1. El siguiente algoritmo obtiene el índice de un elemento x en el arreglo **ordenado** A de n datos, entre los índices i y f. Si el elemento x no está en $A[i, f]$, entonces retorna -1.

```
search(A, x, i, f):
    if f < i:
        return -1
    m = ⌊(i+f)/2⌋
    if A[m] > x:
        return search(A, x, i, m-1)
    else if A[m] < x:
        return search(A, x, m+1, f)
    else:
        return m
```

a) **[3pt]** Demuestra que este algoritmo es correcto. En particular,
   a. **[1pt]** Que termina.
      Solución: Dado un intervalo inicial $[i, f]$, se define $d = f - i$ como el tamaño del intervalo.
      En cada llamada recursiva de ***search*** el tamaño del intervalo disminuye al menos a la mitad.
      El algoritmo llega a su fin cuando el tamaño del intervalo es 1.
      Eventualmente al dividir el tamaño del intervalo a la mitad se va a cumplir que el nuevo intervalo tiene solo 1 dato. En este momento el algoritmo termina.
      Ojo: Esta no es la única manera de demostrarlo, esto es solo un ejemplo.
   b. **[2pt]** Que si x está en A, entonces el algoritmo encuentra el índice en el que está.
      Solución: Se definen 2 conjuntos: R inicialmente vació y P inicialmente con todos los datos del arreglo. En R se va a almacenar todos los datos que el algoritmo deja de considerar al momento de achicar el intervalo y en P se almacenan los datos que aún están en el intervalo de búsqueda.
      Si el dato a buscar x está en el intervalo inicial entonces $x \in P$ al inicio del programa.
      Si ejecutamos ***search*** se pueden dar 2 casos: el valor en la posición m es x, en este caso el algoritmo retorna la posición m y termina correctamente encontrando la posición. En el caso de que en m no esté x el algoritmo disminuye el intervalo a la mitad de su tamaño, lo que quiere decir que los datos que ya no están en el intervalo pasan del conjunto P al conjunto R. Además, se cumple que el dato x se mantiene en R ya que la mitad del intervalo escogida es la que contiene a x.
      Dado lo anterior el elemento x no sale del conjunto P hasta que la posición m sea exactamente la posición de x.
      Dijimos en el ejercicio pasado que el algoritmo es finito, por lo tanto eventualmente si no se encuentra al elemento x el conjunto P va a quedar solo con el valor x, lo que se traduce a que el intervalo a revisar por el algoritmo es solo la posición de x. Por lo tanto el algoritmo termina retornando la posición del elemento a buscar.

Ojo: no es la única manera de demostrarlo, este es solo un ejemplo.

b) **[1.5pt]** Plantea la recurrencia **exacta** del tiempo $T(n)$ que toma el algoritmo.
Solución: El algoritmo en el peor caso no va a encontrar el elemento a buscar hasta el último paso. Por lo tanto $T(1) = 1$. En el resto de los casos el algoritmo revisa la posición m dividiendo el intervalo en 2 mitades: la que tiene el elemento x y la que no. En el peor caso la mitad que tiene el elemento a buscar es la mitad más grande (puede ser que una mitad tenga un elemento más que la otra). Además, la posición m no queda en ningún intervalo. Por lo tanto, la recurrencia es:

$$T(n) = \begin{cases} 1, & n = 1 \\ T\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + 1, & n > 1 \end{cases}$$

En el caso de que no de ponga el techo, el -1 o el caso base, se descuenta 0.25 (por cada uno).

c) **[1.5pt]** Obtén la complejidad de este algoritmo **formalmente** a partir de esta recurrencia.
Solución: Para resolver la recurrencia primero acotamos $T(n)$ por arriba por la siguiente expresión:

$$T(n) \leq T'(n) = \begin{cases} 1, & n = 1 \\ T'\left(\frac{n}{2}\right) + 1, & n > 1 \end{cases}$$

Ahora vamos a calcular el valor de $T(2^k)$ donde $n \leq 2^k < 2n$:

$$T'(2^k) = T'\left(2^{k-1}\right) + 1$$

Desarrollando varias iteraciones:

$$T'\left(2^k\right) = T'\left(2^{k-2}\right) + 1 + 1$$
$$T'\left(2^k\right) = T'\left(2^{k-3}\right) + 1 + 1 + 1$$
$$T'\left(2^k\right) = T'\left(2^{k-i}\right) + i$$

Cuando $i = k$:

$$T'\left(2^k\right) = T'(1) + k$$

Sabemos que $T'(1) = 1$ y que $2^k < 2n$ por lo tanto $k < \log_2 2n$:

$$T(n) \leq T'(n) \leq T'\left(2^k\right) < 1 + \log_2 2n \in O(\log n)$$

Si se usa la recurrencia simple sin justificar la simplificación o si se calcula usando 2^k sin definir correctamente se descuenta 0.25 (por cada uno).

2. Tienes a tu disposición 4 ejecutables compilados E_1, E_2, E_3, E_4 correspondientes a 4 algoritmos de ordenación: InsertionSort, SelectionSort, QuickSort con pivote aleatorio, QuickSort con pivote del primer elemento. El problema es que no sabes cual archivo corresponde a que algoritmo. Para descubrir a qué algoritmo corresponde cada archivo creas 3 archivos con 1 millón de números para ordenar: El primer archivo tiene los números ya ordenados, el segundo tiene los números de manera aleatoria y el tercero tiene los números ordenados en el orden inverso. Al ejecutar cada programa con cada set de números obtienes la siguiente tabla de tiempos:

Archivo	Datos ordenados	Datos aleatorios	Datos al revés
E_1	44.88s	48.72s	43.35s
E_2	40.87s	9.97s	43.22s
E_3	7.35s	25.47s	46.11s
E_4	10.05s	9.81s	10.56s

[6pt] Diga cuál archivo corresponde a qué algoritmo y justifique cómo llegó a esa conclusión a partir de los datos obtenidos.

Solución: De los algoritmos dados sabemos sus mejores y peores casos con sus complejidades respectivas. En particular sabemos que:

i. InsertionSort toma tiempo lineal cuando los datos están ordenados, toma tiempo cuadrático cuando los datos están en el orden inverso y toma tiempo cuadrático en el caso promedio. Además, sabemos que el caso en el que realiza más intercambios es cuando el arreglo está al revés ya que es el caso en el que todos los pares están invertidos.

ii. SelectionSort toma tiempo cuadrático en todos los casos ya que siempre revisa todos los datos del arreglo en cada iteración.

iii. QuickSort aleatorio: El caso promedio toma tiempo $O(n \cdot \log n)$ y el peor caso tiempo cuadrático. Ya que el pivote es aleatorio lo más probable es que el tiempo en los 3 inputs sea similar y que el número de comparaciones sea proporcional a $n \cdot \log n$.

iv. QuickSort con pivote primer elemento: El peor caso es cuadrático y se da cuando el escoge como pivote el mayor o el menor elemento del intervalo. En el caso de que el arreglo esté ordenado normalmente o al revés se dará el peor caso. En el caso aleatorio se espera que el tiempo sea proporcional a $n \cdot \log n$.

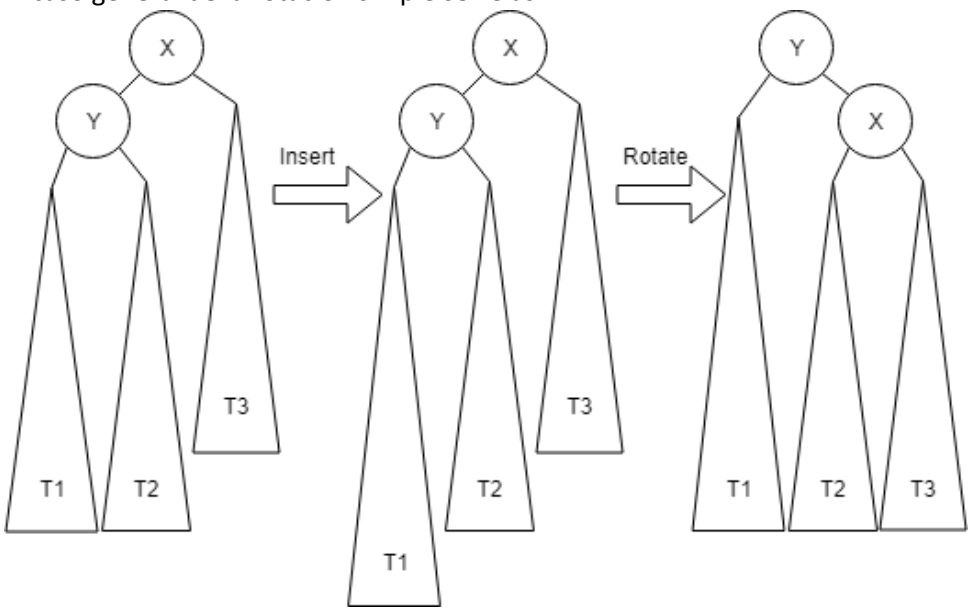
Por lo tanto, $E_2 =$ QuickSort con pivote primer elemento ya que toma mucho más tiempo cuando el arreglo está completamente ordenado y cuando está completamente ordenado en el orden contrario. $E_1 =$ SelectionSort ya que siempre toma mucho tiempo ordenar con el algoritmo independiente del orden. $E_3 =$ InsertionSort ya que hay una clara tendencia a que mientras más ordenado, menos tiempo toma. $E_4 =$ QuickSort con pivote aleatorio ya que en todos los casos tuvo un comportamiento eficiente independiente del orden, lo cual es lo esperable.

Distribución de puntajes:

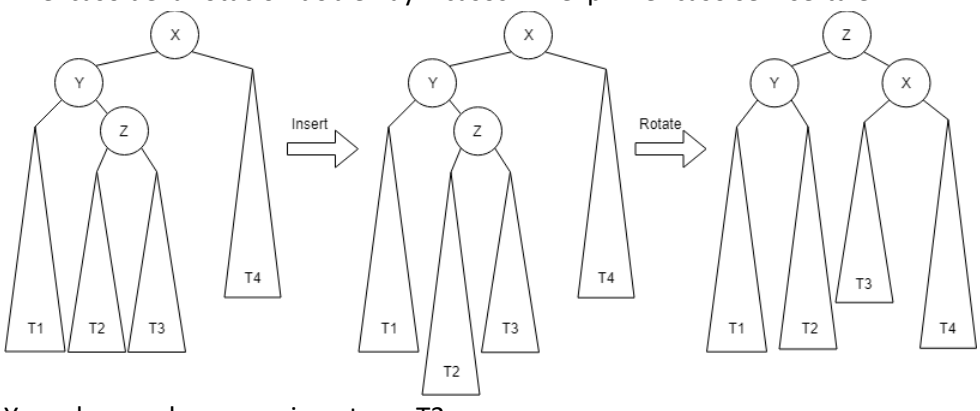
1.5 puntos por identificar correctamente todos los algoritmos.

1.5 puntos por cada justificación (solo son necesarias 3 ya que se puede dejar uno por descarte)

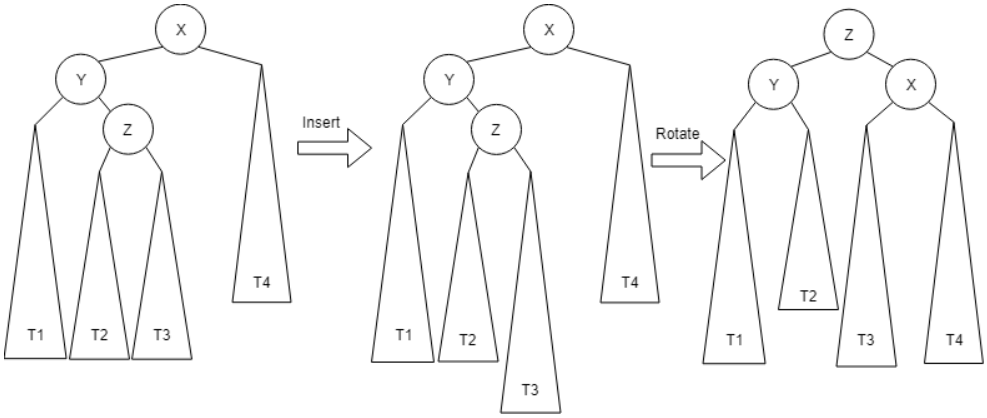
3. Se tiene un árbol AVL $\boldsymbol{T}$ en el que se inserta una clave $\boldsymbol{k}$ que produce un desbalance. Sea $\boldsymbol{x}$ el nodo más profundo en la ruta de inserción de $\boldsymbol{k}$ tal que el subárbol que tiene como raíz a $\boldsymbol{x}$ está desbalanceado, y sean $\boldsymbol{y}$ y $\boldsymbol{z}$ los siguientes dos nodos bajando por esa ruta de inserción, como se definió en clases. Demuestra que:
    a. [3pt] Luego de hacer una rotación simple o doble en torno a $\boldsymbol{x}$, $\boldsymbol{y}$ y $\boldsymbol{z}$, el subárbol que tiene como raíz $\boldsymbol{x}$ vuelve a tener la misma altura que tenía antes de insertar $\boldsymbol{k}$.
    Solución: Para demostrar esto se analizará el caso general de la rotación simple y el caso general de la rotación doble en una dirección cada una ya que el caso simétrico es equivalente.
    El caso general de la rotación simple se ve así:



    Separamos el subárbol que rota en 5 partes: el nodo X, el nodo Y, y los subárboles T1, T2 y T3. La altura inicial antes de insertar es 2 + k donde k es la altura de T1 y T2. T3 tiene altura k – 1.
    Al insertar el árbol T1 aumenta su altura en 1 lo que produce el desbalance (si se inserta en T2 y crece entonces es una rotación doble).
    Al hacer la rotación simple el subárbol T2 queda a la misma profundidad de antes, el subárbol T1 sube 1 nivel y el subárbol T3 baja 1 nivel.
    Ya que T3 estaba 1 nivel más arriba que T2 antes de rotar, ahora llega a la misma profundidad y ya que T1 subió un nivel, ahora llega a la misma profundidad que antes de insertar. Por lo tanto, la altura del árbol sigue siendo k + 2.
    En el caso de la rotación doble hay 2 casos. En el primer caso se inserta en T2:



    Y en el segundo caso se inserta en T3:



El análisis es el mismo para ambos casos ya que en ambos casos suben tanto T2 como T3, T1 se queda a la misma altura y T4 baja 1 nivel. Esto hace que 3 de los 4 árboles queden a la profundidad máxima antes de insertar y uno de ellos quede más arriba. Por lo tanto, la altura del árbol se mantiene.

b. [1.5pt] Demuestra que esta rotación no genera otros desbalances en el árbol $\boldsymbol{T}$.
Solución: Para demostrar esto hay que demostrar que en el subárbol rotado no queda ningún desbalance y que no se producen desbalances fuera del subárbol rotado.
En el caso de la rotación simple y la doble, los árboles T1, T2, T3 y T4 se mantienen intactos luego de insertar y rotar. Además, ya que el nodo X es el de más abajo en estar desbalanceado, ninguno de los subárboles está desbalanceados luego de insertar. Por lo tanto, luego de rotar siguen balanceados.
Por otro lado, el subárbol que se rota mantiene su altura (demostrado en a), por lo tanto, si se produjo en desbalance en un nodo más arriba que X al insertar, este se corrige al momento de rotar.

c. [1.5pt] Demuestra que esta rotación basta para solucionar el desbalance del árbol $\boldsymbol{T}$.
Solución: Dado lo demostrado en a y en b y sabiendo que el árbol estaba balanceado antes de insertar, sabemos que todos los subárboles superiores al árbol rotado quedan balanceados y que todos los subárboles que están incluidos en la rotación también quedan balanceados. Por lo tanto, la rotación hecha fue suficiente para balancear el árbol luego de insertar.

4. Queremos extender la funcionalidad de un *Heap Binario*. Asumiendo que tienes las funciones $\boldsymbol{insert}$, $\boldsymbol{extract}$, $\boldsymbol{sift-up}$ y $\boldsymbol{sift-down}$, escribe un algoritmo para las siguientes operaciones:
    a. **[3pt]** Modificar la prioridad del elemento en la posición $\boldsymbol{i}$
    Solución: El algoritmo de cambio de prioridad es el siguiente:
    $update\ priority(H, i, value)$:
        $H[i] = value$
        $sift - up(i)$
        $sift - down(i)$
    Notar que solo sift up o sift down moverán el elemento, pero no ambos.

    b. **[3pt]** Eliminar del *Heap* el elemento en la posición $\boldsymbol{i}$
    Solución: El algoritmo de eliminación es similar al anterior pero un poco distinto:
    $pop(H, i)$:
        $value = H[i]$
        $H[i] = H[H.count]$
        $H.count - -$
        $sift - up(i)$
        $sift - down(i)$

    Tanto el algoritmo en a como el algoritmo en b funcionan en O(logn)