Universidad de Chile
Departamento de Ciencias de la Computación
CC4102 - Diseño y Análisis de Algoritmos

Auxiliar 5 - "Dominios Discretos y Finitos"

Profesor: Gonzalo Navarro
Auxiliar: ~~Manuel~~ Ariel Cáceres Reyes

02 de Octubre del 2017

P1. Ordenando Strings

Tenemos los `Strings` $w_1, w_2, \ldots w_n \in \Sigma$, con $\sigma = |\Sigma| \in \mathcal{O}(n)$ y queremos ordenarlos alfabéticamente. Además, el largo total de los `Strings` es $N = \sum |w_i|$.
Diseñe algoritmos de tiempo $\mathcal{O}(N)$ en los casos:

a) Todas las cadenas del mismo largo.

b) Cadenas de largo variable.

P2. Arreglo de Sufijos

Si $T = t_1 t_2 \ldots t_n$ es un texto de largo n, definimos su i-ésimo sufijo como $T_i = t_i t_{i+1} \ldots t_n$.
Un arreglo de sufijos de un texto T, SA_T, es un arreglo que cumple:

$$SA_T[i] = k \Leftrightarrow T_k \text{ es el i-ésimo en orden lexicográfico entre los sufijos de } T$$

Diseñe un algoritmo que encuentre el arreglo de sufijos de T en tiempo $\mathcal{O}(n \log n)$.

P3. Rank

Sea B una secuencia de bits de largo n. Se define $RANK(B,i)$ como el número de bits en 1 en $B[1,i]$, es decir:

$$RANK(B,i) = \sum_{0<j\leq i} B[j],\ 1 \leq i \leq n$$

a) Construya una estructura que permita calcular $RANK(B,i)$ en tiempo constante y utilice $2n + o(n)$ **bits** de espacio.

b) Resuelva el mismo problema, esta vez utilizando $o(n)$ **bits** de espacio.

"... supporting the desired data operations as efficiently as possible while increasing the space as little as possible ..."
Gonzalo Navarro

Soluciones

P1. a) Si adaptamos CountingSort para ordenar `Strings` de una letra su complejidad será $\mathcal{O}(n+\sigma)$, y por lo tanto hacer un RadixSort sobre estos *Strings* tiene costo $\mathcal{O}\left(\frac{N}{n}(n+\sigma)\right) = \mathcal{O}(N)$.

b) Si los `Strings` tienen diferente largo no podemos usar el RadixSort de la parte anterior pues este ordena desde el dígito menos significativo, a si es que por ejemplo *ba* quedaría antes de *a* al ordenar ascendentemente.

Otra opción es usar `padding` en las cadenas de largo menor a la más larga, de modo que tengamos n cadenas de largo $\max |w_i|$. Sin embargo, este procedimiento podría llegar a ser $\mathcal{O}(N^2)$ (si por ejemplo tenemos 1 cadena larga y las demás cortas).

La solución viene dada por procesar los `Strings` desde el dígito más significativo y hacer $\leq \sigma$ llamados recursivos para ordenar por el siguiente dígito. Este algoritmo es llamado MSD-RadixSort y se presenta a continuación:

```
//W es el arreglo de Strings y d el dígito por el que se está ordenando
//Asumimos también que CountingSort además retorna el arreglo que contiene las
  posiciones iniciales de cada letra
Funcion sort (W, d)
    if |W| ≤ 1 o last(W[1]) = −1 then
        return
    end
    count ← CountingSort(W, d)
    for r ← 1 . . . σ do
        sort(W[count[r]: count[r+1]-1], d+1)
    end
```

Por razones prácticas de implementación le agregaremos un -1 al final de las cadenas, lo que agrega solo n al tamaño del input.

Finalmente considerando que el costo amortizado por letra para CountingSort es de $\mathcal{O}(1)$ y notando que en el `peor` caso MSD-RadixSort procesa todas las letras del input con CountingSort, el costo total es $\mathcal{O}(N)$.

P2. Un primer intento es ordenar con cualquier algoritmo óptimo basado en comparaciones, lo que haría $\mathcal{O}(n \log n)$ comparaciones. Sin embargo, estas comparaciones (de cadenas) cuestan $\mathcal{O}(n)$ comparaciones de letras, por lo que la complejidad de esta solución es $\mathcal{O}(n^2 \log n)$. Otra forma sería usar RadixSort de la pregunta anterior para ordenar, que tomaría $\mathcal{O}(N) = \mathcal{O}(n^2)$. Veremos a continuación una forma de realizar esto en $\mathcal{O}(n \log n)$ gracias a Manber & Myers.

Este algoritmo va ordenando prefijos de los sufijos del texto, en fases, cuyos largos crecen exponencialmente. El análisis es simple, cada ordenación de prefijos de largo 2^i (dado que ya se ordenaron los de largo 2^{i-1}) se hace en $\mathcal{O}(n)$, por lo que el costo total del algoritmo es de $\mathcal{O}(n \log n)$. La observación clave que permite hacer saltos exponenciales en tiempo lineal es que "la parte que aún me falta comparar de los sufijos corresponden a prefijos de otras sufijos cuyo orden relativo puedo inferir a partir de lo que ya tengo ordenado".

Una descripción del algoritmo (sin muchos detalles de implementación) es la siguiente:

```
Ordenar los sufijos con CountingSort por su letra más significativa.
for i ← 1...log n do
    RadixSort(i)
end
```

Donde RadixSort(i) presupone que los sufijos están ordenados según los prefijos de tamaño 2^{i-1} y entrega como resultado el SA ordenado según los prefijos de tamaño 2^i. Para realizar esto RadixSort(i) utiliza el inverso del SA ya construido, lo que le indica el orden relativo de las primeras 2^{i-1} letras y también el orden relativo de las siguientes 2^i letras. De hecho SA^{-1} entrega un número $\in [n]$, por lo que puedo usar estos dos "caracteres" para hacer un RadixSort sobre ellos y tener el SA ordenado por las primeras 2^i letras en $\mathcal{O}(n)$. 😊

P3. a) Una primera idea consiste en tener un arreglo de n elementos que en la posición i contenga el valor de $RANK(B, i)$, sin embargo, como el $RANK$ puede llegar hasta n necesito $\mathcal{O}(\log n)$ bits para representar uno de estos valores y en total $\mathcal{O}(n \log n)$. 😭

💡

Dividir B en partes de tamaño $\frac{\lceil \log n \rceil}{2}$ y guardar el $RANK$ de los últimos elementos de cada una de las partes. Esto toma $\frac{n}{\frac{\lceil \log n \rceil}{2}} \log n = \frac{2n}{\lceil \log n \rceil} \log n \leq 2n$ bits de espacio.

Ahora que tengo el $RANK$ cada $\frac{\lceil \log n \rceil}{2}$ solo debo identificar el $RANK$ más cercano y luego calcular el resto en tiempo $\mathcal{O}\left(\frac{\lceil \log n \rceil}{2}\right)$.

💡

Para reducir la complejidad a $\mathcal{O}(1)$ precalcularemos todos los $RANK$ de todas las secuencias de largo $\frac{\lceil \log n \rceil}{2}$. Es decir, tendremos en una tabla $T[w, r] = RANK(w, r)$ que ocupará espacio $2^{\frac{\lceil \log n \rceil}{2}} \cdot \frac{\lceil \log n \rceil}{2} \cdot \log \frac{\lceil \log n \rceil}{2} = \mathcal{O}(\sqrt{n}) \cdot \mathcal{O}(\log n) \cdot \mathcal{O}(\log \log n) \in o(n)$ bits.🙏

b) El problema con la solución anterior es que el arreglo que guarda los $RANK$ de las partes ocupa $\mathcal{O}(n)$ bits de espacio.

💡

Aumentaremos el tamaño de las partes a $\frac{\log^2 n}{2}$, de este modo solo necesitamos $\frac{n}{\frac{\log^2 n}{2}} \log n = \frac{2n}{\log n} = o(n)$ bits de espacio.
Lamentablemente ya no podemos guardar todos los $RANK$ de todas las secuencias de tamaño $\frac{\log^2 n}{2}$ en espacio $o(n)$. 😭

"... supporting the desired data operations as efficiently as possible while increasing the space as little as possible ..."
Gonzalo Navarro

💡
Dividimos cada parte en $\log n$ subpartes de tamaño $\frac{\log n}{2}$ cada una y guardamos el $RANK$ del final de cada una, pero relativo a la parte en la cual se encuentra esa subparte. De este modo necesitamos $\log n \cdot \frac{n}{\frac{\log^2 n}{2}} \cdot \log\left(\frac{\log^2 n}{2}\right) \leq \frac{4n \log \log n}{\log n} = o(n)$. 🙏