DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Problemas Algoritmos y Programación
Segundo Cuatrimestre de 2016

Integrante	LU	Correo electrónico
Ariel FUTORANSKI	1/0	elenmascarado@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Problema A: Ayudando a los gorilas

1.1. Introducción:

En el siguiente problema se desea saber si dado un nombre n y un apodo a, si a esta contenido dentro de n o sea, si a es un substring de n. Además se pide que resolvamos el problema con una complejidad O(longitud(n)).

1.2. Desarrollo:

Para resolver este problema, se decidió utilizar el algoritmo de $Knuth - Morris - Pratt$ mas conocido como KMP. El último es un algoritmo que dados dos textos,t_1 y t_2, permite saber en O(longitud(t_1)) si t_2 es un substring de t_1.

Para lograr esta complejidad, KMP aprovecha información que guarda sobre fallos previos mientras se fija si t_2 esta contenido en t_1 y una tabla precalculada, utilizando información de t_2. El objetivo de esta tabla es no permitir que cada carácter de t_1 sea analizado más de una vez, y lograr de esta manera una complejidad lineal en función del tamaño de t_1.

1.3. Pseudocodigo:

Algorithm 1 completarTablaKMP

```
1:  procedure COMPLETARTABLAKMP(palabra, tablaKMP[int])
2:      actual ← 1
3:      principio ← 0
4:      tablaKMP[0] ← 0
5:      while actual < |palabra| do
6:          if palabra[actual] == palabra[principio] then
7:              tablaKMP[actual] ← principio + 1
8:              principio ← principio + 1
9:              actual ← actual + 1
10:         else
11:             if principio > 0 then
12:                 principio ← tablaKMP[principio-1]
13:             else
14:                 tablaKMP[actual] ← 0
15:                 actual ← actual + 1
16:             end if
17:         end if
18:     end while
19: end procedure
```

Algorithm 2 stringMatching

```
1: procedure STRINGMATCHING(palabra, texto)
2:     // inicializamos todas las posciones de tablaKMP en 0
3:     tablaKMP[—palabra—] ← 0
4:     completarTablaKMP(palabra, tablaKMP)
5:     m ← 0
6:     i ← 0
7:     while m < |texto| do
8:         if palabra[i] == texto[m] then
9:             if i == |palabra| − 1 then
10:                retornar m
11:            end if
12:            i ← i + 1
13:            m ← m + 1
14:        else
15:            if i ≠ 0 then
16:                i ← tablaKMP[i-1]
17:            else
18:                m ← m + 1
19:                i ← 0
20:            end if
21:        end if
22:    end while
23:    retornar —texto—
24: end procedure
```

1.4. Correctitud:

Como se menciono anteriormente, dados dos textos t y p, se utiliza el algoritmo KMP para ver si p esta contenido dentro de t, el cual primero precalcula una tabla con información sobre p.

Será un arreglo de longitud $|p|$, en el cual habrá un k en la posición i si existe un subarreglo de p longitud k (que termina en i) que es a la vez sufijo y prefijo del subarreglo que va desde la posición 0 hasta la posición i. Es decir, si $p[i-k+1..i] = p[0..k-1]$. Este k es el máximo posible para cada posición i

Veamos que calculamos este arreglo de manera correcta. Llamemos a este arreglo $TablaKMP$.

En la primera posición del arreglo, colocamos un 0, pues en realidad para cualquier i, el subarreglo $p[0..i]$ es sufijo y prefijo. Entonces no lo contamos como tal.

Mantendremos dos índices, i que será la posición del arreglo que vamos a querer calcular, y el otro será j, que indica el elemento con el que tendremos que comparar a $p[i]$. El i comenzará en 1 y el j en 0.

Entonces dados i y j proseguimos de la siguiente manera:

- Si $p[i] = p[j]$, entonces $TablaKMP[i]$ es igual a j+1. Luego aumentamos en uno a i y a j.
- Si $p[i] \neq p[j]$ entonces:
  - Si $j \neq 0$, j es igual a $tablaKMP[j-1]$
  - Si $j = 0$, $TablaKMP[i]$ es igual a 0. Aumentamos en uno a i

(1) Dados i, j (surgidos luego de algunas iteraciones de nuestro algoritmo) vale lo siguiente: $p[0..j-1] = p[i-j..i-1]$. Es decir, el subarreglo de los j elementos anteriores a i son prefijos y sufijos del subarreglo correspondiente.

(1) Demo: Por inducción.

- Caso base: $j = 0$, $i = 1$. $p[0..j-1] = p[0..-1] = [] = p[1.,0] = p[i-j..i-1]$
- Paso inductivo: Asumo que vale para j, i, quiero ver que vale para los próximos valores de ambos, j' y i'. Dividamos en los 3 casos que puede tomar nuestro algoritmo

- Si $p[i] = p[j]$, entonces por HI vale que $p[0..j] = p[i-j..i]$. Los nuevos valores j' y i', son $j' = j+1$ y $i' = i+1$. Entonces $p[0..j'-1] = p[i'-j'..i'-1]$
- Si $p[i] \neq p[j]$ y $j \neq 0$, entonces $j' = tablaKMP[j-1]$ y $i' = i$. Por HI vale que $p[0..j-1] = p[i-j..i-1]$. Como sabemos que $tablaKMP[j-1]$ es menor a j entonces también vale que $p[0..tablaKMP[j-1]-1] = p[i-tablaKMP[j-1]..i-1]$. Entonces $p[0..j'-1] = p[i-j'..i-1]$. Entonces $p[0..j'-1] = p[i'-j'..i'-1]$
- Si $p[i] \neq p[j]$ y $j = 0$ entonces $i' = i$. Vale entonces que $p[0..j'-1] = p[0..,0-1] = [] = p[i'..i'-1]$ $= p[i'-j'..i'-1]$

Entonces si $p[i] = p[j]$ sabemos que $p[i]$ es sufijo y prefijo. Pero además, si $j > 0$, quiere decir que $p[i-j..i-1]$ es prefijo y sufijo. Entonces $p[i-j..i]$ tambien será prefijo y sufijo (pues $p[i] = p[j]$). Entonces el valor que corresponde a $TablaKMP[i]$ debe ser $j+1$, pues extendemos un prefijo-sufijo ya encontrado. Incrementamos en uno ambos indices, pues ahora nos interesa comparar las posiciones siguientes, para ver si encontramos un prefijo-sufijo de longitud $j+2$.

Si $p[i] \neq p[j]$ (y $j \neq 0$) entonces no podemos extender el prefijo-sufijo ya encontrado. Entonces debemos encontrar el máximo posible prefijo-sufijo que podamos extender. Es decir, un $k < j$ tal que $p[0..k-1]$ $= p[i-k..i-1]$. Este mismo k se haya en $tablaKMP[j-1]$, pues el segundo máximo prefijo-sufijo es el máximo prefijo-sufijo del máximo prefijo-sufijo. Entonces j debe ser igual a $tablaKMP[j-1]$. De esta manera, en la próxima iteración, analizaremos si existe un prefijo-sufijo de longitud $k+1$.

En el caso restante, que j sea igual a 0, entonces no existe un prefijo-sufijo con el caracter $p[i]$ (es $p[i]$ es distinto a $p[0]$, y no hay prefijos-sufijos con los predecesores de i). Entonces el valor de $tablaKMP[i]$ debe ser 0. Luego hay que incrementar el i para seguir con la busqueda de prefijos.

Entonces estamos llenando $tablaKMP$ correctamente. Veamos ahora cómo utilizamos esto para buscar a p en t.

Tenemos dos índices i y m. Vamos a iterar a i por la p y a m por t. Comenzamos desde la posición 0 de ambos strings.

La idea es que, dados los valores de i, m luego de algunos ciclos de ejecución, valga que: $p[0..i-1] = t[m-i..m-1]$. Es decir, que el substring de p desde 0 hasta $i-1$ sea igual al substring de longitud i que finaliza en $m-1$. De esta manera, luego, analizaremos los elementos $p[i]$ y $t[m]$.

- Si $p[i] = t[m]$, entonces aumentamos i y m, pues ahora vale que $p[0..i] = t[m-i..m]$, es decir, podemos agrandar el substring de p que se encuentra en el t. Si $p[i]$ era la ultima letra de p, entonces encontramos un matching.
- Si $p[i] \neq t[m]$ y $i = 0$, es decir, estamos comparando la primer letra de p y no son iguales, entonces simplemente incrementamos m, pues no va a existir matching desde m.
- Si $p[i] \neq t[m]$ y $i \neq 0$ tambien sabemos que no va a haber matching (desde la posición $m-i$ del t). Sabemos dos cosas. En primer lugar, que había matching parcial de longitud i, es decir, $p[0..i-1] = t[m-i..m-1]$. En segundo lugar, utilizando la $TablaKMP$ calculada anteriormente, que $p[0..f-1]$ $= p[i-f..i-1]$ siendo $f = TablaKMP[i-1]$. Entonces vale que $t[m-f..m-1] = p[i-f..i-1]$. Además como, por lo dicho anteriormente vale que $t[m-f..m-1] = p[0..f-1]$. Por lo tanto necesitamos decrementar i hasta f, pues luego seguiremos con la comparación entre $p[f+1]$ y $t[m]$, ya que ya sabemos que el substring $p[0..f-1]$ es igual a $t[m-f..m-1]$.

Entonces, de esta manera, si existe matching lo vamos a encontrar.

1.5. Complejidad:

Precalculo sobre t_2:

El ciclo principal de la función que precalcula la tabla $TablaKMP$, itera desde $i = 0$ hasta $i = |t_2|$. El indice i solamente aumenta de a uno (si es que aumenta). Entonces, si la cantidad de veces que no aumenta es $\mathbf{O}(|t_2|)$, el ciclo va a terminar en $\mathbf{O}(|t_2|+|t_2|) = \mathbf{O}(|t_2|)$. Esto es valido porque las operaciones dentro del ciclo son $\mathbf{O}(1)$. Veamos que sucede esto.

Si i no aumenta, entonces estamos en el caso que el indice j va a disminuir. Para un i fijo, el j podría disminuir varias veces (en diferentes iteraciones) hasta ser igual a 0. En el peor caso, podría disminuir j veces, es decir, la cantidad de iteraciones que ese i no aumentará será j. Entonces la cantidad total de veces que el indice i no aumenta es igual a la sumatoria de los distintos j_a, con $0 \leq a \leq |t_2| - 1$. Veamos que esta sumatoria es menor o igual a $|t_2|$.

Propiedad: El algoritmo finaliza con el indice i a $|t_2|$. Esto se ve fácilmente del algoritmo, pues i sólo aumenta de a uno, y el ciclo finaliza cuando es mayor o igual a $|t_2|$.

Por absurdo. Supongamos que la suma es mayor a $|t_2|$. Como cada vez que j aumenta, i también aumenta. Y como cada j_a disminuye hasta cero, entonces necesariamente el indice j se tuvo que haber aumentado $\sum j_a$. Por lo tanto el índice i resultó aumentado por lo menos $\sum j_a$ veces. $\sum j_a$ ¿$|t_2|$, entonces i es mayor $|t_2|$. Pero habíamos probado que el i finalizaba en $|t_2|$. ABS!

Entonces queda probado que este precalculo es $\mathbf{O}(|t_2|)$

Complejidad KMP:

Demostremos que la complejidad es $\mathbf{O}(|t_1|)$ de forma similar a lo hecho anteriormente. Sea m el índice que itera por el texto e i el que itera por la palabra.

Tenemos el ciclo principal, que itera desde $m = 0$ hasta $m = |t_1|$. El indice m aumenta de a uno, si aumenta. Las operaciones dentro del ciclo son $\mathbf{O}(1)$, por lo tanto el ciclo termina en $\mathbf{O}(|t_1|)$ si la cantidad de veces que no aumenta es $\mathbf{O}(|t_1|)$. Probemos que vale esto.

Igual que antes, si el índice m no aumenta, entonces el índice i disminuye. Sea i_a la cantidad de veces que disminuye i cuando se encontraba en la posición a, con $\leq a \leq |t_2| - 1$. Entonces la cantidad de veces que m no aumenta es igual a la sumatoria de las i_a.

Veamos que esta sumatoria es menor o igual a $|t_1|$. Por absurdo. Supongamos que es mayor. Sabemos que i se debe haber aumentado por lo menos $\sum i_a$ veces. Como cada vez que i aumenta, también lo hace m, quiere decir que m es mayor a $\sum i_a$ ¿$|t_1|$. Pero m debía finalizar en $|t_1|$ (misma propiedad que en la demo anterior). ABS!

Entonces la complejidad es $\mathbf{O}(|t_1|)$.

Conclusion:

La primera parte tiene un costo $\mathbf{O}(|\ t_2\ |)$, mientras que la segunda parte tiene un costo en complejidad igual a $\mathbf{O}(|\ t_1\ |)$. Entonces la complejidad seria igual a $\mathbf{O}(|\ t_2\ | + |\ t_1\ |)$ pero como en este problema $|\ t_1\ | \leq |\ t_1\ |$, la complejidad final es de $\mathbf{O}(|\ t_1\ |)$.

2. Problema B: Buscando a los alumnos

2.1. Introducción:

La cátedra de la materia Problemas, Algoritmos y Programación (PAP) recuerda los prefijos de los correos de sus estudiantes pero nunca los correos completos lo cual siempre le trae problemas al momento de querer comunicar las notas de parciales.

Para solucionar esto decidieron configurar su cliente de correos electrónicos de manera que al ingresar un prefijo les muestre a lo sumo una cantidad T de correos que comienzan con él, donde $T \in \mathbb{N}_0$ es un valor fijo seteado en las configuraciones.

Se desea encontrar el mínimo T tal que para todo prefijo puedan verse todos los correos que comiencen con él.

Más formal, dados $D_1, D_2, \ldots, D_N$ correos y $P_1, P_2, \ldots, P_N$ sus respectivos prefijos, con $N \in \mathbb{N}$.

Se busca

$$Min_{\,T \in \mathbb{N}_0}\ (\forall 1 \leq i \leq N)\,|\{D_j \,|\, (\forall 1 \leq j \leq N)\, P_i \textit{ es prejifo de } D_j\}| \leq T$$

Lo que es igual a hayar

$$Max_{\,1 \leq i \leq N}\,|\{D_j \,|\, (\forall 1 \leq j \leq N)\, P_i \textit{ es prejifo de } D_j\}|$$

Se pide una complejidad temporal $\mathcal{O}$(S), con $S = \sum_{i=1}^{N} |D_i|$, donde $|D_i|$ la longitud en caracteres del *i-ésimo* correo.

2.2. Desarrollo:

Notación 2.1. *CANT_LETRAS es la cantidad de caracteres posibles. Se asumen caracteres alfabéticos sin acentos y sin la letra 'ñ'.*

Observación 2.1. *Se considera un string como un arreglo de caracteres.*

Observación 2.2. *De no haber correos, se asume $T = 0$ ya que no hay prefijos ni tampoco correos que mostrar.*

Observación 2.3. $(\forall 1 \leq i \leq N)$ *P_i es prefijo de $D_i \Rightarrow 1 \leq T$.*

Se planteó construir un *Trie* donde cada uno de sus elementos representa un caracter de una palabra y persiste la información indicada en la figura 1. Donde,

- ***cantidad_ocurrencias*** persiste la cantidad de veces que el string formado desde la raíz del *Trie* hasta el caracter actual es prefijo de las palabras ingresadas. Se inicializa en 0 y se incrementa cada vez que es recorrido al ingresar una nueva palabra.
- ***es_prefijo*** $= 1$ si el string formado desde la raíz hasta el caracter actual es uno de los prefijos P_i definidos en la introducción del problema. 0 en caso contrario.
- ***hijos*** simboliza la continuación del caracter actual para formar una palabra. Se inicializan todos en *NULL* y se les asigna una dirección válida a medida que tengan caracteres que le continúen en una palabra ingresada.

A modo de ejemplo, dada la entrada de la figura 2 se forma el *Trie* de la figura 3.

Observación 2.4. *Al sólo interesarse por el valor de cantidad_ocurrencias en los caracteres finales de cada prefijo basta ingresar al Trie hasta dicho caracter.*

Una vez ingresadas todas las palabras se devuelve el máximo valor *cantidad_ocurrencias* de los nodos con *es_prefijo* $= 1$.

```
estrucutra nodo:
    Nat cantidad_ocurrencias
    Bool es_prefijo
    nodo hijos[CANT_LETRAS]
```

Figura 1: Nodo Trie

```
correo: estados | longitud_prefijo:5
correo: ester | longitud_prefijo:4
correo: quito | longitud_prefijo:3
correo: esta | longitud_prefijo:3
```

Figura 2: Un ejemplo de input del problema

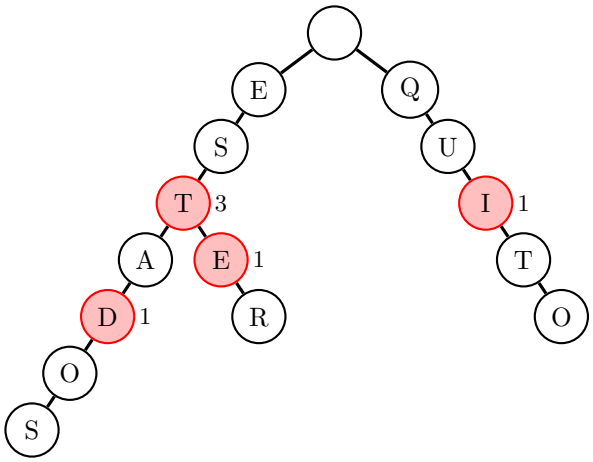


Figura 3: Representación gráfica del *Trie* para el caso en la figura 2. En rojo los nodos con *es_prefijo* = 1 junto a su valor de *cantidad_ocurrencias*. Los caracteres 'E' y 'Q' son hijos del caracter vacío (nodo Raíz).

2.3. Complejidad:

Para demostrar la complejidad se detallan los algoritmos más significativos de la solución.

Algorithm 3 inicializarNodo

```
1: procedure INICIALIZARNODO(nodo)
2:     nodo.cantidad_ocurrencias ← 0
3:     nodo.es_prefijo ← false
4:     for i : 0 → CANT_LETRAS do
5:         nodo.hijos_i ← NULL
6:     end for
7: end procedure
```

Se itera una cantidad acotada de veces ($CANT\_LETRAS$) una operación $\mathcal{O}(1)$, y se realizan otras operaciones $\mathcal{O}(1)$. Se deduce entonces que *inicializarNodo* tiene una complejidad temporal $\mathcal{O}(1)$.

Algorithm 4 posicionLetraEnHijo

```
1: procedure POSICIONLETRAENHIJO(caracter c)
2:     if ASCII(c) ≤ ASCII('Z') then
3:         pos ← ASCII(c) − ASCII('A')
4:     else
5:         pos ← 26 + ASCII(c) − ASCII('a')
6:     end if
7:     retornar pos
8: end procedure
```

El algoritmo se basa fuertemente en la definición de la tabla ASCII de C++ [1] y en que el valor de una variable de tipo *caracter* es su código en la misma. Se asume que obtenerlo es $\mathcal{O}(1)$. Por lo tanto se realizan todas operaciones $\mathcal{O}(1)$ y se deduce entonces que *posicionLetraEnHijo* tiene una complejidad temporal $\mathcal{O}(1)$.

[1] Ver tabla ASCII de C++ http://en.cppreference.com/w/cpp/language/ascii

Algorithm 5 agregarPalabra

```
1: procedure AGREGARPALABRA(string palabra, nat long_prefijo)
2:     actual ← Trie.raiz
3:     for i : 0 → long_prefijo do
4:         pos ← posicionLetraEnHijo( palabra_i )
5:         if actual.hijos_pos = NULL then
6:             inicializarNodo( actual.hijos_pos)
7:         end if
8:         actual ← actual.hijos_pos
9:         actual.cantidad_ocurrencias ← actual.cantidad_ocurrencias + 1
10:        if actual.es_prefijo then
11:            minT ← max( minT, actual.cantidad_ocurrencias)
12:        end if
13:    end for
14:    actual.es_prefijo ← True
15:    minT ← max( minT, actual.cantidad_ocurrencias)
16: end procedure
```

Donde

- *minT* es una variable global que persiste el máximo de los valores de *cantidad_ocurrencias* de cada nodo que cumpla *es_prefijo* = 1.
- *long_prefijo* es la longitud del prefijo del *string palabra* recordado por la cátedra.

Se itera *long_prefijo* veces realizando asignaciones en $\mathcal{O}(1)$ y llamados a funciones con costo temporal $\mathcal{O}(1)$. Se deduce entonces que *agregarPalabra* tiene una complejidad temporal $\mathcal{O}$(long_prefijo).

La solución propuesta realiza N llamados a *agregarPalabra* y luego se retorna el valor *minT*, donde N es la cantidad de palabras del input. El *i-ésimo* llamado tiene una complejidad temporal $\mathcal{O}(|P_i|) = \mathcal{O}(|D_i|)$ ya que todo prefijo tiene una longitud a lo sumo igual a la palabra de la cual proviene. Se deduce entonces que la solución propuesta tiene una complejidad temporal $\mathcal{O}(\sum_{i=1}^{N} |D_i|)$.

2.4. Correctitud:

Proposición 2.1. *Si P_i es prefijo de $P_j \wedge P_j$ es prefijo de $D_j \Rightarrow P_i$ es prefijo de D_j (La relación entre prefijos es transitiva).*

Una vez completado el *Trie*, el valor de *cantidad_ocurrencias* de cada caracter que indique ser el último de un prefijo (*es_prefijo* = 1) será por su definición la cantidad de veces que este sea prefijo de todas los otros prefijos ingresados. Más formal, dado el nodo V tal que el string formado desde la raíz del *Trie* hasta este sea P_i

$$V.cantidad\_ocurrencias = |\{P_j \,|\, (\forall 1 \leq j \leq N)\, P_i \text{ es prejifo de } P_j\}|$$

La variable global *minT* persiste el máximo de todos estos, por lo tanto

$$minT = max_{0 \leq i \leq N} \, |\{P_j \,|\, (\forall 1 \leq j \leq N)\, P_i \text{ es prejifo de } P_j\}|$$

Y por la proposición 2.1,

$$minT = max_{0 \leq i \leq N} \, |\{D_j \,|\, (\forall 1 \leq j \leq N)\, P_i \text{ es prejifo de } D_j\}|$$

Por lo tanto

$$(\forall 1 \leq i \leq N)\, |\{D_j \,|\, (\forall 1 \leq j \leq N)\, P_i \text{ es prejifo de } D_j\}| \leq minT$$

Entonces *minT* es exactamente el valor que se busca.

3. Problema C: Ciencia argentina

3.1. Introducción:

Para estar seguros de que los sueldos docentes son realmente adecuados, un grupo de docentes nos ha pedido que les ayudemos a decidir si la escala de sueldos es apropiada o no. Para ello, dada una tabla de sueldos de tamaño C x A, donde C es la cantidad de cargos distintos y A es la cantidad de niveles de antigüedad, necesitamos hacer Q consultas del estilo: dados el cargo c_1 y la antigüedad a_1 de un docente, y el cargo c_2 y la antigüedad a_2 a la que aspira a llegar para fin de año (asumiendo que se conserva la escala salarial). ¿Qué parte de su sueldo cobraría, si tuviera cargo c_2 y antigüedad a_2, pero no podría cobrar teniendo cargo c_1 (sin importar si consigue la antigüedad a_2) o antigüedad a_1 (sin importar si tiene el cargo c_2)?

Se nos pide una solución que responda las Q consultas con complejidad temporal $O(AC + Q)$. Para solucionar este problema, planteamos un algoritmo basado en programación dinámica, que construye una tabla aditiva, y luego, en base a la misma, obtiene los valores necesarios para calcular el resultado de cada una de las consultas.

3.2. Desarrollo:

El algoritmo que construimos es bastante simple, por lo que podemos dividirlo en dos partes bien distinguidas: por un lado la construcción de una tabla aditiva, y por el otro, el cálculo de las consultas solicitadas utilizando la tabla.

En primer lugar, veamos cómo es una tabla aditiva, y qué información contiene en el contexto de nuestro problema. Por simplicidad llamaremos s a la tabla de sueldos dada como parámetro y S a la tabla aditiva que construimos. Entonces, S es una tabla de tamaño C+1 x A+1 que esta definida como:

$$S_{i,j} = \sum_{a=0}^{i-1} \sum_{b=0}^{j-1} s_{a,b} \qquad 0 \leq i \leq C, \quad 0 \leq j \leq A$$

De esta manera, en la posición (i,j) de S se almacena la suma de los elementos contenidos en el rectángulo determinado por [0,i) x [0,j) de la tabla s, que en el contexto de nuestro problema representa el sueldo que cobra un docente que tiene cargo i-1 y antigüedad j-1.

Sin embargo, para construir la tabla, el algoritmo utiliza un método alternativo al recién explicado, pero que es equivalente (correctamente justificado en sección de correctitud). Este se basa en programación dinámica, de manera que utiliza resultados previamente calculados para obtener los nuevos. Entonces, el algoritmo que implementamos utiliza exactamente la siguiente definición:

$$S_{i,j} = \begin{cases} 0 & \text{si } i = 0 \vee j = 0 \\ s_{i-1,j-1} + S_{i-1,j} + S_{i,j-1} - S_{i-1,j-1} & \text{si } i > 0 \wedge j > 0 \end{cases}$$

Una vez construida S, resta resolver las Q consultas. Para ello, el algoritmo utiliza S. Cada query está constituida por dos cargos (c_1 y c_2) y dos antigüedades (a_1 y a_2). Utilizando estos datos como índices en S el algoritmo determina mediante la siguiente cuenta el resultado solicitado por la consulta:

$$Q(c_1, c_2, a_1, a_2) = S_{c_2,a_2} - S_{c_1,a_2} - S_{c_2,a_1} + S_{c_1,a_1}$$

De esta manera obtenemos los resultados de todas las consultas solicitadas.

3.3. Pseudocódigo:

Algorithm 6 construirTablaAditiva

```
1: procedure CONSTRUIRTABLAADITIVA(matriz sueldos, int c, int a)
2:     int tabla[c+1][a+1]
3:     for i : 0 → c do
4:         tabla[i][0] ← 0
5:     end for
6:     for i : 0 → a do
7:         tabla[0][i] ← 0
8:     end for
9:     for i : 0 → c-1 do
10:        for j : 0 → a-1 do
11:            tabla[i+1][j+1] ← sueldos[i][j] + tabla[i][j+1] + tabla[i+1][j] - tabla[i][j]
12:        end for
13:    end for
14:    retornar tabla
15: end procedure
```

Algorithm 7 calcularConsultas

```
1: procedure CALCULARCONSULTAS(matriz sueldos, int c, int a, int q, matriz queries)
2:     int tabla[c+1][a+1] ← construirTablaAditiva(sueldos, c, a)
3:     int resultados[q]
4:     for i : 0 → q-1 do
5:         int c_1 ← queries[i][0]
6:         int a_1 ← queries[i][1]
7:         int c_2 ← queries[i][2]
8:         int a_2 ← queries[i][3]
9:         int query_i ← tabla[c_2][a_2] - tabla[c_1][a_2] - tabla[c_2][a_1] + tabla[c_1][a_1];
10:        resultados[i] ← query_i
11:    end for
12:    retornar resultados
13: end procedure
```

3.4. Correctitud:

En esta sección explicaremos y justificaremos por qué el algoritmo que construimos resuelve el problema dado. La sección más compleja y menos intuitiva del algoritmo es la cuenta que este realiza para calcular la tabla aditiva y el resultado de cada consulta. Sin embargo, en el siguiente ejemplo podemos observar fácilmente qué representan dichos cálculos.

Tabla de sueldos

	0	1	2	3	4
0	1	2	3	4	5
1	6	7	8	9	10
2	11	12	13	14	15
3	16	17	18	19	20
4	21	22	23	24	25

Figura 4: Sueldo de docente con cargo 2 y antigüedad 3.

Un docente con cargo 2 y antigüedad 3 tiene un sueldo de

$$\sum_{i=0}^{1}\sum_{j=0}^{2} s_{i,j} = 1+2+3+6+7+8 = 27$$

(Como la matriz empieza desde cero restamos 1 al cargo y a la antigüedad en la sumatoria)

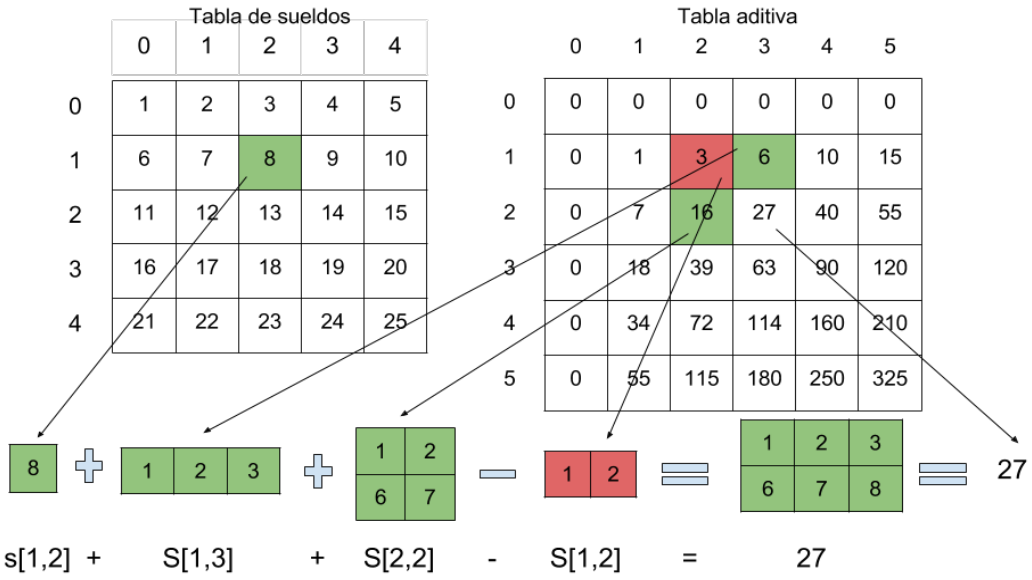


Figura 5: Cálculo de tabla aditiva mediante programación dinámica.

Aquí podemos observar gráficamente el cálculo de cada elemento de la tabla aditiva. Sin embargo, para ser más formales, también lo demostramos por inducción. Queremos demostrar que:

$$S_{i,j} = \sum_{a=0}^{i-1}\sum_{b=0}^{j-1} s_{a,b} = \begin{cases} 0 & \text{si } i = 0 \vee j = 0 \\ s_{i-1,j-1} + S_{i-1,j} + S_{i,j-1} - S_{i-1,j-1} & \text{si } i > 0 \wedge j > 0 \end{cases}$$

Entonces fijamos j tal que $j > 0$ ya que si $j = 0$ la demostración es trivial. Entonces demostraremos que:

$$\forall i \in N_0 \quad con\ j\ fijo\ (j > 0)$$

$$P(i) : \sum_{a=0}^{i-1}\sum_{b=0}^{j-1} s_{a,b} = \begin{cases} 0 & \text{si } i = 0 \vee j = 0 \\ s_{i-1,j-1} + \sum_{a=0}^{i-2}\sum_{b=0}^{j-1} s_{a,b} + \sum_{a=0}^{i-1}\sum_{b=0}^{j-2} s_{a,b} - \sum_{a=0}^{i-2}\sum_{b=0}^{j-2} s_{a,b} & \text{si } i > 0 \wedge j > 0 \end{cases}$$

Caso base) $i = 0$

$$\sum_{a=0}^{-1}\sum_{b=0}^{j-1} s_{a,b} = 0$$

Paso inductivo) Suponemos que P(i) es válido y queremos demostrar que P(i+1) también lo es. Es decir queremos ver que vale lo siguiente:

$$P(i{+}1) : \sum_{a=0}^{i}\sum_{b=0}^{j-1} s_{a,b} = \begin{cases} 0 & \text{si } i+1 = 0 \vee j = 0 \\ s_{i,j-1} + \sum_{a=0}^{i-1}\sum_{b=0}^{j-1} s_{a,b} + \sum_{a=0}^{i}\sum_{b=0}^{j-2} s_{a,b} - \sum_{a=0}^{i-1}\sum_{b=0}^{j-2} s_{a,b} & \text{si } i+1 > 0 \wedge j > 0 \end{cases}$$

Como supusimos j > 0 entonces podemos descartar el primer caso. Entonces debemos demostrar que la igualdad se cumple para el caso i+1 > 0 ∧ j > 0.

$$\sum_{a=0}^{i}\sum_{b=0}^{j-1} s_{a,b} = \sum_{a=0}^{i-1}\sum_{b=0}^{j-1} s_{a,b} + \sum_{b=0}^{j-1} s_{i,b}$$

Aplicamos la hipótesis inductiva:

$$= s_{i-1,j-1} + \sum_{a=0}^{i-2}\sum_{b=0}^{j-1} s_{a,b} + \sum_{a=0}^{i-1}\sum_{b=0}^{j-2} s_{a,b} - \sum_{a=0}^{i-2}\sum_{b=0}^{j-2} s_{a,b} + \sum_{b=0}^{j-1} s_{i,b}$$

Separamos un término de la última sumatoria:

$$= s_{i-1,j-1} + \sum_{a=0}^{i-2}\sum_{b=0}^{j-1} s_{a,b} + \sum_{a=0}^{i-1}\sum_{b=0}^{j-2} s_{a,b} - \sum_{a=0}^{i-2}\sum_{b=0}^{j-2} s_{a,b} + \sum_{b=0}^{j-2} s_{i,b} + s_{i,j-1}$$

Sumamos y restamos $\sum_{b=0}^{j-2} s_{i-1,b}$:

$$= s_{i-1,j-1} + \sum_{a=0}^{i-2}\sum_{b=0}^{j-1} s_{a,b} + \sum_{a=0}^{i-1}\sum_{b=0}^{j-2} s_{a,b} - \sum_{a=0}^{i-2}\sum_{b=0}^{j-2} s_{a,b} + \sum_{b=0}^{j-2} s_{i,b} + s_{i,j-1} + \sum_{b=0}^{j-2} s_{i-1,b} - \sum_{b=0}^{j-2} s_{i-1,b}$$

Reagrupamos términos y queda la igualdad buscada:

$$= s_{i,j-1} + \sum_{a=0}^{i-1}\sum_{b=0}^{j-1} s_{a,b} + \sum_{a=0}^{i}\sum_{b=0}^{j-2} s_{a,b} - \sum_{a=0}^{i-1}\sum_{b=0}^{j-2} s_{a,b}$$

$$\Big(\sum_{a=0}^{i-1}\sum_{b=0}^{j-1} s_{a,b} = s_{i-1,j-1} + \sum_{a=0}^{i-2}\sum_{b=0}^{j-1} s_{a,b} + \sum_{b=0}^{j-2} s_{i-1,b}\Big)$$

$$\Big(-\sum_{a=0}^{i-1}\sum_{b=0}^{j-2} s_{a,b} = -\sum_{a=0}^{i-2}\sum_{b=0}^{j-2} s_{a,b} - \sum_{b=0}^{j-2} s_{i-1,b}\Big)$$

La demostración fijando i y haciendo inducción sobre j es análoga. De esta manera queda demostrado que el cálculo que realizamos es equivalente a la cuenta que pide el enunciado.

Volviendo al ejemplo, si el docente que mencionamos al principio aspira a tener el cargo 4 y la antigüedad 4 a fin de año, entonces buscamos el resultado de la consulta Q(2, 4, 3, 4). Observemos el siguiente gráfico para entender por qué es correcto el cálculo de la consulta:

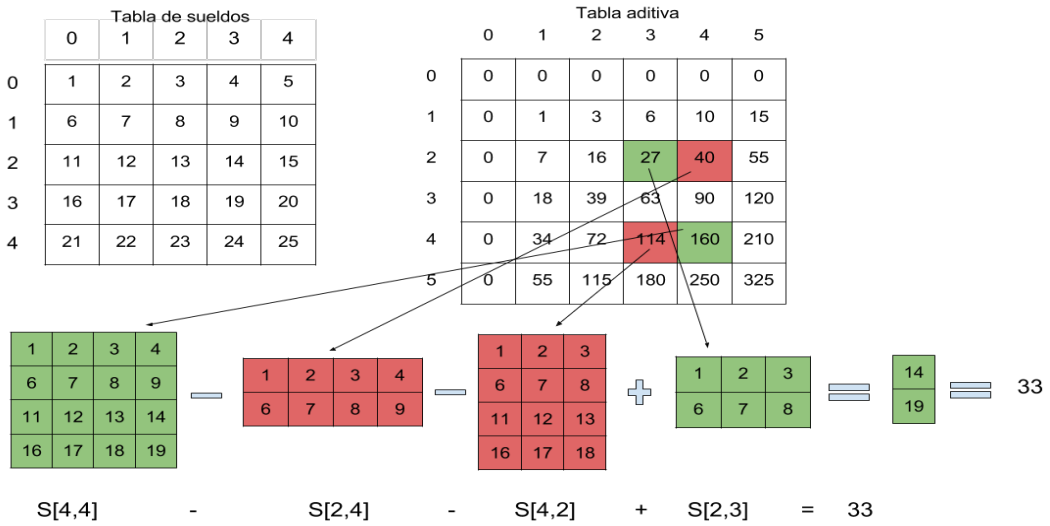


Figura 6: Cálculo de query mediante tabla aditiva.

De esta manera, el algoritmo propuesto, resuelve el problema dado.

3.5. Complejidad:

Veamos la complejidad de cada paso del algoritmo para luego calcular la complejidad total del mismo. En primer lugar, el algoritmo crea una matriz vacía de tamaño C+1 x A+1, donde almacenará la tabla aditiva. Comienza completando con ceros la primer columna y la primer fila, y el resto de las posiciones las calcula en función de otras previamente calculadas. Como se trata de operaciones aritméticas y accesos a estructuras en O(1), completar cada posición toma O(1). Luego el conjunto de todas las posiciones de la tabla toma O(C*A).

Por otro lado, el cálculo de cada query también toma O(1), ya que lo único necesario es acceder a 4 posiciones de la matriz y hacer una cuenta. Entonces para calcular todas las queries el algoritmo toma O(Q). Finalmente, la complejidad total es O(C*A+Q) como pide el enunciado.

4. Problema D:Diversión asegurada

4.1. Introducción:

El problema a resolver es el siguiente: dado una cantidad D de días, y el valor de diversión de cada día (valores no negativos), se nos presentan R intervalos de tiempo (en días) y debemos calcular cuánto nos vamos a divertir. Cuánto nos vamos a divertir en un rango es la suma de la diversión de los dos días con mayor valor de diversión (ambos dentro del rango). Se espera que el algoritmo tenga una complejidad temporal de $\mathbf{O}(D + R \log D)$

4.2. Desarrollo:

Decimos resolver el problema dado utilizando una estructura de datos para consultas en intervalos, un Segment Tree.

Asumimos que tenemos D días, con D potencia de 2. Si no lo fuera, podemos extender la cantidad de días para que lo fuese con a lo sumo D días extra (y un valor de diversión cualquiera para cada día extra). Sea D_i el valor de diversión del día i, con $0 \leq i < D$.

Vamos a tener un árbol binario (representado con un arreglo A, de manera que A_0 es la raíz y para cada i, los hijos de A_i son A_{2i+1} y A_{2i+2}), en el que las hojas contendrán el par $(D_i, 0)$, en ese mismo orden. Es decir, cada hoja i, $0 \leq i < D$ será cuanto nos vamos a divertir en el intervalo $[i,i+1)$ (sumando los elementos del par).

Entonces cada nodo interno (es decir, un nodo no hoja) va a contener cuanto nos vamos a divertir (sumando los elementos del par) en el intervalo resultado de la union de los intervalos de sus hijos. Entonces el nodo almacenará los dos elementos máximos entre los 4 elementos de sus hijos.

Luego, haremos las consultas pedidas sobre esta estructura.

4.3. Correctitud:

Veamos que elegir los dos elementos mayores entre dos pares de números ($\triangleright$), es un operador asociativo. Sean A,B,C pares de números. Asumamos sin perdida de generalidad que el primer elemento del par es mayor o igual al segundo. Queremos ver que:

$$\begin{aligned}(A \triangleright B) \triangleright C &=< max(a_1, b_1); max(a_2, b_2, min(a_1, b_1)) > \triangleright C \\ &=< max(max(a_1, b_1), c_1); max(max(a_2, b_2, min(a_1, b_1)), c_2, min(max(a_1, b_1), c_1)) > \\ &=< max(a_1, b_1, c_1); max(a_2, b_2, min(a_1, b_1), c_2, min(max(a_1, b_1), c_1)) >\end{aligned}$$

$$\begin{aligned}A \triangleright (B \triangleright C) &= A \triangleright < max(b_1, c_1); max(b_2, c_2, min(b_1, c_1)) > \\ &=< max(a_1, max(b_1, c_1)); max(a_2, max(b_2, c_2, min(b_1, c_1)), min(a_1, max(b_1, c_1))) > \\ &=< max(a_1, b_1, c_1); max(a_2, b_2, c_2, min(b_1, c_1), min(a_1, max(b_1, c_1))) >\end{aligned}$$

Entonces (A $\triangleright$ B) $\triangleright$ C = A $\triangleright$ (B $\triangleright$ C) $\iff$:

$max(a_2, b_2, min(a_1, b_1), c_2, min(max(a_1, b_1), c_1)) = max(a_2, b_2, c_2, min(b_1, c_1), min(a_1, max(b_1, c_1)))$
$\iff max(min(a_1, b_1), min(max(a_1, b_1), c_1)) = max(min(b_1, c_1), min(a_1, max(b_1, c_1)))$

Hay 3 casos:

- $a_1, b_1 \leq c_1$.
  $max(min(a_1, b_1), min(max(a_1, b_1), c_1)) = max(min(b_1, c_1), min(a_1, max(b_1, c_1)))$
  $\iff max(min(a_1, b_1), max(a_1, b_1)) = max(b_1, min(a_1, c_1))$
  $\iff max(a_1, b_1) = max(b_1, a_1)$ $\square$

- $a_1, c_1 \leq b_1$.

  $max(min(a_1, b_1), min(max(a_1, b_1), c_1)) = max(min(b_1, c_1), min(a_1, max(b_1, c_1)))$

  $\iff max(a_1, min(b_1, c_1)) = max(c_1, min(a_1, b_1))$

  $\iff max(a_1, c_1) = max(c_1, a_1)$ $\square$

- $b_1, c_1 \leq a_1$.

  $max(min(a_1, b_1), min(max(a_1, b_1), c_1)) = max(min(b_1, c_1), min(a_1, max(b_1, c_1)))$

  $\iff max(b_1, min(a_1, c_1)) = max(min(b_1, c_1), max(b_1, c_1))$

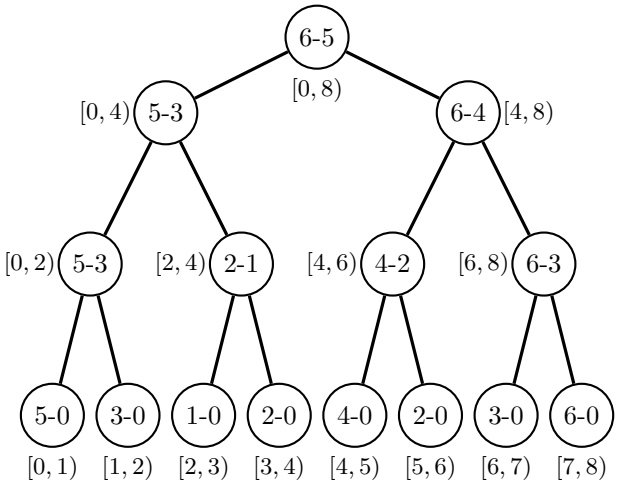
  $\iff max(b_1, c_1) = max(b_1, c_1)$ $\square$

Entonces $\triangleright$ es un operador asociativo.

Veamos ahora que las consultas en esta estructura de datos retorna lo pedido, es decir, cuánto va a ser la diversión en ese rango dado.

Como hallar los dos elementos mayores entre dos pares de números es una operación asociativa, por la forma en que construimos nuestro árbol, cada nodo interno va a contener cuánto nos vamos a divertir en el intervalo resultado de la unión de los intervalos de sus hijos.

Sea $D = 8$ y [5,3,1,2,4,2,3,6] el arreglo que nos dice en la posición i, cual es el valor de diversion del día i. El árbol formado es el siguiente:



Entonces, al realizar una consulta sobre el intervalo $[i, j)$ realizamos lo siguiente:

Empezamos con la raiz y haremos lo siguiente para cada nodo que visitaremos. Sea $[r, s)$ el intervalo que representa ese nodo:

- Si $[r, s)$ está totalmente dentro del intervalo $[i, j)$, es decir, ($\forall$ x $\in [r, s)$) x $\in [i, j)$, entonces devolvemos el par contenido en el nodo.
- Si $[r, s)$ no está dentro del intervalo $[i, j)$, es decir, ($\forall$ x $\in [r, s)$) x $\notin [i, j)$, entonces devolvemos el par (0,0). El par (0,0) es el elemento neutro del operador $\triangleright$. A $\triangleright$ (0,0) = (0,0) $\triangleright$ A = A
- Si Si $[r, s)$ está parcialmente dentro del intervalo $[i, j)$, es decir, ($\exists$ x,y $\in [r, s)$) x $\in [i, j)$ y $\notin [i, j)$, entonces devolvemos A $\triangleright$ B, donde A y B son los pares que resultan de aplicar esto mismo en los hijos izquierdo y derecho respectivamente.

Entonces lo que estamos haciendo es dividir el intervalo dado en intervalos disjuntos, y uniendo los resultados. Y por ser un $\triangleright$ un operador asociativo entonces estamos devolviendo el valor del intervalo pedido.

4.4. Pseudocódigo:

Algorithm 8 SegmentTree

```
1: procedure SegmentTree(Arreglo(entero) val)
2:     if |val| no es potencia de 2 then
3:         extiendo val con 0 hasta que lo sea
4:     end if
5:     arbol ← arreglo de tamaño 2 · |val| − 1
6:     n ← |val|
7:     for i : n − 1 → 2 · n − 2 do
8:         arbol[i] ← (val[i], 0)
9:     end for
10:    for i : n − 2 → 0 do
11:        arbol[i] ← arbol[2 · i + 1] ▷ arbol[2 · i + 2]
12:    end for
13: end procedure
```

Algorithm 9 query

```
1: procedure query(entero i, entero j)
2:     Par res ← funcRecursiva(0, 0, D, i, j)
3:     retornar res.first + res.second
4: end procedure
```

Donde D es la cantidad de días del intervalo.

Algorithm 10 funcRecursiva

```
1: procedure funcRecursiva(entero k, entero l, entero r, entero i, entero j)
2:     if i ≤ 1 ∧ r ≤ j then
3:         retonar arbol[k]
4:     else
5:         if r ≤ i ∨ j ≤ 1 then
6:             retonar (0, 0)
7:         else
8:             Par resIzq ← funcRecursiva(2 · k + 1, 1, (1 + r)/2, i, j)
9:             Par resDer ← funcRecursiva(2 · k + 2, 1, (1 + r)/2, i, j)
10:            retornar resIzq ▷ resDer
11:        end if
12:    end if
13: end procedure
```

4.5. Complejidad:

Sea D la cantidad de días y R la cantidad de intervalos. Analicemos primero el costo de la construcción de la estructura. Comenzamos extendiendo el arreglo dado para que sea potencia de 2, si es necesario. En el peor caso, necesitaremos duplicar el tamaño del arreglo. Esto tiene un costo $\mathbf{O}(D)$. Luego creamos el árbol y lo llenamos. Primero las hojas, con costo $\mathbf{O}(1)$ cada una. Como son D, tenemos costo $\mathbf{O}(D)$. Luego, como la operación $\triangleright$ es $\mathbf{O}(1)$ y visitamos cada nodo interno una única vez, completar el árbol es $\mathbf{O}$(cantidad de nodos del árbol) = $\mathbf{O}(2^*D) = \mathbf{O}(D)$. Entonces la construcción de la estructura es $\mathbf{O}(D)$.

Ahora, se analizara el costo de una consulta sobre un Segment Tree de tamaño D. Dada una consulta sobre un intervalo [i,j) valida, o sea $i \geq 0$, $i \leq j$ y $j \leq D$. Como se comienza desde el nodo raíz, este representa a todo el intervalo, en caso de que i sea igual a 0 y j igual a D, se devuelve simplemente el

valor que contenga el nodo raíz, siendo dicha consulta respondida en **O**(1). En caso de que el intervalo sea mas chico, lo cual es lo más probable, el raíz patea el problema a sus dos hijos. En los hijos de la raíz pueden ocurrir 3 casos

- **Caso 1**

  El nodo representa el intervalo [i,j), entonces procede a devolver los dos máximos de dicho intervalo en **O**(1), pues es el valor que tiene almacenado.

- **Caso 2**

  El nodo representa parte del intervalo [i,j), o sea, el intervalo que representa dicho nodo, contiene una parte del intervalo [i,j) y algunas cosas más. Entonces procede a patearle el problema a sus dos hijos.

- **Caso 3**

  El intervalo que representa el nodo no contiene ningún elemento del intervalo [i,j), entonces devuelve 0, un neutro de la operación $\triangleright$. Esta operación también cuesta **O**(1).

Como cada operación en cada nodo es **O**(1), la complejidad de la consulta depende de cuantos se vean afectados por la consulta y de por cuantos la misma siga bajando. Veamos que en cada consulta, la cantidad de nodos en cada nivel por los que baja la query son como mucho 2.

Supongamos que el algoritmo, para responder la query, baja por 3 nodos de un nivel k del Segment Tree. Sean estos 3 nodos, n_1, n_2 , n_3, dichos nodos son consecutivos, pues la consulta es sobre un intervalo. Esto significa que el nodo n_1 patea la query a los nodos n_{1_2} y n_{1_2}, el nodo n_2 a sus hijos n_{2_1} y n_{2_3} y por ultimo e igual que el resto, el nodo n_3 sus hijos n_{3_1} y n_{3_2}.

Entonces, esto implica que el intervalo representado por los nodos n_{2_1} y n_{2_2} estaba completamente dentro de la query [i,j) pues el nodo n_1 (a su izquierda) representaba un intervalo que contenía cosas del intervalo [i,j) (pues había pateado la query a sus hijos) al igual que el nodo n_3 (a su derecha). Por lo tanto la query nunca hubiera bajado por el nodo n_2 y entonces, el mismo nunca hubiera pateado la query a sus hijos. Se puede asegurar entonces, que la query solo sigue bajando como mucho por dos nodos de un nivel k del Segment Tree.

Por último, veamos cuantos nodos de un mismo nivel pueden ser afectados por la query. Como mostramos antes, solo dos nodos por nivel pueden patear la query para abajo, por ende solamente se pueden ver afectados 4 nodos por nivel, siendo estos nodos, los dos hijos de cada nodo que pateo la query para abajo.

Luego, como se mostró que solamente dos nodos patean la query para abajo, podemos asegurar que la complejidad de una consulta es igual a **O**(2*log(D)), que es igual a **O**(log(D)).

La complejidad final del algoritmo, en el peor caso (D no es potencia de 2) termina siendo **O**(D + Cantidad de querys * log(D)) o sea, **O**(D + R * log(D)).