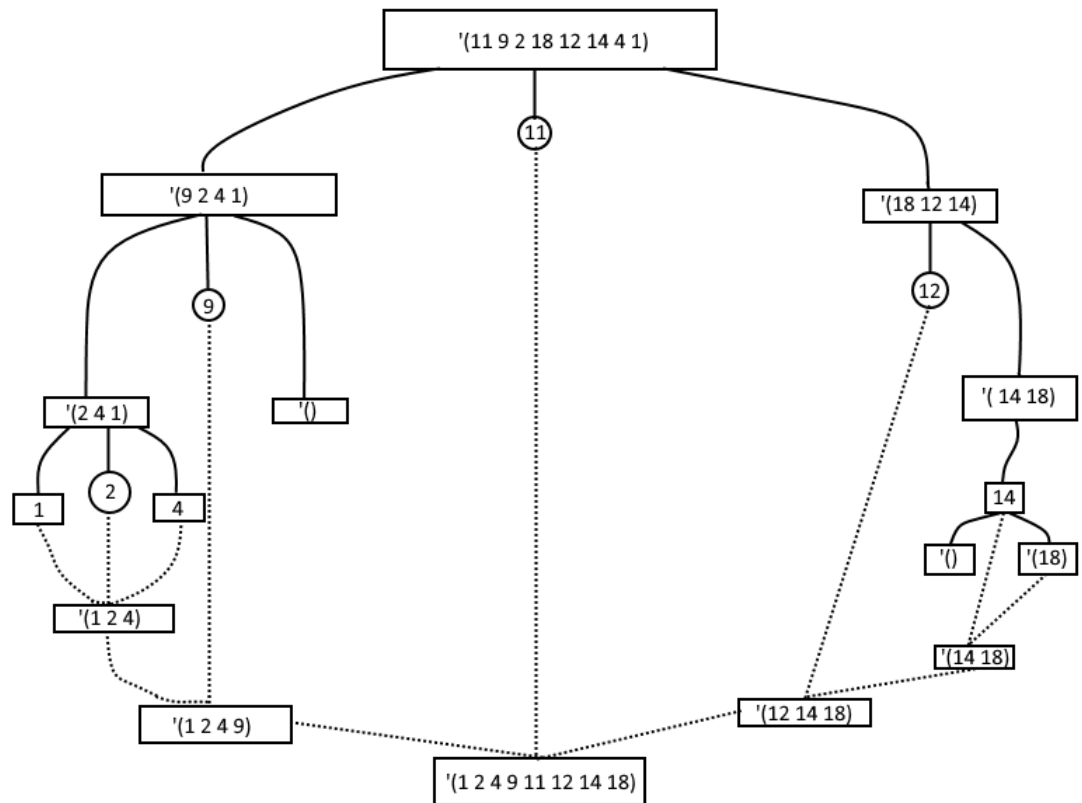


Tarea 2

La llamada (bundle '("a" "b" "c") 0) es un buen uso de bundle? ¿qué produce? ¿por qué?

Problema 9

$$'(11 \ 9 \ 2 \ 18 \ 12 \ 14 \ 4 \ 1).$$


Si la entrada a quicksort contiene varias repeticiones de un número, va a regresar una lista estrictamente más corta que la entrada. Responde el por qué y arregla el problema.

Quicksort no toma en cuenta los duplicados ya que *smallers* y *largers* son estrictos, es decir solo toman en cuenta si son menores o mayores respectivamente. Este problema se puede solucionar formando 3 grupos de datos en vez de 2, uno para los mayores, uno para los menores y uno para los numeros iguales, de esta forma se puede mantener la lista ordenada y con la misma cantidad de datos.

Problema 13

Implementa una versión de quicksort que utilice isort si la longitud de la entrada está por debajo de un umbral. Determina este umbral utilizando la función time, escribe el procedimiento que seguiste para encontrar este umbral.

Si asumimos que isort siempre será más rápido que quicksort bajo una cierta cantidad de datos podemos establecer el umbral cuando isort sea más *lento* que quicksort, ya que este siempre será más rápido bajo este umbral.

Para encontrar el umbral realice pruebas desde los 10 datos hasta los 200 y con estas pruebas pude colocar el umbral en 150 datos. En promedio, después de 150 datos isort era más lento que quicksort, mientras que en 150 datos los algoritmos mostraban 1 o 0 ms de ejecución en quicksort por lo que isort era más rápido.

Problema 18

Considera la siguiente definición de *smallers*, uno de los procedimientos utilizados en quicksort, responde en qué puede fallar al utilizar esta versión modificada en el procedimiento de ordenamiento.

Va a llegar un punto en el que *smallers* tenga al pivote repetidas veces, lo cual va a ocasionar que la lista de *smallers* nunca llegue a *null*, es decir, nunca va a terminar.

Problema 19

Describe con tus propias palabras cómo funciona *find-largest-divisor* de *gcd-structural*. Responde por qué comienza desde ($\min n$ m).

La función itera desde el mínimo entre ambos números para encontrar el número que divida a los dos sin algún residuo. Si un número no divide a ambos se le resta uno al argumento recibido y se continúa con las iteraciones hasta encontrar el GCD o hasta llegar al 1.

Comienza desde el mínimo entre ambos números para evitar iterar en el rango entre el mayor y menor, ya que es imposible que este rango tenga al GCD.

Problema 20

Describe con tus propias palabras cómo funciona *find-largest-divisor* de *gcd-generative*.

Partiendo de la observación de que para dos enteros positivos, el máximo común divisor es igual al máximo común divisor del menor y el residuo del mayor entre el menor, la función calcula la primera iteración de la fórmula, la cual requiere el GCD de otros dos números, y esos otros dos números requieren el GCD de otros dos números, y así hasta llegar a un GCD el cual permite calcular el resto de GCDs hasta llegar a la llamada inicial de la función.

Problema 21

Utiliza la función time para determinar cuál de las dos implementaciones es más eficiente, escribiendo tu respuesta con los tiempos de ejecución obtenidos con ambos procedimientos para valores “pequeños”, “medianos” y “grandes”. Justifica qué valores usaste en cada una de estas mediciones y por qué los consideraste de ese “tamaño”.

Comenzando con los datos 10 y 17, no existe una diferencia, lo mismo sucede con (100, 170), (1000, 1700) y (10000, 17000). La diferencia comienza al entrar a datos grandes mayores a 100,000 ya que el *gcd* estructural comienza a ser más lento que el generativo. Los datos utilizados para probar los datos grandes fueron (10700, 17000) donde *gcd-structural* siempre fue más lento que el generativo por 2 o 3 ms.

Problema 22

Piensa y describe por qué no siempre es la mejor opción elegir el procedimiento más eficiente en tiempo de ejecución. Utiliza criterios que no sean el de “eficiencia”.

La primera razón para no elegir un mejor tiempo de ejecución es lo comprensible que es una solución. Si un algoritmo es más complicado de entender que otro y la diferencia entre eficiencia es muy pequeña, es mejor tener un algoritmo entendible que esa diferencia de eficiencia.

Otro motivo podría ser que un algoritmo sea más rápido pero que utilice demasiada memoria lo cual puede ocasionar muchos problemas con una cantidad grande de datos, por lo que es mejor utilizar un algoritmo que tarde más pero evite estos problemas.