Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

1. (4 ptos) Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j)*,** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                    k <= Math.min(Tmax, j+M); k++) {
                              int opcion  = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

2. (6 ptos) Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla $c_{ij,}$ que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

3. (4 ptos) Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j)*,** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
	int [][] S = new int [H+1][Tmax+1];
	for (int i = H; i>= 0; i--) {
		for (int j = 0; j <= Tmax; j++) {
			if (i == H) {
				S[i][j] = 0;
			}else {
				int maximo = Integer.MIN_VALUE;
				for (int k = Math.max(0, j- M);
						k <= Math.min(Tmax, j+M); k++) {
					int opcion = c[i][k] + S[i+1][k];
					if (opcion > maximo) {
						maximo = opcion;
					}
				}
				S[i][j] = maximo;
			}
		}
	}
	return S[0][T];
}
```

4. (6 ptos) Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```java
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

5. (4 ptos) Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j)*,** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.

Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                   k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

6. (6 ptos) Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
            //Aunque sólo estamos interesados en el tiempo acumulado total.
Vamos a ir creando también la solución
            ArrayList<Integer> estaciones = new ArrayList<>();
            estaciones.add(L);
            return tiempoCocheVoraz(t,c,L,estaciones);
      }


      public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer>
solucion) {
            int tiempoTotal = 0;
            int numEstaciones = t.length;
            int i = 0;//estación 1.
            while (i <  numEstaciones) {
                  //Decidimos si nos mantenemos o cambiamos
                  int lineaActual = solucion.get(i-1);
                  int otraLinea = 1 - lineaActual;
                  int costeMantenerse = t[i][lineaActual];
                  int costeCambio = c[i][lineaActual] + t[i][otraLinea];
                  if (costeMantenerse < costeCambio) {//nos quedamos en la línea
                        solucion.add(lineaActual);
                        tiempoTotal += costeMantenerse;
                  }else {//Cambiamos
                        solucion.add(otraLinea);
                        tiempoTotal += costeCambio;
                  }
                  i++;
            }
            return tiempoTotal;
       }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

7. (4 ptos) Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j)*,** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \{ \begin{array}{lr} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{array}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                  k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

8. (6 ptos) Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla $c_{ij,}$ que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

9. (4 ptos) Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j)*,** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                    k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

10. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```java
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

11. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1) \cdot (T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                  k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

12. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla $c_{ij,}$ que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```java
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

13. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1) \cdot (T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                  k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

14. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```java
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

15. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* $(1 \leq i \leq H)$ la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

a) Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.

b) Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.

c) Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.

Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
	int [][] S = new int [H+1][Tmax+1];
	for (int i = H; i>= 0; i--) {
		for (int j = 0; j <= Tmax; j++) {
			if (i == H) {
				S[i][j] = 0;
			}else {
				int maximo = Integer.MIN_VALUE;
				for (int k = Math.max(0, j- M);
					k <= Math.min(Tmax, j+M); k++) {
				   int opcion = c[i][k] + S[i+1][k];
				   if (opcion > maximo) {
				      maximo = opcion;
				   }
				}
				S[i][j] = maximo;
			}
		}
	}
	return S[0][T];
}
```

16. (6 ptos) Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```java
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

17. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

a) Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.

b) Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.

c) Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                  k <= Math.min(Tmax, j+M); k++) {
                              int opcion  = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

18. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	  }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j))\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

19. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j)*,** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
	int [][] S = new int [H+1][Tmax+1];
	for (int i = H; i>= 0; i--) {
		for (int j = 0; j <= Tmax; j++) {
			if (i == H) {
				S[i][j] = 0;
			}else {
				int maximo = Integer.MIN_VALUE;
				for (int k = Math.max(0, j- M);
					k <= Math.min(Tmax, j+M); k++) {
					int opcion = c[i][k] + S[i+1][k];
					if (opcion > maximo) {
						maximo = opcion;
					}
				}
				S[i][j] = maximo;
			}
		}
	}
	return S[0][T];
}
```

20. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```java
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

21. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1) \cdot (T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                  k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

22. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
            //Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
            ArrayList<Integer> estaciones = new ArrayList<>();
            estaciones.add(L);
            return tiempoCocheVoraz(t,c,L,estaciones);
      }


      public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
            int tiempoTotal = 0;
            int numEstaciones = t.length;
            int i = 0;//estación 1.
            while (i <  numEstaciones) {
                  //Decidimos si nos mantenemos o cambiamos
                  int lineaActual = solucion.get(i-1);
                  int otraLinea = 1 - lineaActual;
                  int costeMantenerse = t[i][lineaActual];
                  int costeCambio = c[i][lineaActual] + t[i][otraLinea];
                  if (costeMantenerse < costeCambio) {//nos quedamos en la línea
                        solucion.add(lineaActual);
                        tiempoTotal += costeMantenerse;
                  }else {//Cambiamos
                        solucion.add(otraLinea);
                        tiempoTotal += costeCambio;
                  }
                  i++;
            }
            return tiempoTotal;
       }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j))\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

23. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* $(1 \leq i \leq H)$ la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

a) Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.

b) Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.

c) Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.

Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                    k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

24. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	  }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j))\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

25. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                  k <= Math.min(Tmax, j+M); k++) {
                              int opcion  = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

26. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
            //Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
            ArrayList<Integer> estaciones = new ArrayList<>();
            estaciones.add(L);
            return tiempoCocheVoraz(t,c,L,estaciones);
      }


      public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
            int tiempoTotal = 0;
            int numEstaciones = t.length;
            int i = 0;//estación 1.
            while (i <  numEstaciones) {
                  //Decidimos si nos mantenemos o cambiamos
                  int lineaActual = solucion.get(i-1);
                  int otraLinea = 1 - lineaActual;
                  int costeMantenerse = t[i][lineaActual];
                  int costeCambio = c[i][lineaActual] + t[i][otraLinea];
                  if (costeMantenerse < costeCambio) {//nos quedamos en la línea
                        solucion.add(lineaActual);
                        tiempoTotal += costeMantenerse;
                  }else {//Cambiamos
                        solucion.add(otraLinea);
                        tiempoTotal += costeCambio;
                  }
                  i++;
            }
            return tiempoTotal;
       }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

27. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1) \cdot (T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
	int [][] S = new int [H+1][Tmax+1];
	for (int i = H; i>= 0; i--) {
		for (int j = 0; j <= Tmax; j++) {
			if (i == H) {
				S[i][j] = 0;
			}else {
				int maximo = Integer.MIN_VALUE;
				for (int k = Math.max(0, j- M);
					k <= Math.min(Tmax, j+M); k++) {
					int opcion = c[i][k] + S[i+1][k];
					if (opcion > maximo) {
						maximo = opcion;
					}
				}
				S[i][j] = maximo;
			}
		}
	}
	return S[0][T];
}
```

28. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	  }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

29. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max} + 1) \cdot (H + 1)) \approx \Theta(H \cdot T_{max})$.

Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M + 1) \cdot (T_{max} + 1) \cdot (H + 1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                    k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

30. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```java
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

31. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
	int [][] S = new int [H+1][Tmax+1];
	for (int i = H; i>= 0; i--) {
		for (int j = 0; j <= Tmax; j++) {
			if (i == H) {
				S[i][j] = 0;
			}else {
				int maximo = Integer.MIN_VALUE;
				for (int k = Math.max(0, j- M);
					k <= Math.min(Tmax, j+M); k++) {
					int opcion = c[i][k] + S[i+1][k];
					if (opcion > maximo) {
						maximo = opcion;
					}
				}
				S[i][j] = maximo;
			}
		}
	}
	return S[0][T];
}
```

32. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla $c_{ij,}$ que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
            //Aunque sólo estamos interesados en el tiempo acumulado total.
Vamos a ir creando también la solución
            ArrayList<Integer> estaciones = new ArrayList<>();
            estaciones.add(L);
            return tiempoCocheVoraz(t,c,L,estaciones);
      }


      public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer>
solucion) {
            int tiempoTotal = 0;
            int numEstaciones = t.length;
            int i = 0;//estación 1.
            while (i <  numEstaciones) {
                  //Decidimos si nos mantenemos o cambiamos
                  int lineaActual = solucion.get(i-1);
                  int otraLinea = 1 - lineaActual;
                  int costeMantenerse = t[i][lineaActual];
                  int costeCambio = c[i][lineaActual] + t[i][otraLinea];
                  if (costeMantenerse < costeCambio) {//nos quedamos en la línea
                        solucion.add(lineaActual);
                        tiempoTotal += costeMantenerse;
                  }else {//Cambiamos
                        solucion.add(otraLinea);
                        tiempoTotal += costeCambio;
                  }
                  i++;
            }
            return tiempoTotal;
       }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j))\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

33. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1) \cdot (T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                  k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

34. (6 ptos) Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla $c_{ij,}$ que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j))\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

35. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* $(1 \le i \le H)$ la temperatura es de *j* grados $0 \le j \le T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j)*,** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \le k \le \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \le H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                    k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

36. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla $c_{ij,}$ que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```java
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j))\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

37. (4 ptos) Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \displaystyle\max_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
	int [][] S = new int [H+1][Tmax+1];
	for (int i = H; i>= 0; i--) {
		for (int j = 0; j <= Tmax; j++) {
			if (i == H) {
				S[i][j] = 0;
			}else {
				int maximo = Integer.MIN_VALUE;
				for (int k = Math.max(0, j- M);
					k <= Math.min(Tmax, j+M); k++) {
				   int opcion  = c[i][k] + S[i+1][k];
				   if (opcion > maximo) {
					maximo = opcion;
				   }
				}
				S[i][j] = maximo;
			}
		}
	}
	return S[0][T];
}
```

38. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j))\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

39. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1) \cdot (T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                   k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

40. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.
Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.
Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.
Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```java
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	  }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos

Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

41. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1) \cdot (T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
	int [][] S = new int [H+1][Tmax+1];
	for (int i = H; i>= 0; i--) {
		for (int j = 0; j <= Tmax; j++) {
			if (i == H) {
				S[i][j] = 0;
			}else {
				int maximo = Integer.MIN_VALUE;
				for (int k = Math.max(0, j- M);
					k <= Math.min(Tmax, j+M); k++) {
				    int opcion = c[i][k] + S[i+1][k];
				    if (opcion > maximo) {
				        maximo = opcion;
				    }
				}
				S[i][j] = maximo;
			}
		}
	}
	return S[0][T];
}
```

42. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```java
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

43. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \le i \le H$) la temperatura es de *j* grados $0 \le j \le T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j)*,** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \le k \le \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \le H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                    k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

44. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
            //Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
            ArrayList<Integer> estaciones = new ArrayList<>();
            estaciones.add(L);
            return tiempoCocheVoraz(t,c,L,estaciones);
      }


      public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
            int tiempoTotal = 0;
            int numEstaciones = t.length;
            int i = 0;//estación 1.
            while (i <  numEstaciones) {
                  //Decidimos si nos mantenemos o cambiamos
                  int lineaActual = solucion.get(i-1);
                  int otraLinea = 1 - lineaActual;
                  int costeMantenerse = t[i][lineaActual];
                  int costeCambio = c[i][lineaActual] + t[i][otraLinea];
                  if (costeMantenerse < costeCambio) {//nos quedamos en la línea
                        solucion.add(lineaActual);
                        tiempoTotal += costeMantenerse;
                  }else {//Cambiamos
                        solucion.add(otraLinea);
                        tiempoTotal += costeCambio;
                  }
                  i++;
            }
            return tiempoTotal;
        }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j))\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \le n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \le n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos

Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

45. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \le i \le H$) la temperatura es de *j* grados $0 \le j \le T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j)*,** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \le k \le \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \le H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
	int [][] S = new int [H+1][Tmax+1];
	for (int i = H; i>= 0; i--) {
		for (int j = 0; j <= Tmax; j++) {
			if (i == H) {
				S[i][j] = 0;
			}else {
				int maximo = Integer.MIN_VALUE;
				for (int k = Math.max(0, j- M);
					k <= Math.min(Tmax, j+M); k++) {
					int opcion = c[i][k] + S[i+1][k];
					if (opcion > maximo) {
						maximo = opcion;
					}
				}
				S[i][j] = maximo;
			}
		}
	}
	return S[0][T];
}
```

46. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j))\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

47. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j)*,** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1) \cdot (T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                    k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

48. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.
Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.
Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.
Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j))\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

49. (4 ptos) Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1) \cdot (T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                    k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

50. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j))\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos

Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

51. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                  k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

52. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. 
Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> 
solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

53. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* $(1 \leq i \leq H)$ la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1)\cdot(T_{max}+1)\cdot(H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                  k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

54. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla $c_{ij,}$ que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total.
Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer>
solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	 }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)

Análisis y Diseño de Algoritmos
Parcial 2 (Noviembre 2022)

Nombre: **Grupo:**

55. **(4 ptos)** Supongamos que tenemos que controlar la calefacción central de una estación antártica a lo largo de las H primeras horas de un intervalo de tiempo. Para ello, disponemos de una tabla c_{ij} con números enteros que representan el confort o satisfacción de los miembros de la base si a la hora *i* ($1 \leq i \leq H$) la temperatura es de *j* grados $0 \leq j \leq T\_max$. Al comienzo del intervalo de tiempo la temperatura se fija a *T* grados y a partir de ahí se puede modificar en como máximo *M* grados en cada hora, pero nunca puede salirse del rango *[0, Tmax]*. Queremos conocer qué temperatura seleccionar cada hora para que se maximice la satisfacción acumulada a lo largo de las horas de todo el intervalo temporal. Para ello, queremos implementar un algoritmo de programación dinámica **basándonos en la ecuación de recurrencia *S(i,j),*** que devuelve el máximo confort total del intervalo que va desde la hora *i* hasta la hora final del intervalo (*H*), sabiendo que la temperatura actual es *j*.

$$S(i,j) = \begin{cases} 0 & i > H \\ \max\limits_{\max(0,\ j-M) \leq k \leq \min(T_{max},\ j+M)} \{c_{ik} + S(i+1,k)\} & i \leq H \end{cases}$$

- **a)** Describir la estructura de datos auxiliar necesaria: dimensiones, celda donde está la solución al problema y donde están los casos base y cómo se rellenaría si quisiéramos utilizar el enfoque bottom-up.
- **b)** Implementar un algoritmo de programación dinámica (top-down o bottom-up) que reciba los datos de entrada y **devuelva el confort total máximo**.
- **c)** Analizar, de forma razonada, la complejidad espacial y temporal del algoritmo implementado suponiendo que el tamaño de entrada depende la longitud del intervalo temporal (*H*) y de la temperatura máxima (T_{max}).

Vamos a necesitar una tabla de (H+1) filas (por el caso base) y (T_{max}+1) columnas. Los casos base están en la última fila.Para resolver el problema partimos de la temperatura T y queremos ver la satisfacción total en el intervalo [1..H], es decir S(1,T). Por tanto, la celda con el resultado será la celda (1,T). Habría que rellenar la tabla de abajo a arriba. El orden de rellenado en una fila es indiferente. Elegimos de izquierda a derecha.

El tamaño de entrada es H, la longitud del intervalo de entrada. Además de los datos de entrada necesitaremos variables de tipos básicos y la tabla S, de dimensiones H+1 x T_{max}+1. Por tanto, $E(H, T_{max}) \in \Theta((T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$.
Por otro lado, necesitamos rellenar todas las filas de la tabla ((H+1)·(T_{max})+1). En el peor caso, hay que calcular y comparar 2M+1 posibles cambios de temperatura por celda que conllevan un coste constante porque M no depende de H. En ese caso $T(H, T_{max}) \in \Theta((2M+1) \cdot (T_{max}+1) \cdot (H+1)) \approx \Theta(H \cdot T_{max})$

```java
//Objetivo: satisfacción máxima en el intervalo [1..H]
//Precondición: c es una tabla con H filas y Tmax+1 columnas.
//Nota: los índices de fila se han ajustado, dado que en Java empiezan en 0.
public int maximoConfort(int [][]c, int H, int T, int M, int Tmax) {
      int [][] S = new int [H+1][Tmax+1];
      for (int i = H; i>= 0; i--) {
            for (int j = 0; j <= Tmax; j++) {
                  if (i == H) {
                        S[i][j] = 0;
                  }else {
                        int maximo = Integer.MIN_VALUE;
                        for (int k = Math.max(0, j- M);
                                  k <= Math.min(Tmax, j+M); k++) {
                              int opcion = c[i][k] + S[i+1][k];
                              if (opcion > maximo) {
                                    maximo = opcion;
                              }
                        }
                        S[i][j] = maximo;
                  }
            }
      }
      return S[0][T];
}
```

56. **(6 ptos)** Una fábrica de coches tiene dos líneas de ensamblaje, cada una con *n* estaciones de trabajo. Cada estación se dedica a alguna tarea concreta como ajuste del motor, colocación de faros, pintura, etc. Las dos líneas tienen el mismo tipo de estaciones y en el mismo orden. Para que un coche esté listo debe pasar a través de los *n* tipos de estaciones. Disponemos del tiempo que tarda cada estación en realizar su labor en una tabla t_{ij} donde *i* es la posición de la estación en la línea de ensamblaje y *j* un identificador (0 o 1) de la línea de ensamblaje a la que pertenece la estación. Normalmente, los coches pasan por todas las estaciones de una misma línea de ensamblaje en secuencia desde la primera a la última, pero a veces es necesario transferirlos a la otra línea porque aquella en la que están tiene problemas o alguna de las estaciones está saturada de trabajo. Cambiar de línea al coche supone un incremento de tiempo dado por la tabla c_{ij}, que indica el coste de pasar el coche de la entrada de la estación *i* de la línea *j* a la entrada de la estación *i* de la otra línea. Es decir, c_{30} indicaría el coste de transferir el coche del inicio de la estación *3* de la línea *0* al principio de la estación 3 de la línea *1* Queremos encontrar un algoritmo que devuelva el mínimo tiempo necesario para construir un coche que comienza en la línea dada $L \in \{0,1\}$.

a) Diseñar e implementar un algoritmo voraz que resuelva el problema.

c) Podríamos implementar un algoritmo de programación dinámica basándonos en una ecuación de recurrencia *Coche(i,j)* que devuelva el mínimo tiempo para finalizar la construcción de un coche que está en la línea *j* y la siguiente estación por la que tiene que pasar es la *i*. **Definir dicha ecuación de recurrencia**.

El algoritmo voraz va a ir en cada paso decidiendo si se cambia de línea o no en función del tiempo que necesite para pasar la estación i. Sin comprobar si ese cambio va a repercutir en un mayor tiempo acumulado a largo plazo.

Estructura de la solución: Lista indicando para cada estación *i* (*1<= i <=n*) si estamos en la línea 1 o 0. La casilla de índice cero, siempre tendrá el identificador de la línea en la que empieza el coche. Es decir, el estado inicial de la solución es S={L}. Una solución parcial, en la que se haya pasado por k estaciones tendrá una forma S = {L,s_1, …s_k} donde $k \leq n, s_i \in \{0,1\}$

Función objetivo(minimizarla): tiempo acumulado total de la solución, es decir, el tiempo total en pasar por todas las estaciones. $\sum_{i=1}^{k} tiempo(S,i)$, donde

$$tiempo(S,i) = \begin{cases} c_{iS[i]} + t_{i(1-S[i])} & S[i] \neq S[i-1] \\ t_{iS[i]} & S[i] = S[i-1] \end{cases}$$

Tiempo(S,i) calcula el tiempo en pasar por la estación i, sabiendo si permanecemos en la misma línea o hemos cambiado de línea.

Candidatos factibles: En cada paso de construcción las únicas opciones son quedarnos en la línea (usamos el valor de la posición *i-1*) o cambiarnos (usamos el valor que no está en *i-1*). Los valores para la siguiente posición de nuestra lista pueden ser *0* o *1*.

Función de selección: Elegiremos la opción que permita pasar la siguiente estación *i* con el mínimo tiempo, es decir, si estamos en la línea *0,* la opción mejor entre t_{i0} y $c_{i0} + t_{i1}$. Si estamos en la línea *1*, la opción mejor entre t_{i1} y $c_{i1} + t_{i0}$.

Función de terminación: Cuando *S.length = n+1* (el valor inicial L y las n decisiones tomadas), ya habremos pasado por todas las estaciones.

```
public int tiempoCoche(int [][] t, int [][] c, int L) {
		//Aunque sólo estamos interesados en el tiempo acumulado total. Vamos a ir creando también la solución
		ArrayList<Integer> estaciones = new ArrayList<>();
		estaciones.add(L);
		return tiempoCocheVoraz(t,c,L,estaciones);
	}


	public int tiempoCocheVoraz(int [][] t, int[][]c,int L, ArrayList<Integer> solucion) {
		int tiempoTotal = 0;
		int numEstaciones = t.length;
		int i = 0;//estación 1.
		while (i <  numEstaciones) {
			//Decidimos si nos mantenemos o cambiamos
			int lineaActual = solucion.get(i-1);
			int otraLinea = 1 - lineaActual;
			int costeMantenerse = t[i][lineaActual];
			int costeCambio = c[i][lineaActual] + t[i][otraLinea];
			if (costeMantenerse < costeCambio) {//nos quedamos en la línea
				solucion.add(lineaActual);
				tiempoTotal += costeMantenerse;
			}else {//Cambiamos
				solucion.add(otraLinea);
				tiempoTotal += costeCambio;
			}
			i++;
		}
		return tiempoTotal;
	  }
```

En cuanto al algoritmo de programación dinámica iremos decidiendo en cada estación *i* si se mantiene el coche en la línea *j* o cambiamos de línea con el sobrecoste asociado.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i(1-j)} + Coche(i+1,(1-j)\} & i \leq n \end{cases}$$

En vez de usar (1-j) para representar la otra línea, se podrían utilizar dos ramas para el caso general, una cuando j = 0 y otra cuando j= 1. De esa manera, sabremos el índice de la otra línea en la opción de cambiarnos de línea.

$$Coche(i,j) = \begin{cases} 0 & i = n+1 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i1} + Coche(i+1,1)\} & i \leq n, j = 0 \\ \min\{t_{ij} + Coche(i+1,j), c_{ij} + t_{i0} + Coche(i+1,0)\} & i \leq n, j = 1 \end{cases}$$

El problema se resolvería con Coche(1,L)