Tarea 5
Paralelismo

Ejercicio 1 (Multiplicación de polinomios) **38 Pt**

Una tarea que es muy amena a la paralelización es la multiplicación de polinomios. Si $p(x) = \sum_{i=0}^{m} p_i x^i$ y $q(x) = \sum_{i=0}^{n} q_i x^i$, entonces $(p \cdot q)(x) = \sum_{i=0}^{n+m} r_i x^i$ donde $r_i = \sum_{j=0}^{i} p_j q_{i-j}$. Observe que los coeficientes r_i son independientes entre sí y pueden calcularse en paralelo.

Para construir polinomios y acceder a su información tenemos la siguiente interfaz:

```
type Coeff = Integer

mkPoly :: [Coeff] -> Poly
degree :: Poly -> Int
coeff  :: Poly -> Int -> Coeff
```

Por ejemplo, para definir el polinomio $p(x) = 1 - 2x + 4x^3$, consultar su grado y los coeficientes que acompañan a x y a x^4, hacemos

```
> let p = mkPoly [1,-2,0,4]
> degree p
3
> coeff p 1
-2
> coeff p 4
0
```

(a) [6 Pt] Defina la función `coeffProd :: Poly -> Poly -> Int -> Coeff` que calcula el coeficiente i-esimo (el que acompaña a x^i) del producto de dos polinomios.
Una vez que haya provisto la definición de `coeffProd`, podrá usar la función

```
seqProd :: Poly -> Poly -> Poly
seqProd p1 p2 = mkPoly (map (coeffProd p1 p2) [0..d])  where
  d = degree p1 + degree p2
```

que calcula secuencialmente el producto de dos polinomios.

(b) [1 Pt] Para medir la performance del programa vamos a multiplicar dos polinomios `pa` y `pb` de grado 2000 y 5000, respectivamente. Todo el código necesario ya está en la función `main`. Se pide que compile su programa (activando la optimización `-O2`), y ejecute el programa compilado activando las estadísticas del *runtime* para determinar su tiempo de ejecución (el *elapsed time*).

(c) [9 Pt] Defina ahora la función `parProd :: Poly -> Poly -> Poly` que computa (los coeficientes de) el producto de dos polinomios en paralelo. Para ello, agrupe los coeficientes en grupos de a 15 (es decir, cada *spark* debe albergar el cómputo de 15 coeficientes).

(d) [1 Pt] Vuelva a compilar el programa (usando ahora esta versión paralela `parProd` para calcular el producto de `pa` y `pb`) activando también la misma opción de optimización que antes. Ejecute el programa compilado utilizando 2 *cores* y provea el tiempo de ejecución.

(e) [1 Pt] Compute el *speedup* alcanzado al utilizar 2 *cores*.

(f) [9 Pt] Defina ahora la función `par1Prod :: Poly -> Poly -> Poly` que computa el producto de dos polinomios en paralelo, pero con una granularidad mucho más fina: generando un *spark* por cada coeficiente.

(g) [1 Pt] Determine el *speedup* alcanzado por esta versión paralela (al utilizar 2 *cores*).

(h) [5 Pt] ¿Por qué cree que este *speedup* es tan bajo (o directamente nulo)? Justifique.

(i) [5 Pt] ¿Por qué en el `main` se imprime el número de coeficientes no nulos `nonNullCoeff (seqProd pa pb)` del producto de `pa` por `pb` y no simplemente su producto `seqProd pa pb`?

Ejercicio 2 (Integral definida) **22 Pt**

El *método del trapecio* permite aproximar una integral definida de la siguiente manera:

$$\int_a^b f(x)\,dx \quad \approx \quad \frac{h}{2} \cdot \sum_{i=0}^{n} f(a+ih) + f(a+(i+1)h)\ ,$$

donde $h = \frac{b-a}{n}$ y n es el número de subintervalos en los que se divide el intervalo $[a, b]$ para generar la aproximación (en general, en cuanto más grande es n, mejor es la aproximación). El método asume además que $f \geq 0$ en $[a, b]$.

(a) [10 Pt] Defina la función `integral :: (Number -> Number) -> Number -> Number -> Int -> Number` que implementa el método del trapecio arriba descrito (los parámetros son pasados así: `integral f a b n`).

(b) [10 Pt] Defina una versión paralela `pintegral` que

1. divida la sumatoria $\sum_{i=0}^{n} g(i)$ con $g(i) = f(a+ih) + f(a+(i+1)h)$ en varias subsumatorias de 50 sumandos cada una, de la siguiente manera:

$$\sum_{i=0}^{n} g(i) \ = \ \sum_{i=0}^{49} g(i) + \sum_{i=50}^{99} g(i) + \sum_{i=100}^{149} g(i) + \cdots$$

2. genere un *spark* para evaluar cada una de estas subsumatorias,

3. y finalmente sume los valores de estas subsumatorias para obtener el valor de la sumatoria total.

(c) [2 Pt] Determine el *speedup* de la versión paralela usando dos *cores* al evaluar la integral dada en la función `main`.

Hint. Si su implementación requiere hacer la partición `[1..n] = [[0..49], [50..99], ...]` , le puede resultar útil la función `chunksOf` de la librería `split` (exportada por el módulo `Data.List.Split`).