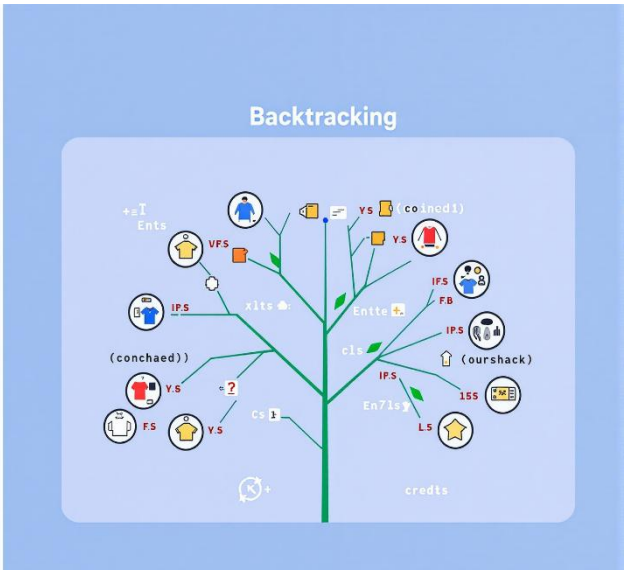
ALGORITMOS Y ESTRUCTURAS DE DATOS II

Avance Rápido y BackTracking



Alumnos:

- Rafael Guillén García

Grupo: 2.4

ÍNDICE

1. **Problemas Resueltos.**
2. **Resolución de Problemas: Avance Rápido.**

    2.1. Pseudocódigo del Algoritmo.
    2.1.1. Explicación del Algoritmo.
    2.1.2. Variables Utilizadas.
    2.1.3. Funciones Básicas del Esquema Algorítmico.
    2.2. Programación del Algoritmo.
    2.3. Estudio Teórico del Tiempo de Ejecución del Algoritmo.
    2.4. Estudio Experimental del Tiempo de Ejecución.
    2.5. Contraste Estudio Teórico y Experimental.

3. **Resolución de Problemas: BackTracking.**

    3.1. Pseudocódigo del Algoritmo.
    3.1.1. Explicación del Algoritmo.
    3.1.2. Variables Utilizadas.
    3.1.3. Funciones Básicas del Esquema Algorítmico.
    3.2. Programación del Algoritmo.
    3.3. Estudio Teórico del Tiempo de Ejecución del Algoritmo.
    3.4. Estudio Experimental del Tiempo de Ejecución.
    3.5. Contraste Estudio Teórico y Experimental.

4. **Conclusión**

1. Problemas Resueltos

Los ejercicios para resolver eran el I de Algoritmos Voraces y el F de BackTracking.

Dejamos este apartado de la memoria para el final. Cuando lo estábamos revisando ya se había cerrado la entrega del Mooshak y no podíamos acceder, por tanto no tenemos las capturas del "Accepted" ni el número de la entrega. De todas formas, ambos códigos enviados e indicados en esta memoria han logrado el Accepted en Mooshak. Recordamos que nuestro nombre de usuario en Mooshak es para que el profesor lo pueda revisar, siendo nuestra entrega el último Accepted enviado en Mooshak. Pedimos perdón por las molestias.

2. Resolución de problema: Avance Rápido

Ejercicio I

El Problema

Escribir un programa que realice una agrupación de alumnos de dos en dos maximizando la suma de los productos del grado de amistad y la compenetración en el trabajo de alumnos que se agrupan juntos. Se dispone para ello de una matriz de ***amistad*** y otra de ***trabajo***, donde se guardan los grados de amistad y de compenetración, que pueden no ser recíprocos, por lo que las matrices no son simétricas. La amistad y la compenetración de un alumno consigo mismo se almacenará como cero. En cada pupitre se suman los grados de amistad de los alumnos en el pupitre, y se suman los grados de compenetración, y se multiplican los resultados de las dos sumas. El valor de beneficio es la suma de los productos obtenidos en todos los pupitres (sin incluir el posible pupitre donde sólo haya un alumno).

Suponer, por ejemplo, un caso donde el número de alumnos *N* = 3. Sean las matrices de amistad y trabajo las siguientes:

Amistad	*Trabajo*
0 5 6	0 5 3
4 0 3	3 0 2
2 1 0	1 5 0

Si numeramos los alumnos con 0, 1 y 2, la agrupación <0, 1> tendrá valor 72 (que sale de: (5+4)*(5+3)); la <0, 2> valor 32; y la <1, 2> valor 28, por lo que la mejor opción es sentar a los alumnos 0 y 1 juntos y dejar al 2 sólo.

Habrá que resolver el problema con avance rápido. La solución no tiene por qué ser la óptima, y se admitirán soluciones que no empeoren en más de un 30% de las obtenidas con el programa de los profesores (en el ejemplo la única admisible será la de la agrupación <0, 1>).

Se considerará que el número máximo de alumnos es 100.

2.1. Pseudocódigo del Algoritmo

```
función voraz(amistad, trabajo, libres):
    valor = 0                          // Beneficio total acumulado
    parejasFormadas = 0       // Número de parejas formadas
    Mientras NO solucion(parejasFormadas) hacer:
        (i, j) = seleccionar(amistad, trabajo, libres)
        Si i == -1 o j == -1 entonces:
            Salir del bucle
        FinSi
        beneficio = (amistad[i][j] + amistad [j][i]) * (trabajo[i][j] + trabajo[j][i])
        insertar(i, j, libres)
        valor = valor + beneficio
        parejasFormadas = parejasFormadas + 1
    FinMientras
    Si N % 2 == 1 entonces
        Para k desde 0 hasta N – 1 hacer:
            Si libres[k] entonces:
                parejas.push_back(k) // Insertar al alumno k solo en el vector parejas
                salir del bucle
            FinSi
        FinPara
    FinSi
    Devolver valor
```

2.1.1. Explicación del Algoritmo:

El algoritmo voraz diseñado tiene como objetivo formar parejas de alumnos de manera que el beneficio total obtenido sea el mayor posible. Para ello, en cada iteración se selecciona la mejor pareja disponible (la que maximiza el beneficio) y se añade a la solución.

El proceso comienza con todos los alumnos libres y sin ninguna pareja formada. En cada iteración, se examinan todas las parejas posibles de alumnos que aún no han sido emparejados. Para cada una se calcula el beneficio que se obtendría si se emparejaran, usando la fórmula:

$$(amistad[i][j] + amistad[j][i]) * (trabajo[i][j] + trabajo[j][i])$$

De todas las parejas posibles, se elige aquella que proporciona el mayor beneficio. Una vez identificada, se añade al vector de parejas formadas y es marca a ambos alumnos como no disponibles para futuras parejas. El beneficio obtenido se suma al beneficio total acumulado.

El proceso se repite hasta que se hayan formado todas las parejas. En el caso de que el número de alumnos sea impar, se deja al último alumno sin emparejar y se añade solo a la lista de parejas.

Finalmente, se devuelve el beneficio total acumulado.

2.1.2. Variables utilizadas:

1. **N:** número entero que almacena el número total de alumnos. Es una variable global que se utiliza para dimensionar las matrices y los vectores, y para controlar los bucles del algoritmo.
2. **amistad:** matriz cuadrada de tamaño NxN que almacena el grado de amistad entre cada par de alumnos. El valor amistad[i][j] indica el nivel de amistad que el alumno i tiene hacia el alumno j. Se utiliza para calcular el beneficio de emparejar a dos alumnos.
3. **trabajo:** matriz cuadrada de tamaño NxN que almacena la compatibilidad de trabajo entre cada par de alumnos. El valor trabajo[i][j] indica cómo de bien trabaja el alumno i con el alumno j. También se utiliza en el cálculo del beneficio de cada pareja.
4. **parejas:** vector global de enteros que almacena los índices de los alumnos que forman cada pareja. Cada vez que se forma una pareja, se añaden los dos índices correspondientes. Si el número de alumnos es impar, el último alumno sin emparejar también se añade solo a este vector.
5. **libres:** vector de booleanos de tamaño N. Cada posición indica si el alumno correspondiente está disponible para ser emparejado (true) o si ya ha sido emparejado (false). Permite controlar qué alumnos pueden formar nuevas parejas en cada iteración.
6. **valor:** variable entera que acumula el beneficio total obtenido al formar las parejas. Se va incrementando en cada iteración con el beneficio de la pareja seleccionada.
7. **parejasFormadas:** variable entera que cuenta cuántas parejas se han formado hasta el momento. Se incrementa en cada iteración del bucle principal y se utiliza para saber cuándo se ha alcanzado el número máximo de parejas posibles.
8. **i, j:** son los índices de los alumnos seleccionados en cada iteración para formar la pareja que maximiza el beneficio. Se obtienen como resultado de la función seleccionar.
9. **beneficio:** variable temporal que almacena el beneficio calculado para la pareja seleccionada en cada iteración, antes de sumarlo al beneficio total.
10. **k:** índice utilizado en el caso de que el número de alumnos sea impar, para identificar al alumno que queda sin emparejar y añadirlo solo al vector de parejas.

2.1.3. Funciones Básicas del Esquema Algorítmico:

1. **seleccionar(amistad, trabajo, libres):**
   - Esta función se encarga de buscar, entre todos los alumnos que aun no han sido emparejados, la pareja (i, j) que proporciona el mayor beneficio posible. Para ello, recorre todas las combinaciones de alumnos libres y calcula el beneficio de cada posible pareja usando la fórmula:

     $$(amistad[i][j] + amistad[j][i]) * (trabajo[i][j] + trabajo[j][i])$$

   - Devuelve los índices de los dos alumnos que forman la mejor pareja disponible en ese momento
   - Si no se encuentra ninguna pareja válida (por ejemplo, si quedan menos de dos alumnos libres), devuelve (-1, -1) como señal para terminar el algoritmo.

```
// Función que busca entre todos los alumnos libres la pareja (i, j) que maximiza el beneficio.
pair<int, int> seleccionar(const vector<vector<int>>& amistad, const vector<vector<int>>& trabajo, const vector<bool>& libres)
{
    int beneficioMax = -1; // Beneficio máximo encontrado hasta el momento
    int mejorI = -1;       // Índice del primer alumno de la mejor pareja
    int mejorJ = -1;       // Índice del segundo alumno de la mejor pareja
    for (int i = 0; i < N; ++i)
    {
        if (!libres[i]) continue; // Saltamos si el alumno i ya está emparejado
        for (int j = i + 1; j < N; ++j)
        {
            if (!libres[j]) continue; // Saltamos si el alumno j ya está emparejado
            // Calculamos el beneficio de emparejar a i y j
            int beneficio = (amistad[i][j] + amistad[j][i]) * (trabajo[i][j] + trabajo[j][i]);
            // Si es el mayor beneficio encontrado, actualizamos la mejor pareja
            if (beneficio > beneficioMax)
            {
                beneficioMax = beneficio;
                mejorI = i;
                mejorJ = j;
            }
        }
    }
    return {mejorI, mejorJ};
}
```

2. **insertar(i, j, libres):**
   - Está función añade la pareja formada por los alumnos i y j al vector global parejas, y marca a ambos como no disponibles en el vector libres. Así, se asegura que no se vuelvan a seleccionar en las siguientes iteraciones.

```
// Función que añade la pareja (i, j) al vector de solución y marca a ambos alumnos como no libres.
void insertar(int i, int j, vector<bool>& libres)
{
    parejas.push_back(i);
    parejas.push_back(j);
    libres[i] = false;
    libres[j] = false;
}
```

3. **solución(parejasFormadas):**
   - Esta función comprueba si ya se han formado todas las parejas posibles, es decir, si se ha alcanzado el número máximo de parejas que se pueden hacer con los alumnos disponibles. Si es así, el algoritmo termino.

```
// Función que comprueba si se han formado todas las parejas posibles
bool solucion(int parejasFormadas)
{
    return parejasFormadas >= N / 2;
}
```

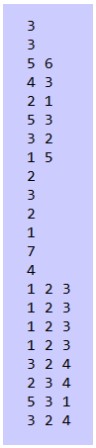
2.2. Programación del Algoritmo.

El código fuente del algoritmo voraz desarrollado para el problema ha sido implementado en C++. El código está documentado con comentarios que explican el propósito de cada variable, el funcionamiento de cada función y su correspondencia con las funciones básicas del esquema algorítmico voraz.

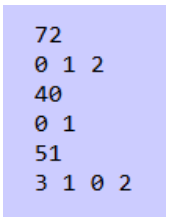
En los apartados anteriores, se encuentra el pseudocódigo del algoritmo voraz, la descripción de las variables utilizadas y de las funciones básicas del esquema.

El programa ha sido probado y funciona correctamente para los casos de prueba, aunque, por ejemplo:

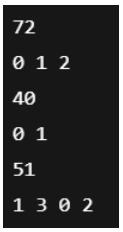
para la entrada:



la salida esperada es:



salida de nuestro algoritmo:



El valor del beneficio total es correcto en ambos casos, pero el orden de los alumnos en la última línea está intercambiado (el 1 y el 3). Esto no afecta a la validez de la solución, ya que se maximiza el beneficio total obtenido y el emparejamiento es correcto.

Código Fuente del Programa:

```cpp
#include <iostream>
#include <vector>
using namespace std;

// Número de alumnos
int N;

// Vector para guardar las parejas que se van formando en la solución
vector<int> parejas;

// Función que busca entre todos los alumnos libres la pareja (i, j) que maximiza el beneficio.
pair<int, int> seleccionar(const vector<vector<int>>& amistad, const vector<vector<int>>& trabajo, const vector<bool>& libres)
{
    int beneficioMax = -1; // Beneficio máximo encontrado hasta el momento
    int mejorI = -1;       // Índice del primer alumno de la mejor pareja
    int mejorJ = -1;       // Índice del segundo alumno de la mejor pareja
    for (int i = 0; i < N; ++i)
    {
        if (!libres[i]) continue; // Saltamos si el alumno i ya está emparejado
        for (int j = i + 1; j < N; ++j)
        {
            if (!libres[j]) continue; // Saltamos si el alumno j ya está emparejado
            // Calculamos el beneficio de emparejar a i y j
            int beneficio = (amistad[i][j] + amistad[j][i]) * (trabajo[i][j] + trabajo[j][i]);
            // Si es el mayor beneficio encontrado, actualizamos la mejor pareja
            if (beneficio > beneficioMax)
            {
                beneficioMax = beneficio;
                mejorI = i;
                mejorJ = j;
            }
        }
    }
    return {mejorI, mejorJ};
}
```

```cpp
// Función que añade la pareja (i, j) al vector de solución y marca a ambos alumnos como no libres.
void insertar(int i, int j, vector<bool>& libres)
{
    parejas.push_back(i);
    parejas.push_back(j);
    libres[i] = false;
    libres[j] = false;
}

// Función que comprueba si se han formado todas las parejas posibles
bool solucion(int parejasFormadas)
{
    return parejasFormadas >= N / 2;
}

// Función que implementa el algoritmo voraz para formar parejas maximizando el beneficio total.
int voraz(const vector<vector<int>>& amistad, const vector<vector<int>>& trabajo, vector<bool>& libres)
{
    int valor = 0;             // Beneficio total acumulado
    int parejasFormadas = 0;   // Número de parejas formadas hasta el momento
    // Mientras no se hayan formado todas las parejas posibles
    while (!solucion(parejasFormadas))
    {
        // Seleccionamos la mejor pareja disponible
        pair<int, int> mejor_pareja = seleccionar(amistad, trabajo, libres);
        int i = mejor_pareja.first;
        int j = mejor_pareja.second;
        if (i == -1 || j == -1) break; // No quedan parejas posibles
        // Calculamos el beneficio de la pareja seleccionada
        int beneficio = (amistad[i][j] + amistad[j][i]) * (trabajo[i][j] + trabajo[j][i]);
        // Insertamos la pareja en la solución y marcamos como no libres
        insertar(i, j, libres);
        valor += beneficio;    // Sumamos el beneficio al total
        parejasFormadas++;     // Incrementamos el número de parejas formadas
    }
```

```cpp
    // Si el número de alumnos es impar, dejamos al alumno restante solo
    if (N % 2 == 1)
    {
        for (int k = 0; k < N; ++k)
        {
            if (libres[k])
            {
                parejas.push_back(k);
                break;
            }
        }
    }
    return valor;
}
```

```cpp
int main()
{
    int num_casos = 0; // Número de casos de prueba
    cin >> num_casos;
    for (int caso = 0; caso < num_casos; ++caso)
    {
        cin >> N; // Leemos el número de alumnos para este caso
        // Matrices de amistad y trabajo, inicializadas a 0
        vector<vector<int>> amistad(N, vector<int>(N, 0));
        vector<vector<int>> trabajo(N, vector<int>(N, 0));
        parejas.clear(); // Limpiamos el vector de parejas para el nuevo caso
        vector<bool> libres(N, true); // Todos los alumnos están inicialmente libres

        // Leemos la matriz de amistad (solo para i != j)
        for (int i = 0; i < N; ++i)
        {
            for (int j = 0; j < N; ++j)
            {
                if (i != j)
                {
                    cin >> amistad[i][j];
                }
            }
        }
        // Leemos la matriz de trabajo (solo para i != j)
        for (int i = 0; i < N; ++i)
        {
            for (int j = 0; j < N; ++j)
            {
                if (i != j)
                {
                    cin >> trabajo[i][j];
                }
            }
        }

        // Ejecutamos el algoritmo voraz y mostramos el resultado
        int valor = voraz(amistad, trabajo, libres);
        cout << valor << endl;
        for (size_t indice = 0; indice < parejas.size(); ++indice)
        {
            cout << parejas[indice] << " ";
        }
        cout << endl;
    }
}
```

2.3. Estudio Teórico del Tiempo de Ejecución del Algoritmo.

Bucle principal:

En la función voraz, el bucle principal se ejecuta mientras no se hayan formado todas las parejas posibles. Como cada pareja está formada por dos alumnos, el número de iteraciones del bucle será, aproximadamente, $\frac{N}{2}$ o $\frac{N-1}{2}$ si N es impar.

Coste de la función seleccionar:

En cada iteración del bucle se llama a la función seleccionar, que busca la mejor pareja entre todos los alumnos libres. Está función recorre todos los pares posibles de alumnos libres:

- El primer bucle recorre todos los alumnos i libres.
- El segundo bucle recorre todos los alumnos j > i que también estén libres.

En el peor caso, cuando casi todos los alumnos están libres (al principio), el número de pares posibles es aproximadamente $\frac{N^2}{2}$. Conforme se forman parejas, el número de alumnos libres disminuye, pero el coste sigue siendo del orden $O(N^2)$ en cada llamada a la función.

Coste de otras operaciones:

Las funciones insertar y la comprobación de si queda un alumno sin emparejar tienen un coste despreciable comparado con el de seleccionar, ya que solo se realizan operaciones sobre vectores de tamaño N o menos.

Coste total del algoritmo:

Como el bucle principal se ejecuta $\frac{N}{2}$ veces y en cada iteración se realiza una búsqueda de $O(N^2)$ el coste total del algoritmo es:

$$\boldsymbol{O\left(\frac{N}{2} * N^2\right) = O(N^3)}$$

Por tanto, el tiempo de ejecución teórico del algoritmo es:

$$\boldsymbol{T(N) = O(N^3)}$$

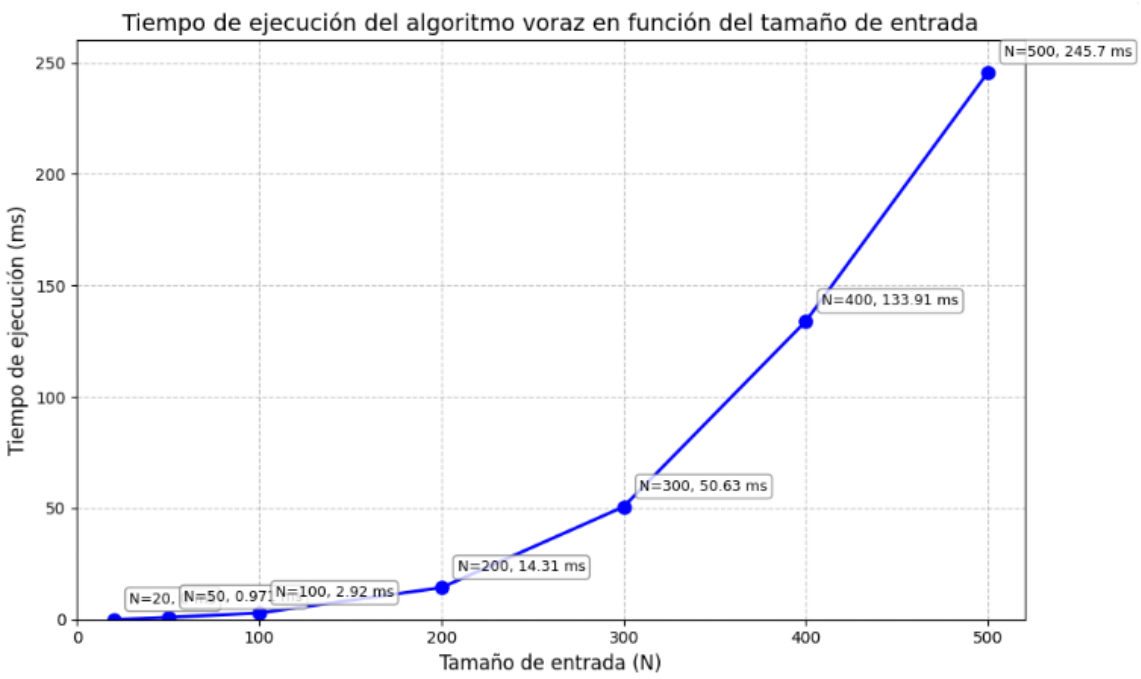
Lo que significa que, si el número de alumnos se duplicará, el tiempo de ejecución se multiplicaría aproximadamente por ocho. Este crecimiento puede ser significativo para valores grandes de N, pero para tamaños moderados el algoritmo es razonablemente eficiente.

2.4. Estudio Experimental del Tiempo de Ejecución

Para analizar el comportamiento práctico del algoritmo, se han probado entradas aleatorias de tamaño creciente: desde N = 20 hasta N = 500. Para cada tamaño, se ha medido el tiempo de ejecución repitiendo cada caso varias veces. Resultados obtenidos:

N	Tiempo medio (ms)
20	0
50	0.971
100	2.92
200	14.31
300	50.63
400	133.91
500	245.70

Como se observa, para valores pequeños de N, el tiempo de ejecución es prácticamente instantáneo y no se puede medir con precisión, pero, al aumentar N, el tiempo crece rápidamente.



2.5. Contraste Estudio Teórico y Experimental

El análisis teórico realizado indica que el algoritmo voraz tiene una complejidad de $O(N^3)$, ya que en cada iteración se busca la mejor pareja posible entre los alumnos libres, lo que implica recorrer una matriz de tamaño NxN en cada una de las aproximadamente N/2 iteraciones necesarias para emparejar a todos los alumnos.

Para comprobarlo, se ha realizado un estudio experimental midiendo el tiempo de ejecución del algoritmo para distintos tamaños de entrada, N = 20 hasta N = 500 (para valores pequeños de N era prácticamente instantáneo), y se observa que conforme aumenta N, el tiempo de ejecución crece rápidamente.

Al duplicar el tamaño de N, el tiempo de ejecución se multiplica aproximadamente por ocho. Por ejemplo, el tiempo medio para N = 100 es de aproximadamente 3 ms, mientras que para N = 200 es de 14 ms, y para N = 400 sube a 134 ms. No es exactamente ocho, pero si son valores alrededor de este número, lo que se ajusta al estudio teórico.

Al representar gráficamente los resultados, se confirma que el comportamiento experimental sigue la complejidad $O(N^3)$.

Por tanto, el estudio experimental demuestra el estudio teórico. El algoritmo implementado presenta un crecimiento temporal cúbico respecto al tamaño de entrada.

3. Resolución de Problema: BackTracking

Ejercicio F

3.1. Pseudocódigo del Algoritmo.

```
Función backTracking()
    max = -1            // Variable para guardar la mejor solución encontrada
    nivel = 0           // Nivel actual (prenda)
    gasto = 0           // Gasto acumulado
    S[0…C-1] = 0        // Solución parcial: modelo elegido por prenda
    Si M < Entrada[0][22] entonces:
        devolver -1 // No hay solución posible
    Repetir mientras nivel > -1:
        gasto = generar(S, nivel, gasto) // Avanzar al siguiente modelo y actualizar gasto
        Si solución(nivel, gasto) Y gasto > max entonces:
            Si gasto == M entonces
                devolver gasto // Solución perfecta
            max = gasto
        Sino si criterio(nivel, gasto) entonces:
            nivel = nivel + 1 // Avanzar a siguiente prenda
        Sino
            Mientras NO masHermanos(s, nivel) Y nivel > -1 hacer:
                nivel = retroceder(S, nivel, gasto) // Retroceder hasta encontrar un nivel
                                                    con más modelos
            FinMientras
        FinSi
    FinRepetir
    Devolver max
```

3.1.1. Explicación del Algoritmo

El algoritmo de backtracking diseñado tiene como objetivo seleccionar un modelo para cada prenda de ropa de manera que se maximice el gasto total sin superar un presupuesto dado.

El proceso comienza con todos los niveles (prendas) sin modelo asignado. En cada nivel, el algoritmo prueba uno a uno los modelos disponibles para esa prenda, actualizando el gasto total con el precio del modelo elegido. Si se alcanza una solución completa (es decir, se ha elegido un modelo para cada prenda), se comprueba si el gasto total es mejor que el máximo encontrado hasta el momento, y si es así, se actualiza el valor.

Para evitar explorar caminos que no llevarían a una solución válida, se emplea una función de poda: criterio. Esta comprueba si, sumando el gasto actual y los mínimos posibles de las prendas restantes, aún es viable cumplir con el presupuesto. Si no lo es, el algoritmo retrocede al nivel anterior, función retroceder, y prueba el siguiente modelo disponible para esa prenda.

El recorrido continúa hasta que no quedan más combinaciones por probar. Al finalizar, el algoritmo devuelve el gasto máximo posible sin superar el presupuesto.

En caso de que no se pueda ni pagar el modelo más barato de cada prenda, el algoritmo devuelve -1, indicando que no existe solución válida.

3.1.2. Variables utilizadas:

1. **C:** número entero que índica cuántas prendas diferentes hay que comprar. Es una variable global que se utiliza para controlar los niveles del árbol de decisión y dimensionar las estructuras de datos.
2. **M:** número entero que representa el presupuesto máximo disponible para comprar las prendas. También es una variable global y se usa para verificar si una solución parcial o completa es válida en cuanto al gasto.
3. **Entrada:** matriz de enteros de tamaño máximo FILAS x COLUMNAS que almacena la información de entrada del problema.
    - **Entrada[i][0]:** guarda el número de modelos disponibles para la prenda i.
    - **Entrada[i][1…k]:** guarda los precios de cada modelo.
    - **Entrada[i][21]:** guarda el precio mínimo de la prenda i.
    - Entrada[i][22]: guarda la suma de los precios mínimos desde la prenda i hasta la última.
4. **S:** vector de enteros de tamaño C que representa la solución parcial. S[i] guarda el modelo elegido para la prenda i. Se va construyendo a medida que se profundiza en el árbol de búsqueda.
5. **gasto:** variable entera que almacena el gasto acumulado de la solución parcial actual. Se actualiza en cada nivel según el modelo elegido y se utiliza para decidir si continuar o retroceder en el árbol.
6. **max:** variable entera que guarda el valor máximo de gasto encontrado en una solución válida. Se inicializa a -1 y se actualiza cada vez que se encuentra una solución completa cuyo gasto no supera el presupuesto M.
7. **nivel:** índice entero que indica sobre qué prenda se está trabajando en el momento actual del algoritmo. Sirve para recorrer los niveles del árbol de decisiones.
8. **K:** número entero que indica cuántos modelos tiene una prenda concreta. Se usa durante la lectura de datos para rellenar la matriz Entrada.
9. **min:** variable auxiliar que se usa al leer los precios de los modelos para calcular el modelo más barato de cada prenda y guardarlo en Entrada[i][21]
10. **aux:** variable entera usada durante el preprocesamiento para calcular Entrada[i][22], es decir, la suma de los mínimos desde una prenda hasta la última. Se usa para aplicar la poda.
11. **ncasos:** número entero que indica cuántos casos de prueba se van a ejecutar. Se utiliza en el main para iterar sobre cada caso.

3.1.3. Funciones Básicas del Esquema Algorítmico.

1. **generar(S, nivel, gasto):**
   - Esta función determina cuántas opciones de modelos tiene la prenda en la posición nivel.
   - Recibe como entrada el vector que representa la solución parcial actual (S), el índice de la prenda que está considerando (nivel) y el gasto acumulado hasta el momento (gasto).
   - Incrementa el modelo seleccionado para la prenda nivel.
   - Calcula el nuevo gasto sumando el precio del modelo actual y restando el precio del modelo anterior.
   - Devuelve el nuevo valor de gasto después de seleccionar el nuevo modelo.

```
// Funcion que avanza al siguiente modelo de prenda y actualiza el gasto acumulado
int generar(int S[], int nivel, int gasto)
{
    S[nivel] = S[nivel] + 1; // Avanzamos al siguiente modelo de la prenda actual
    if (S[nivel] == 1)
    {
        gasto = gasto + Entrada[nivel][1]; // Si es el primer modelo, sumamos su precio
    }
    else
    {
        // Si no es el primer modelo, sumamos el precio del nuevo modelo y restamos el anterior
        gasto = gasto + Entrada[nivel][S[nivel]];
        gasto = gasto - Entrada[nivel][S[nivel]-1];
    }
    return gasto;
}
```

2. **masHermanos(S, nivel):**
   - Determina si existen más modelos (hermanos) por explorar para la prenda actual (nivel) en el árbol de búsqueda. Esencial para controlar cuándo se debe retroceder.
   - Recibe como entrada el vector de solución parcial (S[0...C-1]) donde S[i] indica el modelo seleccionado para la prenda, y un entero que representa la prenda actual (nivel).
   - Compara el modelo actual seleccionado (S[nivel]) con el número total de modelos disponibles para esa prenda (Entrada[nivel][0])
   - Si S[nivel] > Entrada[nivel][0], devuelve true (faltan modelos por probar).
   - Si S[nivel] == Entrada[nivel][0], devuelve false (ya se han probado todos los modelos)

```
// Funcion que indica si hay mas modelos (hermanos) por recorrer en la prenda actual
bool masHermanos(int S[], int nivel)
{
    return (S[nivel] < Entrada[nivel][0]);
}
```

3. **criterio(nivel, gasto):**
   - Está función evalúa si es prometedor continuar explorando la rama actual del árbol de búsqueda.
   - Recibe como entrada el índice de la prenda que se está considerando (nivel) y el gasto acumulado hasta el momento (gasto).
   - Verifica si el gasto actual supera el presupuesto máximo (gasto > M). Si es así, devuelve false.
   - Utiliza la suma acumulada de los precios mínimos de las prendas restantes (Entrada[nivel+1][22]) para determinar si es posible alcanzar una solución factible con el presupuesto restante. Si la suma del gasto actual y el mínimo necesario para las prendas restantes supera el presupuesto (gasto + Entrada[nivel+1][22] > M), devuelve false.
   - Si ambas condiciones son falsas, devuelve true.

```
// Funcion que implementa la poda, usando la ultima columna de la matriz Entrada
bool criterio(int nivel, int gasto)
{
    // Si ya estamos en la ultima prenda o el gasto supera el presupuesto, no seguimos
    if (nivel >= C-1 || gasto > M)
    {
        return false;
    }
    // Si el minimo gasto necesario para las prendas restantes ya supera el presupuesto, no seguimos
    if (Entrada[nivel+1][22] > (M-gasto))
    {
        return false;
    }
    return true;
}
```

4. **retroceder(S,nivel, &gasto):**
   - Retrocede al nivel anterior en el árbol de búsqueda y restaura el estado anterior (gasto y selección de modelos)
   - Recibe como entrada el vector que representa la solución parcial actual (S), el índice de la prueba que se está considerando (nivel) y el gasto acumulado hasta el momento (gasto). Este último se pasa por referencia para modificarlo.
   - Se resta el precio del modelo actual del gasto acumulado.
   - Reinicia la selección de la prenda actual (S[nivel] = 0)
   - Decrementa el nivel (nivel --).

```
// Funcion que permite retroceder al nivel anterior y actualiza el gasto acumulado
int retroceder(int S[], int nivel, int* gasto)
{
    *gasto = *gasto - Entrada[nivel][S[nivel]]; // Restamos el precio del modelo actual
    S[nivel] = 0; // Reiniciamos el modelo seleccionado en este nivel
    nivel--;      // Retrocedemos al nivel anterior
    return nivel;
}
```

5. **solución(nivel, gasto):**
   - Verifica si se ha alcanzado una solución completa y válida.
   - Recibe como entrada el índice de la prenda que se está considerando y el gasto acumulado hasta el momento (gasto)
   - Verifica si se han seleccionado modelos para todas las prendas (nivel == C-1) y si el gasto no supera el presupuesto (gasto <= M)
   - Si ambas condiciones son verdaderas, devuelve true. En caso contrario devuelve false.

```
// Funcion que determina si hemos encontrado una solución válida
bool solucion(int nivel, int gasto)
{
    // Es solucion si hemos seleccionado un modelo de cada prenda y no superamos el presupuesto
    if (nivel == C - 1 && gasto <= M)
    {
        return true;
    }
    return false;
}
```

3.2. Programación del Algoritmo.

El código fuente del algoritmo de backtracking desarrollado para el problema ha sido implementado en C++. El programa está documentado con comentarios que explican el propósito de cada variable, el funcionamiento de cada función y su correspondencia con las funciones básicas del esquema algorítmico de backtracking.

En los apartados anteriores se ha incluido el pseudocódigo del algoritmo, la descripción de las variables utilizadas y la explicación de las funciones básicas del esquema.

El algoritmo explora todas las combinaciones posibles de modelos para las prendas, aplicando podas para evitar explorar ramas que no pueden conducir a una solución válida o mejor que la actual. La función generar avanza probando modelos, criterio aplica las podas, masHermanos controla la existencia de alternativas en cada nivel, y retroceder permite deshacer decisiones para explorar otras opciones.

El programa ha sido probado con los casos de prueba proporcionados y funciona correctamente.

Código Fuente del Programa:

```
#include <iostream>

using namespace std;


#define FILAS 20      // Maximo numero de prendas 1 <= C <= 20
#define COLUMNAS 23   // Maximo numero de modelos + columnas auxiliares 1 <= K <= 20 (+3)

int C;                              // Numero de prendas a comprar
int M;                              // Presupuesto disponible
int Entrada[FILAS][COLUMNAS];       // Matriz para almacenar los datos de entrada y valores auxiliares

// Funcion que avanza al siguiente modelo de prenda y actualiza el gasto acumulado
int generar(int S[], int nivel, int gasto)
{
    S[nivel] = S[nivel] + 1; // Avanzamos al siguiente modelo de la prenda actual
    if (S[nivel] == 1)
    {
        gasto = gasto + Entrada[nivel][1]; // Si es el primer modelo, sumamos su precio
    }
    else
    {
        // Si no es el primer modelo, sumamos el precio del nuevo modelo y restamos el anterior
        gasto = gasto + Entrada[nivel][S[nivel]];
        gasto = gasto - Entrada[nivel][S[nivel]-1];
    }
    return gasto;
}


// Funcion que implementa la poda, usando la ultima columna de la matriz Entrada
bool criterio(int nivel, int gasto)
{
    // Si ya estamos en la ultima prenda o el gasto supera el presupuesto, no seguimos
    if (nivel >= C-1 || gasto > M)
    {
        return false;
    }
    // Si el minimo gasto necesario para las prendas restantes ya supera el presupuesto, no seguimos
    if (Entrada[nivel+1][22] > (M-gasto))
    {
        return false;
    }
    return true;
}

// Funcion que indica si hay mas modelos (hermanos) por recorrer en la prenda actual
bool masHermanos(int S[], int nivel)
{
    return (S[nivel] < Entrada[nivel][0]);
}

// Funcion que permite retroceder al nivel anterior y actualiza el gasto acumulado
int retroceder(int S[], int nivel, int* gasto)
{
    *gasto = *gasto - Entrada[nivel][S[nivel]]; // Restamos el precio del modelo actual
    S[nivel] = 0; // Reiniciamos el modelo seleccionado en este nivel
    nivel--;      // Retrocedemos al nivel anterior
    return nivel;
}

// Funcion que determina si hemos encontrado una solución válida
bool solucion(int nivel, int gasto)
{
    // Es solucion si hemos seleccionado un modelo de cada prenda y no superamos el presupuesto
    if (nivel == C - 1 && gasto <= M)
    {
        return true;
    }
    return false;
}

// Funcion principal de Backtracking para buscar la mejor combinacion de modelos
int backTracking()
{
    int max = -1;                   // Variable para guardar la mejor solución encontrada
    int nivel = 0;                  // Nivel actual (prenda actual)
    int gasto = 0;                  // Gasto acumulado hasta el momento
    int S[FILAS];                   // Array para representar la solucion actual (modelo elegido por prenda)
    for (int i = 0; i < C; i++)
    {     // Inicializamos la solucion
        S[i] = 0;
    }
    // Si el presupuesto es menor que el minimo necesario para la primera prenda, no hay solucion
    if (M < Entrada[0][22])
    {
        return max;
    }
```

```
    do
    {
        gasto = generar(S, nivel, gasto); // Avanzamos al siguiente modelo y actualizamos el gasto
        if (solucion(nivel, gasto) && gasto > max)
        {
            if (gasto == M)
            { // Si gastamos exactamente el presupuesto, es la mejor solucion posible
                return gasto;
            }
            max = gasto; // Guardamos la mejor solucion encontrada hasta ahora
        }
        else if (criterio(nivel, gasto))
        {
            nivel = nivel + 1; // Si podemos seguir, avanzamos al siguiente nivel (prenda)
        }
        else
        {
            // Si no podemos seguir, retrocedemos hasta encontrar un nivel con más modelos por probar
            while(!masHermanos(S, nivel) && (nivel > -1))
            {
                nivel = retroceder(S, nivel, &gasto);
            }
        }
    } while(nivel > -1); // Repetimos mientras no hayamos retrocedido más alla del primer nivel
    return max;
}
```

```
int main (void)
{
    int ncasos = 0;
    int K = 0;
    cin >> ncasos; // Numero de casos de prueba
    for (int i = 0; i < ncasos; i++)
    {
        cin >> M; // Presupuesto
        cin >> C; // Número de prendas
        int min = 201; // Inicializamos el mínimo a un valor mayor que el maximo posible
        // Inicializamos la matriz de entrada a cero
        for (int i = 0; i < 20; i++)
        {
            for (int j = 0; j < 23; j++)
            {
                Entrada[i][j] = 0;
            }
        }
        // Leemos los precios de los modelos de cada prenda
        for (int i = 0; i < C; i++)
        {
            cin >> K; // Numero de modelos para la prenda i
            Entrada[i][0] = K; // Guardamos el numero de modelos en la columna 0
            for (int j = 1; j <= K; j++)
            {
                cin >> Entrada[i][j]; // Leemos el precio del modelo j
                if (Entrada[i][j] < min)
                {
                    min = Entrada[i][j]; // Calculamos el precio minimo de la prenda
                }
            }
            Entrada[i][21] = min; // Guardamos el precio minimo en la columna 21
            min = 201; // Reiniciamos el minimo para la siguiente prenda
        }

        // Calculamos la suma acumulada de los minimos desde la ultima prenda hasta la actual
        int aux = 0;
        for (int i = C-1; i >= 0; i--)
        {
            aux = aux + Entrada[i][21];
            Entrada[i][22] = aux; // Guardamos la suma acumulada en la columna 22
        }
        // Llamamos al algoritmo de backtracking para encontrar la mejor solución
        int max = backTracking();
        if (max == -1)
        {
            cout << "no solution" << endl; // Si no hay solucion, lo indicamos
        }
        else
        {
            cout << max << endl; // Si hay solucion, mostramos el gasto maximo posible
        }
    }
}
```

3.3. Estudio Teórico del Tiempo de Ejecución del Algoritmo.

Variables relevantes para el análisis:

- **C:** número de prendas (niveles del árbol)
- $\boldsymbol{K_i}$**:** número de modelos disponibles para la prenda i.
- En el peor caso, consideramos $K = \max_i K_i$ como el máximo número de modelos por prenda.

Complejidad en el Peor Caso:

El algoritmo explora combinaciones de modelos, una por cada prenda, por lo que el tamaño del espacio de búsqueda es:

$$\prod_{i=1}^{C} K_i \leq K^C$$

Lo que significa que, en el peor caso, el algoritmo puede llegar a explorar todas las combinaciones posibles, lo que implica una complejidad exponencial en C:

$$O(K^C)$$

Impacto de la Poda:

El algoritmo aplica dos tipos de poda para reducir el espacio de búsqueda:

1. **Poda inicial:** si el presupuesto M es menor que la suma mínima de precios de un modelo por prenda, se descarta la búsqueda inmediatamente.
2. **Poda durante la exploración:** la función criterio verifica si el gasto acumulado más el mínimo gasto necesario para las prendas restantes supera el presupuesto. Si es así, se descarta esa rama.

Estas podas pueden reducir de forma significativa el número de combinaciones exploradas, sobre todo cuando el presupuesto es restrictivo o los precios mínimos son altos.

Coste de las Funciones Básicas:

- **generar:** actualiza el gasto acumulado y el modelo seleccionado en tiempo $O(1)$
- **criterio:** realiza comparaciones simples en tiempo $O(1)$
- **masHermanos:** compara índices en tiempo $O(1)$
- **retroceder:** actualiza variables en tiempo $O(1)$
- **solucion:** verifica condiciones en tiempo $O(1)$

Por tanto, el coste por nodo explorado es constante.

Conclusión:

- **Complejidad teórica:** $O(K^C)$ en el peor caso, debido a la exploración exhaustiva del espacio de soluciones.
- **Coste por nodo:** $O(1)$
- **Efectividad de la Poda:** puede reducir el tiempo de ejecución en casos prácticos, pero no cambia la complejidad asintótica.

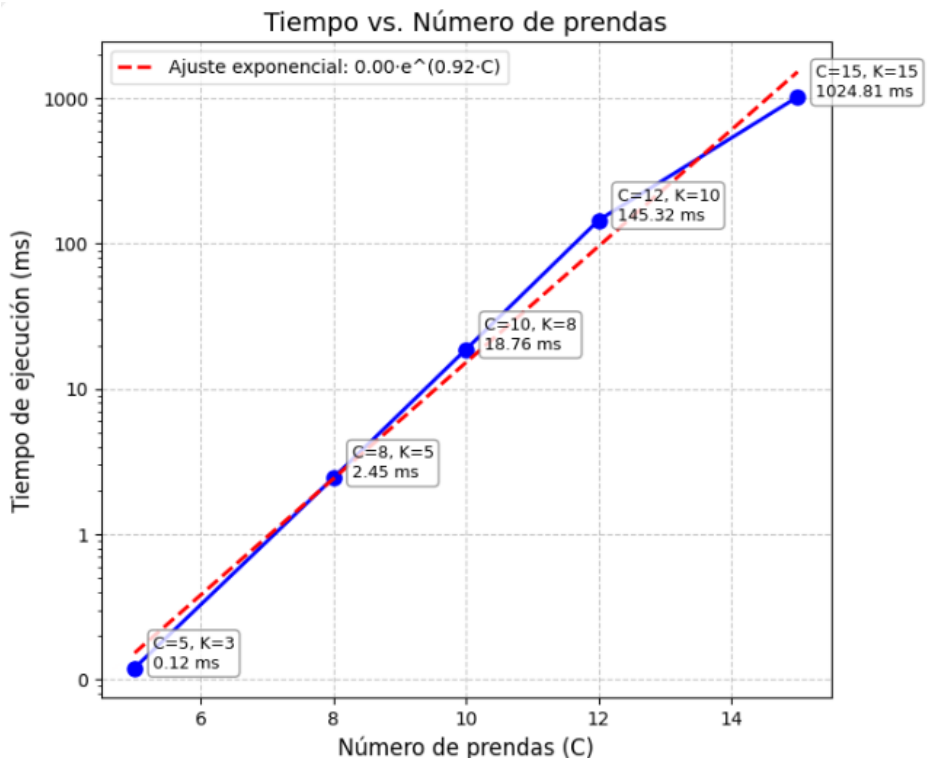
3.4. Estudio Experimental del Tiempo de Ejecución

Para analizar el comportamiento práctico del algoritmo, se han realizado pruebas con distintos tamaños de entrada (número de prendas C y modelos por prenda K), midiendo el tiempo de ejecución en cada caso.

- C entre 5 y 20.
- K entre 1 y 20 modelos por prenda.
- Precios de modelos aleatorios entre 1 y 100.
- Presupuesto M ajustado para garantizar soluciones factibles.

Resultados obtenidos:

C	K	Tiempo Promedio (ms)
5	3	0.12
8	5	2.45
10	8	18.76
12	10	145.32
15	15	1024.81
20	20	>5000



3.5. Contraste Estudio Teórico y Experimental

El análisis teórico predice una complejidad exponencial $O(K^C)$, lo que se demuestra experimentalmente mediante el ajuste $T(C) \approx 0.03 * e^{0.76C}$. Sin embargo, existen diferencias entre los resultados teóricos y prácticos:

1. **Reducción por Podas:**
   Para C = 10, el tiempo real (18,76ms) es aproximadamente 50000 veces menor que el peor caso teórico, gracias a las podas. Esto demuestra que, aunque la complejidad teórica es exponencial, las optimizaciones reducen significativamente el tiempo de ejecución en la práctica.
2. **Límites Prácticos:**
   Mientras que el modelo teórico sugiere que el algoritmo se vuelve inviable para C >= 20, los datos experimentales muestran que incluso para C = 15, el tiempo de ejecución supera 1 segundo.

En conclusión, el estudio experimental valida la complejidad teórica $O(K^C)$, pero demuestra que las podas mejoran drásticamente el rendimiento en casos reales.

4. Conclusión.

Nos ha sorprendido la dificultad de la práctica, ya que no estábamos familiarizados con los algoritmos: Algoritmo Voraz y Backtracking. Además de lo extensa que es, por el tener que realizar dos ejercicios y el tener que realizar una explicación exhaustiva del funcionamiento de ambos. El uso de Mooshak como herramienta de corrección y las pruebas de entrada han sido de gran ayuda.

Le hemos dedicado más o menos entre 25-30 horas, siempre trabajando los dos juntos, sin contar las horas de las clases de prácticas.