Recursividad

Conceptos básicos

Recursividad

Es un enfoque para resolver problemas en el que caracterizamos un problema por medio de ciertas variables y lo resolvemos de la siguiente forma.

- Si el problema es suficientemente simple/pequeño, resolverlo directamente
- Caso contrario, reducir el problema a instancias más pequeñas del mismo, es decir con variables más pequeñas o simples.

Cuando el problema es suficientemente simple o pequeño, este se llama **caso base**. En los otros casos, tenemos **transiciones** que nos llevan a instancias más pequeñas o simples del problema.

Esto es parecido a la **inducción matemática**, en donde para la demostración se tiene un caso base y una hipótesis inductiva.

Conceptos básicos

Recursividad

En matemática existen muchas secuencias de números que están definidas a partir de la recursión.

Ejemplo: Sucesión de Fibonacci

$$F_n = \begin{cases} 0, & si\ n = 0 \\ 1, & si\ n = 1 \\ F_{n-1} + F_{n-2}, & si\ n > 1 \end{cases}$$

Cuando lo pasamos a programación, en C++ podemos definir **funciones recursivas.** Es decir, funciones que se llaman a sí mismas, pero pasándole otros parámetros para resolver un problema más simple.

```
int f(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

Casos base

Correctness

Sin embargo, muchas veces la parte más interesante/complicada es poder demostrar (o convencerse) de que la recursividad es correcta a la hora de resolver un problema. Existen algunas reglas que podemos seguir para esto.

1. **Demuestra que la función termina:**

- **No deben haber ciclos.** Si la solución recursiva a tu problema presenta ciclos, eso hará que matemáticamente tu solución no tenga sentido ya que el valor de tu algoritmo/función nunca estaría definido. A nivel de código, los ciclos van a producir que tu algoritmo entre en un bucle infinito, lo cual se traduciría en un **TLE** en una competencia.

¿Cómo hacemos para evitar los ciclos?

Es útil definir una "propiedad" que resuma los parámetros de la función y que uno pueda demostrar que esa característica siempre aumenta o siempre disminuye, por lo que nunca habrá ciclos. Por ejemplo en la función $f(n,m) = f(n-1,m) + f(n,m-1) + 2$, la propiedad suma $S = n + m$ disminuye en 1 en cada transición. Aparte de la suma también puedo utilizar otras propiedades como el mínimo y el máximo.

- **Asegúrate que llegues a un caso base en un número finito de pasos.** A veces puedes no tener ciclos, pero puede que estés recurseando infinitamente. Por ejemplo, si tus parámetros siempre disminuyen sin llegar a un caso base, puedes seguir disminuyendo hasta $-\infty$

Correctness

Sin embargo, muchas veces la parte más interesante/complicada es poder demostrar (o convencerse) de que la recursividad es correcta a la hora de resolver un problema. Existen algunas reglas que podemos seguir para esto.

2. Demuestra que la función recursiva resuelve el problema:

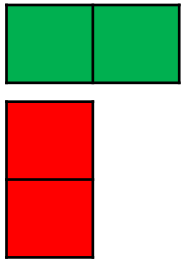
- **Demuestra el resultado en los casos bases.** Suelen ser triviales. Se pueden calcular a mano.
- **Demuestra lo demás por hipótesis inductiva.** Toma la función recursiva como una caja negra y asume que la recursión es correcta para estados “menores” a tu estado actual. Luego demuestra que tu estado actual se sigue calculando correctamente.

Consejo: Ten fe en la recursión. No intentes “entenderla”, tratando de ver qué hace en cada paso. Después de que ya demostraste (o te convenciste) de que la hipótesis inductiva es correcta, ya puedes estar seguro que la recursión es correcta.

Ejemplo: Conteo de dominos

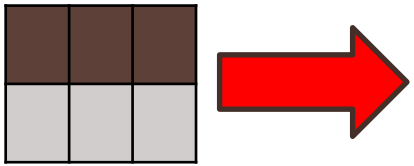
¿De cuántos formas podemos cubrir un tablero de $2 \times n$ usando dominós de 2×1 o 1×2 sin solapamiento?

Dominós



Tablero de $2 \times n$

Tablero de 2×3



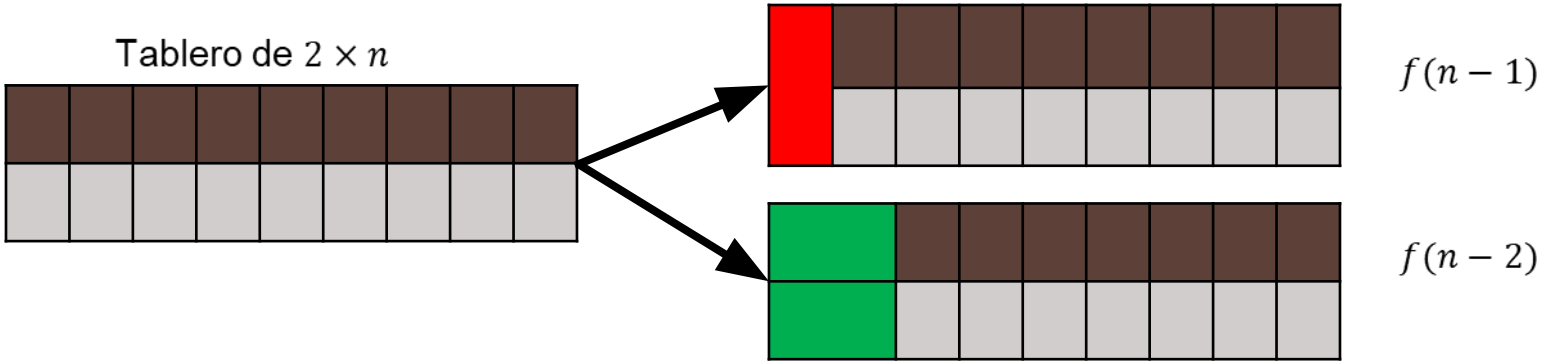
3 posibles formas

Ejemplo: Conteo de dominos

Notemos que la única variable que necesitamos es n, esta variable definirá nuestra estado

Sea $f(n)$ la respuesta al problema. Notemos que no importa el orden por el que empecemos. Podemos empezar tratando de cubrir la primera columna de alguna forma.

Nos daremos cuenta que solo hay 2 formas



$$f(n) = \begin{cases} 1, & n = 1 \\ 2, & n = 2 \\ f(n-1) + f(n-2), & n > 2 \end{cases}$$

Ejemplo: Conteo de dominos

Este ejemplo es muy simple, pero si quisiéramos convencernos que la recursión no entra en un bucle infinito, podemos notar que la variable n siempre disminuye hacia dos estados menores que ella. Seguirá disminuyendo hasta que llegue a los dos casos bases.

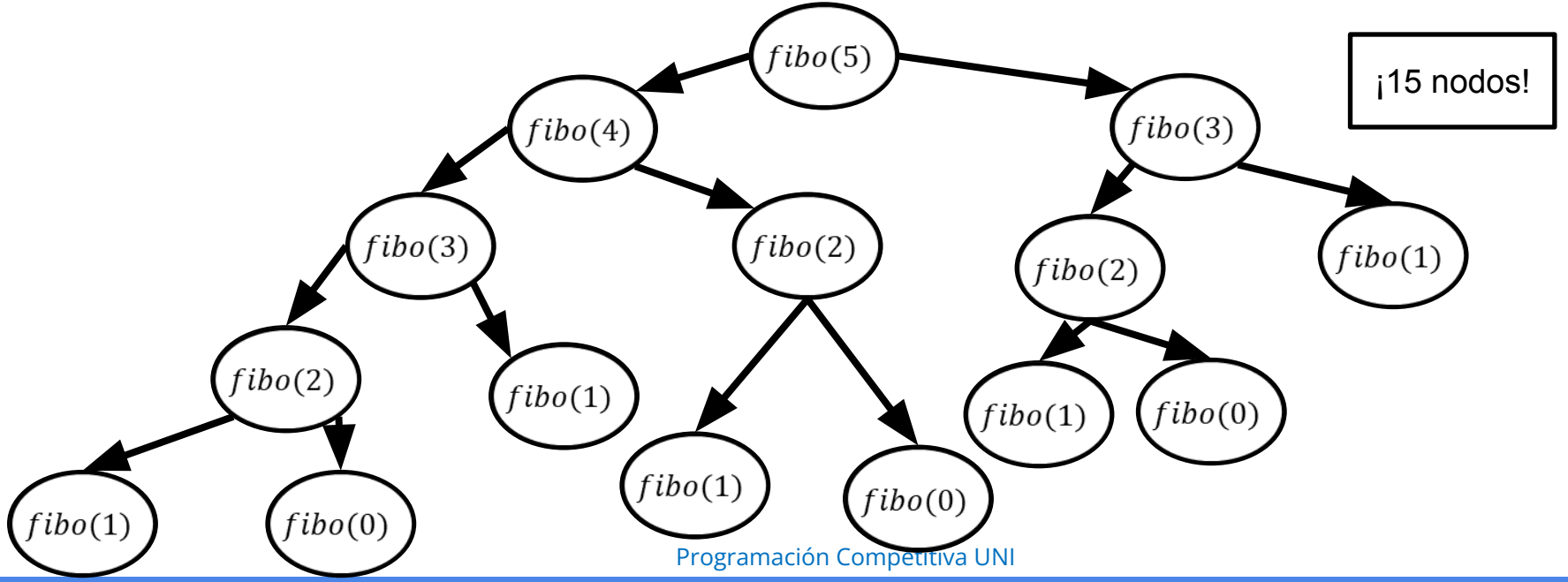
Si uno quisiera demostrar que la fórmula es correcta, se puede hacer por inducción. Los casos bases son triviales porque son cálculos a mano. Luego, suponiendo que $f(x)$ es correcto para $x < n$, $f(n)$ es calculado correctamente debido a que solo podemos desplazarnos a 2 casos excluyentes, $f(n-1)$ y $f(n-2)$.

Nota: Muchas veces es útil empezar el caso base desde el "caso vacío", que en este caso sería $n = 0$. Aunque sea un poco antinatural, si tenemos un tablero vacío, el número de formas de cubrirlo es 1 que sería no poner ningún dominó. Entonces podemos empezar con $f(0) = 1$ y $f(1) = 1$ y llegaríamos al mismo resultado.

$$f(n) = \begin{cases} 1, & n = 0 \\ 1, & n = 1 \\ f(n-1) + f(n-2), & n > 1 \end{cases}$$

Time Complexity

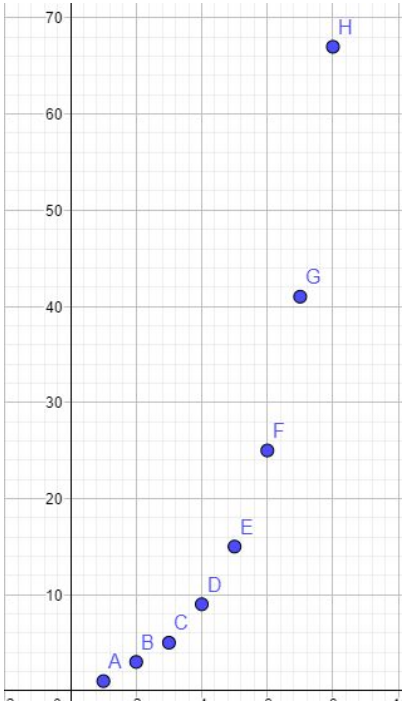
¿Cuál es la complejidad en tiempo de usar recursividad de la forma más simple? ¿Cuántas iteraciones realiza el algoritmo?
Depende mucho del problema pero **no suele ser muy bonita.** Analicemos qué sucede en nuestro código para calcular Fibonacci. En general, para una mejor visualización podemos usar un **árbol de recursión**.



Time Complexity

Si contáramos el número de nodos para cada n tendríamos lo siguiente:

T(1) = 1
T(2) = 3
T(3) = 5
T(4) = 9
T(5) = 15
T(6) = 25
T(7) = 41
T(8) = 67
T(9) = 109
T(10) = 177
T(11) = 287
T(12) = 465
T(13) = 753
T(14) = 1219
T(15) = 1973



¡Exponencial!

No es difícil darse cuenta que
$T(n) = T(n-1) + T(n-2) + 1$

Time Complexity

¿Cómo podemos expresar la complejidad en función de n?
Podemos hallar un primer límite si analizamos el árbol de recursión nivel por nivel. El primer nivel se divide en dos nodos. Luego cada uno de los nodos del siguiente nivel se vuelve a dividir en 2 y así sucesivamente hasta que ya no se puedan dividir más.

Esto nos puede dar la idea intuitiva de que en cada nivel la cantidad de nodos se va duplicando, por lo que la cantidad de nodos $T(n) \leq 2^n - 1$ $(\forall n > 0)$.

Podemos demostrar esto por inducción ya que $T(1) = 1 \leq 2^1 - 1$, $T(2) = 3 \leq 2^2 - 1$
Por hipótesis inductiva suponemos que $T(n-2) \leq 2^{n-2} - 1$ y $T(n-1) \leq 2^{n-1} - 1$.
$T(n) = T(n-1) + T(n-2) + 1 \leq 2^{n-1} - 1 + 2^{n-2} - 1 + 1 \leq 2^{n-1} + 2^{n-1} - 1 = 2^n - 1 \Longrightarrow T(n) \leq 2^n - 1$ ■

Entonces la complejidad de nuestra función recursiva para Fibonacci es $O(2^n)$

Bonus: El hecho de que el mismo $T(n)$ se parezca a la secuencia de Fibonacci nos permite encontrar un mejor bound. $O(\varphi^n)$, donde $\varphi = 1.618$ ... es el número áureo.

Time Complexity

Muchas veces las funciones recursivas tendrán complejidad exponencial si no se tiene cuidado o no se usa ninguna técnica adicional.

En algunos casos, podemos reemplazar un algoritmo recursivo por su versión iterativa. Esto puede mejorar la complejidad a veces. Incluso, aún cuando la complejidad queda igual, los algoritmos iterativos suelen ser un poco más rápidos que los recursivos (constante escondida).

Por ejemplo, versión iterativa de Fibonacci:

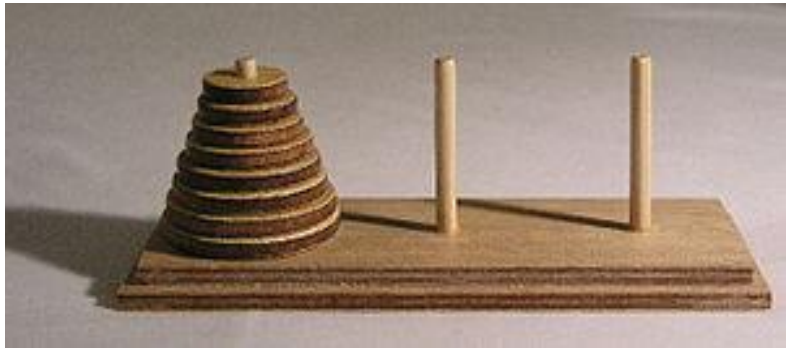
```
int n;
cin >> n;
vector<int> fibo(n);
fibo[0] = 0;
fibo[1] = 1;
for (int i = 2; i < n; i++) {
    fibo[i] = fibo[i - 1] + fibo[i - 2];
}
```

Entonces, ¿por qué usar recursividad?
Porque no siempre tenemos una versión iterativa sencilla

Torres de Hanoi

Las torres de Hanoi corresponde a un juego/rompecabezas en donde existen 3 palos de madera verticales. Uno de ellos contiene n discos con diámetros distintos donde el disco más pequeño se encuentra arriba y el disco más grande abajo. El objetivo es mover los n discos del 1er palo al 3er palo, pudiendo usar el 2do como palo auxiliar, bajo las siguientes reglas.

- Solo se puede mover un disco a la vez.
- Cada movimiento consiste en tomar el disco superior de alguno de los palos y moverlo hacia la parte superior de otro palo.
- No se permite colocar un disco sobre otro de un diámetro menor.



¿Cómo logro ganar el juego?
¿Cuál es el mínimo número de movimientos?

Torres de Hanoi

Pensemos recursivamente.
¿Cuál es el primer disco que llegará a su destino en la posición correcta?
El más grande

Pero para mover el más grande, primero debo mover los $n-1$ anteriores hacia otro lado. Preferiblemente hacia el palo auxiliar, para que así pueda mover el $n-$ ésimo disco a su posición final. Finalmente debo mover los $n-1$ restantes hacia el palo.

Y… ¡listo! Ya no hay que hacer nada más porque ya logramos reducir el problema a una instancia más simple (de n discos a $n-1$) donde se requiere hacer completamente lo mismo. El caso base es cuando $n=0$ en donde no tenemos que hacer nada.

```
Function Hanoi (n, source, aux, target):
    if n == 0 then
        You're done
    end
    Hanoi (n - 1, source, target, aux)
    Move n-th disk to target
    Hanoi (n - 1, aux, source, target)
end
```

Torres de Hanoi

Podemos demostrar por inducción que nuestra estrategia es correcta y nos lleva a ganar el juego en cierto número finito de movimientos.

Para $n = 0$ es trivialmente correcto no hacer nada.
El paso inductivo también es trivial bajo la hipótesis de que nuestra estrategia funciona para $n - 1$ discos.
Se mueven los $n - 1$ al medio, luego muevo el más grande y luego vuelvo a mover los $n - 1$ discos.

¿Cuántos movimientos hacemos?
$T(0) = 0$
$T(n) = 2T(n-1) + 1$

Por inducción podemos demostrar que $T(n) = 2^n - 1$

Es más, esta es la cantidad mínima de movimientos ya que la estrategia óptima debe mover sí o sí los $n - 1$ discos a cualquier lado para sacar el grande, eso hace que tengamos al menos $T(n-1)$ movimientos. Luego sí o sí tiene que hacer al menos un movimiento más para mover al grande. Por último tiene que mover los $n - 1$ restantes a la fuente. Entonces la estrategia óptima hace $\geq 2T(n-1) + 1$ movimientos

Referencias

- Jeff Erickson. Algoritms. Chapter 1 – Recursion. https://jeffe.cs.illinois.edu/teaching/algorithms/book/01-recursion.pdf

¡Gracias por su atención!