IIC2133 - Pauta Examen 2020-1

Problema 1

Parte A

Podemos ver en el primer for que lo que hace el algoritmo es contar, en $INDEX[key - min]$, la cantidad de tuplas que tienen key como clave.

Luego, en el segundo for, va sumando hacia adelante estos contadores, de manera que $INDEX[key - min]$ tiene la cantidad de tuplas con clave menor o igual a key.

Por ultimo, el último for recorre el arreglo de atrás para adelante, y para cada elemento de clave key, mira cuantos elementos tienen clave menor o igual a él y lo pone en esa posición en el arreglo solución S, y disminuye ese contador en uno para poder posicionar correctamente el siguiente elemento con esa clave. Eso si esta disminución se hace antes de posicionar el elemento ya que los arreglos parten en 0.

Por lo tanto, si tenemos k elementos con claves menor a key, y m elementos de clave key, y estos van a quedar entre las posiciones k y $k + m$. Por lo tanto el arreglo S está ordenado.

[1.5 pts] Por identificar lo que hace el algoritmo en cada paso.
[0.5 pts] Por justificar por qué eso hace que ordene.

La pregunta pedía justificar. Si demuestran también se considera correcto.

Parte B

En primer lugar, encontrar el mínimo y el máximo elemento toma $O(n)$.

Luego recorremos A, que tiene n elementos. Como cada operación del for toma $O(1)$ ya que es acceso a un arreglo, este for aporta $O(n)$ a la complejidad.

El siguiente for recorre desde 1 hasta range - 1, y cada operación nuevamente es $O(1)$, por lo que ese for aporta $O(range)$.

El último for recorre A. Nuevamente, todas las operaciones adentro son $O(1)$, por lo que este for aporta $O(n)$ a la complejidad.

Por lo tanto, la complejidad total del algoritmo es

$$O(n) + O(n) + O(range) + O(n)$$

$$O(n + range)$$

El término $range$ no es una constante ya que depende de los datos, por lo que no lo podemos simplificar.

[1.5pt] Por calcular correctamente la complejidad.
[0.5pt] Por NO simplificar el $range$

Parte C

El puntaje se recibe por identificar diferencias específicas de la siguiente lista. Se eligen las dos mejores.

A diferencia de los algoritmos vistos en clases:

- ◇ **[1pt]** Este algoritmo no compara los elementos: todos los algoritmos que vimos ordenaban mediante comparación de elementos (se puede ver incluso que el algoritmo no tiene ningun if)
- ◇ **[1pt]** La complejidad de este algoritmo depende de sus datos, además del n. Los algoritmos que vimos en clases podían depender del orden en que venían los datos (QuickSort, InsertionSort), pero jamás de qué números se estaban ordenando.
- ◇ **[1pt]** Este algoritmo es lineal, asumiendo que el *range* no depende de n. Los algoritmos más rapidos que estudiamos en clases ordenaban en tiempo $O(nlog(n))$.
- ◇ **[0.5pt]** Este algoritmo solo funciona si las claves son números enteros positivos. Esto recibe menos puntaje ya que es algo que viene en el enunciado.
- ◇ **[0.5pt]** (Si analizaron el algoritmo como estaba originalmente) Este algoritmo ordena los datos de mayor a menor. Recibe menos puntaje ya que es una diferencia trivial y es fácilmente solucionable en $O(n)$.

Problema 2

Lo que se pide es, básicamente, encontrar un *bosque de cobertura mínimo*. Es decir, un conjunto de $|S|$ árboles que cubren el grafo, tal que cada árbol contiene exactamente un nodo de S, y el costo total de las aristas de todos los árboles es mínimo.

Este problema se puede resolver de varias maneras, y son todas equivalentes:

Opción 1

Conectamos todos los nodos de S a un nuevo nodo $s^\star$ mediante aristas de costo 0, y luego aplicamos el algoritmo de Prim o Kruskal tal como se vieron en clases.

Teniendo el MST del grafo, eliminamos $s^\star$ y las aristas que lo conectan con los nodos de S para separar el árbol en sub-árboles, formando así el bosque que se pide.

Agregar y quitar este nodo y sus aristas no agrega complejidad a los algoritmos, por lo que están dentro de la complejidad esperada.

Opción 2

En lugar de agregar un nodo extra y aristas de costo cero, podemos modificar uno de los algoritmos para poder manejar este caso:

Opción 2.1: Prim

El algoritmo de Prim parte poniendo de un nodo "de partida" en el heap y genera el árbol a partir de ahi, marcando cuales nodos ya fueron "incluidos". Como para este problema debemos generar un árbol por cada nodo de S, simplemente hay que partir desde todos al mismo tiempo y construir los árboles del bosque mínimo de manera simultánea. Para eso basta con partir poniendo **todos** los nodos de S en el heap con prioridad mínima, para que todos partan siendo incluidos.

Estos nodos eventualmente iban a ser agregados al heap: forzar que se agreguen al comienzo no modifica la complejidad del algoritmo.

Opción 2.2: Kruskal

El algoritmo de Kruskal parte diciendo que cada nodo del grafo es su propio conjunto, y va agregando aristas hasta que todos los nodos son parte del mismo conjunto. Sabemos que los nodos de S no pueden pertenecer al mismo sub-arbol, asique sus conjuntos jamás deberian unirse. Para esto, al principio del algoritmo, podemos unir todos los nodos de S en un mismo conjunto: así jamás se eligiran aristas que los conecten.

Estos nodos eventualmente iban a ser unidos al conjunto final, forzar su union al comienzo no modifica la complejidad del algoritmo.

[6pts] Por proponer un algoritmo correcto y de complejidad correcta
[4pts] Si el algoritmo es correcto pero la complejidad no lo es
[1pts] Por calcular correctamente la complejidad, pero el resto está mal.
[1pts] Por identificar las características de la solución (*bosque de cobertura mínimo*), pero no resuelve el problema.

Problema 3

Parte A

OPCION 1

La cantidad de nodos negros en la rama más corta debe ser la misma que en la más larga (por propiedad 4 de ARN vista en clases). Llamemos a esta cantidad "n" [**0,5 por incorporar propiedad y usar cantidad de nodos negros para comparar ambas ramas**]. La rama más corta posible tendrá todos los nodos negros (será de n nodos) [**0,5 mínimo de nodos de rama más corta**].

Ahora, la rama más larga posible será de $n + r$ nodos, donde r es la cantidad máxima de nodos rojos que puede tener.

Para analizar el máximo r, consideremos la propiedad 3 vista en clases, con la que sabemos que si un nodo es rojo, sus hijos deben ser negros (hojas nulas se consideran negros). Si consideramos además que la raíz es negra, sabemos que la cantidad de nodos rojos será menor a la de negros (por cada rojo en una rama estará su hijo negro ($(n-1) >= r$), y además está la raíz) [Si nulos no se cuentan como negros, la cantidad de rojos puede ser a lo más la de negros ($(n-1) >= r$)] [**0,5 máximo de nodos de rama más larga considerando propiedad 3**].

Es decir, la rama más larga posible tendrá $n + r$ nodos donde $r = n$ [$r = n - 1$]. Entonces tendrá $2n$ [$2n - 2$] nodos, el doble de los n nodos de la más corta posible- Entonces, podemos decir que la rama más larga de un ARN tendrá a lo más el doble de nodos que la más corta [**0,5 por enunciar conclusión correctamente, sólo si tenía bien los pasos anteriores**]

OPCION 2

Usamos análisis del árbol 2-3-4. Notamos que en él, todas las ramas tienen la misma altura h [**0.5**]. Además, como los nodos de tipo 2 corresponden a 1 nodo negro en ARN (un nivel), los de tipo 3 a 1 negro con un hijo rojo en ARN (dos niveles), y los de tipo 4 a 1 negro con dos hijos rojos en ARN (dos niveles), vemos que la rama más corta posible en el ARN tendrá solo nodos tipo 2 en el 2-3-4, y se convertirá en el ARN en una rama con altura h [**0.5 altura más corta**]. Por otro lado, la más larga posible estará constituida por nodos tipo 3 o 4, y esto se traducirá en ARN en una rama de altura $2h$ [**0.5 altura más larga**]. Así, vemos que la rama más larga posible tendrá 2 veces la altura de la más corta posible. Es decir, la más larga tendrá a lo más el doble de nodos de la más corta [**0.5**].

Parte B

i

Porque para que un árbol sea RN, debe cumplir con la propiedad 4. Si insertamos una hoja negra a un árbol que ya es RN y queda con más de una rama, siempre romperemos la propiedad (excepto para la raíz) [**0.6 por mencionar que propiedad se rompe siempre o siempre que no sea la raíz**]. Esto se explica ya que, si antes tenía "n" nodos negros en cada rama, ahora tendrá "n+1" en una de ellas, y "n" en las demás. Así, al insertar el nodo como negro la propiedad se rompe siempre, mientras que si insertamos el nodo como rojo, romperá la propiedad 3 solo a veces. [**0.2 por hacer distinción por frecuencia**] Además, restaurar la propiedad 4 será en general más costoso, ya que incide globalmente en el árbol, mientras que la propiedad 3 es más fácil de arreglar con cambios locales [**0.2 por hacer distinción por complejidad**]

ii

Deberíamos restaurar la propiedad 4 (sin violar las otras, cuidando sobre todo la propiedad 3) **[0,2]**. Si quisieramos priorizar mantener el nodo insertado como negro, esto lo podríamos conseguir con un mecanismo que parta localmente con rotaciones y cambios de color que se complejizaría mucho, o con un mecanismo que revise todo el árbol globalmente **[0.3]**.

Si pensamos en el árbol 2-3-4 equivalente, lo que estamos haciendo es insertar un nodo como tipo 2, aún cuando el nivel padre no se haya llenado. Esto, además de no asegurar que se cumplan las propiedades, no es lo más eficiente (y el árbol dejaría de tener la propiedad de 2-3-4 de todas las ramas a la misma profundidad, y se ropmería el mecanismo de split. Por lo tanto, pensemos en los casos en que se puede mantener el nodo insertado m como negro y aún así respetar la forma equivalente de 2-3-4.

(a) Se inserta bajo un nodo n tipo 2 (En ARN, sería un nodo negro sin hijos). Acá podemos forzar el nodo a subir, y podemos elegir dejarlo como el nodo negro (y n pasa a ser rojo). En ARN, n sería un nodo negro sin hijos. Lo que se haría es una rotación simple con el insertado, y cambio de color.

(b) Se inserta bajo un nodo n tipo 3, que contiene a p y q (En ARN, sería un nodo negro p con 1 hijo rojo q). Nuevamente, forzamos el nodo a subir. Si entra por el medio ($p < m < q$ ó $p > m > q$) (m entra como hijo interno de q; hacia el lado de p), podemos dejarlo como nodo negro con p y q como hijos. En caso contrario, sería muy costoso intentar mantener a m negro, y conviene simplemente pintarlo rojo.

(c) Se inserta bajo un nodo n tipo 4, que contiene a p, q y r (En ARN, sería un nodo negro p con 2 hijos rojos, q y r). Nuevamente, forzamos a m a subir. Si entra por el medio ($r < m < q$ ó $r > m > q$) (m entra como hijo interno de r o q), podemos dejarlo como nodo negro de hijo tipo 3 generado al subir a p mediante split. En caso contrario, es similar a (b) y se pinta rojo.

[0,5 Por mencionar solo uno de los casos, o ejemplo (con condiciones declaradas), en que nodo insertado quede definitivamente negro. Alternativamente, se acepta un mecanismo global que revise el árbol completo y deje un ARN que cumpla las propiedades]

Un ejemplo de mecanismo global sería insertar como en AVL, luego contar la altura h, y colorear desde las hojas hacia arriba, nivel por medio, hasta que cada rama tenga h nodos negros.

Parte C

OPCIÓN 1

En clases se usa la equivalencia del árbol Rojo Negro (ARN) con un árbol 2-3-4 (A) para la inserción **[0.2]**. En el árbol A, los nodos de tipo 2 corresponden a 1 nodo negro en ARN, los de tipo 3 a 1 negro y 1 rojo en ARN, y los de tipo 4 a 1 negro y 2 rojos en ARN [0.3 Puede estar a lo largo de la explicación, deben estar los tres]. Entonces, solo necesitamos saber que en un árbol 2-3-4 con más de un elemento, siempre habrá al menos un nodo tipo 3 o 4. **[0.3]**

Al insertar el primer nodo, (A) queda con un único nodo tipo 2. Al insertar el 2do (n=2 es caso base con n¿1), (A) queda con un único nodo tipo 3 y la condición se cumple **[0.2 Caso base]**. En adelante al ir insertando nodos que aumenten n, la única forma en que un nodo deje de ser tipo 3 es que pase a ser tipo 4 al recibir una nueva llave (seguiría cumpliendo la condición) **[0.4]**, y la única forma en que deje de ser tipo 4 es que ocurra un split al recibir una nueva llave, pero esto siempre deja como hijos dos nodos en A, uno tipo 2 y otro tipo 3 (seguiría cumpliendo la condición) **[0.4]**.
Entonces, como con n=2 se cumple la condición y al ir insertando nodos en RN (elementos en (A)) se sigue cumpliendo, podemos decir que siempre se cumple para n¿1 y que, para n¿1, un ARN formado con el método visto en clases siempre tiene al menos un nodo rojo **[0.5 si enuncia conclusión, solo si justificación es correcta]**

OPCIÓN 2

CB: n= 2, Raíz negra con un hijo rojo. $R = 1 > 0$ **[0.2 Necesario solo si en cada caso se argumenta que se mantiene o aumenta número de rojos. Si se usa argumento de que cada caso deja algún rojo, se suman los 0,2 a 0.5 de conclusión]**

Hay cinco casos de inserción:

(a) C1: Raíz. Se da solo con n = 1, no aplica a condición [Puede omitirse]

(b) C2: Padre negro. Nodo se inserta rojo y queda rojo. Rojos aumentan [$R > 0$] **[0.2 por presentar caso y explicar bien que R se mantiene positivo]**

(c) C3: Padre rojo y tío rojo. Se mantiene número de nodos rojos. [Rojos resultantes entre abuelo, tíos e insertado = $2 > 0$] [0.3 por presentar caso y explicar bien que R se mantiene positivo]

(d) C4: Padre rojo, tío negro, es hijo interno (su valor está entre su padre y su abuelo). Lleva a C5 aumentando nivel previo a inserción de rojos en uno, y como después de C5 siempre se mantiene el número de rojos (considerando el insertado), el caso cumple. [0.3 por presentar C4 y C5 y explicar bien que R se mantiene positivo. Se pueden presentar juntos].

(e) C5: Padre rojo, tío negro, es hijo externo (su valor no está entre padre y abuelo]. Rojos entre abuelo, tíos e insertado pasa de 1 a 2 [> 0].

Como en n=2 hay un nodo rojo y para todos los casos posibles de inserción con $n > 1$ se mantiene o aumenta R [Como siempre al insertar un nodo con $n > 1$ se asegura $R > 0$], podemos decir que, para $n > 1$, un ARN formado con el método visto en clases siempre tiene al menos un nodo rojo [0.5 por llegar a conclusión correcta en base a todos los casos bien justificados (solo si están bien los pasos anteriores)]

Problema 4

Primero, llamaremos J al conjunto de jarros. Definimos la funcion $V(J, x)$ que es el costo mínimo de sumar x con los jarros de J.

Parte A

En primer lugar, por definición, si queremos sumar un número x y hay un jarro de ese volumen, entonces el costo es 1 ya que es llegar y hacer $\infty \to x$. Es decir:

$$V(J, x) = \begin{cases} 1, & x \in J \\ \dots \end{cases}$$

[1pt] por el caso base.

Luego, los números de la forma $v - v_j$ se pueden construir haciendo $v_i \to v_j$, donde v_i es el jarro que contiene el volumen v. El costo de esto será (el costo de construir v) + 1. Llamaremos a este costo $T(J, x)$

$$T(J, x) = \begin{cases} V(J, v) + 1, & \exists\ v_j \in J \mid x = v - v_j \\ \dots \end{cases}$$

Ya que no sabemos cual es el más barato, debemos revisar cada uno. Considerando que $v = x + v_j$:

$$T(J, x) = \begin{cases} \min\limits_{v_j\ \in\ J} [V(J, x + v_j) + 1] \\ \dots \end{cases}$$

Cambiando v_j de nombre a j para que sea más legible:

$$T(J, x) = \begin{cases} \min\limits_{j\ \in\ J} [V(J, x + j) + 1] \\ \dots \end{cases}$$

[1pt] Por establecer el caso del traspaso de un jarro a otro.

Ojo que así como está, nada impide que se llame a $V(J, x + j)$ con números cada vez más grandes y que la recursión jamás termine. Y lo cierto es que para poder hacer el traspaso de un jarro a otro, este volumen tiene estar contenido en algun jarro. Eso significa que no sirve sumar volumenes más grandes que el jarro más grande (llamémoslo $J_{\max}$) para luego traspasarlo a otro jarro, así que lo dejamos fuera del MIN. Si el MIN queda vacío, es decir, x más el mínimo de J ya supera al máximo, entonces retornamos infinito.

$$T(J, x) = \begin{cases} \infty, & x + J_{\min} > J_{\max} \\ \min\limits_{\substack{j\ \in\ J \\ x+j\ \leq\ J_{\max}}} [V(J, x + j) + 1], & else \end{cases}$$

Además, todos los números mayores a 1 pueden ser expresados como la suma de otros dos números, por lo que para los números que no cumplan con ninguna de las dos anteriores, debemos descomponerlos y buscar la forma más barata de sumarlos. Llamaremos a esto $S(J, x)$

$$S(J,x) = \begin{cases} \infty, & x = 1 \\ \min\limits_{y \in [1,\cdots,\lfloor \frac{x}{2} \rfloor]} [V(J,y) + V(J,x-y)], & else \end{cases}$$

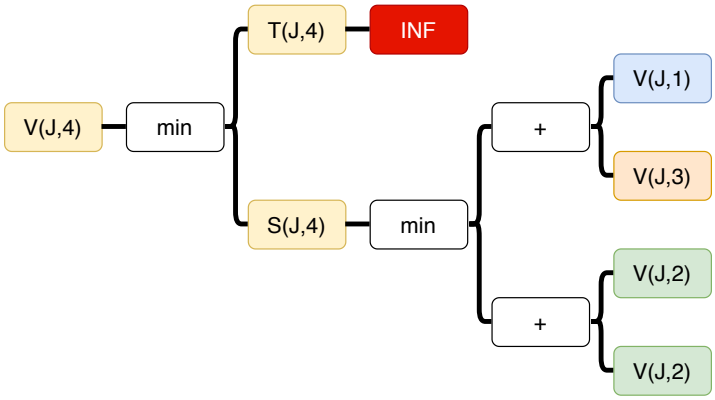
[1pt] Por establecer el caso de la descomposición

Uniendo todo, la manera más barata de crear un volumen va a ser o la más barata traspasando, o la más barata sumando.

$$V(J,x) = \begin{cases} 1, & x \in J \\ \min\left[T(J,x), S(J,x)\right], & else \end{cases}$$

Parte B

Tomemos el ejemplo del enunciado: $J = \{5, 7\}$, $x = 4$. Haremos un diagrama para mostrar la recursión.

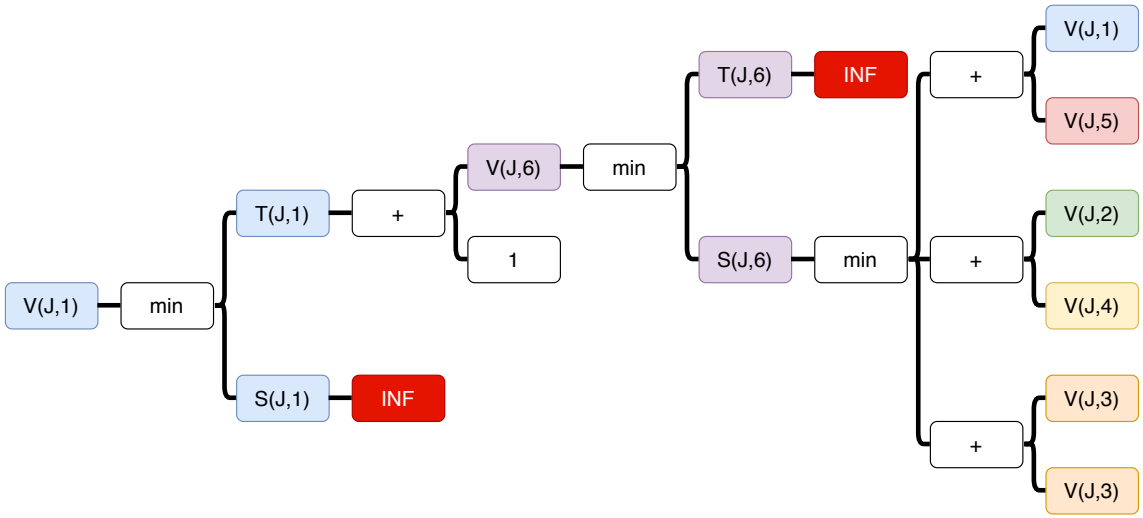


Podemos ver que ya en la primera llamada se nos repite el $V(J, 2)$: para que calcularlo dos veces si podemos calcularlo una vez y guardar su resultado? Para esto sirve programación dinámica: para no tener que calcular dos veces un resultado que ya conocemos.

[1pt] Por indicar casos en que se repiten las llamadas a la funcion con los mismos parámetros.
[1pt] Por explicarlo mostrando un diagrama (se pedía explicítamente en el enunciado)
[1pt] Por explicar por qué es util usar programación dinámica en este caso.

Lo siguiente se incluye solo para que la respuesta esté completa.

Veamos qué pasa en $V(J,1)$:



Podemos ver que DENTRO de la llamada a $V(J,1)$ hay nuevamente una llamada a $V(J,1)$. Esto significa que el algoritmo jamás va a terminar. Considerando que no tiene sentido argumentar circularmente el mínimo costo de de sumar un número, entonces deberíamos dejar fuera esos casos, al igual que el $V(J,4)$ (recordemos que seguimos dentro de la llamada original a $V(J,4)$).

Para solucionar esto podemos agregar un "contexto" a la ecuación de recurrencia, para ignorar llamadas a $V(J,x)$ si ya estamos dentro de una llamada a $V(J,x)$.

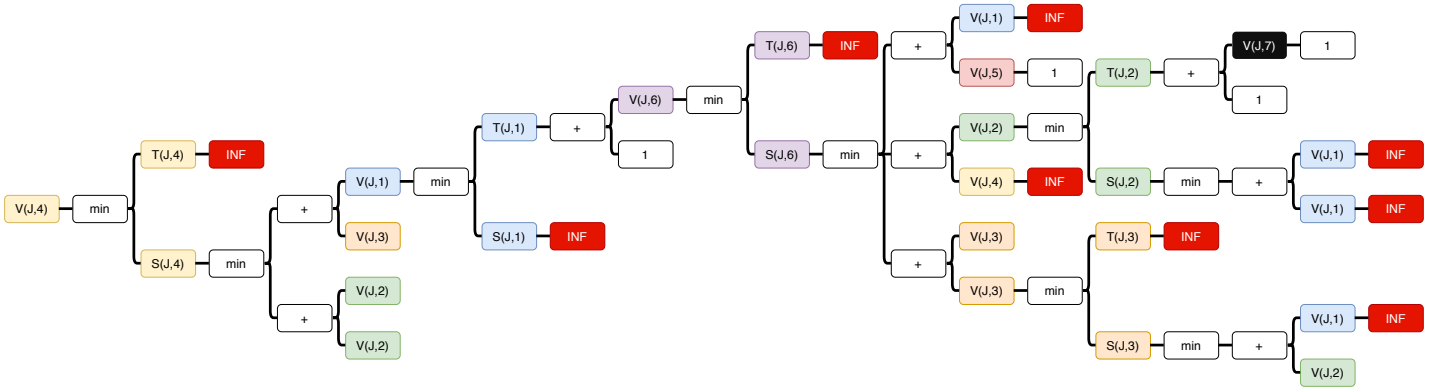
Esto lo podemos hacer con un conjunto C que indica todos los x dentro de los cuales estamos, partiendo por el x original, en nuestro caso $= 4$.

$$V(J,C,x) = \begin{cases} \infty, & x \in C \\ 1, & x \in J \\ \min\left[T(J,C,x), S(J,C,x)\right], & else \end{cases}$$

$$T(J,C,x) = \begin{cases} \infty, & x + J_{\min} > J_{\max} \\ \min\limits_{\substack{j \in J \\ x+j \leq J_{\max}}} [V(J, C \cup \{x+j\}, x+j) + 1], & else \end{cases}$$

$$S(J,C,x) = \begin{cases} \infty, & x = 1 \\ \min\limits_{y \in [1,\cdots,\lfloor \frac{x}{2} \rfloor]} [V(J, C \cup \{y\}, y) + V(J, C \cup \{x-y\}, x-y)], & else \end{cases}$$

Con contexto y programación dinámica el árbol de llamadas queda como sigue (se obvia el contexto en los parámetros).



Así, $V(J, 4) = 4$

Nota del ayudante: elegí el ejemplo del enunciado por simplicidad, no se me ocurrió que fuera el caso más complejo para ese J :c