Tarea 8

Saul Ivan Rivas Vega

Diseño y análisis de algoritmos

Equipo Completo:
Yadira Fleitas Toranzo
Diego de Jesús Isla Lopez
Saul Ivan Rivas Vega

27 de noviembre de 2019

1. Ejercicio 1.

Considera el siguiente problema de *formato de textos*. Como entrada tenemos un arreglo $W[1, ..., n]$ tal que cada $W[i]$ es una palabra; supondremos que el último símbolo en $W[i]$ es un espacio en blanco.También recibimos un entero **long_linea** > 0 como parte de la entrada, tal que **long_linea** es mayor o igual a la longitud de cada palabra $W[i]$, denotada $|W[i]|$. Lo que buscamos es dar *fomato* al texto en W para esto hay que decidir cuales palabras van juntas en la misma línea. Si decidimos poner en una misma línea las palabras $W[i], ..., W[i']$, $i \leq i'$, lo que se denota $l[i, i']$, entonces la *penalización* de $l[i, i']$ es:

- ∞ si **long_linea** $< |W[i]| + ... + |W[i']|$ (es decir, las palabras no caben en una sola línea);
- de otra forma, $(\textbf{long\_linea} - (|W[i]| + ... + |W[i']|))^3$

Entonces, la *penalización* de un *formato* $l_1[1, i_1], l_2[i_1 + 1, i_2], ..., l_j[i_{j-1} + 1, n]$ de W es la suma de las penalizaciones de sus líneas.

1.1. a) Demuestra que la siguiente estrategia *greedy* produce *formatos* arbitrariamente malos, es decir, con penalizaciones arbitrariamente grandes

Procesamos las palabras de $W[1]$ a $W[n]$; si aún hay espacio suficiente para meter $W[i]$ en la línea actual, entonces la metemos; de otra forma iniciamos una nueva línea en la que ponemos a $W[i]$.

Demostración. Podemos demostrarlo dando un ejemplo donde esto sucede.
Sea $n = 3$, $long\_linea = 6$ y con las longitudes $|W[1]| = 3$, $|W[2]| = 3$ y $|W[3]| = 2$.

- El algoritmo tomará la línea actual con espacio disponible de 6 y procesará la primera palabra, $W[1]$.
- Como el espacio disponible es 6 y $|W[1]| = 3$ entonces la metemos en la línea actual.
- Ahora el espacio disponible es 3 y como $|W[2]| = 3$ entonces la metemos en la línea actual.
- Como el espacio disponible es 0 y $|W[3]| = 2$ creamos una nueva linea y metemos a $W[3]$ en ella.

Al terminar el algoritmo terminamos con un formato f_1 con las lineas $l_1[1, 2]$ y la linea $l_2[3, 3]$. Veamos sus *penalizaciones*.

En el caso de $l_1[1,2]$, $|W[1]|+|W[2]|$ es igual a *long_linea* entonces entra en el segundo caso de la función de *penalización* entonces la calculamos con:

$$\begin{aligned} penalizacion(l_1[1,2]) =&(long\_linea - (|W_1| + |W_2|))^3 \\ =&(long\_linea - (3+3))^3 \\ =&(long\_linea - 6)^3 \\ =&(6-6)^3 \\ =&(0)^3 \\ =&0 \end{aligned} \tag{1}$$

Para $l_2[3,3]$, $|W[3]|$ es menor a *long_linea* entonces entra en el segundo caso de la función de *penalización* entonces la calculamos con:

$$\begin{aligned} penalizacion(l_2[3,3]) =&(long\_linea - (|W_3|))^3 \\ =&(long\_linea - (2))^3 \\ =&(long\_linea - 2)^3 \\ =&(6-2)^3 \\ =&(4)^3 \\ =&64 \end{aligned} \tag{2}$$

Ahora la *penalización* del formato f_1 es la suma de las dos penalizaciones de sus líneas:

$$penalización\ f_1 = 64 + 0 = 64 \tag{3}$$

Sin embargo, tomemos el siguiente formato f_2 con las líneas, $l_1[1,1]$ y $l_2[2,3]$. Para $l_1[1,1]$ tenemos:

$$\begin{aligned} penalizacion(l_1[1,1]) =&(long\_linea - (|W_1|))^3 \\ =&(long\_linea - (3))^3 \\ =&(long\_linea - 3)^3 \\ =&(6-3)^3 \\ =&(3)^3 \\ =&27 \end{aligned} \tag{4}$$

Y para $l_2[2,3]$ tenemos:

$$\begin{aligned} penalizacion(l_2[2,3]) =&(long\_linea - (|W_2| + |W_3|))^3 \\ =&(long\_linea - (3+2))^3 \\ =&(long\_linea - 5)^3 \\ =&(6-5)^3 \\ =&(1)^3 \\ =&1 \end{aligned} \tag{5}$$

Por lo tanto la *penalización* total para el formato f_2 es:

$$\textit{penalización } f_2 = 27 + 1 = 28 \tag{6}$$

Finalmente como f_1 es mayor que f_2 y además podemos formar arreglos agregando las mismas 3 longitudes del ejemplo en ese orden decimos que el algoritmo produce *formatos* con penalizaciones arbitrariamente grandes.

1.2. b) Usa la técnica de programación dinámica para diseñar un algoritmo que encuentre un formato con penalización mínima. Demuestra la correctitud de tu algoritmo y haz un análisis de tiempo y espacio.

Tomemos la siguiente función para calcular el formato óptimo para las primeras i palabras:

$$OPT(i) = \begin{cases} 0, & i = 0 \\ min_{j \in [1,\ldots,i]}(OPT(j-1) + FuncionP(l[j,i])), & \text{en otro caso} \end{cases} \tag{7}$$

OPT toma la i-ésima palabra y busca la penalización mínima de colocarla en una linea que empieza en j donde j esta en [1,...,i], así se calcula la penalización de la línea l[j,i] y se suma al óptimo con las primeras (j-1) palabras, lo cual se calcula con OPT(j-1).
Donde la función de penalización $FuncionP$ esta definida como en el enunciado del ejercicio:

$$FuncionP(l[i,j]) = \begin{cases} \infty, & \textbf{long\_linea} < |W[i]| + \ldots + |W[j]| \\ (\textbf{long\_linea} - (|W[i]| + \ldots + |W[j]|))^3, & \text{en otro caso} \end{cases} \tag{8}$$

Ahora con la función de optimización podemos escribir un algoritmo de programación dinámica como el siguiente:

Data: Arreglo de palabras W, entero positivo *long_linea*.
Result: Tupla $(minp, L)$ Que representa el valor de la penalización mínima y un arreglo de lineas L donde cada elemento es una tupla (i, j) que representa una linea que comienza con la palabra en i y termina con la palabra en j.

```
/* Inicializamos un arreglo para guardar las penalizaciones minimas de tamaño n,
   donde n = |W|.                                                          */
1  M = [n];
   /* Inicializamos el caso base.                                          */
2  M[0] = 0;
   /* Recorremos todos los valores desde 1 hasta n.                        */
3  for i = 1, i ≤ n do
       /* Ahora para cada posición i revisaremos cual es la mejor línea que incluye a
          i, que es revisar las líneas que comienzan en una posición j ∈ [1, ..., i].  */
4      minp = ∞;
5      for j = 1, j ≤ i do
           /* Nos vamos quedando con el menor minp que veamos, en cada iteración
              obtenemos la penalización de la línea más el formato óptimo de las
              primeras j − 1 palabras.                                     */
6          minp = min(minp, M[j − 1] + FuncionP(W, long_linea, j, i))
7      end
       /* Guardamos la menor penalización para las primeras i palabras.    */
8      M[i] = minp;
9  end
   /* Inicializamos el arreglo respuesta L.                                */
10 L = [];
   /* La menor penalización se encuentra en M[n], ahora reconstruyamos la respuesta
      que es recorrer las posiciones que registraron el mínimo en cada i desde el
      final, Recorreremos el arreglo con un while por que no es necesario recorrer
      todas las posiciones i ∈ [1, ..., n], sino solamente las que pertenecen a la
      respuesta óptima                                                     */
11 i = n;
12 while i > 0 do
13     for j = 1, j ≤ i do
           /* revisamos si para la línea que termina en la palabra i empezar en j fue
              lo óptimo                                                    */
14         if M[j − 1] + FuncionP(W, long_linea, j, i) = M[i] then
               /* En ese caso agregamos la tupla a la respuesta y además pasamos a la
                  siguiente posición i                                     */
15             L ← (j, i);
16             i = j − 1;
               /* Interrumpimos el actual ciclo For                        */
17             break;
18         end
19     end
20 end
21 return (M[n], L);
```

Algorithm 1: Algoritmo FORMATODP.

Data: Arreglo de palabras W, entero positivo $long\_linea$, indice de inicio ini, indice de fin fin.
Result: Un valor de penalización de incluir a las palabras de $W[ini, ..., fin]$ en una sola línea.

```
/* Obtenemos la suma de las longitudes de las líneas.                    */
sumaW = 0;
for i = ini, i ≤ fin do
    sumaW = sumaW + |W[i]|;
end
/* Evaluamos la condición para la penalización con respecto a long_linea. */
if sumaW > long_linea then
    /* En este caso la suma de las longitudes es mayor a long_linea.     */
    return ∞;
end
return (long_linea − sumW)^3;
```

Algorithm 2: Algoritmo FuncionP.

1.2.1. Demostración de correctitud

El algoritmo FuncionP termina. El algoritmo termina pues las únicas operaciones que realiza son evaluaciones y asignaciones que podemos considerar constantes, con algunas en un ciclo For, el cual realizara $fin - ini + 1$ iteraciones, donde $ini, fin \in [1, ..., n]$, por lo tanto la mayor cantidad de iteraciones que se realizan en una invocación del algoritmo es cuando $ini = 1$ y $fin = n$ y se harán $n - 1 + 1 = n$ iteraciones. Finalmente decimos que el algoritmo FuncionP termina después de n iteraciones.

El algoritmo FORMATODP termina. El algoritmo en una primera parte realiza dos ciclos For anidados, como estos recorren n posiciones a lo mas podemos decir que realiza n^2 iteraciones, sin embargo en cada iteración ademas de realizar comparaciones y asignaciones de tiempo constante el algoritmo además llama a FuncionP, y como vimos en el párrafo anterior este termina después de a lo más n iteraciones. Por lo tanto decimos que realiza n^3 operaciones. En una segunda parte de igual forma tiene dos ciclos anidados y de igual forma se recorren a lo mas las n posiciones del arreglo pues i empieza igual a n y termina hasta llegar a 0, puede brincar en una sola iteración, sin embargo también puede ser el caso que se recorran todas las n posiciones, haciendo que los ciclos anidados de igual forma realicen n^2 iteraciones. En cada iteración se llama también a FuncionP, agregando otro factor de n. Entonces decimos que este segundo ciclo anidado realiza n^3 iteraciones. Finalmente el algoritmo después de recorrer los dos ciclos anidados tenemos que termina realizando $2(n^3)$ iteraciones.

El algoritmo FuncionP es correcto. El algoritmo implementa la evaluación y los resultados de los dos casos que se presentaron en el enunciado del problema entonces corresponde a los valores que la función debe retornar.

El algoritmo FORMATODP es correcto. Por inducción.
Invariante: En el formato óptimo las líneas tienen como fin una palabra i, además tienen como inicio una palabra $j \in [1, ..., i]$, y la $(j-1)-$ésima palabra es la palabra al final de la siguiente línea.
Caso base: Hay 0 palabras en el arreglo, el formato óptimo no contempla ninguna línea, la penalización es 0. **Hipótesis de inducción:** El algoritmo obtiene el inicio $j \in [1, ..., i]$ con el que se obtiene el formato de penalización mínima que incluye a la línea $l[j, i]$. Suponer verdadero hasta la i-ésima palabra, esta i-ésima palabra debe pertenecer a alguna línea en el formato final, dicha linea puede empezar y terminar en i, siendo una línea con solo una palabra, formalmente la linea es de la forma $l[j, i]$ donde $j \in [1, ..., i]$, por nuestra suposición inicial si tomamos a algún j como nuestro inicio, ya sabemos cual es el formato de penalización mínima que incluye a la línea que termina en $j-1$. Por lo tanto debemos obtener solamente aquel j tal que minimice la suma del formato óptimo con las primeras $j-1$ palabras mas la penalización de la línea $l[j, i]$. Que es recorrer todos los $j \in [1, ..., i]$ para obtener el mínimo de todos. Finalmente como es exactamente lo que realiza el algoritmo decimos que el algoritmo FORMATODP es correcto.

1.2.2. Análisis de tiempo.

El algoritmo FuncionP tiene complejidad $O(n)$. Como vimos en la sección anterior el algoritmo solo realiza un ciclo con n iteraciones ademas de operaciones como asignaciones y evaluaciones que podemos considerar de tiempo constante. Por lo tanto decimos que el algoritmo es de orden $O(n)$.

El algoritmo FormatoDP tiene complejidad $O(n^3)$. Como vimos en la sección anterior el algoritmo termina después de realizar $2n^3$ iteraciones adicionalmente se realizan operaciones como asignaciones y evaluaciones que podemos considerar de tiempo $O(1)$. Por lo tanto decimos que el algoritmo es de orden $O(2n^3)$ y tomando el factor dominante finalmente decimos que la complejidad es $O(n^3)$.

1.2.3. Análisis de espacio.

El algoritmo FuncionP tiene espacio de orden $O(1)$. Adicional a la entrada el algoritmo solo hace uso de una variable $sumaW$ e i para el ciclo. Dichas variables no ocupan mas espacio conforme crece n, por lo tanto se mantiene constante y finalmente decimos que el algoritmo ocupa espacio de orden $O(1)$.

El algoritmo FORMATODP ocupa espacio de orden $O(n)$. Adicional a la entrada el algoritmo solo usa un arreglo de n posiciones para guardar los mínimos y un arreglo de tuplas L que en el caso que cada palabra este en su propia linea, este será de tamaño $2n$, pues se guarda una variable de inicio y una de fin. Por lo tanto el algoritmo ocupa $3n$ mas unas variables constantes de asignación o para recorrer ciclos. Finalmente decimos que el algoritmo ocupa espacio de orden $O(n)$.

1.3. c) Modifica tu algoritmo para que funcione con una función de penalización de línea dada; la penalización de un formato sigue siendo la suma de las penalizaciones de sus líneas.

2. Ejercicio 2

Considera un arreglo $A[1, ..., n]$ con enteros positivos en sus entradas. Decimos que una pareja (i, j) es un *declive* si $i \leq j$ y $A[i] \geq A[j]$; la *longitud* de (i, j) es $A[i] - A[j]$. Diseña un algoritmo de tiempo y espacio $o(n^2)$ que calcule un declive de A de longitud máxima (es o pequeña, investiga ese concepto). Demuestra que tu algoritmo es correcto y haz el análisis de tiempo y espacio.

Data: Arreglo A.
Result: Tupla (max_long, r_i, r_j) Que representa el valor de la longitud máxima de un declive en A y los índices r_i, r_j de dicho declive (r_i, r_j).

```
   /* Inicializamos una cola de prioridad donde guardaremos tuplas (A[i], i) donde
      ordenaremos mayor a menor por A[i].                                          */
 1 Q = {};
   /* Inicializamos las respuestas actuales.                                       */
 2 max_long = −∞;
 3 r_i = 0;
 4 r_j = 0;
   /* Recorremos del final al inicio el arrego A, pasando por todas las posiciones
      desde n hasta 1.                                                             */
 5 for i = n, i ≥ 1 do
       /* Insertamos el elemento actual a la cola de prioridad.                   */
 6     (A[i], i) → Q;
       /* Obtenemos el elemento al frente de Q, es decir la tupla (A[i], i) con menor
          A[i], no lo removemos de la cola de prioridad solo obtenemos sus valores. */
 7     (valor, indice) ← Q;
       /* Como procesamos de fin a inicio este elemento cumple que i ≤ indice, ahora
          entonces revisamos si A[i] ≥ valor lo cual sería un declive (i, indice).  */
 8     if A[i] ≥ valor then
           /* Calculamos la longitud del declive (i, indice).                     */
 9         current_long = A[i] − valor;
           /* Actualizamos nuestra respuesta si es mayor a la que llevabamos.     */
10         if current_long > max_long then
11             max_long = current_long;
12             r_i = i;
13             r_j = indice;
14         end
15     end
16 end
17 return (max_long, r_i, r_j);
```

Algorithm 3: Algoritmo MAXDECLIVE.

2.1. Demostración correctitud.

El algoritmo termina. El algoritmo termina puesto que solo realiza un loop con n iteraciones en las cuales se realizan asignaciones y comparaciones que podemos asumir de tiempo constante, sin embargo también se realizan inserciones en una cola de prioridad que como hemos revisado anteriormente, son de tiempo $logn$. Por lo tanto podemos decir que el algoritmo termina después de n iteraciones, posteriormente en el análisis temporal desarrollaremos la complejidad de cada iteración.

El algoritmo devuelve un declive de longitud máxima. Podemos partir de explicar cuál es un algoritmo que por fuerza bruta encuentra la respuesta. Es un algoritmo que solo intentará todas las posibles parejas i, j donde $i \leq j$ y verificará si es un declive, conservando el declive con longitud máxima, lo cuál sería de tiempo $O(n^2)$ lo cual no cumple con nuestra restricción temporal de $o(n^2)$. Sin embargo tomemos la siguiente observación.

Observación 1 No es necesario revisar todos los j tales que $j \geq i$ para encontrar un declive de longitud máxima que tiene a i como primer elemento, solo es necesario revisar al elemento j tal que $j \in [i, ..., n]$ y además tenga el menor valor en $A[i, ..., n]$.

Demostración de la Observación 1 Por contradicción.
Supongamos que el declive de longitud máxima que tiene a i como primer elemento no es con el j tal que $j \in [i, ..., n]$ y además tenga el menor $A[j]$, sino con un j' tal que por lo menos cumple que $j' \in [i, ..., n]$ y además $A[j] \leq A[i]$ para ser un declive. Si j' tiene el menor valor en $A[i, ..., n]$ entonces significa que $j' = j$, en otro caso significa que $A[j] < A[j']$, y podemos desarrollar la siguiente ecuación:

$$\begin{aligned} A[j] &< A[j'] \\ -A[j] &> -A[j'] \\ A[i] - A[j] &> A[i] - A[j'] \end{aligned} \tag{9}$$

Lo cual significa que la longitud del declive tomando al elemento j es mayor al declive tomando al elemento j', pero esto es una contradicción puesto que en la suposición inicial el declive (i, j') era el de longitud máxima. Finalmente como no puede existir dicho j' decimos que el declive de longitud máxima que tiene como primer elemento a i es (i, j) donde el elemento j cumple que $j \in [i, ..., n]$ y además tiene el menor valor en $A[i, ..., n]$

Ahora tomando la **Observación 1** podemos demostrar la correctitud del algoritmo por contradicción. Supongamos que el declive de longitud máxima (i', j') no fue devuelto por el algoritmo. Por la **Observación 1** para dicho declive significa que j' cumple que $j' \in [i', ..., n]$ y además tiene el menor valor en $A[i', ..., n]$. El algoritmo debió haber ignorado a i' puesto que para cuando se encuentra en el índice i justamente revisa el elemento j tal que cumple que $j \in [i, ..., n]$ y además tiene el menor valor en $A[i, ..., n]$. Sin embargo, el algoritmo si revisa todos los índices $1 \leq i \leq n$ pero esto es una contradicción puesto que no puede haber revisado e ignorado al índice i'. Finalmente decimos que el algoritmo devuelve un declive de longitud máxima.

2.2. Análisis de tiempo

El algoritmo es de complejidad $O(nlogn)$ y por lo tanto $o(n^2)$ Como vimos en la sección donde se explicaba que el algoritmo termina, el algoritmo solo realiza un loop de fin a inicio por los n elementos en A, sin embargo en cada iteración además de las operaciones de asignación y comparación que podemos asumir de tiempo $O(1)$, también tenemos operaciones de inserción y consulta a una cola de prioridad que como anteriormente hemos visto son de complejidad $O(logn)$. Por lo tanto el algoritmo es de complejidad $O(nlogn)$ y como para toda constante $c > 0$ existe un n_0 tal que para toda $n \geq n_0$ se cumple que $nlogn < n^2$ decimos que el algoritmo es $o(n^2)$.

2.3. Análisis de espacio

El algoritmo ocupa espacio de orden $O(n)$ Adicional a la entrada el algoritmo ocupa algunas variables individuales para las respuestas y consultar a la cola de prioridad las cuales podemos considerar de orden $O(1)$. Sin embargo también hacemos uso de una cola de prioridad y adicionalmente no removemos los elementos que insertamos en ella y como cada elemento es una pareja, tendremos al final $2n$ valores en la cola de prioridad. Como $2n$ es el factor dominante decimos que el algoritmo tiene espacio de orden $O(2n)$ y finalmente $O(n)$.

3. Ejercicio 3

Considera el problema de selección de centros visto en clase. Demuestra que el siguiente algoritmo devuelve un conjunto C con a lo más k centros tal que $rc(C) \leq 2rc(C^*)$, donde C^* es un conjunto con a lo más k centros óptimo, es decir con radio de covertura mínimo. Puedes suponer como correctos todos los algoritmos y afirmaciones vistas en clase.

```
Alg. SelecCentros(S, k)
   If |S| <= k then
      return S
   Else
      Inicializa C con cualquier elemento s de S
      While |C| < k do
         Elegir cualquier elemento s en S que maximice dist(s,C)
         Agregar s a C
      endwhile
      return C
```

4. Ejercicio 4

Escribe una versión recursiva del algoritmo probabilístico de corte mínimo visto en clase (basado en contracciones de aristas). Demuestra que el conjunto de aristas devuelto por el algoritmo es efectivamente un corte de la gráfica inicial.

5. Ejercicio 5

Tenemos n servidores que buscan coordinarse para ejecutar localmente la misma acción. De forma *abstracta*, los servidores llegan a un *consenso* sobre un bit, y ejecutan localmente la acción asociada. Por ejemplo, si el bit consensuado es 0, cada servidor hace *rollback* en su base de datos local, pero si el bit consensuado es 1, cada servidor hace *commit* localmente. Considera el siguiente algoritmo probabilístico para este problema:

```
Alg. ConsensoBinario
   r = 0
   While TRUE do
      1. r = r+1
      2. Cada servidor obtiene un bit aleatorio b_r con probabilidad uniforme
      3. Cada servidor comunica su bit b_r a todos los servidores
      4. Si todos los bits b_r de los n servidores son iguales, cada servidor
         ejecuta localmente la tarea correspondiente y termina
      5. De otra forma, continua
   endwhile
```

Responde a lo siguiente:

1. Demuestra que el número esperado de iteraciones para ejecutar la acción es $O(2^n)$. Tip: Modela el problema con una variable aleatoria con distribución geométrica.

2. ¿Cuál es el número esperado de iteraciones si en cada una de ellas los servidores obtienen su $r-$ésimo bit, b_r, llamando una función **shared_random_bit**(r) que devuelve el mismo $r-$ésimo bit a todos los servidores con probabilidad p, para alguna constante $0 < p < 1$?