Backtracking [†]

1. 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.

Solución:

a)

$$separar(S) = \begin{cases} True & \text{si } |S| = 0 \\ \bigvee_{k=1}^{|S|} (separar(S[i+1:] \wedge palabra(S[:i]))) & \text{si } |S| > 0 \end{cases}$$

¿Con qué parametros llamamos a la función para resolver el problema?

b) Analicemos la complejidad, en principio no sabemos cuanto cuesta $palabra(\dots)$, asi que lo que vamos a hacer es contar la cantidad de llamados a la función, tenemos entonces la siguiente recurrencia:

$$T(n) = \sum_{k=0}^{n-1} T(k) + O(n)^{\P}$$

Donde el $O(n)$ viene de llamar a $palabra(S[:i])$ para todo i en el rango. Vamos a usar el siguiente truco que es abrir el O con una constante $C > 0$ y comparar con la recurrencia de un valor menos.

$$T(n-1) = \sum_{k=0}^{n-2} T(k) + C(n-1)$$

Si le restamos esto a la primer recurrencia obtenemos

$$T(n) - T(n-1) = T(n-1) + C$$

Reordenando

$$T(n) = 2T(n-1) + C$$

Si expandimos esta recursión nos queda un árbol binario totalmente balanceado. Ahora queremos evaluar el costo computacional que es la suma de todos los nodos con su costo. En este caso el costo de cada nodo es una constante asi que solo necesitamos contar la cantidad total de nodos. Esto nos da la siguiente complejidad[¶], usando la fórmula de una suma geométrica

$$\sum_{i=0}^{n-1} 2^i = \frac{1-2^n}{1-2} = O(2^n)$$

Osea que una cota que podemos dar es

$$O(costo(palabra(n))2^n))$$

c) Vamos a hacer inducción sobre la longitud de la cadena n. Queremos probar que $separar(S)$ computa correctamente si podemos separar a S en palabras o no para toda para toda cadena S de tamaño

[†] Inspirado fuertemente en `https://jeffe.cs.illinois.edu/teaching/algorithms/book/02-backtracking.pdf`

n. Hacemos inducción fuerte sobre la longitud de la cadena S. Esto significa que nuestro predicado va a valer para **todos** los elementos de tamaño más chico no solo el anterior. Nuestro predicado es:

$P(n) =$ **Para toda cadena S de tamaño n vale que $separar(S)$ nos dice si hay una separación válida o no.**

Caso base: $P(0)$, tenemos que probar que vale para toda cadena de longitud 0. Como no podemos subdividir más la cadena deberíamos devolver $True$ que es exáctamente lo que devuelve nuestra función recursiva si le damos una cadena vacia.

Caso inductivo: Como hacemos inducción fuerte lo que queremos ver es que si vale $P(n)$ para todo $n \leq j$ entonces vale $P(j+1)$. Es decir podemos asumir que vale para **toda** cadena de tamaño $n \leq j$ y queremos ver entonces que pasa con una cadena de $n = j+1$ elementos. Como tenemos que que $j+1 > 0$, entonces si evaluamos la función recursiva caemos en el segundo caso, donde separamos S en dos subcadenas de tamaño k y $n-k$ para $1 \leq k \leq n$, podemos de vuelta separar en dos casos[‖], donde S era una cadena subdividible en el cual deberíamos poder responder $True$ y en el que S no era una cadena subdividible:

i. Si S es subdividible en particular existe un k que da el primer prefijo de la subdivision, entonces $palabra(s[:k])$ devuevle $True$ y nos queda ver que pasa con $separar(S[k:])$, como vale nuestra HI y $|S[k:]| < |S| = j+1$ podemos decir que la función recursiva devuelve $True$ a la subdivisión del sufijo de S y por lo tanto tenemos un $\wedge$ de dos $True$ que da $True$.

ii. El anterior fue el caso fácil, tenemos que ver que pasa si S no admite una subdivisión en palabras, esto significa que dado el primer prefijo, o bien es invalido o el sufijo lo es. Veamos que nuestra función es correcta con respecto a esto. Dado un k, Si $palabra(S[:k])$ es $False$ listo no tenemos que hacer nada, pero si $palabra(S[:k]) = True$, tiene que ser que $separar(S[k:]) = False$ (¿Por qué?). Como sabemos que estamos en el caso que S no tiene ninguna subdivisión válida y $separar(S[k:])$ dio $True$ y como sabemos que por **HI** $separar$ funciona correctamente para cadenas de tamaño $\leq j$ (¿Por qué?[**]) tiene que devolver $False$.

Probamos los dos casos S separable y S no separable, entonces lo probamos para toda cadena, que es lo que queríamos.

2. 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.

Solución:

[¶]De hecho se puede probar fácilmente por un argumento combinatorio que es $\Theta(2^n)$.

[‖]Se pueden hacer ambos a la vez pero tal vez es más claro verlo separado.

[**]Como estamos en el caso que S no es separable tiene que ser que el k que elegimos es mayor que 1 ya que sino toda la palabra sería valida y por lo tanto separable.

[†]Para el interesado por el problema `https://link.springer.com/article/10.1007/BF00264289`

a) Tenemos la siguiente función recursiva AO[§]

$$AO(i,j) = \begin{cases} 0 & \text{si } i > j \\ \sum_{r=i}^{j} f(i) + \min_{1 \le r \le j} AO(i, r-1) + AO(r+1, j) & \text{si no} \end{cases}$$

¿Con qué parametros llamamos a la función para resolver el problema?

b) La función anterior respeta la siguiente recursión

$$T(n) = \sum_{k=1}^{n} (T(k-1) + T(n-k)) + O(n)$$

Notemos que cada parte de la suma se repite ($k = n-(n-k)$), es decir cada $T(k)$ aparece dos veces, además podemos "abrir" el O con una constante $C > 0$ y tenemos que

$$T(n) = 2\sum_{k=0}^{n-1} T(k) + Cn$$

Ahora si tomamos la recursión con un elemento menos tenemos

$$T(n-1) = 2\sum_{k=0}^{n-2} T(k) + C(n-1)$$

Restando $T(n) - T(n-1)$, se cancela casi todo y nos queda

$$T(n) - T(n-1) = 2T(n-1) + C$$

Finalmente reordenando

$$T(n) = 3T(n-1) + C$$

De la misma manera que el ejercicio anterior podemos concluir que $T(n) = O(3^n)$.

c) Veamos que nuestra función recursiva consigue el costo de un ABB de menor costo. La función tiene dos parámetros y nos gustaria hacer inducción sobre el tamaño de la lista, para esto podemos usar como parámetro para la inducción la diferencia (o suma en algunos casos) de los dos índices, es decir hacemos inducción sobre $j - i$. El predicado que obtenemos es el siguiente:

$P(n) = AO(i,j)$ **computa el costo del ABB de costo óptimo para un array de claves que van de** i **a** j **de tamaño** $j - i = n$**.**

Notar que este predicado funciona tambien para $j - i$ negativos en los cuales devuelve 0, lo cual tiene sentido porque es un rango vacío de claves. Seguimos con la inducción para los otros valores:

Caso base: $P(0)$, queremos ver que $AO(i,j)$ computa el arbol binario de búsqueda óptimo para todo $j-i = 0$. Si $j-i = 0$ tenemos un array de un elemento, no caemos en el caso base de la función recursiva sino en el segundo, pero abriendo la recursión una vez vemos que las dos funciones recursivas devuelven 0 y solo evaluamos las frecuencias del $i-$ésimo elemento, que es lo que esperamos que tenga el costo de un ABB de un solo nodo/hoja., vamos a caer en la segunda cláusula pero expandiendo una vez la recursión vemos que ambas llamadas recursivas son 0 y solo sumamos los accesos de la clave $j-$ésima, que es lo que esperamos para un árbol que consiste de una sola hoja. **Caso base:** $P(0)$, queremos ver que $AO(i,j)$ computa el arbol binario de búsqueda óptimo para todo $j - i = 0$. Si $j - i = 0$ tenemos un array de un elemento, no caemos en el caso base de la función recursiva sino

en el segundo, pero abriendo la recursión una vez vemos que las dos funciones recursivas devuelven 0 y solo evaluamos las frecuencias del $i-$ésimo elemento, que es lo que esperamos que tenga el costo de un ABB de un solo nodo/hoja.

Caso inductivo: de vuelta hacemos inducción fuerte, es decir vamos asumir que vale $P(0)$ a $P(k)$ y queremos ver que vale $P(k+1)$.

Como $k+1 > 0$ caemos en el caso recursivo, de acá en adelante sale parecido al ejercicio anterior, solo que no tenemos un caso donde no podamos construir un ABB no óptimo, existe un óptimo y por lo tanto un r que podemos fijar y aplicar la **HI** sobre los subárboles recursivos.

3. 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 llenar estos comodines con letras más aceptables para crear palabras que considera más buenas. Dobra considera una palabra como buena si no contiene 3 vocales consecutivas, 3 consonantes consecutivas y al menos contiene una E. Queremos calcular cuantas palabras buenas se pueden armar a partir de una palabra con comodines.

   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. ‡

   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. ¶

Solución:

a) Tarea para el lector.

b)

$$Dobra(S,i) = \begin{cases} verificar(S) & \text{si } i = n \\ Dobra(S, i+1) & \text{si } S[i] \neq \_ \\ \sum_{c \leftarrow \text{ABC}} Dobra(S[i] \leftarrow c, i+1) & \text{si } |S| \neq i \wedge S[i] = \_ \end{cases}$$

¿Con qué parametros llamamos a la función para resolver el problema?

En este caso no podemos escribir fácilmente fórmula recursiva exacta pero podemos asumir que en todos los casos se llama a Dobra, es decir que son todos _. En tal caso la fórmula sería

$$T(n) = 26T(n-1)$$

Esto nos dice que solo los nodos de la recursión son $O(26^n)$. Pero no nos olvidemos de la función verificar, si esta toma $O(n)$ tendríamos $O(n)$ llamados en las hojas que son $O(26^{n-1})$, sobreacotando obtenemos que la cota superior es $O(n26^n)$.

c) Veamos que $Dobra(S,k)$ es una función válida que resuelve nuestro problema. Lo que queremos probar es que dada una palabra S con comodines, $Dobra(S,0)$ calcula correctamente la cantidad de palabras lindas o válidas que se pueden armar dadas las restricciones que impuso nuestro amigo

§No confundir con Argentum Online
‡Se puede asumir que $verificar$ funciona correctamente.
¶La mejor complejidad que conocemos es $O(n2^n)$

Dobra. Para esto vamos a hacer inducción sobre k el tamaño del prefijo de S. Nuestro predicado es el siguiente

$P(k) = Dobra(S, k)$ **devuelve la cantidad de palabras válidas que se pueden formar modificando el sufijo de S que tiene tamaño $n - k$.**

Caso base $P(0)$ nos dice calcular la cantidad de palabras que puede formar $Dobra(S, n - 0)$, este es el caso base y solo mira la palabra entera, si S es una palabra válida devuelve 1 y si no 0.

Caso inductivo: Esta vez no hace falta hacer inducción fuerte, supongamos que lo probamos para un sufijo de tamaño k, es decir que vale $P(k)$ y queremos ver que vale para un sufijo de tamaño $k + 1$, es decir que vale $P(k + 1)$.

Ahora como $k + 1 > 1$ tenemos que $n - (k + 1) < n$ y caemos en alguno de los casos recursivos. En el primer caso recursivo si el $k + 1-$ésimo elemento no es un comodín aplicamos la **HI** directo que nos dice cuantas palabras válidas hay apartir de acá. En el caso de que si fuese un comodín es lo mismo pero además agregamos una de las 26 letras posibles, para cada una podemos aplicar la HI que nos dice cuantas palabras formamos a partir del siguiente sufijo y sumamos, como lo hacemos sobre todas las posibles (no hay ñ) estamos sumando sobre todas las váldas.

Otra forma de pensarlo es igual al ejercicio 1, para todas las palabras posibles válidas que empiezan en el sufijo $n - k + 1$ hay alguna que es válida por lo cual podemos elegir la letra correcta y por recursión sabemos que nos devuelve la cantidad de válidas para un sufijo más chico, de la misma manera podemos elegir una palabra que nunca sea posible armar y tenemos un caso similar.

d) Nuestro algoritmo del item a) es muy costoso, ya que tenemos que verificar un montón de soluciones de más que sabemos que no son factibles, por ejemplo porque no tienen una E. Podemos sacar el checkeo de verificar si hacemos movimientos válidos y además llevamos un historial de si pusimos o no una E. Si llegamos al final y nunca pusimos o vimos una E descartamos la solución sin llamar a $verificar$ y si había una E la damos como válida. Además, poner las 26 letras no es necesario, si en una posición una vocal era válida podríamos haber puesto las 5, solo nos importa si va una vocal o no, lo mismo con las consonantes. Aplicando todo esto obtenemos la siguiente función recursiva.

$$Dobra(S, i, hayE) = \begin{cases} 1 & \text{si } |S| = i \wedge hayE \\ Dobra(S, i+1, hayE \vee S[i] =' E') & \text{si } S[i] \neq \_ \wedge \text{es una combinación válida} \\ 4Dobra(S \leftarrow' A', i+1, hayE) & \\ \quad +Dobra(S[i] \leftarrow' E', i+1, True) & \text{si } S[i] = \_ \wedge \text{solo una vocal es válida} \\ 21Dobra(S[i] \leftarrow' B', i+1, hayE) & \text{si } S[i] = \_ \wedge \text{solo consonante} \\ 4Dobra(S \leftarrow' A', i+1, hayE) & \\ \quad +21Dobra(S[i] \leftarrow' B', i+1, hayE) & \\ \quad +Dobra(S[i] \leftarrow' E', i+1, True) & \text{si } S[i] = \_ \wedge \text{admite ambas} \\ 0 & \text{caso contrario} \end{cases}$$

Queremos dar una recurrencia, como siempre podemos acotar de más y suponer que caemos en el tercer caso en vez de particularmente en vocal/no vocal. Este tiene dos llamados recursivos, si además tomamos el que no tiene _ son 3 y tenemos la siguiente recurrencia.

$$T(n) = 3T(n-1)$$

Como vimos antes esto es $O(3^n)$.

Todavía podemos hacer algo mejor, una vez que $hayE = True$ no hace falta seguir separando en casos, podemos recorrer cada posición poner o no una E y luego hacer siempre dos llamados recursivos de vocal/consonante. Esto nos da

$$T(n) = 2T(n)$$

Que es $O(2^n)$ por cada elección que hicimos de poner o no una E. Como tenemos n posibles elecciones como mucho tenemos $O(n2^n)$.

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

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

   $\sim$ 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.

   b) Dar alguna poda por factibilidad/optimalidad para el algoritmo anterior.

Solución:

a) Idea: una cadena de adición es un subconjunto de $\{1, \ldots, n\} = [n]$, si listamos todos los conjuntos estos y buscamos la cadena de adición resolveríamos el problema. Es decir queremos $\mathcal{P}([n])$. Podemos pensar esto como iterar sobre todas las cadenas de bits de longitud n dónde cada bit i dice si está el elemento $i-$ésimo en el subconjunto. Iterar esto es ir de 0 a 2^n, es decir sumar a un entero de a 1 y ver los bits que se forman. Luego por cada entero tenemos que verificar si el subconjunto es una cadena de adición, para esto simplemente hay que ir de atras para adelante y verificar que se puede sumar con las posiciones en dos bits anteriores, esto cuesta $O(n^2)$, como hacemos esto para todos los subconjuntos de $\mathcal{P}([n])$ tenemos una complejidad final de $O(n^2 2^n)$.

b) Notar que una cadena de adición que esperamos sea válida tiene que tener siempre el bit $n-$ésimo prendido, es decir podemos sumar hsata n (o al menos esperamos que la cadena lo haga), asi que podemos tirar todos los números que no tengan el bit $n-$ésimo prendido sin verificar si es una caden válida, esto es una poda por factibilidad.

Otra "poda" que podemos hacer es contar la cantidad de bits de la mejor solución vista hasta ahora, si la actual es candidata a ser una solución factible pero tiene más bits usados que la que mejor encontramos podemos también descartarla ya que va a ser una cadena más larga en caso de ser válida.