Trabajo Práctico II - EDyA II

Laureano Hess, Lautaro Peralta Aguilera

Junio 2022

1 Implementación con listas

1.1 mapS

```haskell
myMapS :: (a -> b) -> [a] -> [b]
myMapS f [] = []
myMapS f (x:xs) = y:ys
    where (y, ys) = f x ||| myMapS f xs
```

Los costos secuencial y paralelos de $myMapS$ puede expresarse como las siguientes recurrencias,donde n es la longitud de la lista, es decir $|xs|$ y xs_n es el elemento enésimo de la lista :

$$W(n) = W(n-1) + W(f\ xs_n) + c_1$$
$$S(n) = max\left\{S(n-1), W(f\ xs_n)\right\} + c_1$$

donde $W(f\ xs_i)$ y $S(f\ xs_i)$ son el trabajo y profundidad de la aplicación de f a xs_i
Veamos que:

- $$W(n) \in \mathcal{O}\left(\sum_{i=0}^{n-1} W(f\ xs_i)\right)$$

  - Caso base

    $$W(1) = W(f\ xs_i) + a_1 \leq a_1 \cdot W(f\ x_i) \leq c \cdot W(f\ xs_i) + 1 \qquad , a_1 \leq c$$

  - Caso inductivo
    Supongamos que

    $$W(x) \leq c \sum_{i=0}^{x-1} W(f\ xs_i), \quad \forall x < n$$

Luego:

$$
\begin{aligned}
W(n) &= W(n-1) + W(f\ xs_n) + c_1 \\
&\overset{(HI)}{\leq} c \sum_{i=0}^{n-1-1} W(f\ xs_i) + W(f\ xs_n) + c_1 \\
&\leq c \sum_{i=0}^{n-2} W(f\ xs_i) + c_1 W(f\ xs_n) \\
&\leq c \sum_{i=0}^{n-2} W(f\ xs_i) + cW(f\ xs_n) \qquad c_1 < c \\
&\leq c \sum_{i=0}^{n-1} W(f\ xs_i)
\end{aligned}
$$

Luego, tomando $c = max\{a_1, c_1\}$ y $n_0 = 1$ tenemos que:

$$0 < W(n) \leq c \sum_{i=0}^{n-1} W(f\ xs_i) \qquad \forall c > 0, n \geq n_0$$

Es decir,

$$W(n) \in \mathcal{O}\left(\sum_{i=0}^{n-1} W(f\ xs_i)\right) \tag{1}$$

- $S(n) \in \mathcal{O}\left(max_{i=0}^{n-1}(f\ xs_i) + n\right)$
  - Caso base

$$S(1) = S(f\ xs_i) + a_1 \leq a_1 \cdot S(f\ xs_i) \leq c \cdot S(f\ x_i) \leq c \cdot \max_{i=0}^{0}(S(f\ xs_i) + 1) \qquad , a_1 \leq c$$

  - Caso inductivo

Supongamos que $S(x) \leq c \cdot \max_{i=0}^{x-1}(f\ xs_i) + x$ Luego:

$$\begin{aligned}
S(n) &= \max\left\{S(n-1), W(f\ x_n)\right\} + c_1 \\
&\leq \max\left\{c \cdot \max_{i=0}^{n-1-1}(f\ xs_i) + (n-1), W(f\ xs_n)\right\} + c_1 \\
&\leq \max\left\{c \cdot \max_{i=0}^{n-2}(f\ xs_i) + (n-1), c \cdot W(f\ xs_n)\right\} + c_1 \\
&\leq c \cdot \max\left\{\max_{i=0}^{n-2}(f\ xs_i) + (n-1), W(f\ xs_n)\right\} + c_1 \\
&\leq c \cdot \max\left\{\max_{i=0}^{n-2}(f\ xs_i) + (n-1), W(f\ xs_n + (n-1))\right\} + c_1 \\
&\leq c \cdot \left(\max_{i=0}^{n-1}(f\ xs_i) + (n-1)\right) + c_1 \\
&\leq c \cdot \left(\max_{i=0}^{n-1}(f\ xs_i) + (n-1)\right) + c \qquad , c_1 \leq c \\
&\leq (c+1) \cdot \left(\max_{i=0}^{n-1}(f\ x_i) + (n-1)\right) \\
&\leq c \cdot \left(\max_{i=0}^{n-1}(f\ xs_i) + (n-1) + 1\right) \\
&\leq c \cdot \left(\max_{i=0}^{n-1}(f\ xs_i) + n\right)
\end{aligned}$$

Luego, tomando $c = max\{a_1, c_1\}$ y $n_0 = 1$ tenemos que:

$$0 < S(n) \leq c \cdot \left(\max_{i=0}^{n-1}(f\ xs_i) + n\right) \qquad \forall c > 0, n \geq n_0$$

Es decir,

$$S(n) \in \mathcal{O}\left(\max_{i=0}^{n-1}(f\ xs_i) + n\right) \tag{2}$$

1.2 appendS

```haskell
appendS = (++)
(++) []     ys = ys
(++) (x:xs) ys = x : xs ++ ys
```

Implementamos appendS con el operador (++) del preludio, el cual se define como se puede ver en el codigo explicitado.

$$\begin{aligned}
W_{++}(0) &= c_0 \\
W_{++}(n) &= c_1 + W_{++}(n-1)
\end{aligned}$$

Donde n es la longitud de la primer lista (xs). y como, no hay paralelización, $S(n)$ será igual a $W(n)$ y por lo tanto trabajo y profundidad serán iguales. Como se intuye que el costo de (++) es $\Theta(n)$ entonces procederemos a demostrarlo por substitución.

Supongo que vale

$$\forall n \in \mathbb{N}, \exists c \in \mathbb{R}, n_0 \in \mathbb{N}/$$
$$W_{++}(x) \leq cx, \forall x < n$$

- Caso base

$$W_{++}(1) = a_1 \leq 1c \leftrightarrow a_1 \leq c$$

- Caso inductivo

$$\begin{aligned} W_{++}(n) &= W_{++}(n-1) + c_1 \\ &\overset{(H.I)}{\leq} c(n-1) + c_1 \\ &\leq cn - c + c_1 \\ &\leq cn \leftrightarrow \forall c \geq c1 \end{aligned}$$

Luego tomando $c = max(a_1, c_1)$ y $n_0 = 1$

$$W_{++}(n) \leq cn, \qquad \forall n \geq n_0$$

Es decir,

$$W_{++}(n) \in \mathcal{O}(n)$$
$$S_{++}(n) \in \mathcal{O}(n)$$

1.3 reduceS

Primero veamos el costo de contraer la entrada a una lista más simple para aplicar luego la función myReduceS

```
contract :: (a -> a -> a) -> [a] -> [a]
contract _ [] = []
contract _ [x] = [x]
contract f (x:y:xs) = l : r
    where
        (l,r) = f x y ||| contract f xs
```

Supondremos que $W(f(xs_i, xs_{(i+1)})) = S(f(xs_i, xs_{(i+1)})) \in \mathcal{O}(1)$. Como la paralelización depende del costo de f y en este caso es constante, $S(n)$ será igual a $W(n)$ y por lo tanto trabajo y profundidad tendrá el mismo costo. (Donde n esla longitud de la lista, en este caso $|xs|$)

$$W_{contract}(n) = S_{contract}(n) = W(n-2) + c_1$$

Sabemos que esta recurrencia es de orden $\mathcal{O}(n)$ (demostración en apéndice 3.3))

```
myReduceS :: (a -> a -> a) -> a -> [a] -> a
myReduceS f b [] = b
myReduceS f b [x] = b `f` x
myReduceS f b xs = myReduceS f b xs'
     where xs' = contract f xs
```

Veamos las funciones de costo suponiendo que $f \in O(1)$: (Donde n es la longitud de la lista, en este caso $|xs|$)

$$\begin{aligned} S_{reduceS}(n) = W_{reduceS}(n) &= W\left(\left\lceil \frac{n}{2} \right\rceil\right) + W_{contract}(n) + c_2 \\ &\leq W\left(\left\lceil \frac{n}{2} \right\rceil\right) + a_1 n + a_2 \end{aligned}$$

Sabemos que esta recurrencia es orden $\mathcal{O}(n)$ (demostración en apéndice 3.1)

1.4 scanS

Primero veamos el costo de contraer la entrada a una lista más simple para aplicar luego la función $myScanS$

```haskell
contract :: (a -> a -> a) -> [a] -> [a]
contract _ [] = []
contract _ [x] = [x]
contract f (x:y:xs) = l : r
    where
        (l,r) = f x y ||| contract f xs
```

Supondremos que $W(f(xs_i, xs_{(i+1)})) = S(f(xs_i, xs_{(i+1)})) \in \mathcal{O}(1)$. Como la paralelización depende del costo de f y en este caso es constante, $S(n)$ será igual a $W(n)$ y por lo tanto trabajo y profundidad tendrá el mismo costo. (Donde n es la longitud de la lista, en este caso $|xs|$)

$$W_{contract}(n) = S_{contract}(n) = W(n-2) + c_1$$

Sabemos que esta recurrencia es de orden $\mathcal{O}(n)$ (demostración en apéndice 3.3)
Ahora analizaremos el costo de la función $comparate$ la cual utilizaremos para expandir una solución más simple de la entrada original luego de haberlo contraido y posteriormente evaluado por $myScanS$

```haskell
comparate :: (a -> a -> a) -> a -> [a] -> [a] -> Int -> [a]
comparate f b [] _ _ = []
comparate f b [x] xs'' _ = [head xs'']
comparate f b l@(x:y:xs) r@(x'':xs'') n
    | even n = x'' : comparate f b l r (n+1)
    | otherwise = l' : r'
    where
        (l',r') = f x'' x ||| comparate f b xs xs'' (n+1)
```

Podemos observar que el costo de comparate depende de la f dada, la cual supondremos que será de costo constante, es decir $W(f(xs_i)) = S(f(xs_i)) \in \mathcal{O}(1)$.
Como la paralelización depende del costo de f y en este caso es constante, $S(n)$ será igual a $W(n)$ y por lo tanto trabajo y profundidad tendrá el mismo costo,resultando en la siguiente recurrencia, donde n es la longitud de la lista xs, es decir $|xs|$

$$S_{comparate}(n) = W_{comparate}(n) = W(n-2) + c_1 + c_2$$

Sabemos que esta recurrencia es de orden $\mathcal{O}(n)$ (demostración en apéndice 3.3))

```haskell
myScanS :: (a -> a -> a) -> a -> [a] -> ([a],a)
myScanS f b [] = ([],b)
myScanS f b [x] = ([b], f b x)
myScanS f b xs = (r,t)
    where
        xs' = contract f xs
        (xs'', t) = myScanS f b xs'
        r = comparate f b xs xs''
```

Por último, tendremos la función $myScanS$, donde aplicaremos los costos anteriormente mencionados:

$$W_{myScanS}(n) = W_{myScanS}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \underbrace{a_1 n}_{W_{contract}(n) \in O(n)} + \underbrace{a_2 n}_{W_{comparate}(n) \in O(n)} + a_3$$

Sabemos que esta recurrencia es de orden $\mathcal{O}(n)$ (demostración en apéndice 3.1)

Y el costo de la profundidad:

$$S_{myScanS}(n) = S_{myScanS}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \underbrace{a_1 n}_{S_{contract} \in O(n)} + \underbrace{a_2 n}_{S_{comparate} \in O(n)} + a_3$$

Sabemos también que esta recurrencia es de orden $\mathcal{O}(n)$ (demostración en apéndice 3.1)

2 Implementación con arreglos paralelos

2.1 mapS

```
getElement :: Arr a -> (a -> b) -> Int -> b
getElement a f n = f (a ! n)

mapS f xs = tabulate (getElement xs f) (length xs)
```

Sean $W(f \ \ s_i)$ y $S(f \ \ s_i)$ los costos sencuenciales y paralelos de aplicar f a s_i, y $getElement$ es la función que le pasaremos a tabulate la cual accede a la posición n del arreglo, esto es

$$W(getElement\, n) = W(f\, s_n)$$
$$S(getElement\, n) = S(f\, s_n)$$

Como $mapS$ está definida únicamente a partir de $tabulate$ que tiene costos conocidos, vamos a tener, (Donde n es la longitud del arreglo, es decir $|s|$):

$$
\begin{aligned}
W_{mapS}(f, n) &= W_{tabulate}(f, n) \\
&\leq c \cdot \left(\sum_{i=0}^{n-1} W(getElement\ i) \right) \\
&\leq c \cdot \left(\sum_{i=0}^{n-1} W(f\ s_i) \right) \\
&\in \mathcal{O}\left(\sum_{i=0}^{n-1} W(f\ s_i) \right) \\
S_{mapS}(f, n) &= S_{tabulate}(f, n) \\
&\leq c \cdot \left(\max_{i=0}^{n-1} S(getElement\ i) \right) \\
&\leq c \cdot \left(\max_{i=0}^{n-1} S(f\ s_i) \right) \\
&\in \mathcal{O}\left(\max_{i=0}^{n-1} S(f\ s_i) \right)
\end{aligned}
$$

2.2 appendS

```haskell
getAppendedElement :: Arr a -> Arr a -> Int -> a
getAppendedElement xs ys n
    | n < lxs = xs ! n
    | otherwise = ys ! (n-lxs)
    where lxs = length xs

appendS :: Arr a -> Arr a -> Arr a
appendS xs ys = tabulate (getAppendedElement xs ys) (length xs + length ys
```

Como $W_{(!)}(n) \in \Theta(1)$, $W_{getAppendedElement}(n) \in \Theta(1)$ en donde n es la longitud del arreglo para $W_{(!)}(n)$ y el entero el cual usaremos para acceder al elemento en $W_{getAppendedElement}(n)$

Luego:

$$
\begin{aligned}
W_{appendS}(s, t) &= c_0 + W_{tabulate}(|s| + |t|) \\
&= c_0 + \sum_{i=0}^{|s|+|t|} W(getAppendedElement\,(i)) \\
&= c_0 + c_1\,(|s| + |t|) \\
W_{appendS}(s, t) &\in \Theta\,(|s| + |t|)
\end{aligned}
$$

y:

$$
\begin{aligned}
S_{appendS}(s, t) &= c_0 + W_{tabulate}(|s| + |t|) \\
&= c_0 + \max_{i=0}^{|s|+|t|} W(getAppendedElement\,(i)) \\
&= c_0 + c_2 \\
S_{appendS}(s, t) &\in \Theta\,(1)
\end{aligned}
$$

2.3 reduceS

Primero, veamos dos funciones auxiliares:

```
ceilDivTwo :: Int -> Int
ceilDivTwo n =  n `div` 2 + mod n 2

inPair :: (a -> a -> a) -> Arr a -> Int -> Int -> a
inPair f xs n k
    | dk == (n - 1) = xs ! (n - 1)
    | otherwise = f (xs ! dk) (xs ! (dk+1))
    where dk = 2*k
```

Es fácil ver que $W_{ceilDivTwo}(n) \in \Theta(1)$, en donde n es el entero que se divide, dado que son operaciones de costo constante. Si suponemos que $W(f(a,b)) \in \Theta(1)$, en donde a y b son los argumentos de la operación, entonces podemos analizar *contract* fácilmente:

```
contract :: (a -> a -> a) -> Arr a -> Arr a
contract f xs = tabulate (inPair f xs n) (ceilDivTwo n)
    where n = length xs
```

Como *contract* está basada en *tabulate* usando funciones de costo constante, tendrán coste secuencial: (Donde n es la longitud del arreglo)

$$\begin{aligned} W_{contract}(n) = W_{tabulate}(n) &\in O\left(\sum_{i=0}^{n} W(f(i))\right) \le O\left(\sum_{i=0}^{n} c\right) \\ &= O(nc) \\ W_{contract}(n) &\in O(n) \end{aligned}$$

y paralelo:

$$\begin{aligned} S_{contract}(n) = S_{tabulate}(n) &\in O\left(\max_{i=0}^{n} S(f(i))\right) \le O\left(\max_{i=0}^{n} c\right) \\ &= O(c) \\ S_{contract}(n) &\in O(1) \end{aligned}$$

Con esto ya podemos analizar *reduceS*:

```
myReduceS :: (a -> a -> a) -> a -> Arr a -> a
myReduceS f b xs
    | n == 0 = b
    | n == 1 = b `f` (xs ! 0)
    | n  > 1 = myReduceS f b reduced
    where
        n = length xs
        reduced = contract f xs
```

Podemos escribir el trabajo de la longitud de la secuencia xs (n):

$$W_{reduceS}(n) = \underbrace{W\left(\left\lceil \frac{n}{2} \right\rceil\right)}_{\text{length } reduced} + \underbrace{a_1 n}_{W_{contract} \in O(n)} + a_2$$

Sabemos que esta recurrencia es de orden $\mathcal{O}(n)$ (demostración en apéndice 3.1), es decir $W_{reduceS}(n) \in \mathcal{O}(n)$ Y podemos escribir a la profundidad de la longitud de la secuencia $xs(n)$ como:

$$S_{reduceS}(n) = \underbrace{S\left(\left\lceil\frac{n}{2}\right\rceil\right)}_{\text{length } reduced} + \underbrace{a_1}_{S_{contract} \in O(1)} + a_2$$

Sabemos que esta recurrencia es de orden $\mathcal{O}(lg(n))$ (demostración en apéndice 3.2), es decir $S_{reduceS}(n) \in \mathcal{O}(lg(n))$

2.4 scanS

Primero, veamos el costo de expandir la solución de un problema menor al original, aplicándolo con un tabulate

```
comparate :: (a -> a -> a) -> a -> Arr a -> Arr a -> Int -> a
comparate f b s s' 0 = b
comparate f b s s' n
    | even n = s' ! (n `div` 2)
    | otherwise = f (s' ! (n `div` 2)) (s ! (n-1))
```

Es fácil ver que comparate está constituido por operaciones de costo constante y por $W(f(a,b))$, donde a y b son los argumentos de la operación f, el cual suponemos de costo constante, por ende $W_{comparate}(n) = O(1)$
, donde n es el entero con el cual accederemos al arreglo. Por la misma razón, tambien suponemos que $S(f(a,b))$ es de costo constante, resultando así que $S_{comparate}(n) = O(1)$
Luego, $tabulate\,(comparate\, f\, b\, s\, r)\, n$ tiene costo de trabajo: (Donde n será el largo del arreglo por el cual tabulate lo construirá, es decir $|s|$)

$$\begin{aligned} W_{tabulate}(n) &= W_{tabulate}(n) \in O\left(\sum_{i=0}^{n} W(comparate(i))\right) \leq O\left(\sum_{i=0}^{n} c\right) \\ &= O(nc) \\ W_{tabulate}(n) &\in O(n) \end{aligned}$$

Y costo de profundidad:

$$\begin{aligned} S_{tabulate}(n) &= S_{tabulate}(n) \in O\left(\max_{i=0}^{n} S(comparate(i))\right) \leq O\left(\max_{i=0}^{n} c\right) \\ &= O(c) \\ S_{tabulate}(n) &\in O(1) \end{aligned}$$

```haskell
myScanS :: (a -> a -> a) -> a -> Arr a -> (Arr a,a)
myScanS f b s | n == 0 = (empty,b)
              | n == 1 = (Arr.fromList [b],b `f` (s ! 0))
              | n > 1 = (s'',t)
              where
               n = length s
               s' = contract f s
               (r,t) = myScanS f b s'
               s'' = tabulate (comparate f b s r) n
```

Como *contract* está basada en *tabulate* usando funciones de costo constante, tendrán coste secuencial: (Donde n es la longitud del arreglo, es decir $|s|$)

$$
\begin{aligned}
W_{contract}(n) = W_{tabulate}(n) &\in O\left(\sum_{i=0}^{n} W(f(i))\right) = O\left(\sum_{i=0}^{n} c\right) \\
&= O(nc) \\
W_{contract}(n) &\in O(n)
\end{aligned}
$$

y paralelo:

$$
\begin{aligned}
S_{contract}(n) = S_{tabulate}(n) &\in O\left(\max_{i=0}^{n} S(f(i))\right) = O\left(\max_{i=0}^{n} c\right) \\
&= O(c) \\
S_{contract}(n) &\in O(1)
\end{aligned}
$$

Con esto ya podemos analizar $scanS$. Podemos escribir a n como el trabajo de la longitud de la secuencia s ($|s|$):

$$W_{myScanS}(n) = W_{myScanS}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \underbrace{a_1 n}_{W_{contract}(n) \in O(n)} + \underbrace{a_2 n}_{W_{tabulate}(n) \in O(n)} + a_3$$

Sabemos que esta recurrencia es de orden $\mathcal{O}(n)$ (demostración en apéndice 3.1))

Y al costo de la profundidad:

$$S_{myScanS}(n) = S_{myScanS}(n)\left(\left\lceil \frac{n}{2} \right\rceil\right) + \underbrace{a_1}_{S_{contract}(n) \in O(n)} + \underbrace{a_2}_{S_{tabulate}(n) \in O(n)} + a_3$$

Sabemos que esta recurrencia es de orden $\mathcal{O}(lg(n))$ (demostración en apéndice 3.2))

3 Apendice

Aquí se justificaran recurrencias en común encontradas en las funciones, aplicando diversos métodos de resolución de recurrencia para demostrarlas.

3.1 $W\left(\left\lceil \frac{n}{2} \right\rceil\right) + a_1 n + a_2 \in \mathcal{O}(n)$

Si $W(n)$ es de la forma

$$W(n) = W\left(\left\lceil \frac{n}{2} \right\rceil\right) + a_1 n + a_2$$

podemos demostrar que $W \in O(n)$ usando la regla de suavidad.

Sabemos que n es una función suave, debemos ver que W es eventualmente no decreciente.

- Caso base.

$$
\begin{aligned}
W(1) &= a \\
W(2) = W(1+1) = W(1) + a_1 2 + a_2 &\geq W(1)
\end{aligned}
$$

- Caso inductivo
  Supongamos que $W(x) \leq W(x+1) \quad \forall x < n$ Luego:

$$
\begin{aligned}
W(n) &= W\left(\left\lceil \frac{n}{2} \right\rceil\right) + a_1 n + a_2 \\
&\overset{(HI)}{\leq} W\left(\left\lceil \frac{n+1}{2} \right\rceil\right) + a_1 n + a_2 \\
&\leq W\left(\left\lceil \frac{n+1}{2} \right\rceil\right) + a_1(n+1) + a_2 \\
&= W(n+1) \\
W(n) &\leq W(n+1)
\end{aligned}
$$

Luego, $W(n)$ es eventualmente no decreciente. Ahora veamos que $W(2^k) \in O(2^k)$ Supongamos que $W(2^x) < c2^x \quad \forall x < k$ Luego:

$$
\begin{aligned}
W(2^k) &= W\left(\left\lceil \frac{2^k}{2} \right\rceil\right) + a_1 2^k + a_2 \\
&= W\left(\frac{2^k}{2}\right) + a_1 2^k + a_2 \\
&= W\left(2^{k-1}\right) + a_1 2^k + a_2 \\
&\leq c2^{k-1} + a_1 2^k + a_2 \\
&\leq c2^{k-1} + a_1 2 \cdot 2^{k-1} + a_2 \\
&\leq c2^{k-1} + a_1 2 \cdot 2^{k-1} + a_2 2^{k-1} \\
&\leq (c + 2a_1 + a_2) 2^{k-1} \\
&\leq 2c \cdot 2^{k-1} \qquad\qquad 2a_1 + a_2 <= c \\
&\leq c \cdot 2^k
\end{aligned}
$$

Para el caso base:

$$
\begin{aligned}
W(1) &= a \\
W(2) &= W(1) + a_1 2 + a_2 \\
&= a + a_1 2 + a_2 \\
&\leq c \cdot 2 \qquad\qquad a + 2a_1 + a_2 \leq c
\end{aligned}
$$

Luego, tomando $c = a_3 + 2a_1 + a_2$ y $n = 2$:

$$W(2^k) \leq c \cdot n \qquad \forall 2^k \geq 2$$

Es decir, $W(2^k) \in O(2^k)$

Aplicando la regla de suavidad tenemos que

$$W(n) \in O(n)$$

3.2 $S\left(\left\lceil\frac{n}{2}\right\rceil\right) + a_0 \in \mathcal{O}(lg(n))$

Si $S(n)$ es de la forma

$$S(n) = S\left(\left\lceil\frac{n}{2}\right\rceil\right) + a_0$$

podemos demostrar que $W \in O(n)$ usando la regla de suavidad.
Sabemos que n es una función suave, debemos ver que W es eventualmente no decreciente.

- Caso base.

$$\begin{aligned} W(1) &= a \\ W(2) &= W(1+1) = W(1) + a_0 \geq W(1) \end{aligned}$$

- Caso inductivo
  Supongamos que $S(x) \leq S(x+1) \quad \forall x < n$ Luego:

$$\begin{aligned} S(n) &= S\left(\left\lceil\frac{n}{2}\right\rceil\right) + a_0 \\ &\overset{(HI)}{\leq} S\left(\left\lceil\frac{n+1}{2}\right\rceil\right) + a_0 \\ &\leq S\left(\left\lceil\frac{n+1}{2}\right\rceil\right) + a_0 \\ &= S(n+1) \\ S(n) &\leq S(n+1) \end{aligned}$$

Luego, $S(n)$ es eventualmente no decreciente. Ahora veamos que $S(2^k) \in O\left(lg\left(2^k\right)\right)$ Supongamos que $S(2^x) < c lg\left(2^x\right) \quad \forall x < k$ Luego:

$$\begin{aligned} S(2^k) &= S\left(\left\lceil\frac{2^k}{2}\right\rceil\right) + a_0 \\ &\leq S\left(\frac{2^k}{2}\right) + a_0 \\ &\leq S\left(2^{k-1}\right) + a_0 \\ &\leq c \cdot lg\left(2^{k-1}\right) + a_0 \\ &\leq c \cdot lg\left(2^{k-1}\right) + c \qquad , a_0 \leq c \\ &\leq c \cdot (lg\left(2^{k-1}\right) + 1) \\ &\leq c \cdot (lg\left(2^{k-1}\right) + lg(2)) \\ &\leq c \cdot lg\left(2 \cdot 2^{k-1}\right) \\ &\leq c \cdot lg\left(2^k\right) \end{aligned}$$

Para el caso base:

$$\begin{aligned} S(1) &= a \\ S(2) &= W(1) + a_0 \\ &= a + a_0 \\ &\leq c \cdot 2 \qquad a + a_0 \leq c \end{aligned}$$

Luego, tomando $c = a + a_0$ y $n = 2$:

$$W(2^k) \leq c \cdot lg\left(2^k\right) \qquad \forall 2^k \geq 2$$

Es decir, $S(2^k) \in O(lg\left(2^k\right))$

Aplicando la regla de suavidad tenemos que

$$S(n) \in O(lg(n))$$

3.3 $W(n-2) + c_1 \in \mathcal{O}(n)$

Si $W(n)$ es de la forma

$$W(n-2) + c_1$$

Se puede intuir que la recurrencia es de costo $\mathcal{O}(n)$ por lo cual procederemos a demostrarlo por substitución.
Supongo que vale que

$$\begin{aligned} &\forall n \in \mathbb{N}, \exists c \in \mathbb{R}, n_0 \in \mathbb{N}/ \\ &W(x) \leq cx, \forall x < n \end{aligned}$$

- Caso base

$$W(1) = a_1 \leq 1c \leftrightarrow a_1 \leq c$$

- Caso inductivo

$$\begin{aligned} W(n) &= W(n-2) + c_1 \\ &\stackrel{(H.I)}{\leq} c(n-2) + c_1 \\ &\leq cn - 2c + c_1 \\ &\leq cn \leftrightarrow \forall c \geq \frac{c_1}{2} \end{aligned}$$

Luego tomando $c = max\left(a_1, \frac{c_1}{2}\right)$ y $n_0 = 1$

$$W(n) \leq cn, \qquad \forall n \geq n_0$$

Es decir, $W(n) \in \mathcal{O}(n)$