son los $\lceil (n/i)/2 \rceil$ nodos de la izquierda al centro, los de la derecha son la copia simétrica, por lo que solo hay que decidir cómo son los de este lado. Si tenemos que decidir g grupos y cada uno tiene $f(i)$ formas, tenemos un total de $f(i)^g$ combinaciones. En nuestro caso sería $f(i)^{\lceil (n/i)/2 \rceil}$.

Esto únicamente para un divisor i. El resultado final de $f(n)$ sería la suma de esto para todos sus divisores, es decir, $f(n) = \sum_{i|n} f(i)^{\lceil (n/i)/2 \rceil}$

Para calcular $f(n)$ haremos una DP, en la que hay n posibles estados.
Cada una se caula pasando por los divisores en $\mathcal{O}(\sqrt{n})$ y calculando $f(i)$ elevado a un exponente del orden de n, que usando exponenciación rápida se consigue en $\mathcal{O}(\log n)$ por lo que cada $f(n)$ requiere de tiempo $\mathcal{O}(\sqrt{n} \log n)$ y en total tenemos un algoritmo que se ejecuta en tiempo $\mathcal{O}(n\sqrt{n} \log n)$, que teniendo en cuenta el tamaño del input es eficiente.

Código

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

using ll = long long;

vector<ll> dp;
vector<bool> prime;

ll ipow(ll x, int n) { // calcula x^n usando exponenciacion rápida O(log n)
    if (n == 0) return 1;
    if (n % 2 == 0) return ipow(x*x, n/2);
    return x * ipow(x*x, (n-1)/2);
}

ll f(int n) {
    if (n == 1 or n == 2) return 1;
    if (dp[n] == -1) {
        int k = n-1;
        ll ans = 0;
        for (int i = 1; i*i <= k; ++i) {
            if (k % i == 0) {
                int j = k/i;
                if (prime[i])
                    ans += ipow(f(i), (j+1)/2); // f(i)^floor((n/i)/2), i divisor de n
                if (j != i and prime[j])
                    ans += ipow(f(j), (i+1)/2); // f(j)^floor((n/j)/2), j divisor de n
            }
        }
        dp[n] = ans;
    }
    return dp[n];
}

int main() {
    prime.assign(501, true);
    prime[0] = prime[1] = false;
    for (int i = 2; i <= 500; ++i) {
        if (prime[i]) {
            for (int j = i*i; j <= 500; j += i) {
                prime[j] = false;
            }
        }
    }
    dp.assign(501, -1);
    int n;
    while (cin >> n) {
        cout << f(n) << endl;
    }
}
```