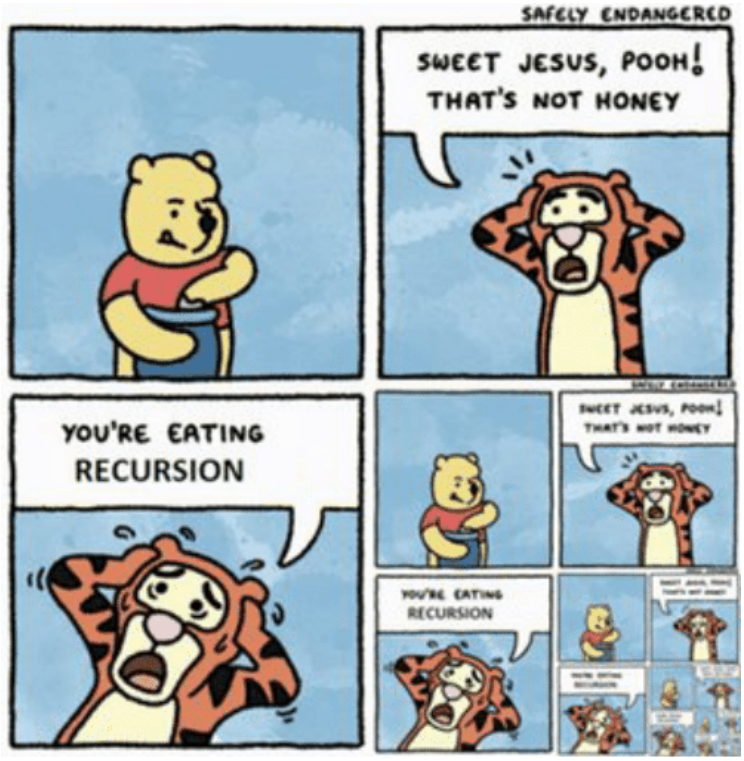
Ayudantía 4 - Algoritmos y Complejidad
Recursion, Backtracking and shitte

Sassy Complexes



La forma clasica de implementar la operacion $x!(factorial)$ es de forma recursiva, o sea, definir la solucion en terminos de si misma, por ejemplo, para calcular el factorial de 5(5!) usted se preguntaria:

- Yo soy el 5 y no conozco mi factorial, pero se que me puedo multiplicar con el factorial del 4, entonces le pregunto al 4 por su factorial.
- ... Y el 4 dice, "Yo soy el 4 y no conozco mi factorial, pero se que me puedo multiplicar con el factorial del 3, entonces le pregunto al 3 por su factorial".

$$\vdots$$

Así descubriamos que el factorial de 5 es $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

Simples son de pensar los algoritmos recursivos, el resto del curso se trata de esto; Pero, como el nombre del ramo lo dice: ¿Cual es su complejidad? Lo revisaremos mas tarde, antes:

1. Reducción - Magia funcional

Como lo hicimos con el factorial, ¿que hacer cuando mi problema es demasiado dificil? lo **reduzco** a un problema mas simple ($5! \rightarrow 4!$) y no me importa lo dificil que sea $4!$, el verá como se las arregla, y sabemos que $4!$ garantiza una respuesta correcta (*para usar* $4!$ *como caja negra, debemos asegurarnos que esta funcionando bien primero*). Esta es la magia *funcional*

2. Lo que aprendimos de Hoffmann

Todos los lenguajes y todos los sistemas operativos (sanos) que manejan la memoria, la manejan implementando las estructuras *Stack* y *Heap* (ya deberia saber que son). Cada vez que una funcion es llamada, se reserva un bloque de memoria en el stack, para ejecutar esa funcion.

Existe una optimización muy divertida llamada *tail call optimization (TCO)*, veamos:

Listing 1: No TCO

```
1 def f(text):
2     print(text)
3     return f("sweet jesus, pooh! that's not honey") + f(text)
```

Listing 2: TCO

```
1 def f(text):
2     print(text)
3     return f("sweet jesus, pooh! that's not honey" + text)
```

Algunos lenguajes/compiladores (**NO TODOS!** (*cough*C#*cough*Python*cough*)) la realizan; Por lo tanto, ese ejemplo de Python en realidad es solo demostrativo, por que no va a hacer la optimización.

3. Y su complejidad?

Listing 3: Factorial Recursivo Tradicional

```
1 int factorial(int n) {
2    if (n <= 1){
3        return 1;
4    }else {
5        return n * factorial(n-1);
6    }
7 }
```

Analicemos: Podemos escribir una relacion de recurrencia para el total de trabajo hecho. Como caso base: "tu haces una unidad de trabajo cuando el algoritmo corre con un input de tamaño 1", entonces:

$$T(1) = 1$$

Para un input de tamaño $n+1$,tu algoritmo hace una unidad de trabajo junto con la función misma, entonces:

$$T(n+1) = T(n) + 1$$

Expandiendo:

$$\begin{aligned} T(1) &= 1 \\ T(2) &= T(1) + 1 = 2 \\ T(3) &= T(2) + 1 = 3 \\ T(4) &= T(3) + 1 = 4 \\ &\dots \\ T(n) &= n \end{aligned}$$

Que es como:

$$T(n) = n = 2^{\log_2 n} = O(n)$$

Se ve bonito, pero es (casi)mentira. Si el input es entendido como un número, efectivamente la complejidad es polinomial. Aunque formalmente "tiempo polinomial" esta definido respecto al número de bits usados para el input. Aqui, el input n puede se especificado en $\Theta(\log n))$ bits, entonces, el tiempo de ejecución $2^{\log_2 n}$ puede ser considerado como **exponencial**.

4. Church-Turing Thesis

En las palabras mas simples que se pueden inventar, nos dice que el calculo-λ es equivalente a las maquinas de Turing, y viceversa, si extrapolamos eso, tenemos que las funciones recursivas existen por que existen las funciones iterativas, y lo mismo al reves. Entonces:

4.0.1. ¿Hay una version del Factorial iterativa?

Listing 4: Esta!

```
1 int factorial(int n){
2     int acc = 1;
3     for(int i=1; i<=n; i++){
4         acc *= i;
5     }
6     return acc;
7 }
```

De la misma forma, puede transformar cualquier función iterativa en una recursiva, por ejemplo "sumar los elementos de una lista" (Python's `sum`)

Listing 5: sum

```python
def suma_iterativa(A):
    acc = 0
    for a in A:
        acc += a
    return acc

def suma_recursiva(A, acc=0):
    if not A:
        return acc
    return suma_recursiva(A[1:], acc + A[0])
```

5. Backtracking

La idea de backtracking puede entenderse como "llevar el rastro" (*lit.* Back-Track), tiene mucho que ver con la idea de resolver laberintos, recuerde la antigua leyenda griega del heroe Theseus que entro al laberinto del minotauro con un hilo magico con el que iba dejando el rastro para despues poder salir, backtracking sigue la misma idea, uno va resolviendo el problema de forma recursiva, pero si comete un error, regresamos atras y probamos otra alternativa. Probamos todas las alternativas posibles, **una por una**.

```
BT(A, PROBLEM):
    IF A es solucion a PROBLEM:
        RETURN A
    ELSE:
        computar S' (usando  A y PROBLEM)
        WHILE |S'| > 0:
            a_i = algun elemento de S'
            borrar a_i de S'
            sol = BT( A + [a_i], PROBLEM)
            IF sol IS NOT NULL:
                añadir sol a conjunto de soluciones factibles o retornar
        RETURN NULL
```

A seria la serie de decisiones que he tomado y S' las decisiones que puedo tomar (a veces calcularla usando a para no pisarse la cola)
En el fondo solo se trata de una BUSQUEDA EN PROFUNDIDAD, pero teniendo en cuenta las decisiones tomadas, en primer lugar porque impliocan restricciones diferentes para los subproblemas y en segundo lugar para evitar entrar en bucles. Vea la animacion en Wikipedia del Sudoku (Intente programarla tambien, es facil)

6. Branch and Bound

Tomando la misma idea de Backtracking, pero en vez de hacer una busqueda exhaustiva, dividimos el espacio de busqueda en espacios mas pequeños, y los vamos probando en casos limites, si de verdad vamos consiguiendo una mejora, seguimos expandiendo este espacio de busqueda hasta resolver el problema completo.

6.1. $A^\star$

Suponga que quiere buscar un camino de un punto a otro, penso en Dijkstra. Ahora agregue obstaculos, terreno pasable, terreno impasable, dificultades, etc. Dijkstra si bien sirve se vuelve un poco lento pues explora todo el grafo antes de entregar la respuesta. $A^\star$ se puede pensar como una mejora de Dijkstra, pues utiliza una heuristica para moverse rapidamente a traves del grafo. $A^\star$ no necesariamente es un algoritmo voraz como Disjktra, o DP o Backtracking, sino mas bien es un menjunje de muchas cosas. (Es una algoritmo de busqueda Informado) (Aunque alguien una vez demostró una versión general de B&B que resumia $A^\star$, por eso se pasa esta cosa aquí) En cada iteracion se busca determinar cual de los caminos parciales avanzar. Lo hace estimando el costo de llegar hasta el objetivo desde donde estamos actualmente, es decir, se busca minimizar:

$$f(n) = g(n) + h(n)$$

donde n es el nodo que estamos explorando actualmente, g (n) es el costo desde el inicio hasta n y h(n) es una heuristica que estima el costo del supuesto camino mas corto desde n hasta el objetivo. La heuristica depende del problema y debe ser una heuristica admisible, es decir, nunca sobreestima el costo de ir al objetivo. Una implementacion tipica utiliza una cola de prioridad para seleccionar los nodos de costo minimo estimado. Esta cola se conoce como el Set Abierto o Frontera. En cada iteracion el nodo con el menor f (x) es removido y los f y g de sus vecinos son actualizados. El algoritmo continua hasta haber encontrado el f minimo o se vacia la cola. Entonces f valdra el largo del camino en el objetivo, puesto que la heuristica es 0 en el destino. Notese que estamos expandiendo una frontera, utilizando la heuristica para expandirnos de una forma mas **inteligente** hacia el objetivo, note que si $h(x) = 0 \forall x$ estamos corriendo el algoritmo de Dijkstra.

6.1.1. La heuristica

depende del problema, pero siempre puede usar una metrica de distancia tradicional como:

Euclides:

$$h(A,B) = \sqrt{(A_x - B_x)^2 + \left(A_y - B_y\right)^2}$$

Manhattan:

$$h(A,B) = |A_x - B_x| + \left|A_y - B_y\right|$$

Chebyshev:

$$h(A,B) = \max\left\{|A_x - B_x|, \left|A_y - B_y\right|\right\}$$

Octile:

$$h(A,B) = \max\left\{A_x - B_x, A_y - B_y\right\} - \left(\sqrt{2} - 1\right) \cdot \min\left\{A_x - B_x, A_y - B_y\right\}$$

6.1.2. Vease las animaciones:

- https://www.redblobgames.com/pathfinding/a-star/introduction.html
- https://qiao.github.io/PathFinding.js/visual/
- http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

7. Ejercicios

1. Extienda la `suma_recursiva` para aceptar cualquier tipo de operador(*callable*), implementando la función fold izquierdo

Listing 6: Haskell's foldl in Python3

```
    def reduce(func, list, initval=0):
        if not list: return initval
        return reduce(func, list[1:], func(initval, list[0]))
```

Listing 7: Haskell's foldl in Haskell

```
        foldl :: (b -> a -> b) -> b -> [a] -> b
        foldl f z []     = z
        foldl f z (x:xs) = foldl f (f z x) xs
```

2. **#SPOILER: DP** Transforme una función que retorne cierto elemento de la secuencia de Fibonacci a versión iterativa

Listing 8: Fibonacci iterativo

```
int fib(int n){
    int prev = 1;
    int curr = 0;
    int next;
    for (int i = 1; i <= n; ++i){
        next = curr + prev;
        prev = curr;
        curr = next;
    }
    return curr;
}
```

3. esboze un algoritmo de *Backtracking* para, dada una matriz representando un laberinto, retorne la ruta que se debe seguir para salir.
   Pensó en el algoritmo de Tremaux, nosotros si[1]

4. Dado un multiconjunto de enteros, ¿existe un subconjunto no vacio cuya suma sea un numero X? Por ejemplo: para el conjunto $\{-7,-3,-2,5,8\}$ y con objetivo $X=0$ la respuesta es **sí** puesto que $\{-3,-2,5\}$ suman 0. Plantee un algoritmo de backtracking para responder si o no.
   Algo bien simple, tomamos el primer elemento y lo probamos recursivamente contra los demas, al elegir el primer elemento $\hat{p}$ quedan dos opciones:

   - Existe un subset que contiene a $\hat{p}$ y suma X
   - Existe un subset que no contiene a $\hat{p}$ y suma X

[1] https://youtu.be/czChNtKn8l8?t=111

Listing 9: ¿Como puede hacer que retorne el subset encontrado?

```
#! /bin/env python3
def subset_sum(A, X):
    if sum(A) == X:
        # Si estan sumando X...
        return True
    elif len(A) == 0:
        # Conjunto vacio no permitido
        return False
    else:
        # Existe un subset que incluye A[0] o no lo incluye y suma X
        return subset_sum(A[1:], X-A[0]) or subset_sum(A[1:], X)
```

5. A fines del año 2018, el supremo líder inicio una guerra nuclear que destruyo gran parte del mundo en represalia contra un grupo de ingenieros que no pudo escribir un calzador de expresiones regulares que pudiera escapar cualquier caracter seguido de un \. En este contexto, *Kenshiro* es un superviviente y guerrero heredero de un antiguo arte marcial que consiste en tocar los puntos de presión específicos en el cuerpo del enemigo para matarlo.[2].
   Kenshiro ha modelado el cuerpo de sus enemigos como una matriz `booleana` de tamaño $N \times M$, en la cual, al tocar un punto de presión, dicho punto y **todos los otros puntos en esa fila y columna cambian de estado** (si estaba apagado se enciende, si estaba encendido se apaga). Kenshiro necesita encontrar los puntos de presión específicos que debe tocar en el cuerpo enemigo para desactivar completamente todos ellos. Kenshiro sabe que debe usar **Backtracking**. Ayude a Kenshiro a continuar su ola de cinematográficas matanzas diseñando un algoritmo que encuentre los puntos que se deben presionar para apagar todos los puntos del enemigo **usando backtracking**.

6. Para mas ejercicios, dirigase al conglomerado, publicado en moodle.

[2] https://youtu.be/YSgpU70MZno