Cálculo complejidad

La función `void Tokenizador::Tokenizar(const string&, OutputIF&)` contiene la única implementación del algoritmo de tokenización (el resto de funciones dependen de ella). Se analizarán las complejidades temporal y espacial de dicha función, utilizando N como número de caracteres del string recibido y K como número de delimitadores almacenados.

Complejidad temporal

A la hora de estudiar la complejidad temporal, la función Tokenizar puede ser resumida a través del siguiente pseudocódigo:

```
1. para cada carácter n en el string:
2.    para cada carácter k en los delimitadores:
3.          si n==k:
4.                encontradoDelim()
5.                break
6.    estado()
```

La función encontradoDelim recorre de nuevo los delimitadores para reordenar el acceso a los mismos, con lo cual la complejidad temporal de las líneas segunda a quinta está acotada por Ω(1) (cota inferior) y O(2·K) (cota superior). Por otro lado la complejidad de cualquier función de estado viene acotada por Ω(1) y por la cota superior de la función OutputIF::add(const string &).

La interfaz OutputIF representa la salida del algoritmo y tiene dos implementaciones: una basada en una lista de strings y otra basada en un string. La complejidad temporal de la función add en cada una de las implementaciones viene dada por Θ(1) (complejidad de list::push_back[1]) y O(tamRepresentacionInterna + tamPalabraNueva) (complejidad de string::append[2]) respectivamente. Esto significa que en el peor de los casos (implementación basada en string) la complejidad de add depende de las palabras y su tamaño.

Para acotar la complejidad temporal, separaremos el bucle de la línea 1 considerando independientes el bucle de la linea 2 y la función de la linea 6. De esta manera la cota superior pasa a ser O(N·2·K + N·cotaSuperiorAdd).

[1] http://www.cplusplus.com/reference/list/list/push_back/
[2] http://www.cplusplus.com/reference/string/string/append/

Utilizaremos un caso hipotético en el que cada carácter forma una palabra, convirtiendo N·cotaSuperiorAdd en:

$$\sum_{n=0}^{N} 2n - 1$$

Ya que la cota superior de add depende de la representación interna (los caracteres de todas las palabras almacenadas más un '\n' por cada palabra) y del tamaño de la palabra recibida. Este sumatorio sería equivalente a N^2 - N, convirtiendose la cota superior en O(2·N·K + N^2 - N). Siendo que en la mayoría de casos se cumple que 2·K < N, la función Tokenizar vendría acotada temporalmente por Ω(N) y O(N^2).

El mejor caso (que correspondería a la cota inferior) consistiría en un string con el mismo delimitador repetido N veces. El reorden de acceso aseguraría que el bucle de la segunda línea tuviese complejidad constante, y al ser el único delimitador que aparece, una vez colocado en primera posición (después de la primera aparición) la función encontradoDelim no necesitaría reordenar de nuevo.

Para encontrar el peor caso no podemos utilizar el hipotético de la estimación superior, ya que no existe ningún caso real en el que cada carácter sea una palabra distinta sin necesidad de delimitadores. Considerando los casos especiales implementados (URL, decimales, acrónimos y multipalabras) el patrón con mayor cantidad de llamadas a OutputIF::add por carácter sería la combinación de dígito, símbolo del dólar y delimitador (por ejemplo "3$ 3$ 3$ 3$ "). Con este patrón, N·cotaSuperiorAdd se convertiría en:

$$\sum_{n=0}^{2N/3} 2n - 1$$

Siguiendo la lógica anterior y descontando el delimitador del total de palabras. El sumatorio sería equivalente a $4N^2/9$ - 2N/3.

Por último, para maximizar el tiempo empleado en buscar los delimitadores, habría que ir cambiando de delimitadores de manera que siempre se accediese al último en el orden actual (por ejemplo, para los delimitadores ".-/" el patron se convertiría en "3$ 3$/3$-3$.3$ " ya que cuando los casos especiales están activados, el espacio se trata como un "último delimitador"). Con este cambio la sobrestimación N·2·K se convertiría en N·(⅔·K + ⅓·2K) ya que dos tercios de los caracteres recorrerían todos los delimitadores sin ser delimitador y un tercio de los caracteres recorrerían todos los delimitadores siendo el último delimitador, reordenando todos los delimitadores en la función encontradoDelim. Esto nos proporciona una complejidad temporal aproximada de:

$$\frac{4}{9}N^2 + \frac{4}{3}NK - \frac{2}{3}N$$

El último trozo de código de la función Tokenizar se encarga de almacenar la última palabra si el string se acaba en un estado que espera más información. Al ser su complejidad Ω(1) O(N), no afecta a los resultados obtenidos, con lo que se omitió en la simplificación. Además, ni el mejor ni el peor caso hacen uso de este código. Todas las palabras se almacenan al detectar un delimitador.

MEJOR CASO: un string de un solo carácter, que además sea delimitador.
COMPLEJIDAD TEMPORAL: N
EJEMPLO: "....."

PEOR CASO: utilizando OutputString, un string compuesto del patrón dígito, dólar y delimitador, alternando delimitadores.
COMPLEJIDAD TEMPORAL: $\frac{4}{9}N^2 + \frac{4}{3}NK - \frac{2}{3}N$
EJEMPLO: "3$ 3$/3$-3$.3$ "

Complejidad espacial

Para calcular la complejidad espacial se deben considerar las siguientes variables ya que escalan con el tamaño del problema (N y K):

Dentro de la funcion Tokenizar:
- `str`
- `output`
- `word`

Dentro de la clase Tokenizador:
- `delimiters`
- `idxDelims`
- `idx`
- `idxCount`
- `especulativoNum`

El resto de variables tienen tamaño constante con lo que se ignorarán, a pesar de que tengan un tamaño considerable (especialmente los array de conversión a minúscula sin acentos).

También se han de tener en cuenta las concatenaciones y el reajuste automático de tamaño[3]: cada vez que un string no tiene tamaño reservado suficiente para una concatenación, se debe de reservar un nuevo string, del doble de tamaño que el original. Esto mejora la eficiencia temporal a costa de eficiencia espacial. Los 2 strings concatenados deben de existir

[3] https://www.oreilly.com/library/view/optimized-c/9781491922057/ch04.html

en memoria a la vez que el nuevo espacio reservado para concatenarlos. Esto resulta en una complejidad espacial lineal de, llamando T al tamaño del string concatenado, entre T (el tamaño del string concatenado es exactamente el ya reservado) y 3*T-2 (el string concatenado solo excede por 1 carácter la memoria ya reservada, *Figura 1*).

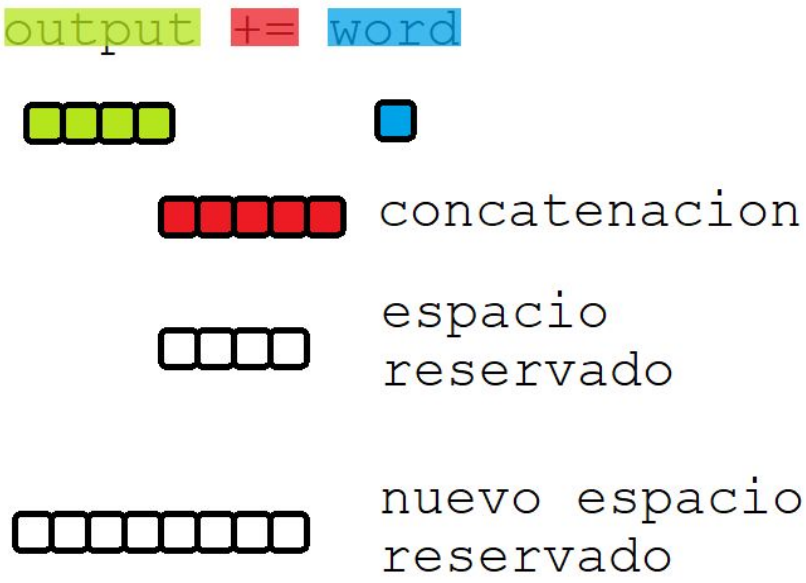

Figura 1: peor caso para el coste espacial de una concatenación. Siendo T el tamaño de la concatenación y S los caracteres que no caben en el espacio reservado, el tamaño necesario sería 3(T-S)+S, ó 3T-2S.

Las siguientes concatenaciones ocurren en el código:

```
output + word
word + especulativoNum
```

El tamaño de delimiters, idxDelims, idx e idxCount sería de K (o de la potencia de dos más cercana a K), el tamaño de str sería N, y el tamaño de output sería:

$$\sum_{w=0}^{numPalabras} tamPalabra_w + 1$$

Siendo el +1 un '\n' o hueco para un puntero en la lista encadenada por cada palabra (entre 0 y 2·N). El tamaño de word sería el de la palabra más grande (de 0 a N), y el tamaño de especulativoNum sería el de la secuencia numérica/palabra/acrónimo más grande después de una coma (de 0 a N-1).

El mejor caso sería de nuevo un string de todo delimitadores, con lo que tanto word como especulativoNum y output serían 0 y no se daría ninguna concatenación, resultando en una complejidad espacial de 4·K + N.

Para el peor caso hay varias posibilidades a considerar. Por un lado podemos buscar aumentar al máximo el tamaño de output, utilizando el mismo patrón que en el apartado anterior. Esto haría que el tamaño de output fuese ⅔·N·2, ya que tendría ⅔·N palabras de 2 caracteres (usando OutputString). Otra opción sería intentar aumentar al máximo el tamaño de word o especulativoNum. Esto convertiría el tamaño de output en N y el de word o especulativoNum en N.

Si tenemos en cuenta la concatenación output + word, el segundo caso tendría un peor tamaño de 3·N-2 (por ser la concatenación de tamaño N) mientras que el primer caso tendría un peor tamaño de 4·N-2 (por ser la concatenación de tamaño ⅔·N·2). Estos peores casos de concatenación se darían con longitudes de string que fuesen potencias de dos más uno.

Sin embargo la ecuación $\frac{4}{3}N = 2^i + 1$ no tiene solución entera[4]. La ecuación más cercana con solución entera sería $\frac{4}{3}N = 2^i + 4$, que representaría una concatenación con 4 caracteres más de los que caben en el espacio ya reservado. La nueva ecuación podría ser satisfecha por $N = 3 \cdot (2^j + 1)$. Estos valores de N con el patrón "dígito dólar delimitador" resultarían en una complejidad espacial de 4·N - 8, convirtiendo la complejidad espacial total en 4·K + 5·N - 8.

MEJOR CASO: un string de un solo carácter, que además sea delimitador.
COMPLEJIDAD ESPACIAL: 4·K + N
EJEMPLO: "....."

PEOR CASO: usando OutputString, un string compuesto del patrón dígito, dólar y delimitador repetido $(2^j + 1)$ veces.
COMPLEJIDAD ESPACIAL: 4·K + 5·N - 8
EJEMPLO: "3$ 3$ 3$ 3$ 3$ "

[4] *i* y *j* representan cualquier número entero positivo.