El Despegue - Tarea № 02

Enrique Giottonini OKTOBERFEST

Problema 5

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

Sea $\gamma = (bundle\, s\, n)$ donde s es una lista de caracteres y n el tamaño de los trozos entonces:

$$(length\, \gamma) = \left\lceil \frac{length(s)}{n} \right\rceil$$

Por lo que no tiene sentido que n sea igual a 0. En la implementacion de <bundle> esto se refleja en la llamada recursiva.

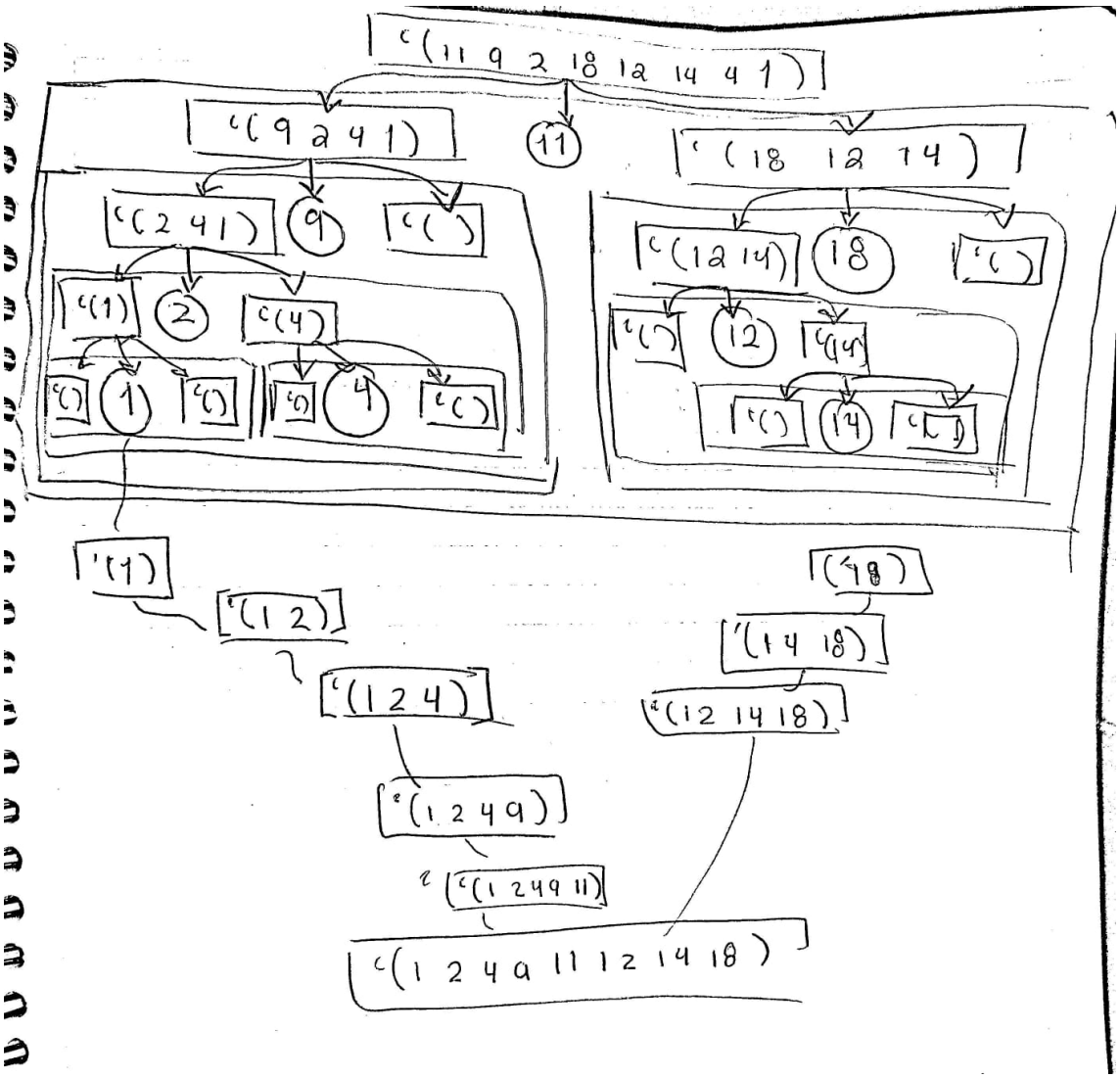
bundle

```
1 (define (bundle s n)
2   (cond
3     [(null? s) null]
4     [else
5      (cons (implode (take s n))
6            (bundle (drop s n) n))]))
```

$(drop\, s\, n) = s$ si $n = 0$ por lo que la llamada recursiva en $(bundle\, s\, 0)$ entraria en un loop infinito.

Problema 9

Dibuja un diagrama como el de la figura anterior($< quicksort >$) pero para la lista '(11 9 2 18 12 14 4 1) .



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.

La llamada recursiva de quicksort divide el problema (la lista) en 3 partes.

1. Los que son estrictamente menores al pivote.
2. El elemento
3. Los que son estrictamente mayores al pivote.

quicksort

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

Una manera de solucionarlo puede ser que en la 2) parte en vez de solo seleccionar el pivote se construya una lista de todas las repeticiones del pivote.

quicksort

```
1 (define (quicksort ls)
2   (cond
3     [(empty? ls) null]
4     [else
5      (define pivot (first ls))
6      (append (quicksort (smallers ls pivot))
7              (filter (lambda (x) (equal? x pivot)) ls)
8              (quicksort (largers ls pivot)))]))
```

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 encontrar el umbral defino una lista de longitud n con elementos aleatorios, luego con $time$ comparo el tiempo de $isort$ y de $quicksort$.

timing quicksort and isort

```
1  (define (randlist n)
2    (shuffle (range n)))
3  (define (greater x y) (> x y))
4
5  n=10,000
6  isort:     cpu time: 14729 real time: 14719 gc time: 2746
7  quicksort: cpu time:   136 real time:   136 gc time:   46
8
9  n=1000
10 isort:     cpu time: 154 real time: 155 gc time: 39
11 quicksort: cpu time:   8 real time:   8 gc time:  0
12
13 n=100
14 isort:     cpu time: 1 real time: 1 gc time: 0
15 quicksort: cpu time: 0 real time: 0 gc time: 0
```

Se toma que el Umbral = 100 ya que no hay diferencia entre un procedimiento con el otro.

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.

smallers

```
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))]))
```

La llamada recursiva de quicksort divide el problema (la lista) en 3 partes.

1. Los que son estrictamente menores al pivote.
2. Los que son igual al elemento.
3. Los que son estrictamente mayores al pivote.

Con esta modificación tenemos el problema de incluír al pivote dentro 1) y por tanto entrar en un bucle infinito.

Problema 19

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

gcd-structural

```
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)))
```

Para encontrar el *greatest common denominator* este algorítmo itera de uno en uno desde el mínimo de los 2 argumentos. Por ejemplo, para encontral el gdc de 100 y 40 checa si el min(100, 40) = 40 divide a ambos y no deja residuo (común denominador), si no es entonces prueba con el 39 y así sucesivamente.

Problema 20

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

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)))
```

Es una implementación del algorítmo de Euclídes. Encontrar el $(gcd\ a, b)$ es encontrar el número más grande que divide a y b, el caso trivial es cuando uno de los dos es 0. De otra forma se sabe, matemáticamente hablando, que al dividir por ejempo a con b nos queda que $a = pb + r$, y un número que divide a b y a debe dividir a r. Si $r = 0$ (caso trivial) entonces el gdc es b. De otra forma encontrar el $(gcd\ b, r)$ nos dará el $(gcd\ a, b)$.

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”.

Dado que es structural es iterativo, un primo lo suficientemente grande va a ser que tenga un mal tiempo, ya que basicamente se va a reducir a que cuente hacia atras desde ese primo:

gcd: structural vs generative

```
1  PRIMES
2  n=1259, m=1889
3  structural: cpu time: 0 real time: 0 gc time: 0
4  generative: cpu time: 0 real time: 0 gc time: 0
5
6  n=241,271, m=31,397
7  structural: cpu time: 3 real time: 3 gc time: 0
8  generative: cpu time: 0 real time: 0 gc time: 0
9
10 n=13,466,917, m=6,972,593
11 structural: cpu time: 665 real time: 664 gc time: 0
12 generative: cpu time:   0 real time:   0 gc time: 0
13
```

```
n=43,112,609, m=37,156,667
structural: cpu time: 3598 real time: 3594 gc time: 0
generative: cpu time:    0 real time:    0 gc time: 0

n=24324324324323243243232343232 7 m=23)
structural: cpu time: 0 real time: 0 gc time: 0
generative: cpu time: 0 real time: 0 gc time: 0
```

Aquí se puede observar la eficiencia de generative aún para números grandes mientras que structural depende del mínimo y que sea primo obliga al algorítmo itere todo el número.

Por otro lado, los números de Fibonacci deberían de ser la debilidad de generative:

gcd: structural vs generative

```
FIBONACCI
> (t 1259 1889)
cpu time: 0 real time: 0 gc time: 0
cpu time: 0 real time: 0 gc time: 0
1
> (t 241271 31397)
cpu time: 3 real time: 3 gc time: 0
cpu time: 0 real time: 0 gc time: 0
1
> (t 13466917 6972593)
cpu time: 665 real time: 664 gc time: 0
cpu time: 0 real time: 0 gc time: 0
1
> (t 43112609 37156667)
cpu time: 3598 real time: 3594 gc time: 0
cpu time: 0 real time: 0 gc time: 0
1
> (t 57885161 13)
cpu time: 0 real time: 0 gc time: 0
cpu time: 0 real time: 0 gc time: 0
1
> (t 82589933 19937)
cpu time: 2 real time: 2 gc time: 0
cpu time: 0 real time: 0 gc time: 0
1
> (t 8258993 19932)
cpu time: 2 real time: 2 gc time: 0
cpu time: 0 real time: 0 gc time: 0
1
> (t 8258992343243 4)
```

```
cpu time: 0 real time: 0 gc time: 0
cpu time: 0 real time: 0 gc time: 0
1
> (t 23432493242348324728374 3)
cpu time: 1 real time: 0 gc time: 0
cpu time: 0 real time: 0 gc time: 0
3
> (t 432743874687324658743265874326583274658732465873246587432 4)
cpu time: 0 real time: 0 gc time: 0
cpu time: 0 real time: 0 gc time: 0
4
> (t 32482349837458235732465943287598327598327532324759843275843275432↩
    3)
cpu time: 0 real time: 0 gc time: 0
cpu time: 0 real time: 0 gc time: 0
3
> (t ↩
   428349827349827349827987432984729847498274928744974287472472874923987429874329874
    2)
cpu time: 0 real time: 0 gc time: 0
cpu time: 0 real time: 0 gc time: 0
2
> (t 4324343243243243 3232)
cpu time: 1 real time: 0 gc time: 0
cpu time: 0 real time: 0 gc time: 0
1
> (t 24324324324323243243232343232 23)
cpu time: 0 real time: 0 gc time: 0
cpu time: 0 real time: 0 gc time: 0
1
```

Pero observamos que el remainder le hace todo el trabajo a generative y sigue siendo más eficiente, aún cuando aumento o disminuyo la distancia entre los parametros, cuando son fibonacci, fibonacci primos, fibonacci consecutivos, y otras variaciones. Por que el algóritmo de Euclídes, junto con una buena implementación de remainder, es exageradamente eficiente.

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”.

Siguiendo la linea de pensamiento del epigráma de Alan Perlis:
"Simplicity does not precede complexity, but follows it."

Los casos donde una solución elegante y eficiente por lo general son resultados de un complejo análisis derivado de otras áreas (matemáticas) por lo que para nosotros programadores promedio, es dificil entender porque funciona, en que casos se comporta mejor, en cuales no, etc. Por tanto es preferible, si no es un sistema crítico, que la solución o algorítmo sea aquel que podamos entender y desmenuzar nosotros mismos.

Los casos donde NO es una solución elegante pero es eficiente o se vale de "trucos" son dificiles de entender para quienes no estuvieron realizando esos "ajuste" que mejoraron el procedimiento.

Para ambos casos es mejor un algorítmo que describa el problema y su solución de manera entendible para quienes la implementan.