Problema de la escalera

Recursión y programación dinámica

Escalera - Introducción

Agustín es un gran programador y odia hacer la misma cosa dos veces. Esto es un problemón porque tiene escaleras en su casa, las cuales tiene que subir y bajar todos los días. ¡En ocasiones, varias veces en el mismo día!

Como Agustín es alto, puede subir de a uno o de dos escalones a la vez, y es esto lo que le permite recorrer las escaleras de distintas formas. Por ejemplo, si una escalera tiene 3 escalones, puede primero subir un escalón y después los otros dos, o puede subir los primeros dos de un paso, y después el tercero.

Si la casa de Agustín tiene una escalera de N escalones y otra de M, ayúdalo a contar la cantidad de formas de subir de la planta baja al primer piso.

Escalera - Introducción

Como siempre, tomamos un enfoque top-down para la implementación de un programa. Los pasos que realiza el programa son:

- tomar dos enteros N y M
- calcular la cantidad de formas de subir
- imprimir el resultado

(demo 1 : esqueleto)

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









int main() {
  int N, M;
  cin >> N >> M;


  int resultado = ???;
  cout << resultado << '\n';
}
```

Principio de la suma

Escalera - Principio de la suma

Agustín tiene dos opciones mutuamente excluyentes (si hace una, no hace la otra): subir por la primera escalera, o por la segunda.

Cuando contamos las formas de hacer algo y hay opciones mutuamente excluyentes, la cantidad total de formas es la suma de las formas de hacer la cosa tomando cada una de las opciones.

En particular, la cantidad de formas de subir al primer piso es la cantidad de formas de subir por la primera escalera más la cantidad de formas de subir por la segunda.

(demo 2 : principio de la suma)

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









int main() {
  int N, M;
  cin >> N >> M;


  int resultado = ???;
  cout << resultado << '\n';
}
```

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

int main() {
  int N, M;
  cin >> N >> M;
  int formas_N = ???;
  int formas_M = ???;
  int resultado = formas_N + formas_M;
  cout << resultado << '\n';
}
```

Funciones

Escalera - Funciones

C++ tiene funciones, un mecanismo para separar el código en partes y evitar la repetición (como le gusta a Agustín). Las funciones también son muy útiles a la hora de hacer programación top-down.

Una función es un bloque de código que tiene “parámetros”. Los parámetros son variables a los que le damos valor al “llamar” a la función. Esto significa que podemos correr el mismo pedazo de código con distintos valores!

Aparte, una función puede “devolver” un resultado con la sentencia “return”. El valor de la llamada a una función será el valor que esta devuelve.

Escalera - Funciones

La sintaxis de las funciones es así:

```
tipo_del_resultado nombre(parámetros) { código }
```

un ejemplo sencillo es una función que calcula el cuadrado de un número

```
int cuadrado(int x) { return x*x; }

int ochenta_y_uno    = cuadrado(9);  // tiene el valor 81

int ciento_veintiuno = cuadrado(11); // tiene el valor 121
```

Por otro lado “el main”, que siempre escribimos, también es una función

```
int main() { … }
```

(demo 3: Funciones)

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









int main() {
  int N, M;
  cin >> N >> M;
  int formas_N = ???;
  int formas_M = ???;
  int resultado = formas_N + formas_M;
  cout << resultado << '\n';
}
```

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


int escalera(int N) {




  return ???;
}

int main() {
  int N, M;
  cin >> N >> M;
  int formas_N = escalera(N);
  int formas_M = escalera(M);
  int resultado = formas_N + formas_M;
  cout << resultado << '\n';
}
```

Escalera - Funciones

Escribimos el esqueleto de una funcion “escalera(n)” que nos va a dar la cantidad de formas de subir una escalera de n escalones.

Notablemente, redujimos el problema de dos escaleras a una sola!

Ahora, veamos cómo resolver el problema con una sola escalera.

Idea: empezar por los casos extremos

Escalera - Casos extremos

Les podemos decir casos borde, o casos especiales pero siempre hablamos de los puntos donde la forma de la solución puede cambiar. Algunas veces el enunciado insinúa estos casos y otras tenemos que descubrirlos estudiando el problema (usando razonamientos lógicos, diagramas, etc)

Muchas veces los casos extremos son más fáciles de resolver.

Por ejemplo, en este problema nada apunta a que 100 sea un caso borde. Pero, como no tiene sentido hablar de cantidades negativas de escalones, no sería raro que 0, 1 y/o 2 sean casos especiales.

Escalera - Casos extremos

Resolvemos a mano para 0, 1.

Para 0 escalones hay una sola solución: apenas arrancamos, ya llegamos al final.

Para 1 escalón hay una sola solución: arrancamos, subimos un escalón y listo.

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


int escalera(int N) {




  return ???;
}

int main() {
  int N, M;
  cin >> N >> M;
  int formas_N = escalera(N);
  int formas_M = escalera(M);
  int resultado = formas_N + formas_M;
  cout << resultado << '\n';
}
```

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


int escalera(int N) {
  if (N == 0) return 1;
  if (N == 1) return 1;



  return ???;
}

int main() {
  int N, M;
  cin >> N >> M;
  int formas_N = escalera(N);
  int formas_M = escalera(M);
  int resultado = formas_N + formas_M;
  cout << resultado << '\n';
}
```

Escalera - Caso general

Buscamos aplicar el principio de la suma. Para esto, necesitamos dos opciones mutuamente excluyentes.

En problemas donde la solución consiste de una secuencia de pasos, una buena idea es mirar las opciones del primer paso.

Para este problema, el primer paso puede ser subir 1 escalón o subir 2 escalones. Esas dos opciones son mutuamente excluyentes.

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


int escalera(int N) {
  if (N == 0) return 1;
  if (N == 1) return 1;



  return ???;
}

int main() {
  int N, M;
  cin >> N >> M;
  int formas_N = escalera(N);
  int formas_M = escalera(M);
  int resultado = formas_N + formas_M;
  cout << resultado << '\n';
}
```

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


int escalera(int N) {
  if (N == 0) return 1;
  if (N == 1) return 1;

  int formas_subiendo_uno = ???;
  int formas_subiendo_dos = ???;
  return formas_subiendo_uno + formas_subiendo_dos;
}

int main() {
  int N, M;
  cin >> N >> M;
  int formas_N = escalera(N);
  int formas_M = escalera(M);
  int resultado = formas_N + formas_M;
  cout << resultado << '\n';
}
```

Escalera - Caso general

Tras dar un paso de un escalón en una escalera de N escalones, nos restaría subir una escalera de N-1 escalones. En cambio, tras dar un paso de dos escalones, nos restaría subir N-2 escalones.

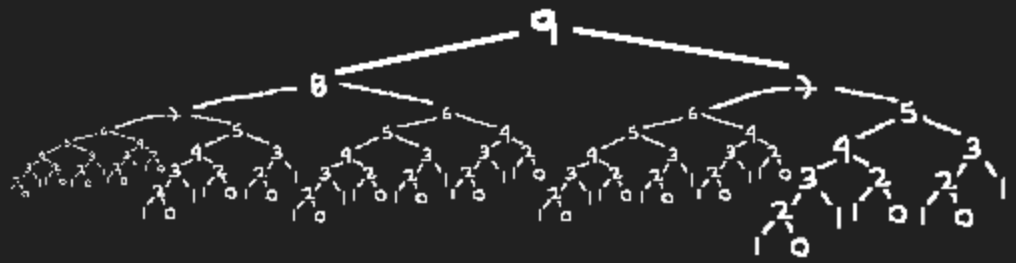
Resulta que ya tenemos una función que calcula la cantidad de formas de subir una escalera de alguna cantidad de escalones… la función escalera(n) que venimos escribiendo!

Idea: definir una función con si misma

(demo 4: recursión)

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


int escalera(int N) {
  if (N == 0) return 1;
  if (N == 1) return 1;

  int formas_subiendo_uno = ???;
  int formas_subiendo_dos = ???;
  return formas_subiendo_uno + formas_subiendo_dos;
}

int main() {
  int N, M;
  cin >> N >> M;
  int formas_N = escalera(N);
  int formas_M = escalera(M);
  int resultado = formas_N + formas_M;
  cout << resultado << '\n';
}
```

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


int escalera(int N) {
  if (N == 0) return 1;
  if (N == 1) return 1;

  int formas_subiendo_uno = escalera(N-1);
  int formas_subiendo_dos = escalera(N-2);
  return formas_subiendo_uno + formas_subiendo_dos;
}

int main() {
  int N, M;
  cin >> N >> M;
  int formas_N = escalera(N);
  int formas_M = escalera(M);
  int resultado = formas_N + formas_M;
  cout << resultado << '\n';
}
```


9
8
5

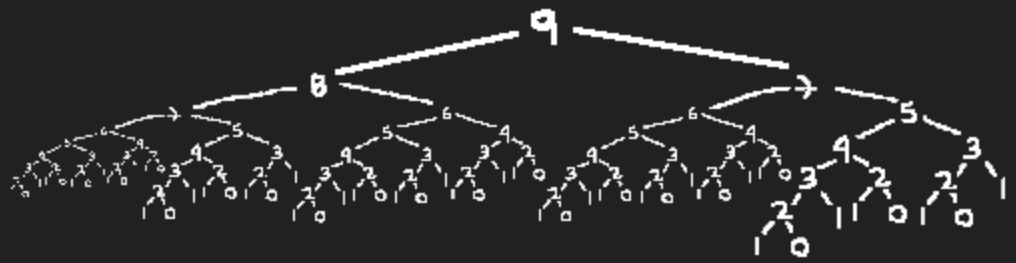
Escaleras - Complejidad

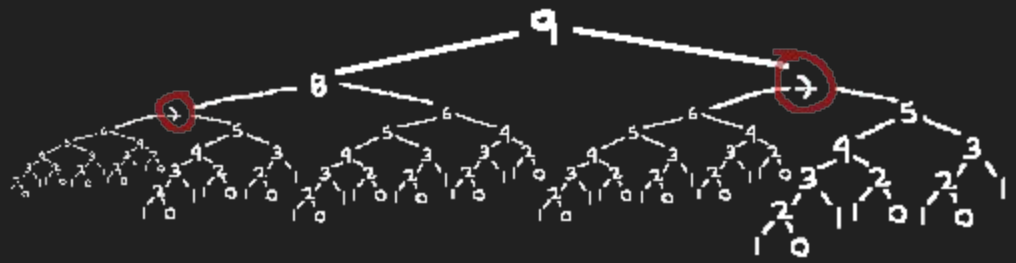
Hagamos un breve análisis de eficiencia. La función toma una cantidad de tiempo proporcional a la cantidad de nodos en el arbolito que vimos.

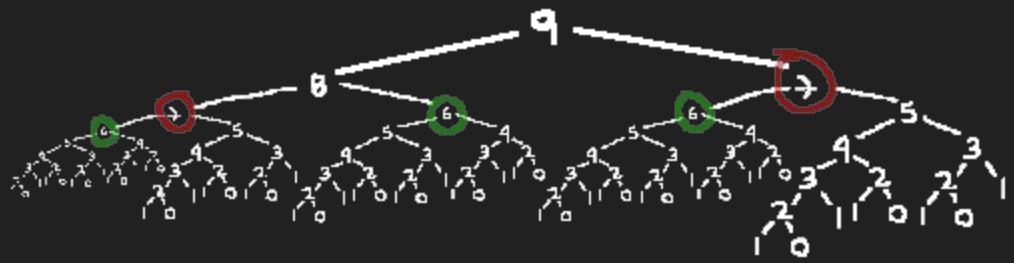
La altura del árbol es N y cada nodo se parte en dos. Entonces, no sería raro que terminemos con 2 * 2 * … N veces … * 2 * 2 nodos en total. O sea 2^N nodos. Así, concluimos que la función tiene complejidad O(2^N)

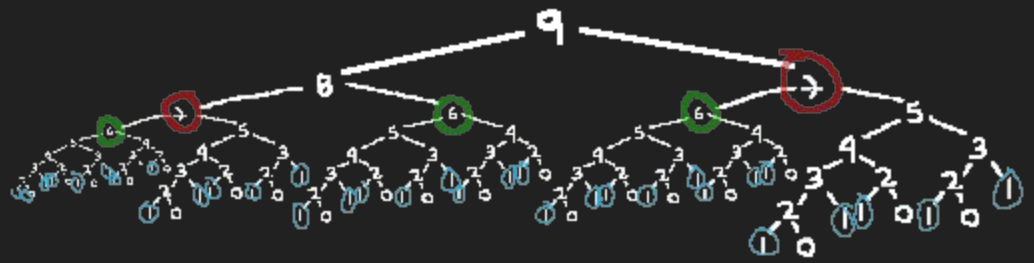
Esto no es del todo acertado, pero le pega cerca. Lo importante es que es una función exponencial, y estas funciones crecen muy rápido. (Eventualmente superan a cualquier polinomio, por ejemplo N^1000)

En resumen, esta función se vuelve muy lenta muy rápido.


9
8
5


9
8
7
6
5
4
3
2
1
0


9
8
7
6
6
5
5
4
3


9
8
6
6
6
5
4
3
2
1
0

Idea: guardar resultados para evitar la repetición

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


int escalera(int N) {
  if (N == 0) return 1;
  if (N == 1) return 1;

  int formas_subiendo_uno = escalera(N-1);
  int formas_subiendo_dos = escalera(N-2);
  return formas_subiendo_uno + formas_subiendo_dos;
}

int main() {
  int N, M;
  cin >> N >> M;
  int formas_N = escalera(N);
  int formas_M = escalera(M);
  int resultado = formas_N + formas_M;
  cout << resultado << '\n';
}
```

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

int memo[1000];
int escalera(int N) {
  if (N == 0) return 1;
  if (N == 1) return 1;

  int formas_subiendo_uno = escalera(N-1);
  int formas_subiendo_dos = escalera(N-2);
  return memo[N] = formas_subiendo_uno + formas_subiendo_dos;
}

int main() {
  int N, M;
  cin >> N >> M;
  int formas_N = escalera(N);
  int formas_M = escalera(M);
  int resultado = formas_N + formas_M;
  cout << resultado << '\n';
}
```

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

int memo[1000];
int escalera(int N) {
  if (N == 0) return 1;
  if (N == 1) return 1;
  if (memo[N] != 0) return memo[N];
  int formas_subiendo_uno = escalera(N-1);
  int formas_subiendo_dos = escalera(N-2);
  return memo[N] = formas_subiendo_uno + formas_subiendo_dos;
}

int main() {
  int N, M;
  cin >> N >> M;
  int formas_N = escalera(N);
  int formas_M = escalera(M);
  int resultado = formas_N + formas_M;
  cout << resultado << '\n';
}
```

Escalera - Complejidad

Ahora la función es mucho más rápida porque se calcula una sola vez por valor. Esta técnica se llama programación dinámica, y aparte de ser más rápida, simplifica el análisis de la complejidad.

Para cualquier función con programación dinámica, basta con multiplicar la cantidad de valores posibles para los argumentos (en este caso O(N)) con la complejidad de calcular cada resultado (en este caso O(1)), para obtener la complejidad total.

Así, resulta que esta versión de la función tiene complejidad O(N).

Escalera - Conclusiones

- Al contar posibilidades, podemos sumar los resultados de casos disjuntos
- Podemos usar funciones para evitar la repetición del código
- Las funciones ayudan con la programación top-down
- A veces lo más natural es escribir funciones que se llaman a sí mismas
- Usamos DP para optimizar funciones recursivas que repiten muchos estados