`Tarea 02` : El Despegue

Ricardo Payan

6 de septiembre de 2022

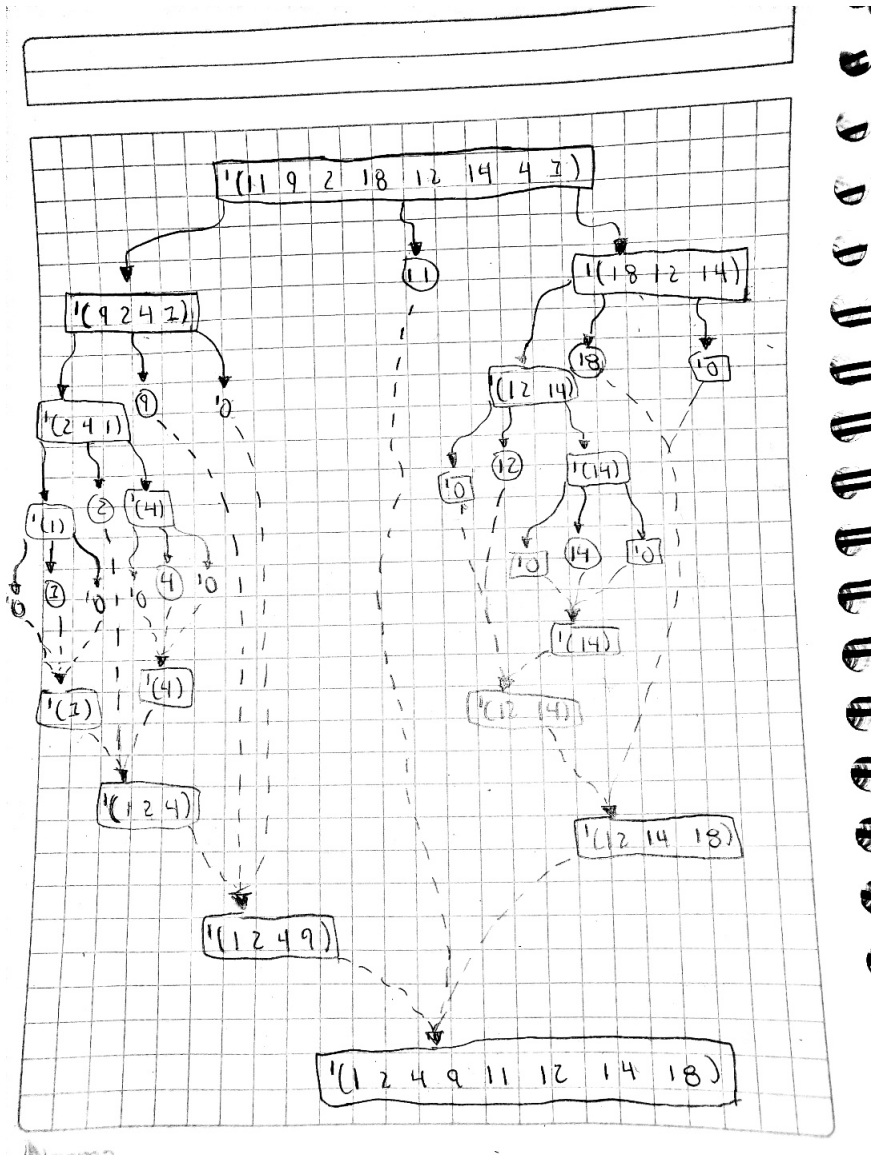
Problema 5.

Si tomamos en cuenta que la finalidad de bundle es juntar strings en un grupos de n cantidad es algo absurdo tratar de juntarlos en grupos de cero. A final esta función funciona como una división y dividir entre 0 es imposible. No creo que sea un buen uso de la función, pero considero que es una prueba interesante.

(bundle '("a" "b" "c") 0)

Devuelve la misma lista '("a" "b" "c"). Al ser imposible dividir lo que sea en trozos de 0, entonces decidí que esto simplemente no afectara la lista en mi implementación.

Problema 9.


1'(11 9 2 18 12 14 4 7)
11
1'(18 12 14)
1'(9 2 4 7)
18
10
1'(12 14)
9
10
1'(2 4 1)
12
1'(14)
10
1'(1)
2
1'(4)
14
10
4
10
10
1'(14)
1'(4)
1'(7)
1'(12 14)
1'(1 2 4)
1'(12 14 18)
1'(1 2 4 9)
1'(1 2 4 9 11 12 14 18)

Problema 11.

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.

Por que los métodos largers y smallers solo agregan los elementos que son estrictamente mayores y menores al pivote respectivamente. En algun punto del ciclo de recursion, el pivote repetido ya estará dentro de la lista y por eso no lo agregara, ya que nuestro algoritmo actual no sabe que hacer con números que son iguales.

Mi solución fue hacer un tercer método llamado "same" que recibe una lista y un pivote.

```
1  (define (same ls pivot)
2    (cond
3      [(empty? ls) null]
4      [(equal? (first ls) pivot) (cons (first ls) (same ↩
           (rest ls) pivot))]
5      [else (same (rest ls) pivot)]))
```

Esta función filtra los elementos de *ls* que son iguales al *pivot* y los agrega a una nueva lista. Entonces modificados la función *Quicksort* para que se unan las tres listas.

```
1  (define (quicksort ls )
2    (cond
3      [(empty? ls) null]
4      [else
5       (define pivot (first ls))
6       (append (quicksort (smallers ls pivot)) (same ls ↩
            pivot) (quicksort (largers ls pivot)))]))
```

Problema 12.

Modifica *quicksort* para ordenar listas con valores de cualquier tipo en cualquier orden aceptando un argumento adicional.

Para resolver este problema implemente la misma idea del problema 8 para el *isort*, agregando un predicado como argumento adicional extra.

```
1       ;Problema 8
2  (define (isort ls predicado)
3    (if (empty? ls)
4        null
5        (insert (first ls)
6                (isort (rest ls) predicado) predicado)))
```

Por supuesto, al modificar *Quicksort*, necesariamente necesite modificar los métodos *largers* y *smallers*; Ya que estos métodos son los que comparan los valores de la lista. El método same se deja igual ya que siempre se usa el mismo comparador.

Al querer modificar *largers* y *smallers* me encontré con el primer subproblema a resolver: Pasando el mismo predicado, ¿Cómo se hace para que uno haga lo contrario al otro?

La solución fue dejar le método *smallers* como el "base", ósea, haría exactamente lo que el predicado indique.

```
1  (define (smallers ls pivot predicado)
2      (cond
3      [(empty? ls) null]
4      [(predicado (first ls) pivot) (append (list (first ↩
           ls)) (smallers (rest ls) pivot predicado))]
5      [else (smallers (rest ls) pivot predicado)]))
```

El metodo largers haria lo contrario a lo que el predicado indique, para eso nos apoyamos con el *not*.

```
1  (define (largers ls pivot predicado)
2    (cond
3      [(empty? ls) null]
4      [(and (not ( predicado (first ls) pivot)) (not ↩
           (equal? (first ls) pivot)) ) (append (list ↩
           (first ls)) (largers (rest ls) pivot predicado))]
5      [else (largers (rest ls) pivot predicado)]))
```

En la primer versión no había agregado la segunda condición:

```
1       not (equal? (first ls) pivot))
```

Y eso hacia mi ejecución llenara la memoria, entre rascarle y quemarse un poco la cabeza al final me di cuenta que el pivot tampoco debia ser igual a elemento que estabamos comparando, ya que eso se haría en el método same.

```
1  (define (same ls pivot)
2    (cond
3      [(empty? ls) null]
4      [(equal? (first ls) pivot) (cons (first ls) (same ↩
           (rest ls) pivot))]
5      [else (same (rest ls) pivot)]))
```

Entonces, al final le agregamos un filtro al metodo quicksort para que solo se ejecute cuando el argumento predicado sea un procedimiento.

```
1  (define (quicksort ls predicado)
2    (unless (procedure? predicado) (error quicksort ↩
         "Esperaba un predicado valido, recibi ¬e" ↩
         predicado))
3    (cond
4      [(empty? ls) null]
5      [else
6       (define pivot (first ls))
7       (append (quicksort (smallers ls pivot predicado) ↩
            predicado) (same ls pivot) (quicksort (largers ↩
            ls pivot predicado) predicado))]))
```

Algunas pruebas:

```
1  > (quicksort '("z" "Apple" "queen" "sexy") string<?)
2      '("Apple" "queen" "sexy" "z")
3
4  > (quicksort (list 'b 'a 'c) symbol<?)
5      '(a b c)
6
7  > (quicksort '(#"b" #"a" #"c") bytes<?)
8          '(#"a" #"b" #"c")
9
10 > (quicksort '(1 2 3 5 2 5 7 9) >)
11     '(9 7 5 5 3 2 2 1)
12
```

```
> (quicksort '(1 2 3 5 2 5 7 9) <)
        '(1 2 2 3 5 5 7 9)

> (quicksort '(1 2 3 5 2 5 7 9) 6)
. . quicksort: Esperaba un predicado valido, recibi 6
```

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.

Para hacer este prueba use las funciones $shuffle$ y $build - list$. A este ultima le pase una cantidad n de elementos.

```
    ;n = 10,000
> (time (quicksort (shuffle (build-list 10000 values)) ↩
   >))
        cpu time: 93 real time: 75 gc time: 31

> (time (isort (shuffle (build-list 10000 values)) >))
        cpu time: 5125 real time: 5252 gc time: 343
```

En esta primer prueba podemos notar una diferencia importante de tiempo, siendo quicksort el algoritmo mas rapido. Ahora lo probaremos con $n = 1000$.

```
    ;n = 1000
> (time (quicksort (shuffle (build-list 1000 values)) >))
        cpu time: 0 real time: 5 gc time: 0

> (time (isort (shuffle (build-list 1000 values)) >))
    cpu time: 46 real time: 49 gc time: 15
```

Ahora le diferencia es menor, pero Quicksort sigue siendo mucho mas rápido. Hagamos otra prueba bajando el umbral a 100.

```
    ;n = 100
> (time (quicksort (shuffle (build-list 100 values)) >))
```

```
3          cpu time: 0 real time: 0 gc time: 0
4
5  > (time (isort (shuffle (build-list 100 values)) >))
6          cpu time: 0 real time: 0 gc time: 0
```

Vemos que no hay diferencia cuando $n = 100$, así que podemos considerar ese el umbral.

Agregamos están condición a Quicksort.

```
1  (define (quicksort ls predicado)
2    (unless (procedure? predicado) (error quicksort ↩
         "Esperaba un predicado valido, recibi ¬e" ↩
         predicado))
3    (cond
4      [(empty? ls) null]
5      [(< (length ls) 100) (isort ls predicado)]
6      [else
7       (define pivot (first ls))
8       (append (quicksort (smallers ls pivot predicado) ↩
            predicado) (same ls pivot) (quicksort (largers ↩
            ls pivot predicado) predicado))]))
9
10
11 > (time (quicksort '(4 6 5 1 9) >))
12         cpu time: 0 real time: 0 gc time: 0
13         '(9 6 5 4 1)
```

Problema 14.

Utiliza filter para definir smallers y largers.

Si tomamos en cuenta que la forma del *filter* es:

```
1      (filter pred lst) -> list?
2
3    pred : procedure?
4    lst : list?
```

Es fácil implementarlo, el *pred* del *filter* serian las condiciones que ya habíamos definido.

```
1  (define (smallers ls pivot predicado)
2    (cond
3      [(empty? ls) null]
4      [(filter (predicado (first ls) pivot) ls) (append ↩
           (list (first ls)) (smallers (rest ls) pivot ↩
           predicado))]
5      [else (smallers (rest ls) pivot predicado)]))
6
7  (define (largers ls pivot predicado)
8    (cond
9      [(empty? ls) null]
10     [(filter (and (not ( predicado (first ls) pivot)) ↩
           (not (equal? (first ls) pivot))) ls) (append ↩
           (list (first ls)) (largers (rest ls) pivot ↩
           predicado))]
11     [else (largers (rest ls) pivot predicado)]))
12
13 (define (same ls pivot)
14   (cond
15     [(empty? ls) null]
16     [(filter (equal? (first ls) pivot) ls) (cons (first ↩
           ls) (same (rest ls) pivot))]
17     [else (same (rest ls) pivot)]))
```

Problema 15.

Implementa smallers y largers como procedimientos locales internos en Quicksort.

Para esto tenemos que usar funciones anónimas dentro de Quicksort, ósea *lambda*; Lo cual, siendo muy sincero, me sigue pareciendo arte de magia. Creo que me acostumbre a como funciona lambda en vez de entenderlo del todo, cada vez que tengo que usarlo le tengo pego una releída la documentación. Con cada una lo entiendo un poco mas, espero...

Entonces definimos *smallers* y *largers*.

```
1  (define (quicksort ls predicado)
2    (unless (procedure? predicado) (error quicksort ↩
```

```
        "Esperaba un predicado valido, recibi ¬e" ↩
        predicado))
3   (cond
4     [(empty? ls) null]
5     [(< (length ls) 100) (isort ls predicado)]
6     [else
7      (define pivot (first ls))
8      (define smallers (filter (lambda (x) (predicado x ↩
           pivot) ls)))
9      (define largers (filter (lambda (x) (filter (and ↩
           (not (predicado x pivot)) (not (equal? x ↩
           pivot))) ) ls)))
10
11     (append (quicksort (smallers ls pivot predicado) ↩
           predicado)
12     (same ls pivot)
13     (quicksort (largers ls pivot predicado) ↩
           predicado))]))
```

También podría definir *same* dentro del Quicksort, pero eso lo dejamos como ejercicio para el lector.

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.

```
1     (define (smallers l n)
2     (cond
3         [(empty? l) '()]
4         [else (if (≤ (first l) n)
5                 (cons (first l) (smallers (rest l) n))
6                 (smallers (rest l) n))]))
```

Fallaria porque usa el comparador <= y el algoritmo de quicksort solo funciona cuando los elementos son estrictamente menores, estrictamente mayores o estrictamente iguales al pivote. De hecho, cuando en alguno de mis intenttos para que el quicksort añadiera los elementos que son iguales al pivote hice exactamente lo mismo que esta definicion de smallers. Se me ciclo el algoritmo.

Problema 19.

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

```
1       (define (gcd-structural n m)
2       (define (find-largest-divisor k)
3       (cond [(= i 1) 1]
4           [(= (remainder n i) (remainder m i) 0) i]
5           [else (find-largest-divisor (- k 1))]))
6   (find-largest-divisor (min n m)))
```

Basicamente, el algoritmo toma dos numeros y busca otro que divida a n y m sin dejar residuo.

Si $i = 1$, el algoritmo se detiene ya que ese es el caso base de GDC. Todos los numeros enteros se pueden divir entre 1 sin dejar residuo.

SI no es el caso, verificamos si el residuo de n y m entre i es igual a cero; si es el caso, significa que i es un comun divisor.

Si no, continuamos el algoritmo restanda una unidad a k.

Empezamos con el minimo de los numeros porque si tratamos de dividir un numero mas chico entre uno mas grande, siempre habra residuo, entonces no tiene sentido empezarlo desde el mas grande.

Problema 20.

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

```
1       (define (gcd-generative n m)
2       (define (find-largest-divisor max min)
3           (if (= min 0)
4           max
5           (find-largest-divisor min (remainder max min))))
6       (find-largest-divisor (max n m) (min n m)))
```

Esto me parece el algoritmo de euclides. Estamos pasando dos numeros enteros y encontramos sus maximo comun divisor.

En algortimo de euclides decimos que si el minimo entre los dos numeros es igual a cero, entonces devolvemos el minimo.

Sabemos que el maximo comun divisor entre dos numeros es igual al maximo comun divisor del menor y el residuo del mayor entre el menor.

$mcd(a, 0) = 0$

$mcd(a, b) = mcd(b, amodb)$

Entonces, la funcion se seguira llamando a si misma hasta que encuentre el GDC.

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".

```
1      ;valores peque os
2  >(time (gcd-structural 100 150))
3          cpu time: 0 real time: 0 gc time: 0
4          50
5
6  > (time (gcd-generative 100 150))
7          cpu time: 0 real time: 0 gc time: 0
8          50
9
10   ;valores medianos
11 > (time (gcd-generative 1000 1500))
12         cpu time: 0 real time: 0 gc time: 0
13         500
14
15 > (time (gcd-structural 1000 1500))
16         cpu time: 0 real time: 0 gc time: 0
17         500
```

```
18
19
20    ;valores grandes
21  >(time (gcd-structural 100000 150000))
22          cpu time: 8 real time: 8 gc time: 2
23          50000
24
25  > (time (gcd-generative 100000 150000))
26      cpu time: 0 real time: 0 gc time: 0
27      50000
```

Podemos notar que empieza a ver diferencias entre los dos algoritmos cuando cruzamos el umbral de cientos de miles. Sinceramente, la selección de estos valores fue un poco albitrari. Al tanteo como se le dice normalmente.

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".

En la seccion 2.3 hablamos de lo importante que es poder explicar nuestros algoritmos; para esa seccion utilizamos firmas. Tal vez se dio cuenta profesor que entendi mucho menos el algoritmo $gcd - generative$ a comparacion de $gcd - structural$ y esto es en gran medida por la forma en que funcionan los algoritmos y como estan escritos. A pesar de que los dos hagan lo mismo me es mas complicado explicar y entender uno que el otro. Con los tiempos, podemos ver que $gcd - generative$ es mas rapido, pero tampoco es una diferencia enorme. Entonces considero que en casos como estos es mejor la opcion que sea mas facil de entender y leer.