

Parcial 1:

1. (25 puntos) Considera el problema de determinar la cantidad de apariciones de un número entero x en un arreglo de números enteros $A[0..N]$. Suponiendo que el arreglo está ordenado ascendenteamente, diseña un algoritmo recurrente que resuelve el problema dado en tiempo $O(\log N)$ y espacio $O(1)$. Es necesario que parta de una especificación y presente, como resultado final, una demostración de la correctitud del algoritmo y de su eficiencia.

Entrada: Un arreglo $A[0..N]$ de numeros enteros ordenados ascendenteamente con $N \geq 0$ y un numero x .

Salida: Cantidad de apariciones de x en A .

Descripción:

Para este algoritmo se utilizarán 2 funciones recurrentes \emptyset y Ψ cuyas descripciones son las siguientes:

Función \emptyset :

Entrada: igual que la entrada del algoritmo principal

Salida: índice de la primera aparición de x en A .

Descripción:

La función \emptyset se basa en el algoritmo de búsqueda binaria con algunas modificaciones. De esta manera se tendrán dos testigos low y hi $0 \leq low \leq hi \leq N$ que permitirán establecer límites dentro del arreglo y obtener el espacio de búsqueda activo. Dado esto la definición recursiva de \emptyset es:

(En la definición, mid es el punto medio entre low y hi)

$\emptyset(low, hi, x) :$ {

- inf (infinito)
- inf
- low
- mid $\downarrow \emptyset(low, mid, x)$
- $\emptyset(mid, hi, x)$
- $\emptyset(low, mid, x)$

$low = hi$
 $low + 1 = hi \wedge A[low] \neq x$
 $low + 1 = hi \wedge A[low] = x$
 $low + 1 < hi \wedge A[mid] = x$
 $low + 1 < hi \wedge A[mid] < x$
 $low + 1 < hi \wedge A[mid] > x$

(si se retorna inf de \emptyset , quiere decir que x no está en A)

Replantamiento de la salida:

Basado en esta definición, el valor objetivo de salida es $\emptyset(0, N, x)$.

Función Ψ :

Entrada: igual que la entrada del algoritmo principal

Salida: índice de la última aparición de x en A .

Descripción:

La función Ψ se basa en el algoritmo de búsqueda binaria con algunas modificaciones. De esta manera se tendrán dos testigos low y hi $0 \leq low \leq hi \leq N$ que permitirán establecer límites dentro del arreglo y obtener el espacio de búsqueda activo. Dado esto la definición recursiva de Ψ es:

(En la definición, mid es el punto medio entre low y hi)

$\Psi(low, hi, x) :$ {

- -1
- -1
- low
- mid $\uparrow \Psi(mid, hi, x)$
- $\Psi(mid, hi, x)$
- $\Psi(low, mid, x)$

$low = hi$
 $low + 1 = hi \wedge A[low] \neq x$
 $low + 1 = hi \wedge A[low] = x$
 $low + 1 < hi \wedge A[mid] = x$
 $low + 1 < hi \wedge A[mid] < x$
 $low + 1 < hi \wedge A[mid] > x$

(Si se retorna -1 de Ψ , es porque x no está en A)
 Replanteamiento de la salida:
 Basandose en la definicion de Ψ , el problema a resolver es $\Psi(0, N, x)$.

Teniendo en cuenta ambas definiciones, se puede plantear una solucion para el problema original. Inicialmente se parte de que si $x \in A$, entonces todas las ocurrencias de x estan contiguas dentro de A (esto se probra en la demostracion de correctitud). De esta manera si se encuentran los indices de la primera y ultima ocurrencia de x es posible encontrar el numero de x 's en A , al restar ambos limites, lo cual permite replantejar la salida:

Replanteamiento de la salida:

Si $x \in A$ entonces $\Psi(0, N, x) = \phi(0, N, x) + 1$, de lo contrario, 0.

A continuacion se muestra una implementacion en python:

```
def phi(low, hi, x, A):
    ans = None
    if(low == hi): ans = float('inf')
    elif(low + 1 == hi):
        if(A[low] == x): ans = low
        else: ans = float('inf')
    else:
        mid = low+((hi-low)>>1)
        if(A[mid] == x): ans = min(mid, phi(low, mid, x, A))
        elif(A[mid] < x): ans = phi(mid, hi, x, A)
        else: ans = phi(low, mid, x, A)
    return ans

def psi(low, hi, x, A):
    ans = None
    if(low == hi): ans = -1
    elif(low + 1 == hi):
        if(A[low] == x): ans = low
        else: ans = -1
    else:
        mid = low+((hi-low)>>1)
        if(A[mid] == x): ans = max(mid, psi(mid, hi, x, A))
        elif(A[mid] < x): ans = psi(mid, hi, x, A)
        else: ans = psi(low, mid, x, A)
    return ans

def solve(x, A):
    ans = None
    low = phi(0, len(A), x, A)
    hi = psi(0, len(A), x, A)
    if(low == float('inf') and hi == -1): ans = 0
    else: ans = hi + 1 - low
    return ans
```

Complejidad temporal:

El tiempo de este algoritmo se compone del tiempo de ambas funciones utilizadas en su construccion mas un par de operaciones constantes:

$$T(n) = T_\phi(n) + T_\psi(n) + O(1)$$

Ahora bien El tiempo para las funciones ϕ y ψ son:

$$T_\phi(n) = T_\phi(n/2) + O(1)$$

$$T_\psi(n) = T_\psi(n/2) + O(1)$$

Utilizando el teorema maestro se tiene que tanto para T_ϕ como T_ψ , $a=1$, $b=2$ y $K=0$, por lo cual la complejidad de ambos algoritmos es $\mathcal{O}(n^0 \log(n)) = \mathcal{O}(\log(n))$. Si se reemplaza en la formula original se tiene que:

$$\begin{aligned} T(n) &\in \mathcal{O}(\log(n)) + \mathcal{O}(\log(n)) + \mathcal{O}(1) \\ T(n) &\in \mathcal{O}(\log(n) + \log(n) + 1) \quad (\text{def } \mathcal{O} \text{ grande}) \\ T(n) &\in \mathcal{O}(\log(n) + \log(n) + 1) \quad (\text{Teorema 2.3.5 notas clase}) \\ T(n) &\in \mathcal{O}(\log(n)) \quad (\text{definición máximo}) \quad \square \end{aligned}$$

Complejidad espacial:

Como se ha visto en la descripción del algoritmo, el almacenamiento es constante ya que solo se utilizan índices que recorren el arreglo original, por lo cual la complejidad espacial claramente pertenece a $\mathcal{O}(1)$. \square

Correctitud del algoritmo:

Este algoritmo se basa en 3 principios:

Teorema 1: Si $x \in A$ entonces todas las ocurrencias de x son contiguas dentro del arreglo.

Teorema 2: ϕ es correcto (ϕ calcula la primera ocurrencia de x en A)

Teorema 3: ψ es correcto (ψ calcula la última ocurrencia de x en A)

Basado estos 3 teoremas es posible concluir que el algoritmo soluciona el problema correctamente, baso lo que se especifica en el replanteamiento de la salida, por lo cual se procede a su demostración:

Prueba teorema 1:

Supongamos por contradicción que $x \in A$ y las ocurrencias de x no están contiguas, lo cual quiere decir que existe un $y \neq x$ tal que:

$$A = [\dots, x, y, x, \dots]$$

Si $y > x$ o $y < x$ sería una contradicción ya que A debe estar organizado ascendentemente. De esta forma se concluye que las ocurrencias de x deben ser contiguas dentro de A . \square

Prueba teorema 2:

Se demuestra por inducción sobre la distancia entre low y hi :

Caso base ($low=hi$): En un espacio de búsqueda vacío no hay elementos, por lo cual $x \notin A$, lo que se representa con el índice inf.

Caso base ($low+1=hi$): Si el espacio de búsqueda solo tiene un elemento, si x es igual al elemento entonces es la menor ocurrencia de x en A , por lo cual se retorna el índice low . En caso que x es distinto al elemento entonces no está en A y se retorna inf.

Caso inductivo ($low+1 < hi$):

Hipótesis inductiva: $\phi(low', hi', x)$ con $low \leq low' < hi' \leq hi$ (básicamente un subespacio de búsqueda) encuentra el índice de la primera ocurrencia de x en A .

En todos los casos inductivos se tiene el índice mid tal que i) $low < mid < hi$ y ii) mid es un índice del arreglo:

Demostración i): 1) $\text{low} + 1 < \text{hi}$ (def caso inductivo)

$$\text{low} < \text{low} + 1 < \text{hi} \quad (\alpha = b \Rightarrow \alpha < b + 1)$$

$\text{low} < \text{hi}$ (transitiva)

$\underline{\text{low}} + 1 < \underline{\text{hi}}$ (ley uniformidad)

$\underline{\text{low}} + 1 < \text{hi}$ (prop desigualdades)

2) $\text{low} \leq \text{low}$ (prop desigualdades)

$\underline{\text{low}} + \text{hi} \leq \underline{\text{low}} + \text{hi}$ (ley uniformidad)

$2\text{low} < \underline{\text{low}} + \text{hi} \leq \underline{\text{low}} + \text{hi}$ ($\text{low} + 1 < \text{hi} \wedge c < d \Rightarrow a + c < a + d$)

$2\text{low} < \underline{\text{low}} + \text{hi}$ (transitiva)

$\text{low} < \underline{\text{low}} + \text{hi}$ (prop desigualdades)

$\text{low} < \underline{\frac{\text{low} + \text{hi}}{2}} < \text{hi}$ (1 y 2)

$\text{low} < \text{mid} < \text{hi}$ (def mid) \square

Demostración ii):

Por definición de low y hi se tiene que $0 \leq \text{low} \leq \text{hi} \leq N$. Como se está en el caso $\text{low} + 1 < \text{hi}$ se tendría que:

$$0 \leq \text{low} < \text{hi} \leq N$$

$$0 \leq \text{low} < \text{mid} < \text{hi} \leq N \quad (\text{por i})$$

Por lo cual mid es un índice del arreglo. \square

De esta manera $\emptyset(\text{low}, \text{mid}, x)$ y $\emptyset(\text{mid}, \text{hi}, x)$ son subproblemas de \emptyset y pueden ser resueltos por H.I. Ahora se procederá a demostrar los subcasos dentro del caso inductivo $\text{low} + 1 < \text{hi}$:

Caso $A[\text{mid}] = x$:

Si se encuentra una ocurrencia de x, entonces el menor índice puede ser mid o la menor ocurrencia de x a la izquierda es decir, $\emptyset(\text{low}, \text{mid}, x)$. De esta manera se llega a $\text{mid} \downarrow \emptyset(\text{low}, \text{mid}, x)$ lo cual resolvería el caso.

Caso $A[\text{mid}] < x$:

Como A está ordenado ascendenteamente entonces no hay ocurrencias de x en $A[\text{low}, \text{mid}]$, por lo cual se busca el índice en $A[\text{mid}, \text{hi}]$, es decir, $\emptyset(\text{mid}, \text{hi}, x)$.

Caso $A[\text{mid}] > x$:

Como A está ordenado ascendenteamente entonces no hay ocurrencias de x en $A[\text{mid}, \text{hi}]$, por lo cual se busca el índice en $A[\text{low}, \text{mid}]$, es decir, $\emptyset(\text{low}, \text{mid}, x)$.

De esta manera se prueban los subcasos del caso inductivo y la correctitud de \emptyset . \square

Prueba teorema 3:.

Se demuestra por inducción sobre la distancia entre low y hi:

(Nota: al ser \emptyset y Ψ funciones similares se hace referencia constantemente a la demostración de \emptyset)

Caso base ($low=hi$): Análogo a demostración en Ø solo que se retorna -1 en vez de inf.

Caso base ($low+1=hi$): Análogo a demostración en Ø solo que si $A[low] \neq x$ entonces se retorna -1 en vez de inf.

Caso inductivo ($low+1 < hi$):

Hipótesis inductiva: $\Psi(low', hi', x)$ con $low \leq low' < hi' \leq hi$ (binariamente un subespacio de búsqueda) encuentre el índice de la última ocurrencia de x en A .

Por i) y ii) de demostración Ø se sabe que por H.I., se cumple $\Psi(low, mid, x)$ y $\Psi(mid, hi, x)$. Se procede entonces a probar los subcasos del caso inductivo:

Caso ($A[mid]=x$):

Si se encuentra una ocurrencia de x , entonces el mayor índice puede ser mid o la mayor ocurrencia de x a la derecha es decir, $\Psi(mid, hi, x)$. De esta manera se llega a $mid \uparrow \Psi(mid, hi, x)$ lo cual resuelve el caso.

Caso ($A[mid] < x$):

Análogo a demostración Ø

Caso ($A[mid] > x$):

Análogo a demostración Ø

De esta manera se prueban los subcasos del caso inductivo y la correctitud de Ψ . □

De esta manera se resuelve la correctitud del algoritmo original. □

Nota: Las demostraciones de correctitud de Ψ y Ø fueron inspiradas en la prueba de correctitud de binary search de las notas de clase.

3. Con base en el enunciado del problema *Inventory Problem* (Ejercicio 6.26, página 333 de *Algorithm Design* por J. Kleinberg y E. Tardos):

(a) (10 puntos) Plantee una función recurrente que permita resolver el problema, enunciando claramente la descripción de la función, restricciones sobre sus parámetros y cómo se puede usar para resolver el problema dado. Es necesario partir de una especificación del problema.

(b) (15 puntos) Diseñe un algoritmo de tabulación con programación dinámica que implemente la función objetivo. Si es posible reducir el espacio de la tabulación, es necesario reducirlo para obtener crédito completo en esta parte.

a) **Entrada:** Un arreglo $D[0..N]$ con $N \geq 1$ y enteros $S, C, K \geq 0$.

Salida: Dadas las condiciones de costo (estipuladas en el problema), se decide como hacer los órdenes de compra en cada mes para minimizar el costo y cumplir la demanda.

Descripción:

Para este algoritmo se utilizará una función recurrente Ø con las siguientes especificaciones:

Función Ø:

Entrada: Valores n y d tal que $0 \leq n \leq N$ y $0 \leq d \leq S$.

Salida: Costo mínimo de satisfacer la demanda del mes n al mes N empezando con d camiones en bodega.

Descripción:

La idea es decidir por cada mes n y su demanda respectiva $D[n]$, registrar el costo mínimo que hay del mes n al

mes N. Para esto se utilizarán 2 parámetros los cuales son el mes n y la cantidad de camiones d que están en bodega. Inicialmente este el caso base que sería el último mes ($n=N-1$), en el cual dependiendo de lo que haya en bodega se calcula el costo: si d es menor a la demanda, se deben comprar camiones para completar lo que costaría K. De lo contrario no es necesario comprar camiones y se debe pagar el costo de almacenamiento de los camiones que quedaron al suplir la demanda (siendo c el costo de almacenamiento por camión, esto sería $(d - D[n]) \cdot c$). Los casos inductivos serían los meses entre el primero y el penúltimo. En caso que en dicho mes $d < D[n]$, necesariamente se deben comprar camiones y se deberá pagar el costo de los camiones que sobren. La idea es comprar un número de camiones x tal que minimicen el costo a futuro. De esta manera el costo en dicho mes sería K más lo que queda en bodega por c. Finalmente si $d \geq D[n]$ entonces existe la posibilidad de continuar con los camiones que quedan al restar la demanda (lo cual tendría un costo $c \cdot (d - D[n])$) o Comprar x nuevos camiones pensando que a futuro podría beneficiarme, lo cual tendría un costo de $K + (d - D[n] + x) \cdot c$. Esta descripción se presenta a continuación en la definición de ϕ :

$$\phi(n, d) = \begin{cases} \begin{aligned} & \bullet K \\ & \bullet (d - D[n]) \cdot c \end{aligned} & \bullet n = N-1 \wedge d < D[n] \\ \left(\begin{aligned} & \downarrow x \mid 0 \leq x \leq S - d + D[n] \wedge x + d \geq D[n] : \\ & K + (x + d - D[n]) \cdot c + \phi(n+1, x + d - D[n]) \end{aligned} \right) & \bullet n = N-1 \wedge d \geq D[n] \\ \left(\begin{aligned} & \downarrow x \mid 0 \leq x \leq S - d + D[n] : \text{ si } x = 0 : (d - D[n]) \cdot c + \phi(n+1, d - D[n]) \\ & \text{ si } x \neq 0 : K + (x + d - D[n]) \cdot c + \phi(n+1, x + d - D[n]) \end{aligned} \right) & \bullet n \neq N-1 \wedge d \geq D[n] \end{cases}$$

Usando esta función es posible obtener la función objetivo de ψ :

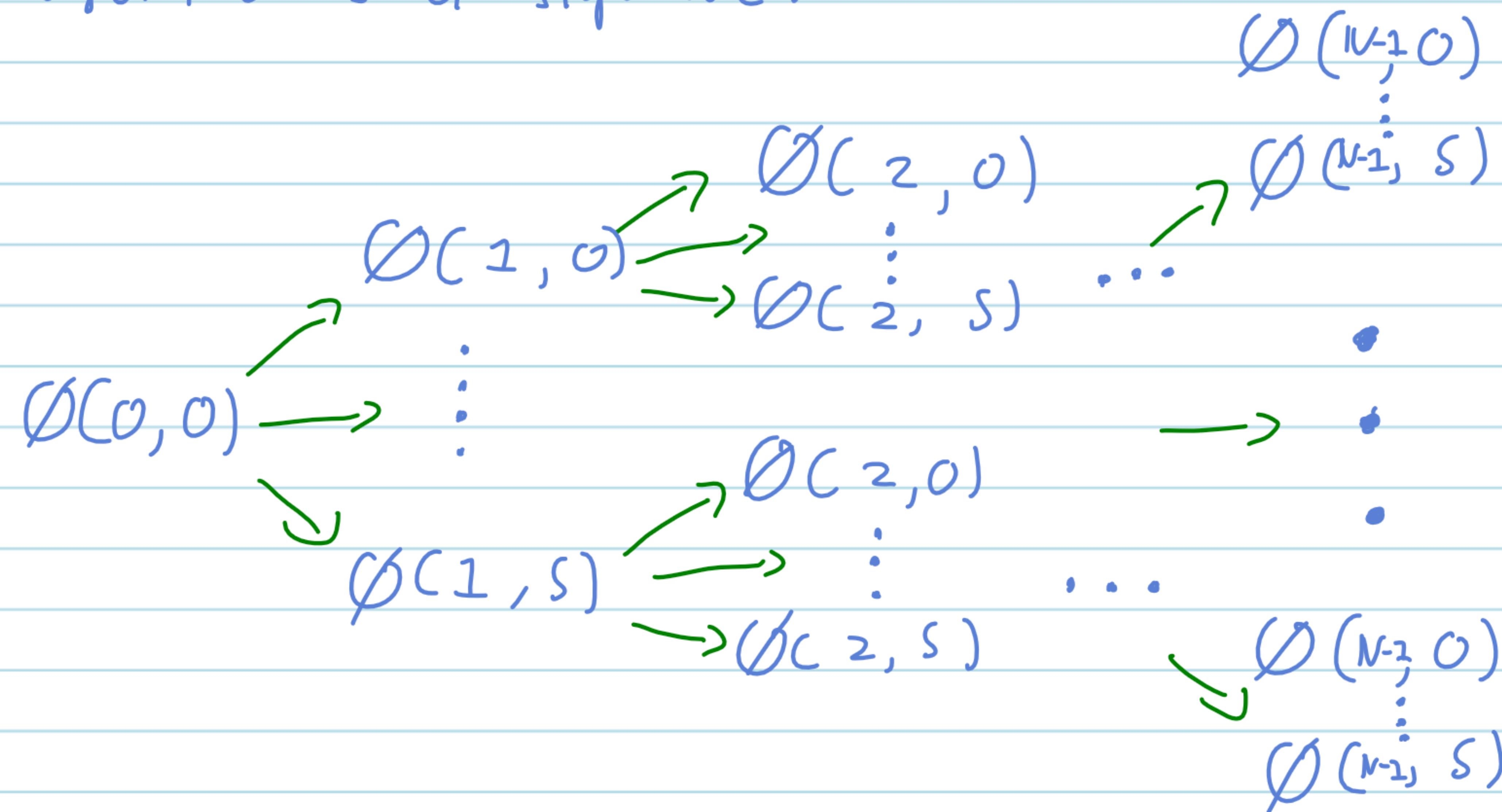
Replanteamiento de la salida:

Se espera obtener el mínimo costo del mes 0 al mes N-1, lo cual se obtendría con $\phi(0, 0)$ ($d=0$ ya que se empieza sin camiones en la bodega).

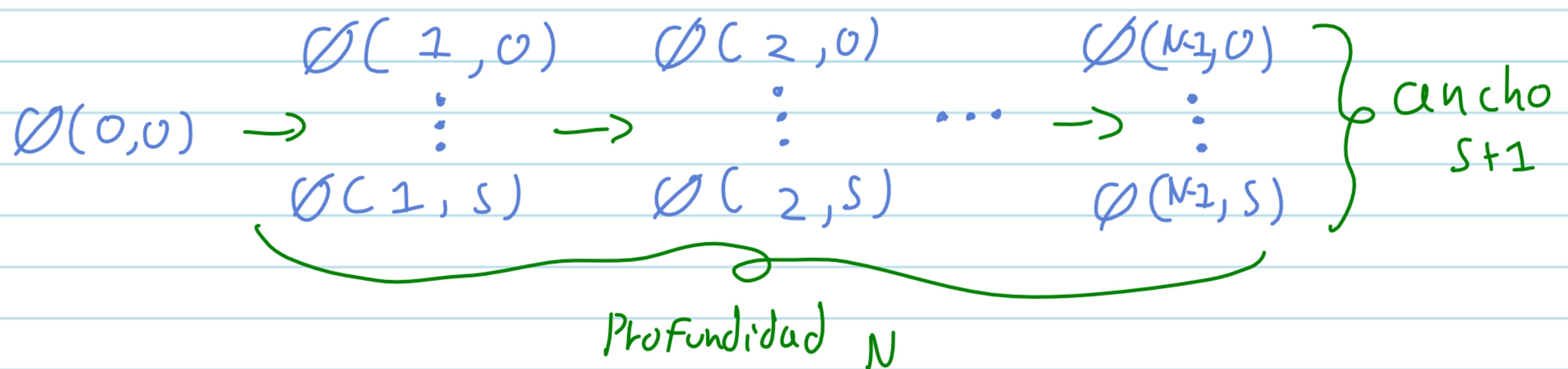
De esta manera siguiendo con la descripción del problema original, para obtener la cantidad de camiones que se deben comprar cada mes para minimizar costos en los N meses, se puede rescatar dicho valor de la función ϕ . En el primer caso base se obtiene el valor al restar $D[n] - d$, mientras que en el segundo caso base la cantidad de camiones sería 0. En cuanto a los casos inductivos, la cantidad de camiones que se deben comprar es la

Esto dada por el x que minimiza el costo. De esta manera se podrían obtener las decisiones de compra para cada mes y solucionar el problema.

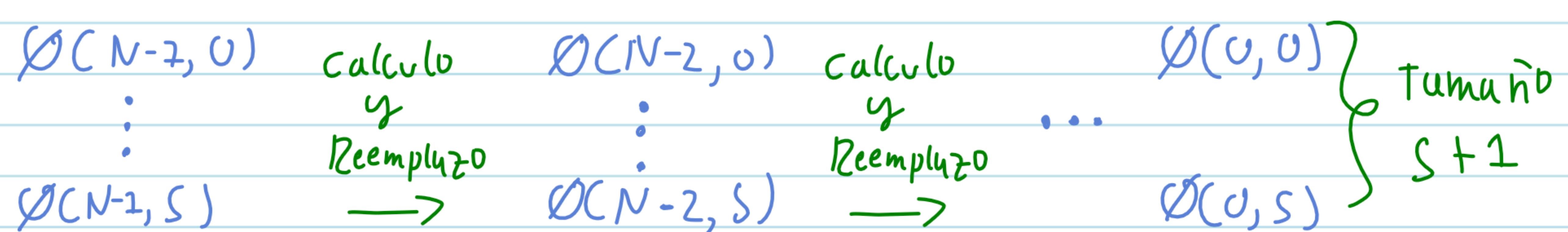
b) Inicialmente se tiene que el árbol de ejecución del algoritmo es el siguiente:



Al utilizar tabulación se reduce el árbol al precalcular instancias solo para:



Esta tabulación exigiría entonces memoria en el orden de $O(N \cdot s)$ como lo indica la figura. Sin embargo, los valores de $\Phi(n, d)$, solo dependen del siguiente valor que sería $n+1$, por lo cual se puede reducir la complejidad espacial, al ir reemplazando en la tabulación por los nuevos valores de las nuevas instancias:



Al final, la complejidad espacial de este algoritmo sería del orden de $O(s)$.

De esta manera para resolver el problema original se creará un arreglo Tab con $s+1$ posiciones. La idea es almacenar en cada posición una tupla (\min, x) . De este modo si estoy en la posición d tal que $0 \leq d \leq s$, entonces $\text{Tab}[d] = (\min, x)$ donde $\min = \phi(n, d)$ (el n depende en que iteración de

llenado en la cual este) y x el pedido que se hizo en el mes h. Al final por cada iteración de n se busca el mayor en un arreglo Answers[0...N) donde estarán almacenados la cantidad de pedidos por cada mes, basándose en el valor mínimo de Tab (es decir el valor mínimo de cada tupla en la primera posición, para cada iteración del mes h). De esta manera se presenta la siguiente implementación, que encapsula este principio:

Implementación φ con tabulación:

```
def phi(n, a, Tab):
    global D, K, C, N, S
    ans = None
    if(n == N - 1 and a < D[n]): ans = (K, D[n] - a)
    elif(n == N - 1 and a >= D[n]): ans = ((a - D[n]) * C, 0)
    elif(n != N - 1 and a < D[n]):
        num = None
        aux = None
        for x in range(D[n] - a, (S - a + D[n]) + 1):
            if(aux == None):
                aux = K + (x + a - D[n]) * C + Tab[x + a - D[n]][0]
                num = x
            else:
                aux = min(aux, K + (x + a - D[n]) * C + Tab[x + a - D[n]][0])
                if(aux == K + (x + a - D[n]) * C + Tab[x + a - D[n]][0]): num = x
        ans = (aux, num)
    elif(n != N - 1 and a >= D[n]):
        num = None
        aux = None
        for x in range((S - a + D[n]) + 1):
            if(aux == None and x == 0):
                aux = (a - D[n]) * C + Tab[a - D[n]][0]
                num = x
            else:
                aux = min(aux, K + (x + a - D[n]) * C + Tab[x + a - D[n]][0])
                if(aux == K + (x + a - D[n]) * C + Tab[x + a - D[n]][0]): num = x
        ans = (aux, num)
    return ans
```

Implementación algoritmo:

```
def solve(D, N, K, C, S):
    Tab = [None for _ in range(S+1)]
    answers = [None for _ in range(N)]
    for i in range(N - 1, -1, -1):
        ans = None
        x = None
        for j in range(S + 1):
            Tab[j] = phi(i, j, Tab)
            if(ans == None):
                ans = Tab[j][0]
                x = Tab[j][1]
            else:
                ans = min(ans, Tab[j][0])
                if(ans == Tab[j][0]): x = Tab[j][1]
        answers[i] = x
    return answers
```

2. Con base en el enunciado del problema *Additions and Subtractions* (Ejercicio 3.33, página 141 de *Algorithms* por J. Erickson) y la metodología propuesta en el curso para resolver problemas usando la técnica de programación dinámica:
 - (15 puntos) Plantee una función recurrente que permita resolver el problema, enunciando claramente la descripción de la función, restricciones sobre sus parámetros y cómo se puede usar para resolver el problema dado (es necesario partir de una especificación del problema).
 - (10 puntos) Determine si es más conveniente utilizar memorización o tabulación, justificando su respuesta, y diseñe un algoritmo coherencia con su decisión y justificación.

a) Entrada: Un arreglo $A[0\dots 2N-1]$ de números enteros y simbolos $\{+, -\}$ con $N \geq 1$
 Salida: Valor máximo posible de la expresión dada en A, al hacer agrupaciones.

Descripción:

Para este algoritmo se utilizará una función recurrente φ con las siguientes especificaciones:

} Bonus

Inicialmente hay que especificar que $A[0..2N-1]$ representa una expresión matemática con sumas y restas, por lo cual hay N números enteros y $N-1$ simbolos de $+$, $-$. Ahora bien, como es una expresión matemática, en las posiciones pares estarían los números y en las impares los simbolos. Teniendo esto en cuenta es posible analizar las casas bases e inductivas. El caso base es cuando hay un solo número ($N=1$), por lo cual trivialmente el máximo valor de la expresión sería el número ($A[n]$). El caso inductivo se da cuando hay más de un número ($N > 1$). La idea es evaluar las distintas maneras de agrupar la expresión, las cuales serían en principio las agrupaciones generadas por cada símbolo de la expresión, e.g. las agrupaciones iniciales de $1+2-3$ por cada símbolo serían $(1)+(2-3)$, $(1+2)-(3)$. De esta manera se tendrían 2 nuevas expresiones que surgen de agrupar con respecto a un símbolo, e.g. si k es la posición del símbolo entonces las expresiones resultantes son $A[0..k] \pm A[k+1..2N-1]$. De esta manera buscando la máxima expresión dentro de ambos subintervalos, se podría encontrar el valor de la máxima expresión al buscar el máximo entre las posibles agrupaciones por cada símbolo. A continuación se presenta dicha función ϕ :

Función ϕ :

Entrada: índices low y hi con $0 \leq low < hi \leq 2N-1$, low es par y hi impar
Salida: valor máximo posible de la expresión en $A[low, hi]$, al hacer agrupaciones.

Definición:

$$\phi(low, hi) = \begin{cases} A[low] & .(\uparrow k | low \leq k \leq hi \wedge k \text{ es impar} : \phi(low, k) + \phi(k+1, hi)) \quad low+1 \leq hi \\ & \text{el } + \text{ o } - \text{ depende de } A: \\ & \text{si } A[k] = '+' \Rightarrow + \\ & \text{si } A[k] = '-' \Rightarrow - \end{cases}$$

Basado en esta definición se puede hacer un replanteamiento de la salida del algoritmo original:

Replanteamiento de la salida:

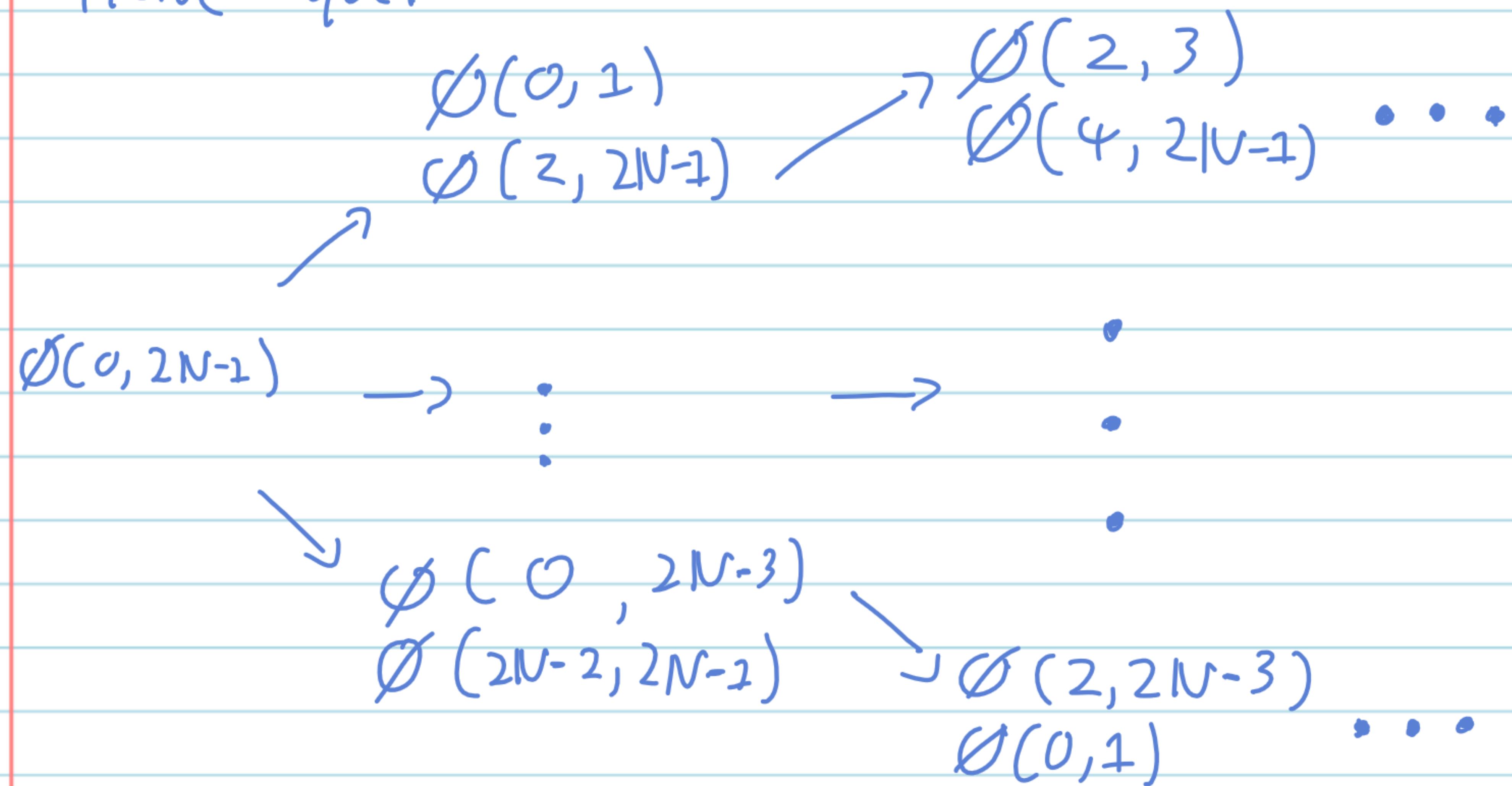
La función objetivo es $\phi(0, 2N-1)$.

De esta forma con la función ϕ se resuelve el problema. A continuación una implementación del algoritmo:

```
def phi(low, hi, A):
    ans = None
    if(low + 1 == hi): ans = A[low]
    else:
        for i in range(low + 1, hi):
            if(i % 2 != 0):
                if(ans == None):
                    if(A[i] == '+'): ans = phi(low, i, A) + phi(i + 1, hi, A)
                    else: ans = phi(low, i, A) - phi(i + 1, hi, A)
                else:
                    if(A[i] == '+'): ans = max(ans, phi(low, i, A) + phi(i + 1, hi, A))
                    else: ans = max(ans, phi(low, i, A) - phi(i + 1, hi, A))
    return ans
```

```
def solve(A):
    ans = None
    ans = phi(0, len(A), A)
    return ans
```

b) Si se mira el arbol de ejecucion del algoritmo se tiene que:



Claramente existen instancias de Ø que se solucionan por lo cual utilizar programación dinámica es beneficioso. En cuanto a la ejecución de tabulación o memorización, es necesario analizar las distintas instancias de Ø que se pueden producir. Utilizando la implementación de Ø, se puede obtener una lista de los distintos llamados según el N:

$N=1$

$[(0, 1)]$

$N=3$

$[(0, 3), (0, 1), (2, 3)]$

$N=5$

$[(0, 5), (0, 1), (0, 3), (2, 5), (2, 3), (4, 5)]$

$N=7$

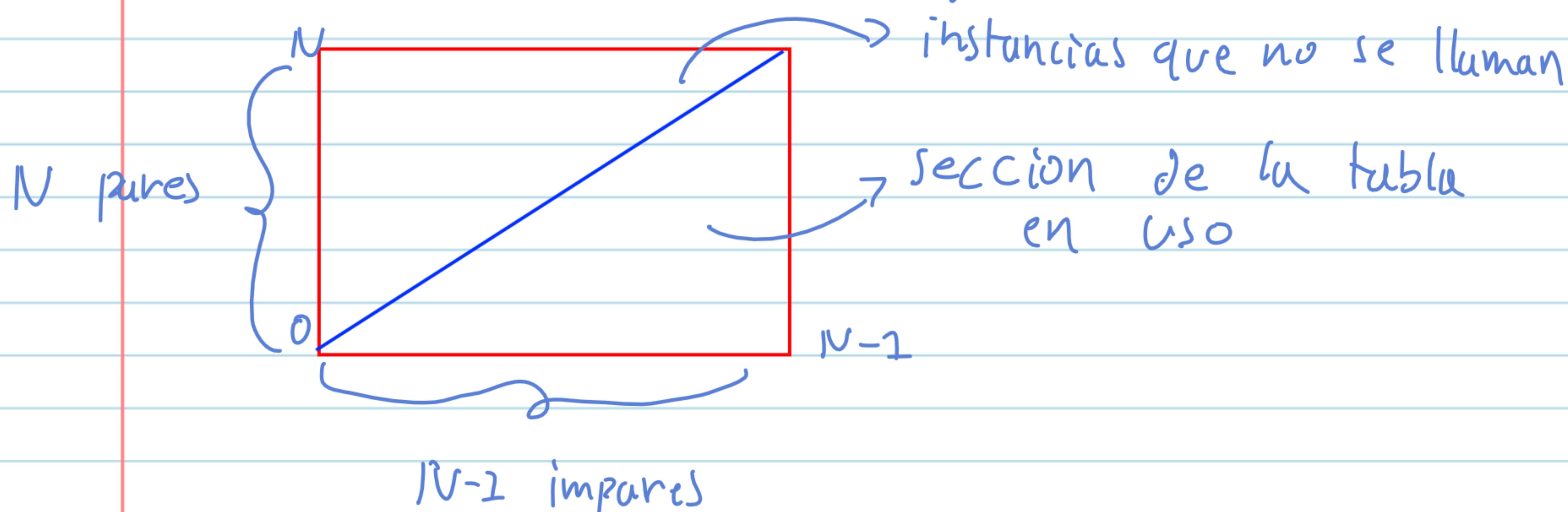
$[(0, 7), (0, 1), (0, 3), (0, 5), (2, 7), (2, 3), (2, 5), (4, 7), (4, 5), (6, 7)]$

(la primera posición de las tuplas es luw y la segunda hi)

Analizando esto y el arbol de ejecucion se observa un patrón en los llamados:

- El llamado con el primer par ocurre con todos los impares, el llamado con el segundo par ocurre con todos los impares menos el primero, el llamado con el tercer par ocurre con todos los impares menos el primero y el segundo...

De esta manera si se tabulara con respecto a los posibles valores de luw (pares) y hi (impares) la sección de la tabla que se llena sería la siguiente:



sentido

Por este motivo tabular no tendría sentido ya se estaria desperdiizando tiempo y memoria al calcular y almacenar instancias que nunca serán accedidas, por lo cual es mejor utilizar memorización.

A continuación se presenta el algoritmo utilizando memorización:

```
def phi(low, hi, A, mem):
    ans = None
    if((low, hi) in mem):
        ans = mem[(low,hi)]
    else:
        if(low + 1 == hi):
            ans = A[low]
            mem[(low,hi)] = ans
        else:
            for i in range(low + 1, hi):
                if(i % 2 != 0):
                    if(ans == None):
                        if(A[i] == '+'):
                            ans = phi(low, i, A, mem) + phi(i + 1, hi, A, mem)
                            mem[(low,hi)] = ans
                        else:
                            ans = phi(low, i, A, mem) - phi(i + 1, hi, A, mem)
                            mem[(low,hi)] = ans
                    else:
                        if(A[i] == '+'):
                            ans = max(ans, phi(low, i, A, mem) + phi(i + 1, hi, A, mem))
                            mem[(low,hi)] = ans
                        else:
                            ans = max(ans, phi(low, i, A, mem) - phi(i + 1, hi, A, mem))
                            mem[(low,hi)] = ans
    return ans
```

```
def solve(A):
    ans = None
    mem = {}
    ans = phi(0, len(A), A, mem)
    return ans
```