Programación Dinámica

Mariano Crosetti

En conjunto con Pablo Zimmermann

Buenos Aires, Argentina
Universidad de Buenos Aires

Training Camp 2018

Contenidos I

Contenidos

Sucesión de Fibonacci

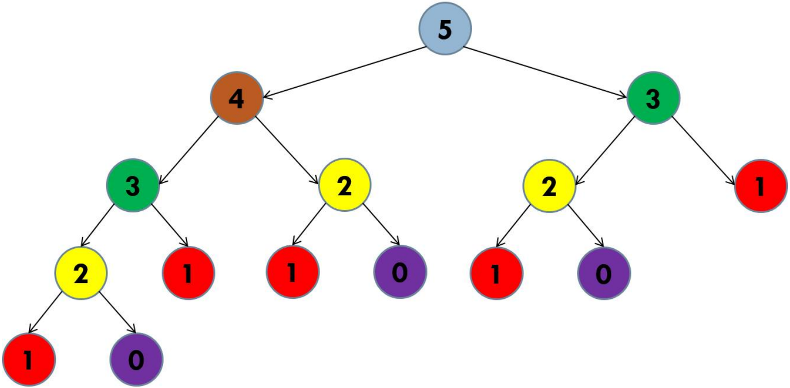
Sucesión de Fibonacci

La sucesión de Fibonacci se define como $f_0 = 1$, $f_1 = 1$ y $f_{n+2} = f_n + f_{n+1}$ para todo $n \geq 0$

- ¿Cómo podemos computar el término 100 de la sucesión de Fibonacci?

```
int fib(int n)
{
    if(n<=1)
        return 1;
    else
        return fib(n-2)+fib(n-1);
}
```

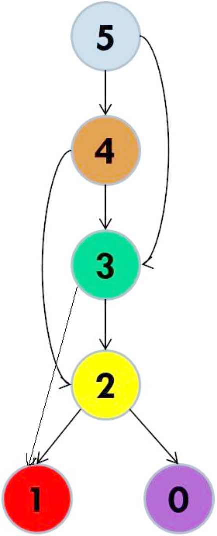
Explicación gráfica de lo que hace el Algoritmo



- Llamamos muchas veces a la misma función con los mismos parámetros
- Tenemos “problemas de memoria”, pues volvemos a calcular algo que ya hemos calculado previamente.

Ahora con Programación Dinámica

```
int fib[100];
int calcFib(int n)
{
    if(fib[n]!=-1)
        return fib[n];
    fib[n] = calcFib(n-2)+calcFib(
        n-1);
    return fib[n];
}
int main()
{
    for(int i=0;i<100;i++)
        fib[i] = -1;
    fib[0] = 1;
    fib[1] = 1;
    int fib50 = calcFib(50);
}
```



Contenidos

Viaje óptimo en matriz

Enunciado

Dada una matriz de $n \times m$, con números positivos, y queremos encontrar el camino de la esquina **superior-izquierda**, a la esquina **inferior-derecha**, que minimize la suma de las casillas recorridas. El camino está restringido a utilizar únicamente movimientos de tipo **derecha** y **abajo**.

Viaje óptimo en matriz

1	7	9	2
8	6	3	2
1	6	7	8
2	9	8	2

Minimum Cost Path: 29

Viaje óptimo en matriz

Enunciado

Dada una matriz de $n \times m$, con números enteros, y queremos encontrar el camino de la esquina **superior-izquierda**, a la esquina **inferior-derecha**, que minimize la suma de las casillas recorridas. El camino está restringido a utilizar únicamente movimientos de tipo **derecha** y **abajo**.

¿Podríamos encontrar una función que devuelva el camino mínimo de una posición a la esquina inferior derecha?

Viaje óptimo en matriz

Enunciado

Dada una matriz de $n \times m$, con números enteros, y queremos encontrar el camino de la esquina **superior-izquierda**, a la esquina **inferior-derecha**, que minimize la suma de las casillas recorridas. El camino está restringido a utilizar únicamente movimientos de tipo **derecha** y **abajo**.

¿Podríamos encontrar una función que devuelva el camino mínimo de una posición a la esquina inferior derecha?

- $f(N-1, M-1) = T_{N-1,M-1}$
- $f(i, M-1) = T_{i,M-1} + f(i+1, M-1))$
- $f(N-1, j) = T_{N-1,j} + f(N-1, j+1))$
- $f(i, j) = T_{i,j} + min(f(i+1, j), f(i, j+1))$

Llevándolo a Programación Dinámica

```
int tab[1010][1010], M, N, dp[1010][1010];
int f(int i, int j)
{
    int &r = dp[i][j];
    if (r != -1) return r;
    if (i == M - 1 && j == N - 1) return r = T[M - 1][N - 1];
    r = INF;
    if (i < M - 1) r = min(r, T[i][j] + f(i+1,j));
    if (j < N - 1) r = min(r, T[i][j] + f(i,j+1));
    return r;
}
...
    memset(dp,-1,sizeof(dp));
    cout << f(0,0) << endl;
```

Para los que no se llevan con la recursión...

```
for (int i = M - 1 ; i >= 0 ; i--) {
    for (int j = N - 1 ; j >= 0 ; j--) {
        int &r = dp[i][j];
        if (i == M - 1 && j == N - 1) r = T[M - 1][N - 1];
        else {
            r = INF;
            if (i < M - 1) r = min(r, T[i][j] + dp[i+1][j]);
            if (j < N - 1) r = min(r, T[i][j] + dp[i][j+1]);
        }
    }
}
cout << dp[0][0] << endl;
```

Top-Down vs Bottom-Up

La primera versión se conoce con el nombre de Top-Down, mientras que la segunda se le dice Bottom-Up.

- Top-Down es una recursión con memoria (se la llama memorización también).
- Top-Down es más fácil de escribir a partir de una función matemática recursiva.
- Bottom-Up construye la solución partiendo de los casos bases "hacia arriba"
- Bottom-Up es más rápida (la recursión tiene costes de tiempo y memoria) si se utilizan la mayoría de las entradas de la tabla.
- Top-Down es mejor en casos que hay muchos estados no visitados.
- Podemos pasar de Top-Down a Bottom-Up copiando y pegando el contenido de la función recursiva y recorriendo los estados de algún modo que nos asegure tener los subproblemas calculados.

Contenidos

Reconstruyendo cualquier cosa!

Enunciado

Reconstruir el camino como una cadena de ”A” (Abajo) y ”D” (Derecha).

```
int tab[1010][1010], M, N, dp[1010][1010];
int f(int i, int j)
{
    int &r = dp[i][j];
    if (r != -1) return r;
    if (i == M - 1 && j == N - 1) return r = T[M - 1][N - 1];
    r = INF;
    if (i < M - 1) r = min(r, T[i][j] + f(i+1,j));
    if (j < N - 1) r = min(r, T[i][j] + f(i,j+1));
    return r;
}
```

Reconstruyendo cualquier cosa!

```
string reconstruccion;
void bt(int i, int j)
{
    if (i == M - 1 && j == N - 1) return;
    if (i < M - 1 && f(i,j) == T[i][j] + f(i+1,j)) {
        reconstruccion.push_back('A');
        bt(i+1,j);
        return;
    }
    if (j < N - 1 && f(i,j) == T[i][j] + f(i,j+1)) {
        reconstruccion.push_back('D');
        bt(i,j+1);
        return;
    }
}
...
```

Receta para reconstruir

Podemos resolver los problemas que requieren reconstruir solución con la siguiente receta:

- Realizar la formulación matemática y programar la recursión.
- Agregar memorization. Tenemos una Top-Down!
- Hacer un backtracking copiando el cuerpo de la función y en cada transición chequear si es óptima, hacer la transición.
- No se olviden los return! no queremos recorrer todos los caminos!!

Contenidos

Cantidad de caminos mínimos

Enunciado

Devolver la cantidad de caminos mínimos para el problem anterior.

```
int dp2[1010][1010];
int cantMinimos(int i, int j)
{
    int &r = dp2[i][j];
    if (r != -1) return r;
    if (i == M - 1 && j == N - 1) return r = 1;
    r = 0;
    if (i < M - 1 && f(i,j) == T[i][j] + f(i+1,j))
        r += cantMinimos(i+1,j);
    if (j < N - 1 && f(i,j) == T[i][j] + f(i,j+1))
        r += cantMinimos(i,j+1);
    return r;
}
```

Receta cantidad respuestas óptimas + Yapa

- Hacer otra función recursiva que devuelva la cantidad de respuestas óptimas desde un estado.
- Utilizar la función anterior para saber si una transición es óptima.

Enunciado

De todos los caminos mínimos, devolver el k-ésimo lexicográfico (considerados como string de "A"' y "D").

- Modificando sencillamante el backtracking podemos resolver el problema anterior.
- El orden lexicográfico nos dice que nos podemos inclinar de forma greedy por una transición.

K-ésimo camino mínimos

```
void bt(int i, int j, int k) {
    if (i == M - 1 && j == N - 1) return;
    if (i < M - 1 && f(i,j) == T[i][j] + f(i+1,j)) {
        if (k<cantMinimos(i+1,j)) {
            reconstruccion.push_back('A');
            bt(i+1,j,k);
            return;
        } else {
            k -= cantMinimos(i+1,j);
        }
    }
    if (j < N - 1 && f(i,j) == T[i][j] + f(i,j+1)) {
        reconstruccion.push_back('D');
        f2(i,j+1,k);
    }
}
```

Contenidos

Rompiendo el Memory Limit

Escribí la solución pero me da Memory Limit. Es necesario guardar todos los estados?

```
dp[(M-1)%2][N-1] = T[M-1][N-1];
for (int i = M-1; i >= 0; i--) {
    for (int j = N-1; j >= 0; j--) {
        int &r = dp[i%2][j];
      if (i == M-1 && j == N-1) r = T[M-1][N-1];
        else {
            r = INF;
            if (i < M-1) r = min(r, T[i][j] + dp[(i+1)%2][j]);
            if (j < N-1) r = min(r, T[i][j] + dp[i%2][j+1]);
        }
    }
}
cout << dp[0%2][0] << endl;
```

Rompiendo el Memory Limit - Receta

Es común tener que optimizar memoria utilizando este truco de la "tira" que se sobreescribe.

- Es muy facil adaptar la solución Bottom-Up agregando %.
- Se generaliza a tiras de mayor anchura.
- Hay que tener cuidado que el parámetro que estemos sobreescribiendo sea el que se recorre en la iteración de menor anidación.
- Siempre es bueno hacer un dibujo e imaginarnos el orden en el cuál estamos llenando la tabla.
- No olvidarse usar % cuando extraemos el resultado.

Contenidos

Agregando una flag - reduciendo complejidad

Enunciado

Resolver el problema anterior pero podemos hacer K movimientos de alfil, pagando el costo de cada casilla que pasamos.

```
int f(int i, int j, int k) {
    int &r = dp[i][j]; if (r != -1) return r;
    if (i == M - 1 && j == N - 1) return r = T[M - 1][N - 1];
    r = INF;
    if (i < M - 1) r = min(r, T[i][j] + f(i+1,j));
    if (j < N - 1) r = min(r, T[i][j] + f(i,j+1));
    if (k>0) {
        int sum = T[i][j];
        for(int d = 1 ; d + i < M && d + j < N ; d ++) {
            r = min (r, f(i + d, j + d, k-1) + sum);
            sum += T[i + d][j + d];
        }
    }
    return r;
}
```

Agregando una flag - reduciendo complejidad

Enunciado

Resolver el problema anterior pero podemos hacer K movimientos de alfil, pagando el costo de cada casilla que pasamos.

```
int f(int i, int j, int k) {
    int &r = dp[i][j]; if (r != -1) return r;
    if (i == M - 1 && j == N - 1) return r = T[M - 1][N - 1];
    r = INF;
    if (i < M - 1) r = min(r, T[i][j] + f(i+1,j));
    if (j < N - 1) r = min(r, T[i][j] + f(i,j+1));
    if (k>0) {
        int sum = T[i][j];
        for(int d = 1 ; d + i < M && d + j < N ; d ++) {
            r = min (r, f(i + d, j + d, k-1) + sum);
            sum += T[i + d][j + d];
        }
    }
    return r;
}
```

Agregando una flag - reduciendo complejidad

```
int f(int i, int j, int k, int b) {
    int &r = dp[i][j]; if (r != -1) return r;
    if (i == M - 1 && j == N - 1) return r = T[M - 1][N - 1];
    r = INF;
    if (i < M - 1) r = min(r, T[i][j] + f(i+1, j, k, 0));
    if (j < N - 1) r = min(r, T[i][j] + f(i, j+1, k, 0));
    if (i < M - 1 && j < N - 1 && k>0) {
        r = min(r, T[i][j] + f(i+1, j+1, k-1, 1));
    }
    if (i < M - 1 && j < N - 1 && b) {
        r = min(r, T[i][j] + f(i+1, j+1, k, 1));
    }
    return r;
}
```

Contenidos

Recuperando un parámetro

Enunciado

Dada una secuencia de enteros, se los quiere particionar en dos subsecuencias minimizando la suma de los cuadrados de las diferencias de elementos consecutivos de cada subsecuencia.
Ejemplo:
27 **2** 30 **1** **2** 31 = $(27 - 30)^2 + (30 - 31)^2 + (2 - 1)^2 + (1 - 2)^2 = 12$

- Podriamos tener una función recursiva: $f_{ultimoRojo,ultimoAzul,posicionActual} = \ldots$
- Obsesrvar que $max(ultimoRojo, ultimoAzul) = posicionActual - 1$
- Una dinámica de 3 estados tendría estados que no visitaríamos.
- Si la planteamos Top-Down sólo estaríamos desperdiciando memoria (no tiempo).
- ¿Se puede evitar?

Recuperando un parámetro

```
int f(int uR, int uA) {
    int &r = dp[uR+1][uA+1]; // offset para permitir
        parametros < 0
    if (r != -1) return r;
    int i = max(uR,uA)+1;
    if (i == N) return r = 0;
    r = INF;
    r = min(r, f(i,uA) + uR == -1 ? 0 : (v[uR]-v[i])**2 );
    r = min(r, f(uR,i) + uA == -1 ? 0 : (v[uA]-v[i])**2 );
    return r;
}
// La respuesta buscada es f(-1,-1)
```

Contenidos

Secuencias parenteseadas

Enunciado

Dada una cadena de caracteres {, }, [,] , (y) de longitud par, dar la mínima cantidad de reemplazos de caracteres que se le deben realizar a este string para dejar una secuencia "bien parenteseada".

T es bien parenteseada si es de la forma:

- $T = \emptyset$
- $T = S_1 S_2$
- $T = (S)$
- $T = [S]$
- $T = \{S\}$

Con S, S_1, S_2 bien parenteseada.

Dinámica de rangos

- El estado es resolver el problema para los subrangos.
- Las transiciones generalmente implican reducir los extremos o partir el subrango en dos (o más subrangos).
- Suele tener ventajas trabajar con rangos [a,b)

Secuencias parenteseadas

```
// f(a,b) = respuesta para S[a,b)
int f(int a, int b) {
    int &r = dp[a][b];
    if (r != -1) return r;
    if (b - a <= 0) return r=0;
    if (b - a == 1) return r=INF;
    r = f(a+1,b-1) + costo_matchear(S[a], S[b-1]);
    for(int i = a + 1 ; i < b ; i++) {
        r = min(r, f(a,i) + f(i,b));
    }
    return r;
}
```

Secuencias parenteseadas - Bottom Up

```
for (int a = 0 ; a < N ; a++) {
    for (int b=a ; b < N ; b++) {
        int &r = dp[a][b];
        if (b - a <= 0) r=0;
        else if (b - a == 1) r=INF;
        else {
            r = f(a+1,b-1) + costo_matchear(S[a], S[b-1]);
            for(int i = a + 1 ; i < b ; i++) {
                r = min(r, f(a,i) + f(i,b));
            }
        }
    }
}
```

Contenidos

Forming Quiz Teams

UVA 10911

Dada una lista de $2N$ alumnos ($N \leq 8$) y la ubicación de sus casas en un plano 2D. Se necesitan formar equipos de a dos minimizando la suma de las distancias entre los dos miembros de cada equipo.

Intentemos calcular $F(S)$ la respuesta en $S \subseteq 0..N-1$:

- $F(\emptyset) = 0$
- $F(S) = min_{x,y \in S} dist(x,y) + F(S - \{x,y\})$

La cantidad de subconjuntos es chica: $2^{16} = 65536$

Tenemos una fórmula recursiva y pocos estados! qué esperamos?

Forming Quiz Teams

```
double f(int msk) {
    double &r = dp[msk];
    if (r>=-0.5) return r;
    if (!msk) return r=0;
    int p = __builtin_ffs(msk) - 1;
    r = INF;
    for(int i=p+1 ; i<n ; i++) if ( (msk>>i)&1 ) {
        r = min( r, f(msk ^ (1<<p) ^ (1<<i)) + dist[i][j]);
    }
    return r;
}
...
forn(i,(1<<n)) dp[i]=-1;
double ans = f((1<<n)-1);
```

Funciones y operadores de bits

Operador	Descripcion
$>>$	shift de bits a la derecha
$<<$	shift de bits a la izquierda
^	xor de bits
&	and de bits
$\vert$	or de bits
~	not de bits
__builtin_popcount(msk)	cantidad de bits en una máscara
__builtin_ffs(msk)	posición del primer 1 desde la derecha

Contenidos

Números variados

Números variados

Dado $X \leq 10^{15}$ devolver la cantidad de números $0 \leq Y \leq X$ tal que Y no contenga números consecutivos en su representación en base 10.

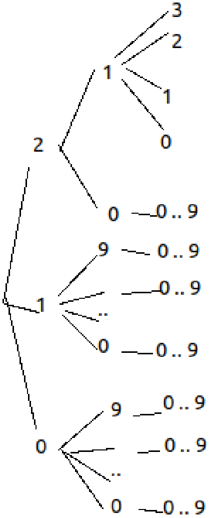
Veamos el campo de posibilidades para X = 213. Y = 0, 1 .. 213

En el problema analizaremos los posibles Y como cadenas de caracteres. Para pensar que todas tienen la misma longitud consideraremos los 0's a la izquierda no significativos. (Para Y = 73, consideraremos "073")

Con esta perspectiva hay que tener cuidado que si bien Y = 3 lo consideramos como "003" es válido pese a tener dígitos repetido (ya que los 0's no significativos pueden estar repetidos).

Números variados

Reordenémoslo un poco:



Números variados

Sólo es importante:

- La posición que estamos completando.
- El último número completado.
- Necesitamos cierta información del prefijo que ya completamos. Ej: saber si venimos "matcheando" el prefijo del número.

¿Podemos hacer DP? Abre las puertas a posibilidades como:

- Reconstruir fácil el K-ésimo!

Números variados

estado posibles:

- 0: el prefijo completado coincide con el de X
- 1: el prefijo completado es lexicograficamente menor que el de X. Además alguno de los dígitos es distinto de 0.
- 2: el prefijo completado son todos 0's.

Encontremos una fórmula para $f(pos, ultimo, estado)$.

- pos puede ser completado con $d \in [0.,9]$ si $estado = 1, 2$ y $d \in [0..X[pos]]$ $estado = 0$.
- si $pos > 0$ tenemos que cuidar que $ultimo \neq d$. Salvo que estemos en $estado = 2$ uy $ultimo = 0$ (son 0's no significativos).
- Debemos actualizar el estado según el estado actual y el caracter completado d.

Números variados

```
int f(int p, int u, int b) {
    int &r = dp[p][u][b]; if (r!=-1) return r;
    if (p==sz(U)) return r = 1;
    r = 0;
    int L = (b==0 ? U[p]-'0' : 9) + 1 ;
    forn(x,L) if ( x!=u || p==0 || (u==0 && b==2) ) {
        int nb;
        if (b==0) {
            if (x == U[p]-'0') {
                nb = 0;
            } else if (x==0 && p==0) {
                nb = 2;
            } else {
                nb = 1;
            }
        } else if (b==1) {
            nb = 1;
        } else {
            if (x==0) {
                nb = 2;
            } else {
                nb = 1;
            }
        }
        r += f(p+1,x,nb);
    }
    return r;
}
```

Contenidos

Tablero disperso

Enunciado (A006506)

Calcular la cantidad de tableros cuadrados de tamaño $N \leq 16$, con 1 ó 0 en sus casillas tal que no existan dos 1 adyacentes (por lado).

Una fuerza bruta sería $O(2^{N^2})$. Pero en realidad sólo necesitamos:

- La última "capa" completada.
- La posición que estamos completando.

1	0	0	1	0	0
0	1	0	0	1	1
1	0	0	1		

Tablero disperso

Enunciado (A006506)

Calcular la cantidad de tableros cuadrados de tamaño $N \leq 16$, con 1 ó 0 en sus casillas tal que no existan dos 1 adyacentes (por lado).

Una fuerza bruta sería $O(2^{N^2})$. Pero en realidad sólo necesitamos:

- La última "capa" completada.
- La posición que estamos completando.

1	0	0	1	0	0
0	1	0	0	1	1
1	0	0	1		

Tablero disperso

```
long long f(int msk, int i, int j) {
    long long &r = dp[msk][i][j];
    if ( r!=-1) return r;
    if ( i == N ) return r = 1;
    r = 0;
    int nmsk, ni = i + (j==N-1), nj = (j+1)%N;
    if ( ( j==0 || ((msk>>(j-1))&1)==0 ) && ((msk>>j)&1)==0
        ) {
        nmsk = msk | (1<<j) ;
        r += f(nmsk, ni, nj);
    }
    nmsk = msk & ~(1<<j) ;
    r += f(nmsk, ni, nj);
    return r;
}
```

Contenidos

Dividing the names

Enunciado

Dada $2N$ palabras se desean dividir entre N calles horizontales y N verticales.
En los letreros que identifican a una calle horizontal (y vertical) se puede escribir un prefijo del nombre de la calle tal que no sea prefijo de ninguna otra calle horizontal (y vertical).

- No existen dos cadenas tales que una sea prefijo de otra.
- Se quiere minimizar la suma de las longitudes de los carteles.

Ejemplos

```
4 GAUSS GALOIS EULER ERDOS
```

Dividing the names

Enunciado

Dada $2N$ palabras se desean dividir entre N calles horizontales y N verticales.
En los letreros que identifican a una calle horizontal (y vertical) se puede escribir un prefijo del nombre de la calle tal que no sea prefijo de ninguna otra calle horizontal (y vertical).

- No existen dos cadenas tales que una sea prefijo de otra.
- Se quiere minimizar la suma de las longitudes de los carteles.

Ejemplos

```
4 GAUSS GALOIS EULER ERDOS Rta: {G,E - G,E}
8 AA AB AC AD BA BB BC BD
```

Dividing the names

Enunciado

Dada $2N$ palabras se desean dividir entre N calles horizontales y N verticales.
En los letreros que identifican a una calle horizontal (y vertical) se puede escribir un prefijo del nombre de la calle tal que no sea prefijo de ninguna otra calle horizontal (y vertical).

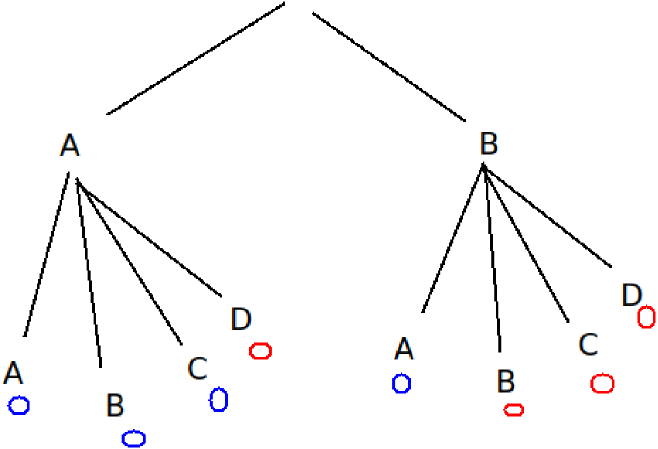
- No existen dos cadenas tales que una sea prefijo de otra.
- Se quiere minimizar la suma de las longitudes de los carteles.

Ejemplos

```
4 GAUSS GALOIS EULER ERDOS Rta: {G,E - G,E}
8 AA AB AC AD BA BB BC BD
Rta: {AA,AB,AC,B - A,BB,BC,BD}
```

Dividing the names

Veamos el Trie de las $2N$ cadenas. Una asignación es como colorear las hojas de dos colores (vertical y horizontal).



Siendo

- $size_v^1$ la cantidad de hojas rojas en el subárbol.
- $size_v^2$ la cantidad de hojas azules en el subárbol.

Hay que minimizar el costo de cada nodo definido como:

$$size_v^1 + size_v^2 - (size_v^1 == 1 + size_v^2 == 1)$$

Que es la cantidad de cadenas distintas en las que aparece.

Observación: basta con pintar N hojas de rojo, las azules quedan definidas.

Hallemos $f(v, k)$ que dado un nodo distribuye de manera óptima k hojas rojas en el subárbol correpondiente al nodo v. De modo que la suma de los costos de los nodos del subárbol sea mínima.

La respuesta al problema es $f(root, N)$

Dividing the names

Observar que parece una mochila en un árbol. Tratemos de hallar f:

$$f(v, k) = k + k - size[v] - (k == 1 + k - size[v] == 1) + ...$$

En ... tenemos que hacer la llamada recursiva a los hijos, hay que distribuir esos k colores rojos disponibles en los hijos de v.

Si fuera un árbol binario sería fácil:

$$... = min_{i=0}^{k}[f(hijo_{derecho}, i) + f(hijo_{izquierdo}, k - i)]$$

Una solución es "binarizar" el árbol:

- A cada nodo le calculamos su primer hijo y su hermano.
- Hacemos una DP en el árbol resultante que es binario.

$$... = min_{i=0}^{k}[f(primerHijo_v, i) + f(hermano_v, k - i)]$$

Dividing the names

```
long long f(int v, int n) { // resultado: (f(0,n) - cant[0] + 2*n)*n
    long long &r = dp[v][n]; if (r != -1) return r;
    if (hijo[v]==-1 && hermano[v]==-1) return r = n>1 ? INF:0;
    if (hijo[v]==-1)
        return r = min( f(hermano[v],n), f(hermano[v],n-1) );
    if (hermano[v]==-1) {
        r = f(hijo[v],n) + (n==1 ? 0ll : n)
                         + (cant[v]-n==1 ? 0ll : cant[v]-n) ;
        return r = min(r,INF);
    }
    r = INF;
    for(int m=0 ; m < n+1 ; m++) {
        r = min(r, f(hijo[v],m) + f(hermano[v],n-m) +
                   (m==1 ? 0ll : m) +
                   (cant[v]-m==1 ? 0ll : cant[v]-m) );
    }
    return r;
}
```

Dividing the names

Otra solucion es hacer una mochila para distribuir los k objetos en los hijos de v.

$$f(v,k) = dp_v(0,k)$$

$$dp_v(i,k) = \sum_{t=0}^{k} [\quad f(hijo[v][i],t) + dp_v(i+1,k-t) \quad]$$

Esto es hacer una mochila en CADA hijo.
Analicemos la complejidad de todas juntas:

$$\sum_{v \in G} O(grado(v) * K^2) = O([\sum_{v \in G} grado(v)] * K^2) = O(N * K^2)$$

Dividing the names

```
forn(i,postC) {
        int v = post[i];
        if (sz(G[v])) {
            int cantH = sz(G[v]);
            dp[cantH][0] = 0;
            forr(x,1,n+1) dp[cantH][x] = INF;
            dforn(i,cantH) {
                int hv = G[v][i];
                forn(m,n+1) {
                    dp[i][m] = INF;
                    forn(mp,min(cant[hv],m)+1) {
                        dp[i][m] = min(dp[i][m],
                            dp[i+1][m-mp]+f[hv][mp]);
                    }
                }
            }
            forn(m,min(cant[v],n)+1) f[v][m] = min(INF, dp[0][m] + (m==1 ? 0ll : m) + (cant[v]-m==1 ? 0ll :
                    cant[v]-m) );
        } else {
            assert(cant[v]==1);
            f[v][0]=f[v][1]=0;
            forr(m,2,n+1) f[v][m]=INF;
        }
    }
    cout << (f[0][n]-cant[0]+2*n)*n << endl;
```