```cpp
int haceAlgo(int n){
    n = n + 1;
    std::cout << "n in function: " << n << std::endl;

    return n;
}

int main() {
    int n = 4;
    int m = haceAlgo(n);
    std::cout << "n in main: " << n << std::endl;
    std::cout << "m returned from function: " << m;

    return 0;
}
```

```cpp
int haceAlgo(int& n){
    n = n + 1;
    std::cout << "n in function: " << n << std::endl;

    return n;
}

int main() {
    int n = 4;
    int m = haceAlgo(n);
    std::cout << "n in main: " << n << std::endl;
    std::cout << "m returned from function: " << m;

    return 0;
}
```

ALGORITMOS DE FUERZA BRUTA

Un algoritmo de fuerza bruta o búsqueda exhaustiva es un enfoque de resolución de problemas donde **se generan todas las posibles soluciones** (todas las soluciones factibles generadas por un problema) **hasta encontrar la correcta.** Este método **explora exhaustivamente el espacio de soluciones posibles**. Si bien es simple de implementar, **puede ser muy ineficiente, especialmente cuando el número de posibles soluciones es grande,** ya que, un algoritmo de fuerza bruta tiene una complejidad exponencial.

Los algoritmos de fuerza bruta se usan cuando:

- El espacio de soluciones es pequeño y manejable.
- Se necesita una solución garantizada y no hay mejor método conocido.
- Se desea un punto de partida básico para comparar con otros algoritmos más eficientes.

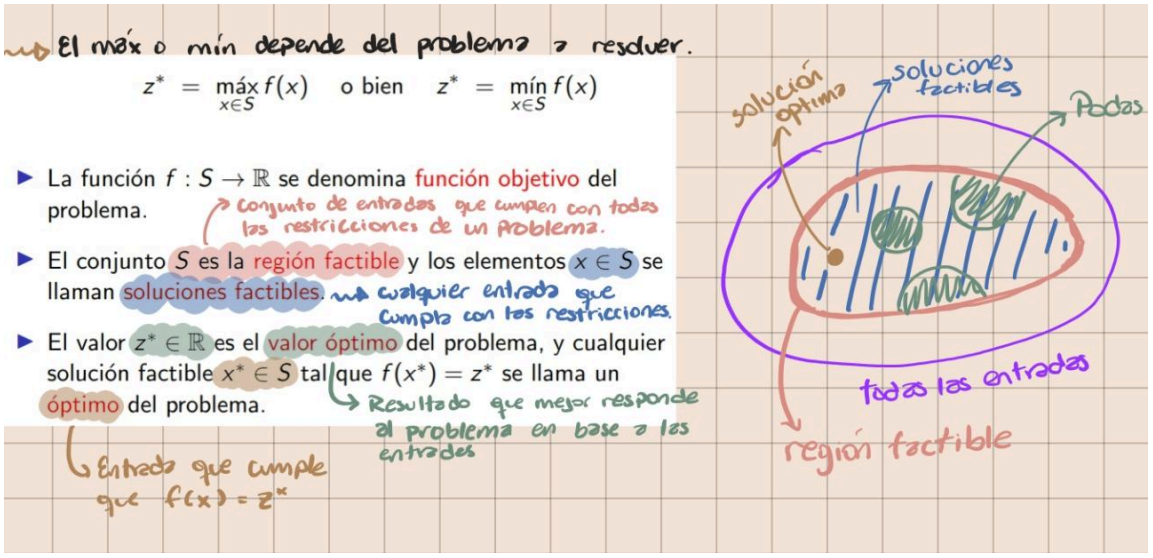
BACKTRACKING

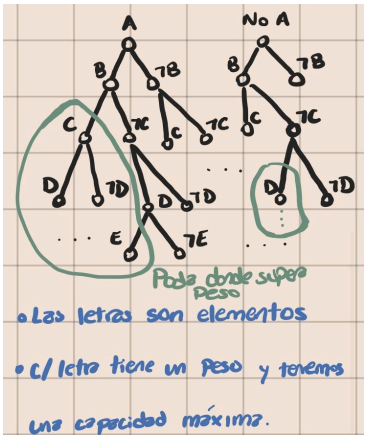
El backtracking es una **técnica de algoritmos** utilizada para resolver problemas de manera sistemática, **probando diferentes posibilidades y retrocediendo** (haciendo "backtrack") **cuando se alcanza un callejón sin salida.** Es una mejora sobre la fuerza bruta porque **evita explorar soluciones que no pueden llevar a una solución válida.**

OPTIMIZACIÓN

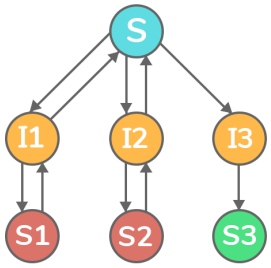
Un problema de optimización consiste en **encontrar la mejor solución dentro de un conjunto.**

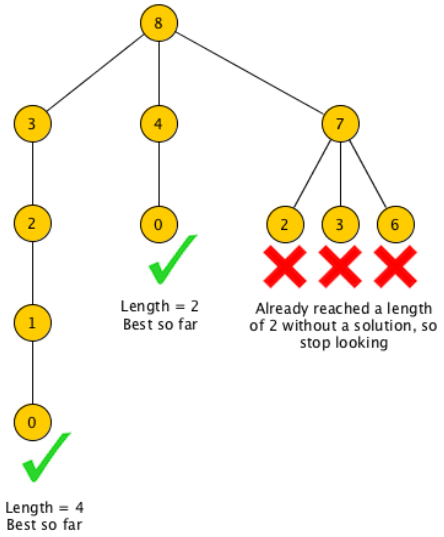
"The first version of a backtracking algorithm does not contain any optimizations. We simply use backtracking to generate all possible paths, optimizations help reduce the amount of possible paths by cutting those who don't follow the conditions given"





Poda → Una poda es una condición o criterio que se usa para evitar explorar ramas del árbol de decisiones que no pueden llevar a una solución válida. Esto es una optimización para el algoritmo de Backtracking, ya que nos ahorra analizar soluciones que no son válidas.
En cada paso, se evalúa si la solución parcial puede llegar a ser una solución completa válida. Si se determina que no puede serlo, el algoritmo "poda" esa rama, es decir, abandona esa dirección y retrocede al último punto de decisión para probar otra opción.





- Soluciones parciales son todos los nodos * * * *
- Soluciones candidatas, son los nodos rojos y verdes * *
- Soluciones válidas, son los nodos verdes *

💡 La cantidad de nodos del árbol las calculamos como 2^N en caso de tener dos hijos por nodo y una cantidad de N elementos.

❖ Algoritmo de Backtracking

El algoritmo toma dos parámetros, a y k.

a → Una **solución parcial** que se está construyendo..

k → **Índice o contador que ayuda a gestionar la construcción de la solución.** Por ejemplo, si nos piden tomar 4 elementos de un conjunto de 10, nuestro k máximo será 4.

```
algoritmo BT(a, k)
    si a es solución entonces
        procesar(a)
        retornar
    sino
        para cada a' ∈ Sucesores(a, k)
            BT(a', k + 1)
        fin para
    fin si
    retornar
```

Si a es solución:

Aquí el algoritmo **verifica si la solución parcial a es una solución completa** (candidata) **al problema**.
Si es una solución entonces, **procesar(a), verifica si es una solución válida** y lo almacena, lo cuenta, o lo imprime. **Después de procesar la solución, el algoritmo termina esta rama de ejecución y retorna para probar otras posibilidades.**

Si a no es solución:

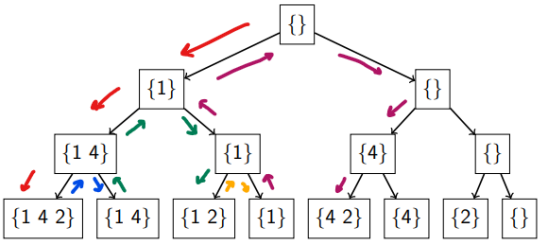
Si a **no es una solución completa**, **entonces se procede a expandirla.**
para cada a' ∈ Sucesores(a, k): El algoritmo genera todos los posibles "sucesores" de la solución parcial a. Cada sucesor **a' es una versión extendida de a que incluye un nuevo elemento o paso.**
BT(a', k + 1): El algoritmo llama recursivamente a sí mismo para cada sucesor a', incrementando k en 1 para reflejar el avance en la construcción de la solución.

❖ Recorrido de Soluciones

- ¿Cuántos nodos tiene el árbol que estamos recorriendo?
- Tiene $\sum_{i=0}^{N} 2^i = O(2^N)$ nodos.

- Cada elemento puede o no estar en el subconjunto.

1.
2.
3.
4.
5.



Para hallar el conjunto S de posibles soluciones usamos backtracking, esta técnica nos permite generar espacios de búsqueda "recursivos", mediante la extensión de soluciones parciales. La idea es definir este método de extensión, y mediante recursión generar de forma ordenada el espacio de soluciones S.

- Generemos S recursivamente, usando backtracking.
- ¿Hay una forma recursiva de generar los subconjuntos de un conjunto?

$$subsets(c : C) = c \times subsets(C) \cup subsets(C)$$

4 {(3),(2)}

{(3,4),(2,4)} U {(3),(2)}

Ejemplo (Reinas): https://www.youtube.com/watch?v=s7kBjMOMWg0&t=23s

Un ejemplo común es el problema del *n-reinas*, donde se deben colocar n reinas en un tablero de ajedrez de n x n de manera que ninguna reina ataque a otra. El algoritmo de *backtracking* coloca una reina y luego procede a colocar la siguiente. Si se coloca una reina en una posición donde no se puede colocar ninguna más, el algoritmo retrocede y prueba una nueva posición para la reina anterior.

- Por ejemplo, para $n = 8$ una implementación directa consiste en generar todos los subconjuntos de casillas.

  $2^{64} = 18,446,744,073,709,551,616$ combinaciones!

- Sabemos que dos damas no pueden estar en la misma casilla.

  $\binom{64}{8} = 4,426,165,368$ combinaciones.

- Sabemos que cada columna debe tener exactamente una dama. Cada solución parcial puede estar representada por $(a_1, \ldots, a_k)$, $k \leq 8$, con $a_i \in \{1, \ldots, 8\}$ indicando la fila de la dama que está en la columna i.

  Tenemos ahora $8^8 = 16,777,216$ combinaciones.

- Adicionalmente, cada fila debe tener exactamente una dama.

  Se reduce a $8! = 40,320$ combinaciones.

S → Es el conjunto de tableros de ajedrez con 8 reinas
P(x) → Verifica que no se ataquen entre ellas
procesar → Lleva la cuenta de la cantidad de tableros que verifican P(x)

```
for x ∈ S do:
    if P(x):
        procesar x
```

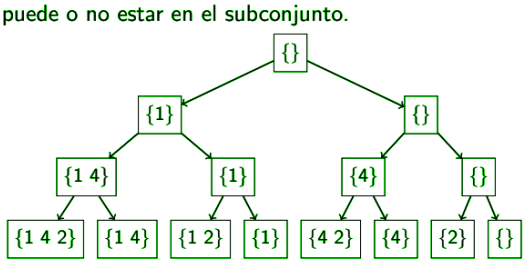
Ejemplo (CD):

Enunciado

Tenemos un CD que soporta hasta P minutos de música, y dado un conjunto de N canciones de duración p_i (con $1 \leq i \leq m$, y $p_i \in \mathbb{N}$) queremos encontrar la mayor cantidad de minutos de música que podemos escuchar.

Con $P = 5$ y una lista de $N = 3$ canciones con duraciones [1, 4, 2] la solución es 5.

- Cada elemento puede o no estar en el subconjunto.



- Cada nodo interno del nivel i representa un subconjunto de los primeros i elementos. Por ende para extender cada solución se agrega o no el elemento $i + 1$.

Algorithm $BT_{CD}(a, i)$ // a es una solución parcial

```
1: if i = N then
2:     if suma(a) ≤ P & suma(a) > mejorSuma then
3:         mejorSuma ← suma(a)
4:     end if
5: else
6:     BT_CD(a ∪ p_i, i + 1)
7:     BT_CD(a, i + 1)
8: end if
```

- ¿Qué podas podemos usar en CD?
- En CD podemos dejar de avanzar si la solución parcial ya superó N (**factibilidad**).
- También podemos ver si poniendo todas las canciones restantes no nos excedemos de k. Si ese es el caso, la mejor solución desde donde estamos es agregar todo (**optimalidad**).

Otra forma de pensarlo:

Plantear funciones de esta pinta nos va a ser muy útil cuando hagamos programación dinámica. Intuitivamente, el árbol de recursión de esta función es el mismo que el de los subconjuntos. Sin embargo, en esta formulación queda claro que no importan qué elementos se fueron eligiendo, sino la suma de los pesos.

El problema se puede pensar de otra forma: si quiero llegar a P, y agrego el primer elemento al CD...

- ¿A cuánto se quiere llegar con el resto de los temas?
- ¿Y si no se lo agrega?
- Si no quedan temas (N=0), ¿Qué valores de P son válidos?

$$CD(i,k) = \begin{cases} -\infty & \text{si } i = N \text{ y } k < 0 \\ 0 & \text{si } i = N \text{ y } k \geq 0 \\ \max(CD(i+1,k), CD(i+1,k-p_i)+p_i) & \text{cc} \end{cases}$$

'La máxima cantidad de música que puedo obtener sin exceder k minutos empleando las canciones desde i hacia delante'

Ejemplo (Prime Ring):

Prime Ring

Dados N números naturales $p_0, \ldots, p_{N-1}$, con $1 < p_i < 10N$, queremos saber cuántas permutaciones j de ellos hay que cumplan que $p_{j_i} + p_{(j_{i+1} \mod n)}$ sea primo para todo $0 \leq i \leq n-1$

- Lo vamos a resolver con backtracking.
- ¿Cuál es el espacio de búsqueda? Todas las permutaciones de los naturales $p_0, \ldots, p_{N-1}$.
- ¿Cuáles son las soluciones parciales? Los prefijos de las permutaciones.
- ¿Cuál es la operación de extensión? Agregar un elemento al prefijo de permutación.

Ejemplo:

- Permutación (1, 2, 3):
  - $1+2=3$ (primo)
  - $2+3=5$ (primo)
  - $3+1=4$ (no es primo)
  - Esta permutación no cumple con la condición.
- Permutación (2, 3, 1):
  - $2+3=5$ (primo)
  - $3+1=4$ (no es primo)
  - $1+2=3$ (primo)
  - Esta permutación no cumple con la condición.

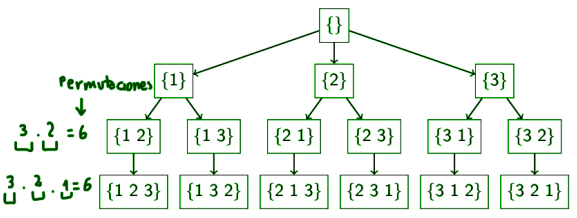
La función que hay que implementar entonces es:

$$primeRing(I) = \begin{cases} esValida(I) & \text{si } |I| = N \\ \sum_{p_i \notin I} primeRing(I \oplus p_i) & \text{cc} \end{cases}$$

'La cantidad de permutaciones que extienden a I, usan todos los elementos de p y generan un anillo de primos'

La solución al problema es $primeRing(\{\})$

- Árbol para $n = 3$, si $p = [1, 2, 3]$.



Si agregamos podas

- Podríamos verificar la condición de primalidad durante la selección de sucesores.
- El árbol cambia: ahora las soluciones parciales son las permutaciones de los subconjuntos que cumplen la condición de primalidad.
- ¿Hay que verificar algo en las hojas?
- En las hojas solo tenemos que verificar que cierre bien el anillo.

La función queda entonces como

$$primeRing(I) = \begin{cases} esPrimo(ultimo(I) + primero(I)) & \text{si } |I| = N \\ \sum_{\substack{p_i \notin I \\ esPrimo(p_i + ultimo(I))}} primeRing(I \oplus p_i) & \text{cc} \end{cases}$$

💡 Solo verifico el primero y último porque en cada paso siempre que agregó un elemento antes de continuar con su sucesor, me fijo que la suma de primo.

Ejemplo (Sudoku):

Enunciado

Dado un tablero de Sudoku de $N \times N$ con algunas casillas ocupadas hay que decidir si se lo puede completar siguiendo las reglas del Sudoku.

- ¿Cuáles son las soluciones parciales? Tableros incompletos.
- ¿Cuál es la función de extensión? Colocar un valor en algún casillero.
- ¿Qué verificamos en las hojas? Revisamos si es un tablero válido.

si estoy en un casillero ya completo

$$sudoku(T,(i,j)) = \begin{cases} esValido(T) & \text{si } i = N \\ sudoku(T, sig(i,j)) & \text{si } T[i][j] \neq 0 \\ \bigvee_{1 \leq k \leq N} sudoku(T \oplus ((i,j) \to k), sig(i,j)) & \text{cc} \end{cases}$$

las casillas son elegidas al azar

Esta función toma un tablero (parcialmente completado) y un índice del mismo, y "prueba" todas las formas de llenar ese casillero y pasa al siguiente (donde el siguiente es el casillero a su derecha o bien el primero de la siguiente fila, si nos encontramos al borde).
La solución es $sudoku(T,(0,0))$

- ¿Qué podas podemos implementar?
- No pongamos números que ya están prohibidos. Para eso revisamos las filas, columnas y subcuadrados en cada paso.
- ¿Hace falta ir en orden?
- No, usemos siempre la posición mas condicionada (o sea, la posición en la cual hay una menor cantidad de opciones válidas restantes).

❖ Branch and Bound

El *branch and bound* (ramificación y acotación) es una técnica de algoritmos usada para resolver problemas de optimización combinatoria. Este método explora de manera sistemática todas las posibles soluciones (similar al *backtracking*), pero con una diferencia clave: en lugar de explorar todas las soluciones posibles, el algoritmo utiliza límites (acotaciones) para descartar grandes porciones del espacio de búsqueda que no pueden contener la solución óptima.

PROGRAMACIÓN DINÁMICA

La programación dinámica es una **técnica de diseño de algoritmos utilizada para resolver problemas complejos dividiéndolos en subproblemas más simples y almacenando los resultados de estos subproblemas para evitar cálculos repetidos.** Es especialmente **útil en problemas** de optimización **donde hay una superposición de subproblemas**, es decir, donde **los mismos subproblemas se resuelven varias veces.**

La idea principal es descomponer el problema original en partes más pequeñas, resolver cada una de esas partes solo una vez, y **almacenar sus soluciones en una tabla** (o matriz)**, de modo que si se necesita resolver el mismo subproblema nuevamente, se pueda obtener la solución directamente de la tabla, en lugar de recalcularla.** Se memorizan las llamadas recursivas con los parámetros dados como índices de la matriz o vector, donde se mapean los valores previamente calculados.

Superposición de estados → **El árbol de llamadas recursivas resuelve el mismo problema varias veces.**

- **Ejemplo.** Cálculo de coeficientes binomiales. Si $n \geq 0$ y $0 \leq k \leq n$, definimos

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- **Teorema.** Si $n \geq 0$ y $0 \leq k \leq n$, entonces

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0 \text{ o } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \end{cases}$$

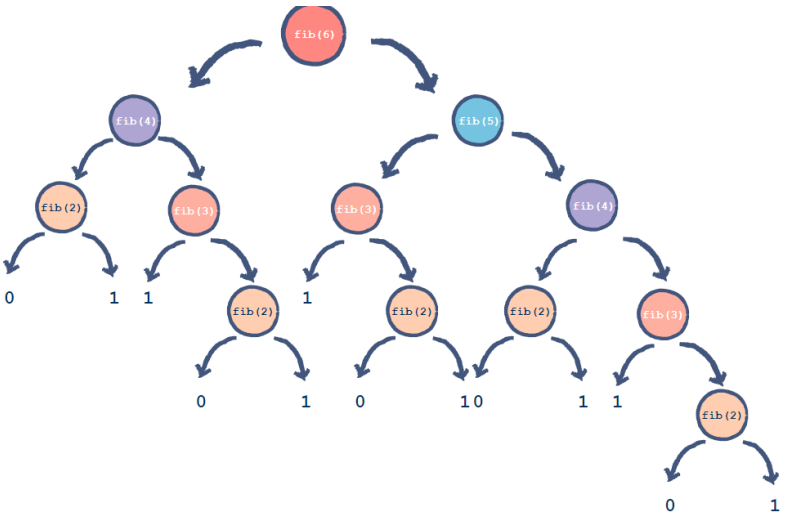
	0	1	2	3	4	⋯	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
⋮	⋮					⋱		
$k-1$	1						1	
k	1							1
⋮	⋮							
$n-1$	1							
n	1							

```
algoritmo combinatorio(n,k)
    entrada: dos enteros n y k
    salida: (n k)

    para i = 1 hasta n hacer
        A[i][0] ← 1
    fin para
    para j = 0 hasta k hacer
        A[j][j] ← 1
    fin para
    para i = 2 hasta n hacer
        para j = 1 hasta min(i − 1,k) hacer
            A[i][j] ← A[i − 1][j − 1] + A[i − 1][j]
        fin para
    fin para
    retornar A[n][k]
```

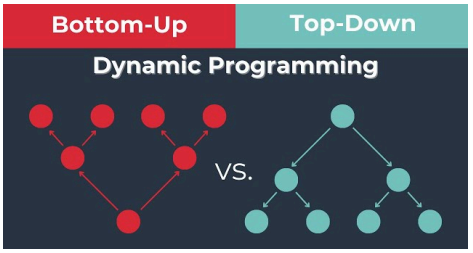
💡Tenemos mapeados todos los valores de la función (bottom-up), entonces en lugar de calcular cada coeficiente binomial, solamente le enviamos dos parámetros a la matriz, y este me devuelve el valor calculado.

- Función recursiva:
  - Complejidad $\Omega(\binom{n}{k})$.
- Programación dinámica (bottom-up):
  - Complejidad $O(nk)$.
  - Espacio $\Theta(k)$: sólo necesitamos almacenar la fila anterior de la que estamos calculando. No es posible con top-down
  - Se podría mejorar un poco sin cambiar la complejidad aprovechando que $\binom{n}{k} = \binom{n}{n-k}$.



TOP-DOWN & BOTTOM-UP

En programación dinámica, los dos **enfoques principales para evitar la repetición de cálculos son top-down y bottom-up.** Ambos enfoques tienen el mismo objetivo: resolver el problema original de manera eficiente, pero difieren en la forma en que abordan los subproblemas.



- **Top-Down** (memorización) **es útil cuando** el problema tiene una estructura recursiva natural y cuando **algunos subproblemas no se necesitan calcular, ahorrando tiempo. Resuelve problemas recursivamente y almacena resultados para evitar repeticiones.**
  💡Usualmente es recursivo, porque no se cuando termino de resolver cada rama. En cada paso recursivo voy guardando en un vector o matriz los resultados y los uso cada vez que necesite calcular algo ya calculado.

- **Bottom-Up** (tabulación) **es preferible cuando todos los subproblemas deben resolverse**, y puede ser más eficiente en términos de espacio porque no requiere mantener la pila de recursión. **Resuelve iterativamente desde los subproblemas más pequeños hasta el problema original.**
  💡Usualmente es iterativo, porque se desde donde quiero empezar y donde quiero terminar. En BottomUp calculo todo (desde el caso inicial hasta el final) y voy guardando cada cálculo en un vector o matriz y luego devuelvo el valor guardado en la matriz o vector con los parámetros.

  https://www.log2base2.com/algorithms/dynamic-programming/dynamic-programming.html (another explanation)

▶ Un algoritmo de programación dinámica evita estas repeticiones con alguno de estos dos esquemas:

1. **Enfoque top-down.** Se implementa recursivamente, pero se guarda el resultado de cada llamada recursiva en una estructura de datos (memorización). Si una llamada recursiva se repite, se toma el resultado de esta estructura.
2. **Enfoque bottom-up.** Resolvemos primero los subproblemas más pequeños y guardamos (habitualmente en una tabla) todos los resultados.

Considera nuevamente el problema de la serie de Fibonacci:

- **Top-Down:** Comenzarías intentando calcular $F(n)$. Si $F(n)$ depende de $F(n-1)$ y $F(n-2)$, calcularías esos valores recursivamente. Si ya has calculado $F(n-1)$, lo reutilizarías desde la memoria en lugar de recalcularlo.
- **Bottom-Up:** Comenzarías calculando $F(0)$ y $F(1)$, almacenando esos valores. Luego calcularías $F(2)$ utilizando $F(0)$ y $F(1)$, y seguirías hasta llegar a $F(n)$.

DIVIDE & CONQUER

Divide and conquer (divide y vencerás) es una estrategia que **consiste en dividir un problema grande en subproblemas más pequeños y similares del mismo tipo de problema, resolver cada uno de estos subproblemas de forma independiente, y finalmente combinar las soluciones** de los subproblemas para obtener la solución del problema original.

Este enfoque **se utiliza** principalmente **cuando un problema puede ser descompuesto en partes más pequeñas** que son más fáciles de manejar y cuando las soluciones a los subproblemas se pueden combinar eficazmente.

Pasos del Algoritmo Divide and Conquer:

- División: **Divide el problema original en varios subproblemas** más pequeños que son instancias del mismo tipo de problema. **Las subpartes tienen que ser más pequeñas.**
- Conquista: **Resuelve los subproblemas de forma recursiva. Si los subproblemas son** lo suficientemente **pequeños**, se resuelven directamente (sin necesidad de más divisiones). **Las tareas a resolver deben ser la misma para todo problema y subproblema.**
- Combinación: **Combina las soluciones de los subproblemas** para formar la solución del problema original.

Un ejemplo clásico del uso de divide and conquer es el algoritmo de Merge Sort:

- División: Divide la lista de elementos a ordenar en dos mitades.
- Conquista: Ordena cada mitad de manera recursiva utilizando el mismo método.
- Combinación: Combina las dos mitades ordenadas en una única lista ordenada.

Otros usos, además de ordenación (Merge Sort y Quick Sort), se utiliza para la multiplicación de matrices (El algoritmo de Strassen, que mejora la multiplicación de matrices tradicionales, utiliza esta técnica), búsqueda en listas (Como en la búsqueda binaria, que divide la lista en mitades para localizar un elemento) entre otros.

Recursos:

Tanto la división como la combinación en divide and conquer implican algún costo. Sin embargo, estos costos no deben ser tan elevados como para que la ventaja de resolver subproblemas más pequeños se pierda. **El éxito de un algoritmo divide and conquer depende de que las fases de división y combinación sean relativamente eficientes en comparación con la solución directa del problema completo.**

- El costo de dividir el problema viene por el lado de que podrías necesitar algún procesamiento, como seleccionar puntos de partición o crear nuevas instancias de subproblemas.
- Combinar las soluciones parciales también implica un costo. En el caso de Mergesort, por ejemplo, hay un proceso de fusión que requiere tiempo lineal respecto al tamaño de las listas.

❖ Algoritmo de D&C

- $F(X)$
  - Si X es suficientemente chico o simple, solucionar de manera ad hoc.
  - Si no,
    - Dividir a X en $X_1, X_2, \ldots, X_k$
    - $\forall i \leq k$, hacer $Y_i = F(X_i)$
    - Combinar los Y_i en un Y que es una solución para X.
    - Devolver Y

F(X) → Es la **función** principal **que aplica la técnica divide and conquer al problema X.**

Caso Base → "Si X es suficientemente chico o simple, solucionar de manera ad hoc". Aquí, **el algoritmo verifica si el problema X es lo suficientemente pequeño o simple como para resolverlo directamente sin dividirlo en subproblemas.**

💡Una solución ad hoc es una solución específica, directa y simple para un problema particular, utilizada cuando el problema es lo suficientemente pequeño o simple para no requerir un enfoque más general o elaborado.

División y Conquista → **Si X no es lo suficientemente pequeño o simple, se procede a dividir el problema.** Se divide el problema X en k subproblemas más pequeños (X_1, ..., X_k). **Para cada subproblema Xi , se aplica la misma función F, es decir, se resuelve recursivamente cada subproblema. Aquí es donde entra en juego la "conquista".**

Combinar → **Una vez que se han resuelto los subproblemas, las soluciones (Yi) se combinan para formar la solución final Y del problema original X.**

❖ Complejidad

- El costo de un algoritmo D&C de tamaño n se puede expresa como $T(n)$, que debe considerar:
  - Dividir el problema en a subproblemas de tamaño máximo n/c siempre que $n/c > n_0$.
  - El costo de efectivamente hacer la subdivisión y luego unir los resultados.
  - Además hay que resolver los subproblemas: $aT(n/c)$.
  - Vamos a utilizar otra función $g(n)$ tal que $g(n) \geq T(n)$ para cualquier valor de n.
  - Sea $b'n^d$, para algún d, una cota superior del costo de dividir en subproblemas y combinar los resultados para un problema de tamaño n. Definimos $g(1) = b = max\{b', T(1)\}$.
- Es decir, $T(n) = aT(n/c) + b'n^d \leq g(n) = ag(n/c) + bn^d$.

1. Forma General de la Recurrencia:

La complejidad de un algoritmo *divide and conquer* se expresa generalmente como una función de recurrencia, denotada como $T(n)$, que describe el tiempo que toma resolver un problema de tamaño n.

Recurrencia básica: $T(n) = a \cdot T\left(\frac{n}{c}\right) + b' \cdot n^d$

Donde:

- a: Número de subproblemas en los que se divide el problema original.
- $\frac{n}{c}$: Tamaño de cada subproblema, donde c es la constante que indica en cuánto se reduce el tamaño del problema.
- $T\left(\frac{n}{c}\right)$: Tiempo que toma resolver cada subproblema.
- $b' \cdot n^d$: Tiempo adicional necesario para dividir el problema y combinar los resultados de los subproblemas.

2. Explicación de Cada Parte:

- **Dividir el problema:** El costo de dividir un problema de tamaño n en subproblemas de tamaño $\frac{n}{c}$ se incluye en la expresión. La función $T(n)$ considera esto y asume que $n/c > n_0$, donde n_0 es un valor pequeño por debajo del cual los problemas se resuelven de manera directa.
- **Costo de subdivisión y combinación:** Este costo no es nulo (como ya discutimos), y se añade como una función adicional, que en este caso se modela como $b' \cdot n^d$. Esta parte de la expresión representa el tiempo que lleva dividir el problema y luego combinar las soluciones de los subproblemas.
- **Resolver los subproblemas:** Aquí, se consideran a subproblemas, cada uno de tamaño $\frac{n}{c}$, con un costo de resolución $T\left(\frac{n}{c}\right)$. La función de recurrencia acumula este costo en cada paso de la recursión.

3. Cota Superior y Función g(n):

- La función $g(n)$ se introduce como una cota superior para $T(n)$. Esto significa que $g(n)$ es una función que siempre es mayor o igual que $T(n)$, para cualquier tamaño de n.
- **Definición de $g(n)$**: En el texto, $g(n)$ se define como la suma del costo de resolver los subproblemas y el costo de combinarlos:

$$g(n) = ag\left(\frac{n}{c}\right) + b \cdot n^d$$

4. Interpretación del Resultado Final:

La recurrencia muestra que el tiempo total $T(n)$ depende tanto del costo de resolver los subproblemas (a través de la función $T\left(\frac{n}{c}\right)$) como del costo de dividir y combinar (representado por $b' \cdot n^d$). El objetivo al analizar esta recurrencia es encontrar cómo crece $T(n)$ a medida que n aumenta, lo cual ayuda a determinar la complejidad del algoritmo.

- Supongamos que $n = c^k$ para algún k y analicemos la recurrencia.
- $T(n) \leq g(n) = ag(n/c) + bn^d$
- $= g(c^k) = ag(c^{k-1}) + b(c^k)^d$
- $= a(ag(c^{k-2}) + (b\ c^{(k-1)d})) + bc^{kd}$
- $= a^2 g(c^{k-2}) + abc^{(k-1)d} + bc^{kd}$
- $= a^3 g(c^{k-3}) + a^2 b(c^{k-2})^d + abc^{(k-1)d} + bc^{kd}$
- $\ldots$
- $= a^j g(c^{k-j}) + \Sigma_{i=0}^{j-1} a^i bc^{(k-i)d}$

La programación dinámica está relacionada con otros paradigmas de diseño de algoritmos como divide y vencerás y greedy. Mientras que divide y vencerás también descompone problemas, no reutiliza los subproblemas, y los algoritmos greedy eligen la mejor opción localmente en cada paso sin mirar el problema completo. En cambio, la programación dinámica explora todas las posibles soluciones parciales y las combina de manera óptima.

GREEDY