Pauta de Corrección

Primer Certamen
Algoritmos y Complejidad

6 de mayo de 2017

1. Estamos escribiendo:

   $$x = g(x) = x + \alpha f(x)$$

   Esto da la iteración:

   $$x_{n+1} = x_n + \alpha f(x_n)$$

   Expandiendo alrededor del cero x^* queda:

   $$x_{n+1} = x_n + \alpha \left(f(x^*) + f'(x^*)(x_n - x^*) + \frac{1}{2} f''(x^*)(x_n - x^*)^2 + \cdots \right)$$

   Con la substitución $e_n = x_n - x^*$, y reconociendo $f(x^*) = 0$, resulta:

   $$e_{n+1} = e_n + \alpha \left(f'(x^*) e_n + \frac{1}{2} f''(x^*) e_n^2 + \cdots \right)$$

   Para maximizar el orden de convergencia, buscamos eliminar el término en e_n, lo que se logra con $\alpha = -1/f'(x^*)$:

   $$e_{n+1} = -\frac{f''(x^*)}{2f'(x^*)} e_n^2 + \cdots$$

   lo que es convergencia cuadrática.

Puntajes

Total	20
Plantear iteración en e_n	5
Aproximar por Taylor	5
Maximizar orden de convergencia	5
Orden resultante	5

2. Cada punto por turno. Son independientes, no se requieren las soluciones de los anteriores en cada caso.

   a) La forma de Lagrange del polinomio interpolador es:

   $$L_n(x) = \sum_{0\le k\le n} \prod_{\substack{0\le j\le n \\ j\neq k}} \frac{x - x_j}{x_k - x_j} f(x_k)$$

   Definamos:

   $$w(x) = \prod_{0\le k\le n} (x - x_k)$$

   Al amplificar la expresión baricéntrica y simplificar, en el numerador queda $L_n(x)$, en el denominador queda la expresión que resulta si $f(x_k) = 1$ siempre. Como indica la pregunta, estos son los polinomios interpoladores de Lagrange de f y 1, con lo que $p_n(x_k) = f(x_k)$.

   b) Si el denominador es el polinomio interpolador de Lagrange de grado n de la función 1 en los puntos x_k, es idénticamente 1, ya que el polinomio interpolador es único.

   c) Si se requiere calcular $p_n(x)$ para varios valores de x, podemos precalcular los w_k y los $w_k f(x_k)$. Hay un total de $n(n-1)/2$ factores $x_i - x_j$ con $i \neq j$, cada w_k es el producto de $n-1$ de ellos, o sea $n-2$ multiplicaciones cada uno. Calcular $w_k f(x_k)$ es una multiplicación adicional. En resumen, la preparación calcula $2n$ valores, con un costo de $n(n-1)/2$ sumas/restas y $n(n-2) + n(n-1) = n(2n-3)$ multiplicaciones.

   Dados los valores precalculados, para evaluar $p_n(x)$ calculamos n restas $x - x_k$, $2n$ divisiones, luego $2n-2$ sumas para calcular las sumatorias, y una división final. En total, son $3n-2$ sumas/restas y $2n+1$ divisiones.

Puntajes

Total			20
a) Interpola		10	
Forma de Lagrange	4		
Amplificar por $w(x)$	3		
Reconocer los polinomios del caso	3		
b) Polinomio		5	
Polinomio interpolante es único	2		
Polinomio que interpola 1 es 1	2		
Por (c) es el polinomio pedido	1		
c) Contar operaciones		10	
Valores precalculados	5		
Cálculo del valor	5		

3. Por la pista, definimos $p[i,t]$ como verdadero si entre $\{a_1,\ldots,a_i\}$ hay un subconjunto que suma t.

   La recurrencia sobre i es bastante obvia. Para incluir a_{i+1}, podemos simplemente no usarlo (es posible obtener t con $\{a_1,\ldots,a_i\}$) o lo incluimos (debemos obtener $t-a_{i+1}$ con $\{a_1,\ldots,a_i\}$). O sea:

   $$p[i+1,t] = p[i,t] \vee p[i,t-a_{i+1}]$$

   Tenemos condiciones iniciales:

   $$p[i,t] = \begin{cases} F & \text{si } t>0 \text{ y } i=0 \\ T & \text{si } t=0 \end{cases}$$

   Es simple llevar esto al algoritmo 1 recursivo. Debemos tener cuidado de no considerar incluir a_i si $a_i > t$, como se muestra.

Algoritmo 1: Hay subconjunto de $\{a_1,\ldots,a_i\}$ con la suma t dada

```
function HaySuma(i, t)
    if t = 0 then
        return T
    else if i = 0 then
        return F
    else
        if t < a_i then
            return HaySuma(i − 1, t)
        else
            return HaySuma(i − 1, t) ∨ HaySuma(i − 1, t − a_i)
        end
    end
```

   Una solución por programación dinámica llena el arreglo desde $t=0$ hacia arriba, para cada $t=0,1,\ldots$ podemos ir llenando $p[t,i]$ sistemáticamente, con lo que los valores requeridos los habremos calculado antes. Ver el algoritmo 2. Luego de ejecutar este algoritmo, $p[n,s]$ nos dice si es posible o no la suma dada.

   Si quisiéramos además obtener un subconjunto que logra la suma pedida, habría que registrar con los $p[t,i]$ verdaderos cuál fue la opción que dio verdadero, y revisar hacia atrás desde el resultado final.

   La complejidad de este algoritmo es $O(ns\log s$, estamos calculando esencialmente ns elementos, cada uno de los cuales significa algunas operaciones entre palabras de $\log_2 s$ bits. Vimos en *Informática Teórica* (INF-155) que este problema (SUBSETSUM) es NP-completo. En términos del largo de la representación en binario este algoritmo no es polinomial, pero sí lo es en términos

Algoritmo 2: Subconjunto de $\{a_1, \ldots, a_n\}$ que suma s, programación dinámica

```
for t ← 1 to s do
    p[0, t] = F
end
for i ← 0 to n do
    p[i, 0] ← T
end
for t ← 1 to s do
    for i ← 1 to n do
        if a_i > t then
            p[i, t] ← p[i − 1, t]
        else
            p[i, t] ← p[i − 1, t] ∨ p[i − 1, t − a_i]
        end
    end
end
```

de los valores de los datos de entrada (los n elementos a_i y s). A tales algoritmos se les llama *pseudopolinomiales*.

Puntajes

Total		30
Plantear la recursión	10	
Condiciones de contorno	5	
Definir orden del cálculo de los valores	15	

4. Cada parte es independiente.

   a) La demostración es por contradicción. Consideremos un grafo $G = (V, E)$ con pesos $w\colon E \to \mathbb{R}^+$. Sea e un arco de costo máximo en un ciclo $C \leq G$, y supongamos que e necesariamente se incluye en un árbol recubridor mínimo T. Si eliminamos e de T, este se descompone en dos árboles T_1 y T_2. Como e es parte de C, hay vértices de C en T_1 y en T_2 (están los extremos de e en cada uno de ellos), por lo que hay otro arco $e' \in C$ que conecta T_1 con T_2. Vale decir, $(T \setminus \{e\}) \cup \{e'\}$ es un árbol recubridor, y $w(e') \leq w(e)$ por como se eligió e. Si $w(e') = w(e)$, podemos construir un árbol recubridor del mismo costo que no incluye e; si $w(e') \leq w(e)$ uno de menor costo. En ambos casos tenemos un árbol recubridor mínimo que no incluye e.

   b) La idea de un algoritmo voraz basado en lo anterior es considerar los arcos en orden de peso decreciente, para cada arco ver si es parte de un ciclo en el grafo actual, eliminándolo de ser así. Nuestras tres propiedades son:

   Greedy choice: Por la proposición anterior, hay un árbol recubridor mínimo que omite el arco de mayor peso que participa en un ciclo.

   Inductive substructure: Al eliminar un arco que es parte de un ciclo, el grafo resultante sigue siendo conexo, y tiene un árbol recubridor.

   Subestructura óptima: Un árbol recubridor mínimo del grafo si el arco de mayor peso que pertenece a un ciclo es un árbol recubridor mínimo del grafo completo, por la misma proposición anterior.

   Como se cumplen las tres propiedades, el algoritmo da una solución óptima.

Puntajes

Total			25
a) Demostrar el teorema		15	
b) Algoritmo		10	
Esbozar el algoritmo	4		
Demostrar las tres propiedades	6		

5. De las tres, backtracking es la más general. Siempre puede aplicarse, explorando exhaustivamente todas las posibilidades.

   Programación dinámica es aplicable siempre que el tomar una decisión restringe opciones futuras, dando lugar a problemas "más chicos" similares. A falta de un criterio que nos indique *a priori* cuál es la decisión correcta, intentamos todas las opciones y elegimos la mejor. Los problemas menores pueden resolverse recursivamente. En caso que la estructura del problema sea tal que hayan problemas menores que se repiten, podemos registrar sus soluciones para referencia futura (*memoización*) o organizar el cálculo de forma de resolver los problemas sistemáticamente en orden de tamaño creciente, de forma que cuando se requieran soluciones de problemas menores estas ya estén registradas.

   Algoritmos voraces son aplicables en situaciones como la anterior, en las cuales hay un criterio "local" (que no usa información resultante de exploración del problema) que permite decidir cuál es la elección correcta, evitando la necesidad comparar alternativas o de búsqueda.

   Comparativamente, los algoritmos voraces son simples y eficientes, pero "nunca son aplicables" (rara vez hay un criterio que permita determinar la elección correcta en forma local, y puede ocurrir que existe pero no se nos ocurre). Programación dinámica (o memoización) reduce una búsqueda extensa a algo más manejable. Programación dinámica es simple de programar, y claramente de mayor costo computacional que un algoritmo voraz. Backtracking es universalmente aplicable y simple de programar, pero el costo computacional puede ser excesivo.

Puntajes

Total		20
Descripción de backtracking	5	
Descripción de programación dinámica	5	
Descripción de algoritmo voraz	5	
Comparar las tres	5	

HvB/LaTeX2ε