Práctica 2: Técnicas Algorítmicas

Compilado: 26 de agosto de 2025

Backtracking

SumaSubconjuntosBT

1. En este ejercicio vamos a resolver el problema de suma de subconjuntos con la técnica de *backtracking*. Dado un multiconjunto $C = \{c_1, \ldots, c_n\}$ de números naturales y un natural k, queremos determinar si existe un subconjunto de C cuya sumatoria sea k. Vamos a suponer fuertemente que C está ordenado de alguna forma arbitraria pero conocida (i.e., C está implementado como la secuencia $c_1, \ldots, c_n$ o, análogamente, tenemos un iterador de C). Las *soluciones (candidatas)* son los vectores $a = (a_1, \ldots, a_n)$ de valores binarios; el subconjunto de C representado por a contiene a c_i si y sólo si $a_i = 1$. Luego, a es una solución *válida* cuando $\sum_{i=1}^{n} a_i c_i = k$. Asimismo, una *solución parcial* es un vector $p = (a_1, \ldots, a_i)$ de números binarios con $0 \leq i \leq n$. Si $i < n$, las soluciones *sucesoras* de p son $p \oplus 0$ y $p \oplus 1$, donde $\oplus$ indica la concatenación.

   a) Escribir el conjunto de soluciones candidatas para $C = \{6, 12, 6\}$ y $k = 12$.

   b) Escribir el conjunto de soluciones válidas para $C = \{6, 12, 6\}$ y $k = 12$.

   c) Escribir el conjunto de soluciones parciales para $C = \{6, 12, 6\}$ y $k = 12$.

   d) Dibujar el árbol de *backtracking* correspondiente al algoritmo descrito arriba para $C = \{6, 12, 6\}$ y $k = 12$, indicando claramente la relación entre las distintas componentes del árbol y los conjuntos de los incisos anteriores.

   e) Sea $\mathcal{C}$ la familia de todos los multiconjuntos de números naturales. Considerar la siguiente función recursiva $\text{ss}\colon \mathcal{C} \times \mathbb{N} \to \{V, F\}$ (donde $\mathbb{N} = \{0, 1, 2, \ldots\}$, V indica verdadero y F falso):

$$\text{ss}(\{c_1, \ldots, c_n\}, k) = \begin{cases} k = 0 & \text{si } n = 0 \\ \text{ss}(\{c_1, \ldots, c_{n-1}\}, k) \vee \text{ss}(\{c_1, \ldots, c_{n-1}\}, k - c_n) & \text{si } n > 0 \end{cases}$$

   Convencerse de que $\text{ss}(C, k) = V$ si y sólo si el problema de subconjuntos tiene una solución válida para la entrada C, k. Para ello, observar que hay dos posibilidades para una solución válida $a = (a_1, \ldots, a_n)$ para el caso $n > 0$: o bien $a_n = 0$ o bien $a_n = 1$. En el primer caso, existe un subconjunto de $\{c_1, \ldots, c_{n-1}\}$ que suma k; en el segundo, existe un subconjunto de $\{c_1, \ldots, c_{n-1}\}$ que suma $k - c_n$.

   f) Convencerse de que la siguiente es una implementación recursiva de ss en un lenguaje imperativo y de que retorna la solución para C, k cuando se llama con $C, |C|, k$. ¿Cuál es su complejidad?

   1) subset_sum(C, i, j): // implementa $\text{ss}(\{c_1, \ldots, c_i\}, j)$
   2) Si $i = 0$, retornar $(j = 0)$
   3) Si no, retornar subset_sum$(C, i - 1, j) \vee$ subset_sum$(C, i - 1, j - C[i])$

   g) Dibujar el árbol de llamadas recursivas para la entrada $C = \{6, 12, 6\}$ y $k = 12$, y compararlo con el árbol de *backtracking*.

h) Considerar la siguiente *regla de factibilidad*: $p = (a_1, \ldots, a_i)$ se puede extender a una solución válida sólo si $\sum_{q=1}^{i} a_q c_q \leq k$. Convencerse de que la siguiente implementación incluye la regla de factibilidad.

1) subset_sum(C, i, j): // implementa ss$(\{c_1, \ldots, c_i\}, j)$
2) Si $j < 0$, retornar **falso** // regla de factibilidad
3) Si $i = 0$, retornar $(j = 0)$
4) Si no, retornar subset_sum$(C, i-1, j) \vee$ subset_sum$(C, i-1, j - C[i])$

i) Definir otra regla de factibilidad, mostrando que la misma es correcta; no es necesario implementarla.

j) Modificar la implementación para imprimir el subconjunto de C que suma k, si existe. **Ayuda:** mantenga un vector con la solución parcial p al que se le agregan y sacan los elementos en cada llamada recursiva; tenga en cuenta de no suponer que este vector se copia en cada llamada recursiva, porque cambia la complejidad.

MagiCuadrados

2. Un *cuadrado mágico de orden n*, es un cuadrado con los números $\{1, \ldots, n^2\}$, tal que todas sus filas, columnas y las dos diagonales suman lo mismo (ver figura). El número que suma cada fila es llamado *número mágico*.

2	7	6
9	5	1
4	3	8

Existen muchos métodos para generar cuadrados mágicos. El objetivo de este ejercicio es contar cuántos cuadrados mágicos de orden n existen.

a) ¿Cuántos cuadrados habría que generar para encontrar todos los cuadrados mágicos si se utiliza una solución de fuerza bruta?

b) Enunciar un algoritmo que use *backtracking* para resolver este problema que se base en la siguientes ideas:

- La solución parcial tiene los valores de las primeras $i-1$ filas establecidos, al igual que los valores de las primeras j columnas de la fila i.
- Para establecer el valor de la posición $(i, j+1)$ (o $(i+1, 1)$ si $j = n$ e $i \neq n$) se consideran todos los valores que aún no se encuentran en el cuadrado. Para cada valor posible, se establece dicho valor en la posición y se cuentan todos los cuadrados mágicos con esta nueva solución parcial.

Mostrar los primeros dos niveles del árbol de *backtracking* para $n = 3$.

c) Demostrar que el árbol de *backtracking* tiene $\mathcal{O}((n^2)!)$ nodos en peor caso.

d) Considere la siguiente poda al árbol de *backtracking*: al momento de elegir el valor de una nueva posición, verificar que la suma parcial de la fila no supere el número mágico. Verificar también que la suma parcial de los valores de las columnas no supere el número mágico. Introducir estas podas al algoritmo e implementarlo en la computadora. ¿Puede mejorar estas podas?

e) Demostrar que el número mágico de un cuadrado mágico de orden n es siempre $(n^3 + n)/2$. Adaptar la poda del algoritmo del ítem anterior para que tenga en cuenta esta nueva información. Modificar la implementación y comparar los tiempos obtenidos para calcular la cantidad de cuadrados mágicos.

MaxiSubconjunto

3. Dada una matriz simétrica M de $n \times n$ números naturales y un número k, queremos encontrar un subconjunto I de $\{1, \ldots, n\}$ con $|I| = k$ que maximice $\sum_{i,j \in I} M_{ij}$. Por ejemplo, si $k = 3$ y:

$$M = \begin{pmatrix} 0 & 10 & 10 & 1 \\ - & 0 & 5 & 2 \\ - & - & 0 & 1 \\ - & - & - & 0 \end{pmatrix},$$

entonces $I = \{1, 2, 3\}$ es una solución óptima.

a) Diseñar un algoritmo de *backtracking* para resolver el problema, indicando claramente cómo se codifica una solución candidata, cuáles soluciones son válidas y qué valor tienen, qué es una solución parcial y cómo se extiende cada solución parcial.

b) Calcular la complejidad temporal y espacial del mismo.

c) Proponer una poda por optimalidad y mostrar que es correcta.

RutaMinima

4. Dada una matriz D de $n \times n$ números naturales, queremos encontrar una permutación π[1] de $\{1, \ldots, n\}$ que minimice $D_{\pi(n)\pi(1)} + \sum_{i=1}^{n-1} D_{\pi(i)\pi(i+1)}$. Por ejemplo, si

$$D = \begin{pmatrix} 0 & 1 & 10 & 10 \\ 10 & 0 & 3 & 15 \\ 21 & 17 & 0 & 2 \\ 3 & 22 & 30 & 0 \end{pmatrix},$$

entonces $\pi(i) = i$ es una solución optima.

a) Diseñar un algoritmo de *backtracking* para resolver el problema, indicando claramente cómo se codifica una solución candidata, cuáles soluciones son válidas y qué valor tienen, qué es una solución parcial y cómo se extiende cada solución parcial.

b) Calcular la complejidad temporal y espacial del mismo.

c) Proponer una poda por optimalidad y mostrar que es correcta.

[1] Una permutacion de un conjunto finito X es simplemente una función biyectiva de X en X.

Palabras en cadena

5. Dada una cadena de letras sin espacios o puntos queremos analizar si se puede subdividir de forma de obtener palabras. Suponiendo que se tiene una función $palabra\colon [a,z] \to bool$ que verifica si una cadena de letras es una palabra en $O(n)$.

   a) Dar una función recursiva que resuelva el problema.

   b) Calcular una cota superior para la complejidad.

   c) Demostrar que el algoritmo es correcto.

Árboles binarios de búsqueda óptimos

6. Dado un conjunto de elementos de $[n]$, y una función $f\colon [n] \to \mathbb{N}$ que nos da la frecuencia de acceso a dichos elementos, decimos que A es un árbol binario de busqueda óptimo si este minimiza el costo de todos los accesos dados por f.

   a) Escribir una función recursiva que devuelva el costo de acceder a todos los elementos usando f.

   b) Dar una cota superior para la complejidad (Ayuda: pasar de la función recursiva a una recurrencia que solo dependa del tamaño de la entrada).

   c) Probar que el algoritmo es correcto

Dobra

7. Dobra se encuentra con muchas palabras en su vida, como es una persona particular la mayoría de estas no le gustan. Para compensar empezó a inventar palabras más agradables. Dobra crea palabras nuevas escribiendo una cadena de caracteres que considera buena, luego borra los caracteres que peor le caen y los reemplaza con _. Luego para mejorar su vida intenta reemplazar estos guiones bajos con letras más aceptables intentando crear palabras más lindas. Dobra considera una palabra como buena si no contiene 3 vocales consecutivas, 3 consonantes consecutivas y al menos contiene una E.

   a) Mostrar alguna solución candidata posible y alguna solución parcial.

   b) Proponer una función recursiva y estimar su complejidad. Asumir que se tiene una función $verificar$ que toma una cadena y devuelve $True$ si es como Dobra quiere o $False$ en caso contrario. [3]

   c) Probar que la función o programa es correcto.

   d) Proponer al menos una poda por factibilidad.

   e) Si b) no tiene una cota superior $O(3^n)$ para la complejidad, analizar el caso donde se separa la recursión en tener o no una letra E y ver si mejora la misma. [5]

[3] Se puede asumir que $verificar$ funciona correctamente.
[5] La mejor complejidad que conocemos es $O(n2^n)$

Cadenas de Adición

8. Dado un entero n decimos que $C = x_1, \ldots, x_k$ es una cadena de adición si cumple lo siguiente

   $\sim$ $1 = x_1 < x_2 < \ldots < x_k = n$

   $\sim$ Para cada $2 \leq j \leq n$ existe $k_1, k_2 < j$ tal que $x_{k_1} + x_{k_2} = x_j$

   a) Encontrar un algoritmo de backtracking que encuentre, si existe, la cadena de adición de longitud mínima.