Enunciado

Sea $S = (x_1, x_2, \dots, x_n)$ una secuencia de n booleanos (1 ó 0) y sea $1 \leq k \leq n$ un entero. Supongamos que se pueden eliminar k ceros, queremos saber la longitud máxima que puede tener una cadena de unos. Por ejemplo si $k = 2$, y $S = 11001010001$, la respuesta es 3, mientras que si $k = 3$ la respuesta es 4.

- Diseñar un algoritmo basado en programación dinámica que indique la longitud más larga de una subsecuencia de
  unos sacando a lo sumo k ceros de S. Debe tener complejidad temporal a lo sumo $O(nk)$.
- Demostrar que es correcto.
- Demostrar su complejidad temporal.

Resolución

La solución simple para este problema es buscar todos los posibles subconjuntos de posiciones donde borrar ceros, y para cada uno, buscar la longitud más larga de unos. Como podemos elegir no borar k ceros, sino *a lo sumo* k, hay $T(k) = \sum_{i=0}^{k} \binom{n}{i}$ posibles subconjuntos de posiciones. Para cada uno, requerimos $O(n)$ tiempo y $O(1)$ espacio para encontrar la subsecuencia más larga de unos. Expandiendo T en k vemos que si $k \geq 1$, entonces $T(k) \in \Omega(\left(\frac{n}{k}\right)^k)$, luego este no va a ser un algoritmo eficiente.

Sin embargo, este problema se puede replantear de una forma que exhiba subestructura óptima. Toda sucesión de unos, borrando a lo sumo k ceros, empieza en alguna posición. Consideremos entonces la sucesión de unos más larga que empieza en la posición i, borrando a lo sumo k ceros. Llamemos L a su longitud. Hay dos posibilidades, $x_i = 0$, y $x_i = 1$.

- Si $x_i = 1$, entonces lo que le sigue a la sucesión es una secuencia de unos de longitud $L - 1$, borrando a lo sumo k ceros, que empieza en la posición $i + 1$.

- Si $x_i = 0$, entonces lo que le sigue a la sucesión es una secuencia de unos de longitud L, borrando a lo sumo $k - 1$ ceros, que empieza en la posición $i + 1$. Si $k = 0$, entonces la solución termina acá, porque no hay forma de usar "-1" ceros, y tenemos $L = 0$.

Vemos que si la sucesión que sigue en la posición $i + 1$ no fuera lo más larga posible, tampoco lo sería la sucesión que comienza en la posición i. Luego, podemos definir una función f, donde $f(i, j)$ nos da la longitud de la secuencia de unos más larga que comienza en la posición i, borrando a lo sumo j ceros en el medio:

$$f(i,j) = \begin{cases} 0 & \text{si } i > n \\ -\infty & \text{si } j < 0 \\ 1 + f(i+1, j) & \text{si } x_i = 1 \\ \max(0, f(i+1, j-1)) & \text{si } x_i = 0 \end{cases}$$

El ejercicio se responde calculando $\max_{1 \le i \le n} \{f(i, k)\}$.

Implementación recursiva

Como en Python los arrays empiezan en el índice 0, en vez de 1, cambiamos los índices de la función que planteamos.

```
def solve(S, k):
  n = len(S)
  # Longitud de la subsecuencia de unos más larga que empieza en i, borrando
  # a lo sumo j ceros en el medio.
  def f(i, j):
    if i == n: return 0
    if j < 0: return -99999
    if S[i] == 1: return 1 + f(i + 1, j)
    if S[i] == 0: return max(0, f(i + 1, j - 1))

  return max(f(i, k) for i in range(n))
```

Usamos `-99999`, pero podríamos usar `-float('inf')` u otras estrategias para significar $-\infty$.

Análisis asintótico

La implementación recursiva de esta función llama a `f` n veces, y como f aumenta i en uno cada vez (o termina), la llamada $f(i,k)$ va a hacer a lo sumo i llamadas recursivas, hasta parar. Para encontrar una familia de casos que produzca nuestro peor caso de número de llamadas, podemos pensar en una secuencia donde $x_i = 1$ para todo i, con $k = 0$. Con esto, caemos siempre en la rama `S[i] == 1`, y al comenzar en la posición i, hacemos $n - i$ llamadas recursivas. Luego, el número de llamadas totales que hacemos en el peor caso es $\sum_{i=0}^{n}(n-i) = \sum_{i=0}^{n} i = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = \Omega(n^2)$. Como hacemos $\Omega(1)$ operaciones en cada una, cuando n es grande, terminamos haciendo $\Omega(n^2)$ operaciones en el peor caso. Esto rápidamente se vuelve intratable. Además, no cumple la complejidad pedida, que es $O(nk)$.

Sin embargo, vemos que el primer argumento de f es un entero entre 0 y $n-1$, y el segundo argumento un entero entre 0 y k. Luego, hay a lo sumo $n(k+1)$ posibles valores distintos que puede tomar f. Al tomar el límite $\lim_{n\to\infty} \frac{n(k+1)}{\frac{n(n-1)}{2}} = \lim_{n\to\infty} \frac{2k+2}{n-1} = 0$, vemos que hay muchos subproblemas compartidos cuando n crece. Esto es entonces un buen candidato para programación dinámica top-down.

Programación dinámica top-down

Recordemos que agregar un cache a una función recursiva es un proceso enteramente mecánico, y no hay nada que pensar.

```
def solve(S, k):
  n = len(S)
  cache = [[None for _ in range(k + 1)] for _ in range(n)]
  # Longitud de la subsecuencia de unos más larga que empieza en i, borrando
  # a lo sumo j ceros en el medio.
  def f(i, j):
    if i == n: return 0
    if j < 0: return -99999
    if cache[i][j] is None:
      if S[i] == 1: cache[i][j] = 1 + f(i + 1, j)
      if S[i] == 0: cache[i][j] = max(0, f(i + 1, j - 1))
    return cache[i][j]
  return max(f(i, k) for i in range(n))
```

En una ejecución, vamos a tener como máximo $n(k+1)$ posibles llamadas recursivas. Cada una va a hacer a lo sumo una llamada no-recursiva. Luego, el número de llamadas totales es $O(nk)$. En ambos casos, hacemos $O(1)$ operaciones. Luego, el número de operaciones que hacemos es $O(nk)$, y la complejidad espacial es también $O(nk)$ porque construímos la matriz `cache`. Esta cota superior vale en todos los casos (peor, mejor, medio, lo que sea), porque no asumimos nada sobre la entrada para nuestro análisis.

Programación dinámica bottom-up

Podemos, en vez de llenar el cache a medida que las llamadas recursivas lo necesiten, llenar el cache manualmente. Vemos que al resolver el subproblema (i, j), necesitamos leer el resultado del subproblema $(i+1, j)$, o $(i+1, j-1)$. Luego, podemos llenar el cache con dos ciclos. El ciclo exterior va a iterar sobre todos los i, decreciente. El ciclo interior va a iterar sobre todos los j, creciente. Luego, al momento de llenar la entrada `cache[i][j]`, vamos a haber escrito la entrada `cache[a][b]` para todo `a > i` y todo `b`.

```python
def solve(S, k):
  n = len(S)
  # cache[i][j] = Longitud de la subsecuencia de unos más larga
  # que empieza en i, borrando a lo sumo j ceros en el medio.
  cache = [[0 for _ in range(k + 1)] for _ in range(n + 1)]
  for i in range(n - 1, -1, -1):
    for j in range(k + 1):
      if S[i] == 1:
        cache[i][j] = 1 + cache[i + 1][j]
      if S[i] == 0 and j > 0:
        cache[i][j] = cache[i + 1][j - 1]
  return max(cache[i][k] for i in range(n))
```

Notar como ya no necesitamos el $-\infty$, puesto que símplemente no leemos esa casilla si nos hubieramos salido de la matriz. Llenamos la matriz de ceros inicialmente, y así no tenemos que manejar el $\max(0, \ldots)$, ni explícitamente el caso $i = n$.
Esta implementación, como siempre que usamos programación dinámica bottom-up, hace muy simple el cálculo de complejidad. Hacemos $O(nk)$ operaciones para inicializar el cache, y usamos $O(nk)$ espacio para eos. Además, tenemos dos ciclos anidados, donde hacemos $O(1)$ operaciones dentro de cada iteración, lo que resulta en $O(nk)$ operaciones en los ciclos. Finalmente, el algoritmo entero usa $O(nk)$ operaciones y espacio.

Mejora de espacio

Una de las ventajas de usar programación dinámica bottom-up, es que podemos notar patrones en el orden de llenado de nuestra cache. En este caso, para llenar la fila i-ésima, sólo necesitamos la fila $(i + 1)$-ésima. Luego, no necesitamos guardar todas las filas anteriores, sólo una. Además, como en cada fila sólo leemos la entrada j o la entrada $j - 1$, si iteramos j en orden decreciente, no necesitamos guardar una copia de la fila al llenar la nueva versión de la misma, podemos sólo usar una misma fila. Esto se conoce como "in-place update".

Si llamamos `dp` a nuestra fila, al comenzar la iteración (i, j) de nuestros ciclos, `dp[r]` va a contener $f(i + 1, r)$ para todo $r \geq j$, y $f(i, r)$ para todo $r < j$. Si `S[i] == 0 and j > 0`, podemos ejecutar `dp[j] = dp[j - 1]`, y la semántica de `dp[j]` pasa de contener $f(i + 1, j - 1)$ a contener $f(i, j)$, que es lo que queremos al comenzar la próxima iteración de j. Si `j == 0`, ejecutamos `dp[j] = 0`. Si, de otra forma, `S[i] == 1`, entonces al escribir `dp[j] += 1`, representamos $f(i, j) = 1 + f(i + 1, j)$, que es lo que queremos que tenga `dp[j]` al comenzar la próxima iteración de j.

Al final, no vamos a tener `cache[i][k]` como antes, para todo `i`. Luego, a medida que calculamos nuevas filas `dp`, tenemos que recordar `dp[k]`, y tomar el máximo `dp[k]` en todas las iteraciones de `i`.

El código entonces queda:

```
def solve(S, k):
  n = len(S)
  dp = [0 for _ in range(k + 1)]
  result = 0
  for i in range(n - 1, -1, -1):
    for j in range(k, -1, -1):
      if S[i] == 1:
        dp[j] += 1
      if S[i] == 0:
        if j > 0: dp[j] = dp[j - 1]
        else: dp[j] = 0
    result = max(result, dp[k])
  return result
```

Esto usa, ahora, sólo $O(k)$ espacio, y sigue usando $O(nk)$ tiempo. Este tipo de mejoras, que dependen del orden en que uno llena el cache, es otro motivo por el cual está bueno usar programación dinámica bottom-up. Uno se ahorra, también, el costo de armar y desarmar el stack de llamadas de funciones recursivas.