El CERN, además de su centro de procesamiento local que procesa petabytes de datos generados por los experimentos del LHC, utiliza la Grid de Computación Mundial WLCG, una red global distribuida de centros de datos (nodos) que colaboran en la gestión del procesamiento de datos.

La WLGC recibe miles de solicitudes diarias para ejecutar trabajos de procesamiento de datos masivos Ti, estos trabajos pueden variar en términos de complejidad Ti.c y tiempo límite de entrega (deadline) Ti.d. Para optimizar el uso de la WLGC y minimizar tiempos de espera, se necesita un sistema que priorice las solicitudes de trabajo, de modo que cuando un nodo se libera pueda recibir el siguiente trabajo más relevante en espera.

a) (2 pt) Proponga una EDD para representar la lista de trabajos en espera de la WLGC de modo de priorizar los trabajos según el tiempo límite de entrega (deadline) Ti.d. Los trabajos que deben completarse más pronto (con menor tiempo hasta el deadline) se deben procesar primero. En caso de que dos trabajos tengan el mismo deadline, se prioriza el trabajo que llegó primero. Incluya el pseudo código de las funciones para insertar y extraer tareas de la cola de trabajos en espera.

b) (3 pt) Además del tiempo límite de entrega, la WLGC debe considerar la complejidad del trabajo Ti.c. Así, si dos trabajos tienen el mismo menor deadline, el trabajo con menor complejidad debe ser procesado primero. Si dos trabajos tienen el mismo deadline y la misma complejidad, se sigue aplicando el criterio de orden de llegada. Identifique las modificaciones necesarias en su EDD de a) para incorporar este criterio, e incluya las versiones modificadas de pseudo código para las funciones para insertar y extraer tareas de la cola de trabajos en espera.

c) (1 pt) Describa (no se requiere pseudo código) los pasos necesarios para actualizar el deadline y/o complejidad de la tarea Ti.

PAUTA:

a) La EDD a utilizar es un Min Heap hDeadline con llave Ti.d. Para cada nodo del Heap hDeadline los trabajos son almacenados en una cola FIFO Q para mantener el orden de llegada en el caso que dos trabajos tienen igual Ti.d . Como un heap no es una EDD de búsqueda, al insertar se debe verificar primero si ya existe un nodo con prioridad Ti.d para insertar directo en la cola Q, antes de insertar un nuevo nodo en el heap. De igual forma, solo se eliminan nodos desde los heap cuando la cola de tareas asociada está vacía.

[Puntajes:

- 0,5 puntos por la EDD, -0,2 puntos si agrega un atributo extra en lugar de usar una cola FIFO.
- 0,6 puntos por insert, -0,3 si no nota que requiere verificar si la llave ya estaba en el heap.
- 0,6 puntos por extract, -0,3 si no nota que no debe extraer el nodo si aun tiene tareas con el mismo deadline (cola no vacía).
- 0,3 puntos por las funciones de manejo de la cola FIFO (o del atributo adicional si lo usaron)

]
←

```
Insert(hDeadline, Ti)
        if buscaHeap(hDeadline, Ti.d, pos)    /* busca en el Heap si ya hay una tarea con deadline
                                               Ti.d, su posición es retornada en pos */
                enQueue(hDeadline, pos , Ti)  /* inserta en la cola FIFO Q del nodo pos de hDeadline
                                               la tarea Ti */
        else                                  /* no hay en el heap un nodo con igual Ti.d. */
                i ← primer nodo vacío de hDeadline
                hDeadline[i] ← Ti.d           /* Asigna al nodo el valor de la llave */
                enQueue(hDeadline, i , Ti)    /* inserta en la cola FIFO Q del nodo de hDeadline la
                                               tarea Ti */
                SiftUp(hDeadline, i)          /* es el mismo visto en clases, con el signo de
                                               comparación para min Heap */

Extract(hDeadline)
        i ← último nodo no vacío de hDeadline
        best ← deQueue(hDeadline, 0)          /* extrae la primera tarea de la cola Q en la raíz
                                               del heap */
        if hDeadline[0].Q.first == null       /* solo si la cola Q esta vacía hay que eliminar el
                                               nodo */
                hDeadline[0] ← hDeadline[i]
                hDeadline[i] ← null
                SiftDown(hDeadline, 0)        /* el mismo de clases ajustado para min Heap */
        return best

buscaHeap(h, key, pos)
        for pos in [0..heapSize -1]
                if h [pos] == key
                        return true
        return false

enQueue(h, pos , Ti)
        /* el nodo pos del heap tiene asociada una cola Q, implementada como lista (o arreglo) */
        /* Q.first es el primer elemento, y Q.last el último y cada nodo nQ de la cola tiene un */
        /* puntero a la tarea asociada y un puntero al siguiente elemento de la cola */
        nuevo ← new nQ
        nuevo.task ← Ti
        nuevo.siguiente ← null
        if h[pos].Q.last <> null
                h[pos].Q.last.siguiente ← nuevo
        h[pos].Q.last ← nuevo
        If h[pos].Q.first == null
                h[pos].Q.first ← nuevo

deQueue(h, pos)
        nodoQ ← h[pos].Q.first                        /* sabemos que la cola no está vacía */
        h[pos].Q.first ← h[pos].Q.first.siguiente
        trabajo ← nodoQ.task.                         /* la tarea almacenada en la cola */
        free(nodoQ)                                   /* liberamos la memoria del nodo de la cola
                                                      que extraemos */
        if h[pos].Q.first == null
                h[pos].Q.last ← null
        return trabajo
```

b) Para lo solicitado, se reemplaza la cola FIFO Q de los nodos del Min Heap hDeadline por un Min Heap hComplejidad con llave Ti.c. Para cada nodo del Heap hComplejidad los trabajos son almacenados en una cola FIFO Q para mantener el orden de llegada en el caso que dos trabajos tienen igual Ti.c . Como un heap no es una EDD de búsqueda, al insertar en ellos se debe verificar primero si ya existe un nodo con prioridad Ti.d o Ti.c para insertar directo en la cola Q, antes de insertar un nuevo nodo en el heap. De igual forma, solo se eliminan nodos desde los heap cuando la cola de tareas asociada está vacía.

[Puntajes:

- 1,0 punto por definir las modificaciones necesarias en la EDD y pseudocódigo
- 1,0 punto por insert modificado
- 1,0 punto por extract modificado

]

```
Insert(hDeadline, Ti)
        if buscaHeap(hDeadline, Ti.d, pos)    /* busca una tarea con deadline Ti.d */
                if buscaHeap(hDeadline[pos].hComplejidad, Ti.c, pos2)
                        enQueue(hDeadline[pos].hComplejidad, pos2 , Ti)
                else                          /* no hay en el heap un nodo con igual Ti.c. */
                        i ← primer nodo vacío de hComplejidad
                        hDeadline[pos].hComplejidad[i] ← Ti.c
                        enQueue(hDeadline[pos].hComplejidad, i , Ti)
                        SiftUp(hDeadline[pos].hComplejidad, i)
        else                                  /* no hay en el heap un nodo con igual Ti.d. */
                i ← primer nodo vacío de hDeadline
                hDeadline[i] ← Ti.d           /* Asigna al nodo el valor de la llave */
                j ← 0                         /* el heap hComplejidad está vacío */
                hDeadline[pos].hComplejidad[j] ← Ti.c
                enQueue(hDeadline[pos].hComplejidad, j , Ti)
                SiftUp(hDeadline, i)

Extract(hDeadline)
        i ← último nodo no vacío de hDeadline
        j ← último nodo no vacío de hDeadline[0].hComplejidad
        best ← deQueue(hDeadline[0].hComplejidad, 0)        /* extrae la primera tarea del heap */
        if hDeadline[0].hComplejidad[0].Q.first == null     /* si vacía eliminar el nodo */
                hDeadline[0].hComplejidad[0] ← hDeadline[0].hComplejidad[j]
                hDeadline[0].hComplejidad[j] ← null
                SiftDown(hDeadline[0].hComplejidad, 0)
        if hDeadline[0].hComplejidad[0] == null             /* si vacío eliminar nodo hDeadline*/
                hDeadline[0] ← hDeadline[i]
                hDeadline[i] ← null
                SiftDown(hDeadline, 0)
        return best
```

c) Se debe buscar en hDeadline si hay un nodo con deadline Ti.d, si existe se debe buscar en el hComplejidad de ese nodo (hDeadline[i]) si hay un nodo con complejidad Ti.c. Si existe se debe buscar en la cola Q de ese nodo (hComplejidad[j]) si algún elemento de esa cola es la tarea Ti. Si la tarea Ti está registrada en la EDD se debe eliminarla de la cola hComplejidad[j].Q donde estaba registrada para luego insertar Ti nuevamente en la EDD con la función Insert con sus atributos actualizados.

[Puntajes:

- 0,4 punto por la descripción de la búsqueda para encontrar la Ti en la EDD
- 0,3 punto por la eliminación de la tarea de la EDD, solo de la cola Q en que está registrada (no requiere el caso cuando era la única tarea en la cola)
- 0,3 punto por insertarla nuevamente como una tarea nueva con sus valores actualizados

]

Pregunta 2

Considera el concepto de *inversión* que vimos cuando estudiamos el algoritmo *insertionSort*: Los índices i y j son una inversión en el arreglo r si $i < j$ pero $r[i] > r[j]$.

a) Escribe un algoritmo basado en (o similar a) *insertionSort* que cuente una a una el número de inversiones de un arreglo r.

insertionSort corrige una a una cada una de las inversiones de r, esto es, cada vez que ejecuta un intercambio corrige una inversión. Por lo tanto, para contar una a una el número de inversiones, basándonos en *insertionSort*, basta con contar uno a uno los intercambios que realiza *insertionSort*.

```
int invertionCount(r, n):
    int count = 0
    for i = 1 . . . n − 1 :
        j = i
        while ( j > 0) ∧ ( r[j] < r[j − 1] ) :
            intercambiar r[j ] con r[j − 1]
            count = count + 1
            j = j−1
    return count
```

b) Determina la complejidad de tiempo —en notación O( )— del peor caso de tu algoritmo de a).

Según se ve en el código de arriba, *invertionCount* realiza básicamente las mismas operaciones que *insertionSort*, excepto que incrementa un contador *count* cada vez que ejecuta un intercambio. Por lo tanto, tiene la misma complejidad que *insertionSort*: $O(n^2)$ en el peor caso.

c) Escribe ahora un algoritmo basado en la estrategia *dividir para conquistar* (y similar a *mergeSort*) que calcule —sin tener necesariamente que contarlas una a una— el número de inversiones de un arreglo *r*.

Modificamos *mergeSort* para que tome un segundo parámetro (por referencia), *count*, de tipo **int**, que parte en 0 (al hacer la llamada inicial a *mergeSort'*) y que al terminar la ejecución de *mergeSort'* almacena la cantidad de inversiones que hay en *r*.

```
mergeSort' ( r, count ):
    if |r| = 1 : return r
    dividir r en r1 y r2
    s1 ← mergeSort' ( r1, count )
    s2 ← mergeSort' ( r2, count )
    s ← merge' ( s1, s2, count )
    return s
```

El verdadero trabajo —ordenar— de *mergeSort* lo hace *merge*; también en esta versión modificada, en que además de ordenar, cuenta —o más precisamente, calcula: cada vez que *b* va a parar a *C*, *count* es incrementado en el número de datos que aún quedan en *A*. ¿Por qué? Dado que *A* representa la "mitad de arriba", o "de la izquierda", del tramo de *r* que estamos ordenando en esta llamada a *mergeSort'*, si $b < a$ significa que un dato en *B* —la "mitad de abajo" o "de la derecha"— es menor que todos los datos que aún quedan en *A* (todos los cuales son mayores que *a*, ya que *A* está ordenado); por lo tanto, cada uno de estos datos en *A* representa exactamente una inversión al compararlo con *b*.

```
merge( A, B, count ):
    iniciamos C vacía
    sean a y b los primeros elementos de A y B
    extraer de su secuencia respectiva el menor entre a y b
    if el menor es b:
        count ← count + |A|     #suponemos que se puede calcular en O(1), lo que es cierto
    insertar el elemento extraído al final de C
    si quedan elementos en A y B, volver a "sean a y b …"
    concatenar C con la secuencia que aún tenga elementos
    return C
```

d) Determina la complejidad de tiempo —en notación O( )— del peor caso de tu algoritmo de c).

Como se ve en el código, *mergeSort'* realiza básicamente las mismas operaciones que *mergeSort*, excepto que *merge'* incrementa *count* en una cierta cantidad cada vez que inserta *b* en *C*. Por lo tanto, tiene la misma complejidad que *mergeSort*: $O(n \log n)$ en el peor caso.

PAUTA 3

a) (2 pt)Demuestre que Timsort es correcto

caso base El tamaño de la lista es 1, por lo tanto está ordenado

HI Timsort divide la lista de k elementos en runs y los ordena con insertion sort que es correcto, luego los mezcla con Merge que también es correcto, lista de tamaño k está ordenada

PD Debemos demostrar que la lista de k+1 elementos está ordenada
Por hipótesis de inducción Timsort divide en runs ordenados una lista de k elementos está ordenada, luego los funde ordenadamente con merge por lo que la lista de k+1 elementos se fusionará bien.

2. (2 pt) Para dividir la secuencia en para fusionar los run timsort debe ordenar los run de a pares O(log(n)) y luego debe fusionarlos O(n). O(n log (n))

3. (2 pt) timsort necesita una memoria adicional O(n) para la fusión de los runs

PAUTA 4
Sea E el número de estaciones y N el número de datos en el arreglo

a) (1 pt) Cada nodo contiene Ex nombre de la estación, flag online/offline y puntero al arreglo. En el arreglo tx timestamp, dx datos meteorológicos.

b) (3 pt)Escriba en Pseudocódigo las funciones de:

- Insert(id, tiempo, dato)

```
Insert(Ei,ti,di)
p=InsertAVL(Ei) p entrega el puntero al nodo insertado con el nombre(Ei)
p.flag=online
InsertDatos(p.array,ti,di)
```

InsertDatos(a,t,d)

```
IF i=Buscarprimer lugar vacío en el arreglo a <>null
    a[i].t<-t; a[i].d<-d
ELSE (el arreglo está lleno)
    for i = n-1 to 0 a[i+1]<-a[i]; se corren los valores una posición (se pier
    a[0].t<-t; a[0]<-t
END
```

```
InsertAVL(nodo, id)
    IF nodo= null, arbol vacío
        return nuevonodo(id)
    ELSE IF id < nodo.id
        nodo.left = insertAVL(nodo.left, id) inserta izquierdo
    ELSE id > nodo.id
        nodo.right = insertAVL(nodo.right, id) inserta nodo derecho
    Balance
    return node
```

- Datos(id, inicio, término)

```
IF p=BuscarNodo(id) <>null el nodo existe
    if i<-busquedabinaria(p,inicio) <> null el rango de inicio existe
    j<-0 indice de recorrido del arreglo de resultado
    while p.a[i].timestamp <=termino b[j]<- a[j]<-a[i]  llenado del resultado
    return b
```

- Offline(id)

```
IF p=BuscarNodo(id) <>null el nodo existe
    page-out(id)
    p.a<-null
    p.flag=offline
```

- Todo

```
    t<- matriz vacía
    For i IN cercanas (id)
        t<-t+Datos(i, inicio, término)
```

c) (2 pt) Cuál es la complejidad de cada una de las funciones implementadas por Ud. La respuesta depende de la implementación que hicieron por lo cual se debe verificar si la argumentación es correcta. En general la complejidad depende de E el número de estaciones y N el número de datos meteorológico

Insert	el Insert en el AVL es O(log (n)) + complejidad de inserción en un arreglo es O(n)
Datos	Búsqueda del nodo O(log(e)), búsqueda binaria en el arreglo O(log(n)) + copia por rango es O(m) con m=final-inicio
Offline	Búsqueda del nodo O(log(n)) + marca Ofline O(1)
Todo	Búsqueda de cada nodo participante O(c log(e))