Trabajo Práctico de Desarrollo

Introducción:

El objetivo del presente informe es realizar una análisis completo del problema "*Instalando Aplicaciones*", tal como se expuso en el Certamen Nacional de la OIA (Olimpiada Informática Argentina) del año 2016. Se considerará un diseño **Orientado a Objetos** y se tendrá en cuenta tanto la complejidad computacional, así como la eficiencia. No solo se exhibirá el algoritmo escogido para resolver el problema, implementado en el lenguaje de programación **Java**, sino que también se demostrará el funcionamiento del mismo, respaldado tanto por inspección de las líneas de código como por casos de prueba integrales. Asimismo, se expondrán posibles alternativas, junto con sus pertenecientes características y consideraciones en aquellas áreas mencionadas previamente. Finalmente, se adjuntará un archivo comprimido en formato *.rar* cuyos contenidos sean el código fuente de la solución, así como los casos de prueba relevantes.

Análisis del Problema:

1. Objetivo

María desea instalar una nueva aplicación en su dispositivo móvil. Sin embargo, dicho celular se encuentra sin espacio de almacenamiento disponible. Para contrarrestar esto, María se dispone a borrar aplicaciones, con la condición de borrar la menor cantidad suficiente para poder instalar aquello que desea. He aquí que el objetivo del problema es encontrar la mínima cantidad de *apps consecutivas* que María requiere borrar, teniendo en cuenta la totalidad de aplicaciones que María posee en su dispositivo, junto con el espacio necesario en **megabytes**.

2. Entrada

- **Primera línea**: Dos números enteros separados por un espacio:
  - C: Número total de aplicaciones instaladas en el celular ($1 \leq C \leq 50{,}000$).
  - A: Cantidad de MB consecutivos requeridos para la instalación de la nueva aplicación ($1 \leq A \leq 1000$).
- **Segunda línea**: Una lista de C números enteros que indican el tamaño en **MB** de cada aplicación instalada. Los tamaños están en el rango de 1 a 1000 y están ordenados según su ubicación en memoria.

3. Salida

- Una única línea con el mínimo número de aplicaciones consecutivas que deben ser borradas para liberar al menos A **MB**.

4. Restricciones y Observaciones

- No se permiten MB "intermedios" sin ocupar, lo que significa que las aplicaciones están contiguas y deben ser borradas en secuencia para liberar espacio.

Requisitos Funcionales

1. **Lectura de Entradas**:
    - Leer desde un archivo **almacenamiento.in** que contiene:
        - Una línea con dos números enteros separados por un espacio: C (cantidad de aplicaciones instaladas) y A (**MB** necesarios para la nueva aplicación).
        - Una línea con C números enteros que representan los tamaños en **MB** de cada aplicación instalada.
2. **Cálculo del Resultado**:
    - Determinar la mínima cantidad de aplicaciones consecutivas que deben borrarse para liberar al menos A **MB**.
3. **Generación de Salida**:
    - Escribir el resultado en un archivo **almacenamiento.out**, que debe contener una única línea con el número mínimo de aplicaciones consecutivas que deben ser borradas.
    - Si no es posible liberar el espacio necesario, el programa debe indicar esto de alguna manera adecuada, por ejemplo, retornando -1 (o un valor específico según lo acordado).

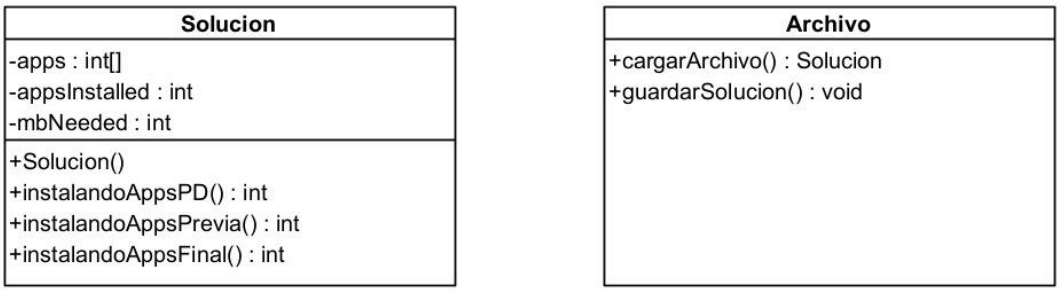
Requisitos No Funcionales

1. **Eficiencia**:
    - El programa debe ser capaz de manejar el tamaño máximo de entrada (50,000 aplicaciones) de manera eficiente.
    - La complejidad temporal del algoritmo debe ser $O(N)$ para garantizar que el programa sea rápido incluso con el mayor número de aplicaciones.
2. **Manejo de Archivos**:
    - El programa debe manejar correctamente la lectura de un archivo de entrada y la escritura en un archivo de salida.
    - Debe verificar que el archivo de entrada existe y contiene datos válidos antes de intentar procesarlos.
3. **Confiabilidad**:
    - El programa debe manejar adecuadamente casos borde y entradas no válidas, como valores fuera del rango especificado.
    - Debe asegurar que siempre haya una salida válida en el archivo **almacenamiento.out**.
4. **Claridad y Mantenibilidad**:

- El código debe estar bien documentado y estructurado para facilitar su comprensión y mantenimiento.
- Deben utilizarse nombres de variables y funciones descriptivos y comentarios donde sea necesario para explicar la lógica del algoritmo.

5. **Uso de Memoria**:
   - El programa debe hacer un uso eficiente de la memoria, especialmente dado el tamaño potencialmente grande de la entrada.
   - Evitar el uso de estructuras de datos innecesarias que puedan aumentar el consumo de memoria.

Diagrama de Clases UML



La clase *"Archivo"* es la que se encarga del manejo de los archivos tanto de salida como de entrada. Con su método **"cargarArchivo"**, trata de leer una dirección en el almacenamiento local, para así recuperar la cantidad de apps instaladas, los **MB** libres necesarios y un vector con el tamaño de cada app. El método **"guardarSolucion"**, en cambio, se invoca una vez se obtenga la solución del problema, y simplemente guardar un entero con la cantidad mínima de apps para eliminar.

La clase *"Solución"* es fundamental para resolver el problema. Posee un vector de enteros **"apps"**, que permite guardar el tamaño de cada aplicación, un entero **"appsInstalled"** que simplemente representa el tamaño del vector, y otro entero **"mbNeeded"**, la cantidad de espacio a liberar. En cuanto a métodos, hay tres de importancia:

- **instalandoAppsPD**
- **instalandoAppsPrevia**
- **instalandoAppsFinal**

Cada uno de ellos se encarga de obtener la solución, devolviendo un entero que representa la cantidad de apps consecutivas para borrar. La diferencia se encuentra en la implementación y eficiencia, que se detallará a continuación.

Implementación:

instalandoAppsPD: El método instalandoAppsPD efectivamente combina la programación dinámica con una variante de la técnica de ventana deslizante para resolver el problema. Precomputando las sumas acumuladas, el algoritmo puede rápidamente ajustar los límites de la ventana y encontrar la mínima cantidad de aplicaciones que deben ser eliminadas para liberar el espacio necesario.

```
public int instalandoAppsPD() {
    int[] dp = new int[this.appsInstalled];
    dp[0] = apps[0];

    for (int i = 1; i < this.appsInstalled; i++) {
        dp[i] = dp[i - 1] + apps[i];
    }

    int minApps = Integer.MAX_VALUE;
    int leftPointer = -1;
    int rightPointer = 0;
    int mbAcum = dp[0];

    while (rightPointer < this.appsInstalled && leftPointer <= rightPointer) {
        if (mbAcum >= this.mbNeeded) {
            minApps = Math.min(minApps, rightPointer - leftPointer);
            leftPointer++;
            mbAcum = dp[rightPointer];
            mbAcum -= dp[leftPointer];
        } else {
            rightPointer++;
            if (rightPointer < this.appsInstalled) {
                mbAcum = dp[rightPointer];
                if (leftPointer >= 0) {
                    mbAcum -= dp[leftPointer];
                }
            }
        }
    }
```

En primer lugar, se inicializa un vector. Su objetivo es precomputar la suma acumulada de los tamaños de las aplicaciones en el arreglo **dp**. Cada posición del vector contiene la suma de las aplicaciones desde la primera hasta la i-ésima aplicación. Esto permite calcular la suma de cualquier subarray en tiempo $O(1)$.

Las variable **minApps** mantiene el tamaño mínimo de subarray encontrado que cumple con la condición, **leftPointer** y **rightPointer** definen los límites de la ventana deslizante y **mbAcum** mantiene la suma acumulada de la ventana actual.

Procedimiento: Mientras **rightPointer** esté dentro del rango de aplicaciones instaladas y **leftPointer** no haya superado a **rightPointer**, pueden ocurrir dos casos:

Si **mbAcum** es suficiente (**mbAcum >= this.mbNeeded**): Se acutualiza **minApps** con la menor cantidad de aplicaciones encontradas. Luego movemos el **leftPointer** hacia la derecha para intentar encontrar una ventana más pequeña. Actualizamos **mbAcum** restando la suma acumulada correspondiente al nuevo **leftPointer**.

Si **mbAcum** no es suficiente (**mbAcum < this.mbNeeded**): Movemos el **rightPointer** hacia la derecha para expandir la ventana. Luego actualizamos **mbAcum** sumando el nuevo valor acumulado de **dp[rightPointer]**. Si **leftPointer** es válido, restamos la suma acumulada correspondiente para mantener el rango correcto de la ventana.

Para finalizar, si **minApps** se ha actualizado desde su valor inicial (Integer.MAX_VALUE), se devuelve **minApps**. De lo contrario, se devuelve -1 indicando que no es posible liberar suficiente espacio eliminando cualquier cantidad de aplicaciones consecutivas.

Complejidad Espacial:

1. **Vector dp:** posee el mismo tamaño que el número de aplicaciones instaladas, lo que requiere $O(N)$ espacio.
2. **Variables**: Las variables **minApps**, **leftPointer**, **rightPointer**, **mbAcum** ocupan espacio constante $O(1)$.
   Complejidad Total: $O(N)$

Complejidad Temporal:

1. **Cálculo de sumas acumuladas:** El bucle recorre todas las aplicaciones una vez, lo que tiene complejidad $O(N)$
2. **Búsqueda de la solución:** Tanto **rightPointer** como **leftPointer** se mueven de izquierda a derecha a lo largo del arreglo, y cada uno de estos punteros avanza a lo sumo N veces, por lo que el bucle global tiene una complejidad de $O(N)$.
   Complejidad Total: $O(N)$

---

instalandoAppsFinal: El método emplea dos punteros (i y j) para mantener una ventana deslizante sobre el arreglo de aplicaciones y calcular la suma de los tamaños de las aplicaciones dentro de esta ventana.

```
public int instalandoAppsFinal() {
    int r = this.appsInstalled;
    int j = 0, tamD = 0;

    for (int i = 0; i < this.appsInstalled; i++) {
        while (j < this.appsInstalled && tamD < this.mbNeeded) {
            tamD += this.apps[j];
            j++;
        }
        if (tamD >= this.mbNeeded)
            r = Math.min(r, j - i);

        tamD -= this.apps[i];
    }
    return r;
}
```

La variable **r** inicialmente se establece en el número total de aplicaciones, representando el peor caso en el que todas las aplicaciones tendrían que ser eliminadas. **J** es un puntero que se mueve hacia adelante para expandir la ventana, y **tamD** es la suma acumulada de los tamaños de las aplicaciones dentro de la ventana actual.

Mientras **j** sea menor que el número total de aplicaciones (**this.appsInstalled**) y la suma acumulada **tamD** sea menor que **this.mbNeeded**, se sigue expandiendo la ventana añadiendo el tamaño de la aplicación en **this.apps[j]** a **tamD** y avanzando **j**.

Si **tamD** es mayor o igual a **this.mbNeeded**, significa que se ha encontrado un conjunto de aplicaciones cuya suma de tamaños es suficiente.

Se actualiza **r** con el menor valor entre su valor y el tamaño actuales de la ventana (**j - i**). Por último, se reduce la ventana restando el tamaño de la aplicación en **this.apps[i]** de **tamD** para preparar la siguiente iteración del bucle for.

Complejidad Espacial: $O(1)$, ya que solo se utilizan unas pocas variables adicionales (no se utilizan estructuras de datos adicionales cuya memoria dependa del tamaño de la entrada).

Complejidad Temporal: $O(N)$, donde N es el número de aplicaciones instaladas (**this.appsInstalled**). Cada aplicación se procesa una vez por cada puntero (**i** y **j**), y ambos punteros solo se mueven hacia adelante, resultando en una complejidad lineal.

Casos de Prueba:

Se utilizó el siguiente caso de prueba:

Cantidad de apps: 10

MB necesarios: 100

Aplicaciones: 42 2 50 10 1 50 30 24 18 23

Resultado: 3 en ambos casos

Se observa que el resultado es el correcto, por lo que el caso de prueba verifica la validez de ambos algoritmos.

Fuentes:

Sliding Window Technique - GeeksforGeeks

Dynamic Programming or DP - GeeksforGeeks