Examen Final de Diseño y Análisis de Algoritmos

Yonatan José Guerra Pérez
Jose Miguel Pérez Pérez
David Cabrera

September 24, 2024

1 PCC

Descripción del problema

Han pasado 20 años desde que José se graduó de Ciencias de la Computación (haciendo una muy buena tesis) y las vueltas de la vida lo llevaron a convertirse en el presidente del Partido Comunista de Cuba. Una de sus muchas responsabilidades consiste en visitar zonas remotas. En esta ocasión debe visitar una ciudad campestre de Pinar del Río.
También han pasado 20 años desde que David consiguió su título en MATCOM. Tras años de viaje por las grandes metrópolis del mundo, en algún punto decidió que prefería vivir una vida tranquila, aislada de la urbanización, en una tranquila ciudad de Pinar del Río. Las vueltas de la vida quisieron que precisamente David fuera el único universitario habitando la ciudad que José se dispone a visitar.
Los habitantes de la zona entraron en pánico ante la visita de una figura tan importante y decidieron reparar las calles de la ciudad por las que transitaría José. El problema está en que nadie sabía qué ruta tomaría el presidente y decidieron pedirle ayuda a David.
La ciudad tiene n puntos importantes, unidos entre sí por calles cuyos tamaños se conoce. Se sabe que José comenzará en alguno de esos puntos (s) y terminará el viaje en otro (t). Los ciudadanos quieren saber, para cada par s, t, cuántas calles participan en algún camino de distancia mínima entre s y t.

Formalización del problema

Nota: No existen calles de longitud negativa

Dado un grafo G, y dada una matriz $A = (\alpha_{ij})$ la matriz de costos de cada punto i a un punto j tal que $A[i, j] = -1$ si no existe una calle que una los puntos i y j del grafo G; devolver una matriz M tal que $M[i, j] = k_{ij}$ donde $k_{ij} = |C_{ij}|$ donde C_{ij} es un camino de costo mínimo entre i y j.

Primera solución: Fuerza bruta

Una primera aproximación es la solución por fuerza bruta, la cual consiste en para cada par de puntos, probar todos los posibles caminos desde uno a otro

y quedarnos con aquel que tuvo un costo mínimo. La complejidad de esto es $O(n^2) * O((n-2)!) = O(n!)$, lo cual es muy ineficiente.

Segunda solución: Dijkstra

Un segundo enfoque puede ser hacer *Dijkstra* n veces, donde se computan los caminos de costo mínimo para cad punto hacia el resto, y contar entonces la cantidad de aristas que participan en estos.

Solución final: Floyd-Warshall

Sabemos que el *Algoritmo de Floyd-Warshall* resuelve el problema de encontrar el camino de costo mínimo entre dos puntos cualesquiera de un grafo donde no existan ciclos de costo negativo, además de que en la práctica, *Dijkstra* no ha demostrado ser más rápido que *Floyd-Warshall*, entonces podemos usar una modificación de este algoritmo para resolver el problema dado.

El *Algoritmo de Floyd-Warshall* es el siguiente:

```
def floyd_warshall(m):
    n = m.shape[0]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if m[i,k] > 0 and m[k,j] > 0:
                    m[i,j] = min(m[i,j],m[i,k] + m[k,j])
    return m
```

La correctitud y complejidad temporal de este algoritmo la podemos encontrar en diversas fuentes, incluyendo las conferencias de EDA II. La modificación del algoritmo propuesto es la siguiente:

```
def init_memory(m):
    n = m.shape[0]
    mem = {}
    for i in range(n):
        for j in range(n):
            if m[i,j] > 0:
                mem[(i,j)] = 1
            if i == j:
                mem[(i,j)] = 0
    return mem

def floyd_warshall_modified(m):
    n = m.shape[0]
    mem = init_memory(m)

    for k in range(n):
        for i in range(n):
            for j in range(n):
                if m[i,k] > 0 and m[k,j] > 0:
                    if m[i,k] + m[k,j] < m[i,j]:
```

```
                    m[i,j] = m[i,k] + m[k,j]
                    mem[(i,j)] = mem[(i,k)] + mem[(k,j)]
    return mem
```

Correctitud

Sabemos que al terminar el algoritmo, la matriz devuelta M cumple que $M[s,t]$ es el menor costo de un camino de s a t para cualesquiera s,t puntos de la ciudad, falta demostrar que para cada par i,j, si existe un camino desde el punto i hasta el punto j, entonces *memory*$[(i,j)]$ contiene la cantidad de calles que participan en algún camino de costo mínimo de i a j, demostremos esto por inducción en k.
De la demostración de la correctitud del *Algoritmo de Floyd-Warshall*, sabemos que después de k pasos, por el triple ciclo, *matriz[i,j]* contiene el costo de un camino de costo mínimo entre i y j que no pasa por un punto de número mayor que k para todo k.
Notemos que dado que no hay calles de longitud negativa, *matriz[i,j]* es mayor que 0 siempre que se pueda llegar de i a j. Supongamos que $\forall k$ se cumple que después de k recorridos por el triple ciclo *for*, *memory*$[(i,j)]$ contiene la cantidad de calles que intervienen en un camino de costo mínimo entre i y j que no pasa por un punto con número mayor que k, para todo i y j. Para $k=0$ es trivial, pues *memory*$[(i,j)]$ es 1 si se puede llegar de i a j en un paso, 0 si $i=j$, y -1 en otro caso.
Supongamos que esto se cumple $\forall m<k$. Como se entra en la condición solo si es posible llegar desde el punto i hasta el punto k, y desde el punto k hasta el punto j, por un camino que no pasa por un punto de número mayor que $k-1$,se cumple entonces que *memory*$[(i,k)]$ y *memory*$[(k,j)]$ fueron calculados anteriormente, dado que *memory*$[(i,j)]$ fue calculado correctamente para todos los pasos anteriores para todo i y j, por hipótesis de inducción. Luego, *memory*$[(i,k)]$ contiene la cantidad de calles que intervienen en un camino de costo mínimo entre i y k que no pasa por un punto de número mayor que $k-1$, lo mismo pasa para k y j. Por tanto, *memory*$[i,j]$ en el paso k tendrá la cantidad de calles que intervienen en un camino de costo mínimo entre i y j que no pasa por un punto de número mayor que k. Que es lo que se quería demostrar.
Luego, como se pasa n veces por el triple ciclo *for*, que es la cantidad de puntos que hay, al terminar el algoritmo habremos contado la cantidad de calles que intervienen en un camino de costo mínimo para cada par de puntos s y t.

Complejidad temporal

Se sabe y está demostrado con antelación que la complejidad temporal del *Algoritmo de Floyd-Warshall* es $O(n^3)$, y como la modificación que se hizo es $O(1)$, entonces el algoritmo resultante es $O(n^3)$.

Casos Especiales

Un caso especial de este problema que es para cuando existe un camino de costo mínimo que es un camino de *Hamilton*, pues para cada par de nodos en dicho camino, el camino de costo mínimo es una secciíon de este; y dado que es un camino hamiltoniano, todos los nodos están presentes en dicho camino.

Bajo esta condición, podemos hacer Dijkstra y comprobar que hay un camino hamiltoniano. Esta operación es $O(|E|log(V)) + O(V) = O(|E|log(V))$.

2 Alianza de robots LGBT

Descripción del problema

En una futurística ciudad llena de robots, cada robot pertenece a una facción identificada por un color específico. Las piezas de los robots fueron además creadas por varios fabricantes distintos, algunos robots tienen piezas hechas por fabricantes en común.
Los robots de diferentes facciones están buscando formar una alianza estratégica multicolor. Esta alianza debe ser un grupo de robots donde todos tengan, dos a dos, alguna pieza hecha por el mismo fabricante, y lo más importante: debe haber al menos un robot de cada color en esta alianza.

Dado un grupo de robots, determine si es posible o no formar una alianza multicolor.

Formalización del problema

Notemos que las facciones de robots forman una partición del conjunto de todos los robots de la ciudad, y que la relación "el robot A tiene al menos una pieza hecha por fabricantes en común con el robot B" es simétrica. Con estas aclaraciones podemos enunciar el problema formalmente como sigue:

Sea R un conjunto tal que $|R| = n$, $P = \{R_1; R_2; \ldots; R_k\}$ una partición de R, y F: R x R una relación simétrica, existirá $C \subseteq R$ tal que:

- $C \cap R_i \neq \emptyset$; $\forall\ i = \overline{1,k}$

- $\forall\ r_i, r_j \in C$: $< r_i, r_j > \in F$?

De esta forma, una instancia X del problema podemos escribirla como la tupla $< R, P, F >$, donde R es un conjunto finito, P una partición de R y F una relación simétrica en RxR. Denotemos M el espacio de todas las tuplas $< R, P, F >$ que existen.

Nota: El problema es *NP*
Demostración

Sea R el conjunto de todos los robots de la ciudad tal que $n = |R|$, y sea C una posible solución del problema, entonces se cumple que:

- *Se puede conocer la cantidad k de facciones en $O(n)$*: basta con iniciar $k = 0$, recorrer todos los robots y, aumentar k en uno cada vez que se vea una facción distinta de las vistas hasta el momento.
- *Comprobar que en C existe al menos un robot por cada facción es $O(n)$*: basta con recorrer todos los robots de C y contar las distintas facciones vistas en cada

momento del recorrido, si en algún momento llegamos a k(la cantidad de facciones) detenemos la iteración pues ya se han visto todas las facciones, o sea, existe al menos un robot de cada facción. Como $|C| \leq n$, entonces el recorrido itera a lo sumo n, luego esta comprobación se puede hacer en $O(n)$.
- *Comprobar que en cualesquiera 2 robots en C comparten al menos una pieza del mismo fabricante es* $O(n^2)$: basta con recorrer el conjunto C y por cada par de robots de C, revisar los pares ordenados contenidos en la relación F, si dicho par existe, entonces ambos robots comparten al menos una pieza hecha por fabricantes en común. Recorrer cada par de robots en C es $O(n^2)$, encontrar el par ordenado correspondiente en F es $O(|F|)$, como F es sobre R x R, entonces $|F| \leq n^2$, luego encontrar dicho par es $O(n^2)$. Por tanto, comprobar que cualesquiera 2 robots de C comparten al menos una pieza hecha por los mismos fabricantes es $O(n^2) * O(n^2)$, luego esta comprobación es $O(n^4)$.

Luego, comprobar si C es una solución del problema se puede hacer en $O(n) + O(n) + O(n^4)$, lo cual es $O(n^4)$, o lo que es lo mismo, se puede comprobar si C es una solución o no del problema en tiempo polinomial. Por tanto, el problema es *NP*.

Lema 1:

Sea $C \subseteq R$ una solución de $X \in M$, $X =< R, P, F >$ una instancia, $k = |P|$ y $|C| = m \geq k$, existe $C_0 \subset R$ con $|C_0| = k$ que también es solución X.

Demostración:
Notemos primero que cualquier solución C de X cumple que $|C| \geq k$, pues $|P| = k$. Supongamos ahora que existe $C \subseteq R$ solución de X tal que existen $r_q, r_t \in C$ tales que $r_q, r_t \in R_s$ para algún $s = \overline{1,k}$, se cumple entonces $|C| > k$. Como C es solución, $\forall\ r_i, r_j \in C$: $< r_i, r_j > \in F$. Luego, el conjunto $C^{'} = C/\{r_q\}$ es también solución de X. Entonces, dado un conjunto $C \subseteq R$ con $|C| = m > k$ solución del problema, podemos construir un conjunto $C^{'} \subset C$ solución de X, por tanto, existe $C_0 \subset R$: $|C_0| = k$ solución de X.

Lema 2:

Sea R un conjunto, $P = \{R_1; R_2; \ldots; R_k\}$ una partición cualquiera sobre R, F: R x R una relación simétrica, $C \subset R$ con $|C| = k$, y $G(R, E)$ un grafo no dirigido tal que:

- $\forall r_i, r_j \in R$: $< r_i, r_j > \in E$ si y solo si $< r_i, r_j > \in F$ y $r_i \in R_s$,$r_j \in R_t$ con $s \neq t$ $(s, t = \overline{1,k})$

Entonces se cumple que C es solución del problema si y solo si $G_0(C, E_0)$ es un clique de G, donde:

$E_0 = \{< r_i, r_j >: r_i, r_j \in C, < r_i, r_j > \in E\}$

Demostración:

1- Si C es solución, entonces $G_0(C, E_0)$ es un clique de $G(R, E)$.

Como C es solución y $|C| = k$, $\forall\ r_i, r_j \in C$: $< r_i, r_j > \in F$; sean R_i, R_j tales $r_i \in R_i$ y $r_j \in R_j$, $\forall\ r_i, r_j \in C$, se cumple que $\forall\ r_i, r_j \in C\ R_i \neq R_j$, luego $< r_i, r_j > \in E$. Como $< r_i, r_j > \in E$ y $r_i, r_j \in C$, entonces $< r_i, r_j > \in E_0$, entonces, $\forall\ r_i, r_j \in C$: $< r_i, r_j > \in E_0$. Por tanto G_0 es completo, o lo que es lo mismo, es un clique de G.

2- Si $G_0(C, E_0)$ es un clique de $G(R, E)$, entonces C es solución.

Como G_0 es un clique de G, $\forall\ r_i, r_j \in C < r_i, r_j > \in F$ y $< r_i, r_j > \in E$. Sean R_i, R_j tales que $r_i \in R_i$ y $r_j \in R_j$, como $< r_i, r_j > \in E$, entonces $R_i \neq R_j$ $\forall\ r_i, r_j \in C$. Por tanto C es solución del problema.

Lema 3:

ea GS el espacio de todos los grafos no dirigidos, sea f: $M \to GS$ que a cada $X \in M$, $(X =< R, P, F >)$ le asigna el grafo no dirigido $G(R, E)$ tal que:

- $\forall r_i, r_j \in R$: $< r_i, r_j > \in E$ si y solo si $< r_i, r_j > \in F$ y $r_i \in R_s$,$r_j \in R_t$ con $s \neq t$ y $R_s, R_t \in P$

Se cumple que f es sobreyectiva, y $\forall G \in GS$, se puede obtener $X \in M$ tal que $f(X) = G$, en tiempo polinomial.

Demostración:
La idea se basa en colorear el grafo de alguna forma. Sea $G(V, E) \in GS$ un grafo no dirigido cualquiera. Tomemos el vértice de mayor grado de G y coloquémoslo en un conjunto R_1, y este a su vez dentro de un conjunto P, a partir de este vértice iniciamos una búsqueda en *BFS* realizando la siguiente operación en cada paso:

Sea v_i el vértice encontrado en el paso i del *BFS*, recorremos cada conjunto en P y hacemos una de las siguientes 2 operaciones:

1- Si encontramos un conjunto $R_s \in P$: $Ady(v_i) \cap R_s = \emptyset$, colocamos v_i en R_s ($Ady(v_i)$ son los vértices adyacentes a v_i)
2- Si no encontramos ningún conjunto tal que se cumpla la condición anterior, agregamos un nuevo conjunto $Q = \{v_i\}$ a P.

Como en cada iteración se consume un vértice del grafo y hay un número finito de estos, el procedimiento termina, y al hacerlo, habremos coloreado el grafo de una forma, dicha coloración define una partición sobre los vértices del grafo.
La complejidad de encontrar el vértice de mayor grado es $O(V)$, la complejidad del *BFS* es $O(V + E)$, y el costo de las operaciones internas del *BFS* es $O(V^2)$, luego, la complejidad total del procedimiento es:

$O(V) + O(V + E) * O(V^2) = O(V^3 + E * V^2)$, que en su caso peor es $O(V^4)$

Si ahora consideramos $X =< V, P, E >$, entonces $X \in M$, $f(X) = G$ y se obtuvo en tiempo polinomial, por tanto f es sobreyectiva.

Solución del problema

El problema es *NP-Completo*

Demostración:
Sea $X =< R, P, F > \in M$, por el lema 1, si $\exists\ C \subseteq R$ es una solución de X, entonces $\exists\ C_0 \subset R$: $|C_0| = |P|$ solución de X. Por el lema 2, encontrar C_0 es equivalente a encontrar un clique sobre el grafo G correspondiente a X; y por el lema 3, $\forall\ G$ grafo no dirigido existe una transformación polinomial a un cierto $X \in M$. Luego, existe una reducción polinomial del *problema del k-clique*, con k que depende de n al problema en cuestión. Como el *problema del k-clique* con k que depende de n es *NP-Completo*, entonces el problema en cuestión es *NP-Completo*

El hecho de que sea un problema *NP-Completo* no quiere decir que no podamos hacer nada más que fuerza bruta solamente. Una primera mejora a la fuerza bruta podría ser, probar solamente los conjuntos de robots que posean solamente k robots, donde k es la cantidad de facciones que hay. La correctitud de esta mejora está dada por el lema 1.

Casos especiales

Notemos que este problema es *NP-Completo* cuando la cantidad de facciones depende de la cantidad de robots, esto es, cuando la cantidad de facciones no está acotada. Sin embargo, existen instancias del problema que sí se pueden resolver en tiempo polinomial. Ejemplo, cuando la cantidad de facciones está acotada, la fuerza bruta es polinomial, pues solo se deben comprobar todos los posibles subconjuntos de robots cuyo tamaño sea menor que la cantidad de facciones k, esta cantidad viene dada por la expresión:

$\sum_{i=1}^{k} \frac{n(n-1)...(n-i+1)}{i!} \leq \sum_{i=1}^{k} \frac{n^k}{i!} \leq \sum_{i=1}^{k} n^k = kn^k$

Luego, la fuerza bruta es $O(n^k) * O(k^2) = O(n^k)$.

La mejora propuesta al algoritmo de fuerza bruta comprueba todos los subconjuntos de tamaño k, esto es:

$C(n, k) = \frac{n^k}{k!}$

Luego, la mejora de la fuerza bruta es $O(n^k) * O(k^2) = O(n^k)$, igual que la fuerza bruta, pero con una constante menor.

Otro caso donde el problema se puede resolver en tiempo polinomial, es para cuando la relación de compartir piezas con otro robot es transitiva. En este caso solo deberíamos ir tomando un robot de una facción distinta a las ya tomadas y comprobar que dicho robot comparte al menos una pieza con el robot anterior

mente tomado, y dado que la relación es transitiva, este comparte piezas con todos los demás; se finaliza el procedimiento cuando hayamos escogido todas las facciones. Esta solución es $O(n^2)$.

Nota

Un posible algoritmo a usar es el *Algoritmo de Bron-Kerbosch*,un algoritmo recursivo usado para enumerar todos los cliques de un grafo.

3 Subsecuencia Buena

Descripción del problema

Supongamos que tienes un array binario B de longitud N. Una secuencia $x_1, x_2, \ldots, x_k$ se llama buena con respecto a B si satisface las siguientes condiciones: spacing

- $1 \leq x_1 < x_2 < \ldots < x_k \leq N + 1$
- Para cada par (i, j) tal que $1 \leq i < j \leq k$, el subarray $B[x_i : x_j - 1]$ contiene $(j - i)$ unos más que ceros. Es decir, si $B[x_i : x_j - 1]$ contiene c_1 unos y c_0 ceros, entonces se debe cumplir que $c_1 - c_0 = j - i$.

Aquí, $B[L : R]$ denota el subarray que consiste en los elementos $[B_L, B_{L+1}, B_{L+2}, \ldots, B_R]$. Nota que, en particular, una secuencia de tamaño 1 siempre es buena.

Por ejemplo, supongamos que $B = [0, 1, 1, 0, 1, 1]$. Entonces: spacing

- La secuencia $[1, 4, 7]$ es una secuencia buena. Los subarrays que necesitan ser revisados son $B[1 : 3]$, $B[1 : 6]$ y $B[4 : 6]$, que todos cumplen con la condición.
- La secuencia $[1, 5]$ no es buena, porque $B[1 : 4] = [0, 1, 1, 0]$ contiene un número igual de ceros y unos (cuando debería contener un 1 extra).

Yonatan le dio a David un array binario A de tamaño N y le pidió que encontrara la secuencia más larga que sea buena con respecto a A. Ayuda a David a encontrar una de estas secuencias.

Si existen múltiples secuencias más largas posibles, puedes devolver cualquiera de ellas.

Primera solución: Fuerza Bruta

La primera solución se basa en generar todas las subsecuencias posibles de índices y verificar si cada una de ellas es una "buena subsecuencia" según las reglas dadas.

Segunda solución: Recursividad

Dado que la fuerza bruta no es eficiente, se intenta optimizar el proceso mediante una solución recursiva. El algoritmo es el siguiente:

- La idea es construir la subsecuencia de manera recursiva. Se empieza desde el índice 1 y, en cada paso, se decide si incluir el índice actual en la subsecuencia o no.
- Se exploran ambas posibilidades (incluir o excluir), y se comprueba cuál opción genera una subsecuencia más larga.
- Se verifica si una subsecuencia es "buena" cada vez que se completa una subsecuencia potencial.

Podríamos intentar mejorar esta solución usando la programación dinámica, pero no habría un cambio considerable con respecto a la solución actual.

Solución final: Greedy

Lema 1:

Sea S una subsecuencia de B ($|S| = k$), y sea C la función que dado un intervalo de un array binario, devuelve la diferencia entre la cantidad de 1's y 0's contenidos en dicho intervalo, se cumple que:

si $\exists i, j, l = \overline{1,k}$: $C(B[x_i : x_j - 1]) = j - i, C(B[x_j : x_l - 1]) = l - j \Rightarrow C(B[x_i : x_l - 1]) = l - i$

Donde x_s es el elemento s-ésimo de S.

Demostración:
Como $C(B[x_i : x_j - 1]) = j - i$ y $C(B[x_j : x_l - 1]) = l - j$, entonces, sean r_1, r_0 la cantidad de 1's y 0's respectivamente en el intervalo $[x_i, x_j - 1]$ de B, y sean $r_1^{'}, r_0^{'}$ la cantidad de 1's y 0's en el intervalo $[x_j, x_l - 1]$ de B, entonces en el intervalo $[x_i, x_l - 1]$ de B hay $r_1 + r_1^{'}$ 1's y $r_0 + r_0^{'}$ 0's, y se sabe que $r_1 - r_0 = j - i, r_1^{'} - r_0^{'} = l - j$. De esto se deduce que $C(B[x_i : x_l - 1]) = r_1 + r_1^{'} - r_0 - r_0^{'} = r_1 - r_0 + r_1^{'} - r_0^{'} = j - i + l - j = l - i$. Por tanto $C(B[x_i : x_l - 1]) = l - i$.

Lema 2:

Sea S ($|S| = k$) una subsecuencia de B un array binario, si $\forall i = \overline{1, k-1}$ se cumple que la cantidad de 1's menos la cantidad de 0's en $B[x_i : x_{i+1} - 1]$ es igual a 1, entonces S es una subsecuencia buena.

Demostración:
Vamos a demostrarlo por inducción en k. Notemos que $k \geq 2$. Para $k = 2$ es trivial, por definición de subsecuencia buena. Supongamos que esta afirmación es verdadera para todo $m < k$. Sea $S_0 = [x_1; x_2; \ldots; x_m]$ y $S_1 = [x_m; x_{m+1}; \ldots; x_k]$ donde $x_i \in S$ subsecuencia de B que cumple con la premisa, entonces, por hipótesis de inducción, S_0 y S_1 son ambas subsecuencias buenas. También se cumple que $\forall i, j (1 \leq i \leq |S_0|, |S_0| \leq j < k)$ tal que $j - i + 1 < k$, la secuencia $S^{'} = x_i, x_{i+1}, \ldots, x_j$, también es una subsecuencia buena, por hipótesis de inducción. Por el lema 1, siendo C la función que dado un intervalo de un array binario devuelve la cantidad de 1's que exceden la cantidad de 0's, se cumple que:

$C(B[x_1 : x_k - 1]) = C(B[x_1 : x_m - 1]) + C(B[x_m : x_k - 1]) = k - 1 \ \forall m < k.$

Luego, S que cumpla con la premisa es una subsecuencia buena para toda $k \geq 2$

Lema 3:

Dado un array binario B y un intervalo cualquiera de B (incluyendo el propio array), y sea c_1, c_0 la cantidad de 1's y 0's respectivamente en dicho intervalo, la mayor subsecuencia buena S que se puede extraer de dicho intervalo cumple que $|S| \leq c_1 - c_0 + 1$

Demostración:
Notemos que dado un array binario B y un intervalo $[i, j]$ de B, existe un multiconjunto M formado por elementos 0's y 1's, tal que $B[i : j]$ es una ordenación de sus elementos, y para una subsecuencia buena S de $B[i : j]$, existe una partición P de M donde cada elemento de P cumple que tiene al menos un 1 más que 0's, llamemos a esta propiedad I. Sea M un multiconjunto binario tal que M contiene c_1 1's y c_0 0's ($c_1 > c_0$), la mayor partición P de M que verifica la propiedad I es aquella donde hay $c_1 - c_0$ elementos con exactamente un 1 más que 0's; pues de existir en P un elemento con más de un 1 más que 0's, se incumpliría la propiedad I, ya que después de haber emparejado los 0's y los 1's, solo quedan $c_1 - c_0$ 1's libres. Luego, dado que una subsecuencia buena S ($|S| = k$) de $B[i : j]$ ($B[i : j]$ con c_1 1's y c_0 0's), define $k - 1$ multiconjuntos binarios, donde cada uno pertence a una misma partición P de M que cumpla la propiedad I, se deduce que $k - 1 \leq c_1 - c_0$.

Por tanto, $|S| \leq c_1 - c_0 + 1$

Lema 4

Sea C la función que calcula la diferencia entre 0's y 1's en cualquier intervalo de un array binario, se cumple que:

$\forall B$ array binario, si $C(B[i : j - 1])$ es máximo para ciertos $i, j = \overline{1, |B| + 1}$, entonces $B[i : j - 1]$ comienza con un 1 y termina con un 1.

Demostración:
Sea $B[i : j - 1]$ un intervalo de B que comienza con 0 tal que $C(B[i : j - 1])$ es máximo, entonces el intervalo $B[i + 1 : j - 1]$ contiene un 0 menos, por tanto $C(B[i + 1 : j - 1] > C(B[i : j - 1])$, contradicción. Con un razonamiento igual para cuando $B[i : j - 1]$ termina en 0, se llega a que $C(B[i : j - 2]) > C(B[i : j - 1])$, contradicción, concluyendo de esta forma la demostración.

Lema 5

Sea C la función que dado un intervalo de un array binario B calcula la diferencia entre 0's y 1's contenidos en dicho intervalo, entonces se cumple que:

Si $C(B[i : j-1])$, con $i, j = \overline{1, |B|+1}$, es máximo, entonces $\exists S$ subsecuencia buena de B contenida en $B[i : j-1]$ tal que $|S|$ es máxima.

Demostración:
La demostración se basa en construir dicha secuencia.

Por el lema 4, sabemos que si $C(B[i : j-1])$ es máximo, entonces $B[i : j-1]$ comienza y termina con 1's. Como $B[i : j-1]$ comienza con 1, hacemos $S = [x_1](x_1 = i)$, declaramos las variables $k_0 = i, k_1 = i+1$, y comenzamos con el siguiente proceso:

1: si $C(B[k_0 : k_1 - 1]) = 1$, declaramos $x_{|S|} = k_1$ y agregamos $x_{|S|}$ a la secuencia S, y vamos al paso 2; si no, vamos al paso 4.

2: si $k_1 = j+1$, terminamos el proceso, y retornamos la secuencia S, si no, vamos al paso 3.

3: hacemos $k_0 = k_1$ y $k_1 = k_1 + 1$, y volvemos al paso 1.

4: hacemos $k_1 = k_1 + 1$ y vamos al paso 1.

Dado que el intervalo de B es finito y siempre termina en 1, el proceso termina. Demostremos que la secuencia obtenida es máxima.
En los pasos 3 y 4, $C(B[k_0 : k_1 - 1])$ aumenta o disminuye en 1, luego, si existe una subsecuencia buena, siempre se entra al caso afirmativo del paso 1 en algún momento. Sea $k = C(B[i : j-1])$, si $k > 0$ entonces el proceso entra al caso afirmativo del paso 1 exactamente k veces, pues por construcción, $\forall p = \overline{1, |S|-1}$, $C(B[x_p : x_{p+1} - 1]) = 1$, esto es, $B[x_p : x_{p+1} - 1]$ es un conjunto con un 1 más que 0's, y por el lema 2, S es buena; dado que se entra al caso afirmativo del paso 1 cada vez que se ve un conjunto con un 1 más que 0's, y hay exactamente k de estos, entonces se agregan a S exactamente k elementos nuevos. Como S tiene exactamente un elemento al comenzar el proceso, al terminar $|S| = k+1$ elementos, por tanto, por el lema 3, S es máxima.

Solución final: Greedy

*Nota:*Cuando B es completamente nulo es un caso trivial, pues no existe subsecuencia buena posible, luego se retorna la cadena vacía.

Encontramos el subarray de B tal que $C(B[i : j-1])$ sea lo mayor posible. Por el lema 4, sabemos que $B[i : j-1]$ comienza y termina con 1's, con el algoritmo usado para la demostración del lema 5, se puede encontrar la subsecuencia buena con la mayor longitud posible.

Complejidad temporal

Encontrar $B[i : j-1]$ tal que $C(B[i : j-1])$ sea lo mayor posible se puede hacer en $O(n)$, basta con declarar B' tal que $B'[s] = 1$ si y solo si $B[s] = 1$, en otro caso $B'[s] = -1$, luego, el subarray de mayor suma en B' es el buscado en B. Este subarray se puede encontrar con una modificación el *algoritmo de Kadane*

```
def Kadane(array):
    current_max = 0
    global_max = 0
    start = 0
    temp_start = 0
    end = 0

    for i in range(len(array)):
        if array[i] > current_max + array[i]:
            temp_start = i
            current_max = array[i]
        else:
            current_max += array[i]

        if current_max > global_max:
            global_max = current_max
            start = temp_start
            end = i

    return start,end
```

Luego de haber hallado el subarray de B de mayor diferencia de 1's y 0's, sabiendo que comienza y termina en 1's por el lema 4, procedemos a aplicar el algoritmo usado en la demostración del lema 5 para construir la secuencia buscada, la cual el propio lema 5 nos asegura su existencia dentro del propio subarray, dicho algoritmo es $O(n)$. Por tanto el algoritmo es $O(n)+O(n) = O(n)$

Correctitud

La demostración de la correctitud del *algoritmo de Kadane* para encontrar el subarray con mayor suma, la podemos encontrar en diversas fuentes, demostremos la correctitud de esta variación. Esta variación solo agrega las variables *start, tempstart y end* para definir donde comienza y donde termina el subarray de mayor suma, por tanto, sabemos que la mayor suma posible es *globalmax*, demostremos entonces que el subarray que da dicha suma comienza en *start* y termina en *end*.
Supongamos que dicho array no comienza en *start*, si dicho subarray comenzara antes, entonces, esto es falso, ya que se entró en la primera condición en la posición $i = start$, esto quiere decir que la suma que se llevaba desde antes más *array[start]* era menor que *array[start]*, entonces el subarray debe comenzar en *start*. Si dicho subarray comenzara después de start, nunca se entró en la primera condición, por tanto la suma anterior más *array[start]* es al menos igual a *array[start]*, por tanto, no se afecta la suma final. Supongamos ahora que el subarray de suma máxima termina antes o después de *end*, esto no puede ser debido a que solo se entra en el segundo bloque *if* cuando la suma actual supera a la mayor suma encontrada. Por tanto, este algoritmo devuelve el subarray de mayor suma.

La correctitud del algoritmo usado para construir la subsecuencia buena de mayor longitud fue demostrada en el lema 5.

Por tanto, el algoritmo dado es correcto.