

Tema 6: Vuelta atrás

Clara María Segura Díaz
Fundamentos de Algoritmia, Curso 2020-21

Dpto. de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

- **Estructuras de datos y métodos algorítmicos: ejercicios resueltos.** N. Martí , Y. Ortega, J.A. Verdejo. Pearson-Prentice Hall, 2004.
Capítulo 14
- **Foundations of algorithms.** R. E. Neapolitan. Jones & Bartlett Learning, Fifth Edition, 2015.
Capítulo 5

VUELTA ATRÁS

- Esquema vuelta atrás

- Coloreado de un mapa
- Cadenas de caracteres

Cual es el esquema general de los problemas de vuelta atrás.

Problema de colorear un mapa de paises

Y desmarcaje es muy importante.

- Vuelta atrás con marcaje

- Problema de las n reinas

Problema de como colocar n reinas en un tablero de forma que no se amenacen entre si.

Una en cada fila y el tablero a de ser NxN

- Encontrar una solución

- Dominó

Dominó circular.

Maximización

- Problemas de optimización

- Problema del viajante

Minimización

- Problema de la mochila

Minimización

Nos sirvió de base para hacer el problema 3 del último examen.

- Funcionarios con trabajo

Maximización

n funcionarios deben hacer n trabajos



Esquema vuelta atrás

Búsqueda en el espacio de soluciones

- No siempre podemos utilizar las técnicas vistas hasta ahora para lograr soluciones eficientes. **No nos basta con utilizar Dv**
 - El último recurso que nos queda para resolver nuestro problema es aplicar la **"fuerza bruta"**. Consiste en probar cada opción del problema de manera manual sin hacer ningún descarte por ejemplo. Probar todas las sols, sin descartes
- Realizar una búsqueda exhaustiva por el espacio de posibles soluciones hasta encontrar una que satisfaga los criterios exigidos.

Explicación de la fuerza bruta.

Búsqueda en el espacio de soluciones

Es decir, la recursión normal o el divide y vencerás no es la mejor opción para todo lo que hacemos, ya que para estos problemas necesitaremos descartar soluciones probando. (A eso se le llama poda)

- No siempre podemos utilizar las técnicas vistas hasta ahora para lograr soluciones eficientes.
- El último recurso que nos queda para resolver nuestro problema es aplicar la **"fuerza bruta"**.
- Realizar una búsqueda exhaustiva por el espacio de posibles soluciones hasta encontrar una que satisfaga los criterios exigidos.
- Esta búsqueda exhaustiva resultará impracticable si el cardinal del conjunto de soluciones posibles es muy grande, lo cual ocurre muy a menudo.

A menos que el número de soluciones a explorar fuera viable. (P-ej 10 soluciones)

No podremos utilizar la fuerza bruta ya que costará mucho. Hay muchas sols a explorar.

Búsqueda en el espacio de soluciones

Es decir, la recursión normal o el divide y vencerás no es resultado para todo lo que hacemos, ya que para estos problemas necesitaremos descartar soluciones probando.

- No siempre podemos utilizar las técnicas vistas hasta ahora para lograr soluciones eficientes.

Esto se refiere a con lo que nosotros sabemos.

- El último recurso que nos queda para resolver nuestro problema es aplicar la

"fuerza bruta"

Realizar fuerza bruta consiste en hacer lo de abajo

- Realizar una búsqueda exhaustiva por el espacio de posibles soluciones hasta encontrar una que satisfaga los criterios exigidos.

- Esta búsqueda exhaustiva resultará impracticable si el cardinal del conjunto de soluciones posibles es muy grande, lo cual ocurre muy a menudo.

Es impracticable porque habrán millones de posibilidades.

- El esquema algorítmico de **vuelta atrás** es una mejora a la estrategia de **fuerza bruta**, ya que la búsqueda se realiza de manera estructurada, descartando grandes bloques de soluciones para reducir dicho espacio de búsqueda. Aumenta la eficiencia

- Al algoritmo básico de **vuelta atrás** se le pueden aplicar una serie de mejoras para hacer más efectiva la **poda**, convirtiéndolo en una herramienta más eficaz para atacar una gran cantidad de problemas, que de otro modo serían inabordables.

RAMAS

Técnica que consiste en evitar la exploración de ramas de búsqueda que no conducen a soluciones validas. Dos tipos de podas según el chat gpt:

Poda de factibilidad: para eliminar ramas que no dirigen a una solución factible.

Poda de optimidad: se utiliza para eliminar una rama que no conduce a una solución óptima.

Veremos ambas



Vuelta atrás vs. fuerza bruta

Además podemos llegar a soluciones que no sabemos si son factibles.

- En la vuelta atrás las soluciones se forman de manera progresiva generando soluciones parciales, comprobando en cada paso si la solución que se está construyendo puede conducir a una solución satisfactoria.
- Sin embargo, en la fuerza bruta la estrategia consiste en probar una solución potencial completa tras otra sin ningún criterio. → Por ello no debemos de utilizar la fuerza bruta
Es muy caro, poco eficiente
- En la vuelta atrás, si una solución parcial no puede llevar a una solución completa satisfactoria, la búsqueda se aborta, evitando así explorar todo el subárbol de búsqueda derivado de una solución parcial no prometedora.
- Cuando se aborta una búsqueda, se vuelve a una solución parcial viable, deshaciendo decisiones previas.
- Debido a este salto hacia atrás, el esquema algorítmico se conoce como vuelta atrás.

Ya que deshacemos una rama volvemos atrás y comprobamos otras ramas.
Por eso el algoritmo se llama vuelta atrás.

Coloreado de un mapa

1er problema

colores

Dado un mapa M con m países y un número $n > 0$ se pide encontrar las formas de colorear los países de M utilizando un máximo de n colores, de tal manera que ningún par de países fronterizos tenga el mismo color.

Hay un vector de países y otro vector de colores. De tal forma que Madrid y Toledo no se pueden pintar del mismo color.

Coloreado de un mapa

- Dado un mapa M con m países y un número $n > 0$ se pide encontrar las formas de colorear los países de M utilizando un máximo de n colores, de tal manera que ningún par de países fronterizos tenga el mismo color.
- Una forma de resolver el problema consiste en generar todas las posibles maneras de colorear el mapa y después desechar aquellas en que dos países fronterizos tienen el mismo color.

Esto no es nada eficiente
Usaremos la vuelta atrás.

Esto es la fuerza bruta, luego hacer esto no es viable, es carísimo.

Coloreado de un mapa

- Dado un mapa M con m países y un número $n > 0$ se pide encontrar las formas de colorear los países de M utilizando un máximo de n colores, de tal manera que ningún par de países fronterizos tenga el mismo color.

Una forma de resolver el problema consiste en generar todas las posibles maneras de colorear el mapa y después desechar aquellas en que dos países fronterizos tienen el mismo color. (FUERZA BRUTA) → No es viable

Pero esto es aceptable solo si n y m son pequeños: el número de posibles formas de colorear el mapa viene dado por las variaciones con repetición de n elementos tomados de m en m , $VR_n^m = n^m$:

n : colores	m : países	nº de posibilidades
3	17	129.140.163
5	17	762.939.453.125
3	50	717.897.987.691.852.588.770.249

Imagínate que por fuerza bruta tengamos que probar todas esas posibilidades. Sería muy costoso.

Probar esta cantidad de casos es prácticamente imposible, ineficiente y por tanto no realizable. Por ello hacemos la vuelta atrás. En ese campo de soluciones se encuentran por ejemplo que todos estén coloreados de un mismo color lo cual no se puede. Tendríamos que hacer una poda de factibilidad.

Coloreado de un mapa

- Dado un mapa M con m países y un número $n > 0$ se pide encontrar las formas de colorear los países de M utilizando un máximo de n colores, de tal manera que ningún par de países fronterizos tenga el mismo color.
- Una forma de resolver el problema consiste en generar todas las posibles maneras de colorear el mapa y después desechar aquellas en que dos países fronterizos tienen el mismo color.  UTILIZAR LA FUERZA BRUTA
- Pero esto es aceptable solo si n y m son pequeños: el número de posibles formas de colorear el mapa viene dado por las variaciones con repetición de n elementos tomados de m en m , $VR_n^m = n^m$:

n : colores	m : países	nº de posibilidades
3	17	129.140.163
5	17	762.939.453.125
3	50	717.897.987.691.852.588.770.249

- Para reducir el espacio de búsqueda, cuando vayamos a elegir un color c para un país, comprobemos que todos los países que tienen frontera con él que ya estén coloreados no lo están con ese color y solo entonces seguiremos con ese coloreado.



De esta forma podemos descartar muchas opciones.

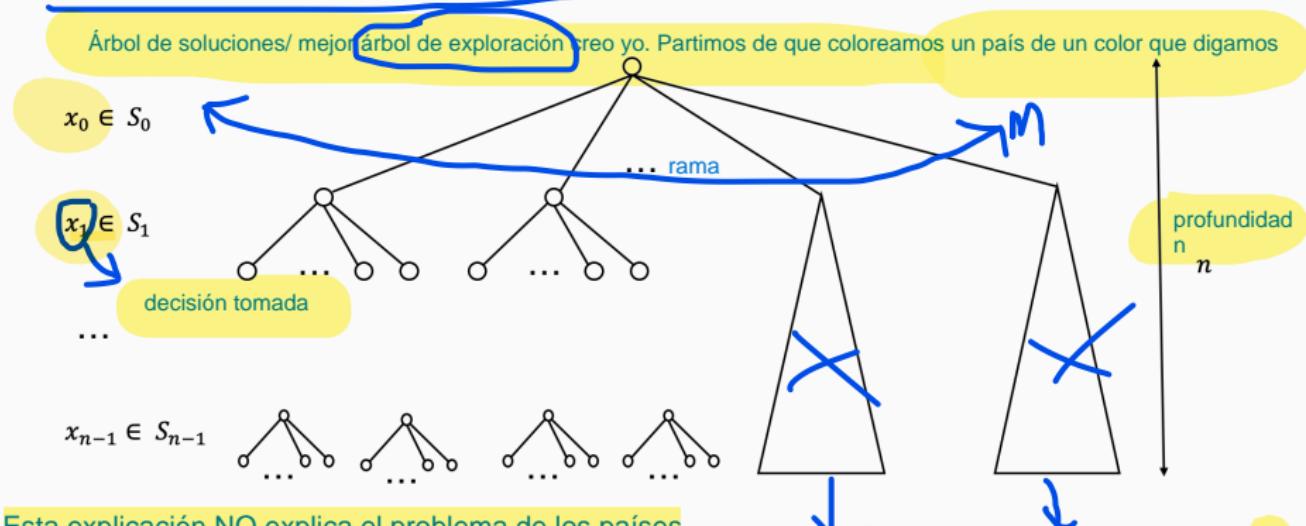
Poda de factibilidad. Exploramos solo aquellas ramas que nos lleven a una solución factible.

Definición del espacio de soluciones

- Consideraremos problemas cuyas soluciones sean construibles por etapas.
- Una solución será expresable en forma de **n -tupla** (x_0, \dots, x_{n-1}) , donde cada $x_i \in S_i$ representa la decisión tomada en la etapa i -ésima.

Definición del espacio de soluciones

- Consideraremos problemas cuyas soluciones sean construibles por etapas.
- Una solución será expresable en forma de n -tupla (x_0, \dots, x_{n-1}) , donde cada $x_i \in S_i$ representa la decisión tomada en la etapa i -ésima.



Esta explicación NO explica el problema de los países.

El triángulo significa que descartamos esa solución bien porque la solución no será factible o bien porque la solución no será la óptima (veremos después las estimaciones)

Definición del espacio de soluciones

De aquí lo importante son los tipos de restricciones que las tendremos que poner en nuestros problemas.

Por eso habrá problemas de maximización/minimización o que cumpla algo

- Además, una solución habrá de minimizar, maximizar o simplemente satisfacer una cierta **función criterio**. La solución debe cumplir lo que diga la función criterio.
- Se establecen entonces dos categorías de **restricciones**:

las **restricciones explícitas**, que indican los conjuntos S_i a los que pertenecen cada una de las componentes de una tupla solución;

Una restricción explícita del problema de países y colores puede ser que el número de colores sea >0

las **restricciones implícitas**, que son las relaciones que se han de establecer entre las componentes de la tupla solución para **satisfacer la función criterio**.

Una implícita podría ser que dos países contiguos no se pueden colorear del mismo color

Por ejemplo, si estás resolviendo un problema de asignación de recursos, las restricciones explícitas podrían ser límites de recursos disponibles en cada categoría o ubicación.

Por ejemplo, si estás optimizando la logística de envío de productos, una restricción implícita podría ser que el tiempo de entrega no supere cierto límite, lo que implica una relación entre la velocidad de entrega y la ruta seleccionada.

Si no me equivoco estas son las más importantes porque son las que tenemos que especificar en nuestro problema

Definición del espacio de soluciones

- El **espacio de soluciones** estará formado por el conjunto de tuplas que satisfacen las **restricciones explícitas** y se puede estructurar como un **árbol de exploración** donde en cada nivel se toma la decisión sobre la etapa correspondiente.

Definición del espacio de soluciones

Importante en nuestros ejercicios hacer uso de estos arboles de exploración.

- El **espacio de soluciones** estará formado por el conjunto de tuplas que satisfacen las restricciones explícitas, y se puede estructurar como un **árbol de exploración** donde en cada nivel se toma la decisión sobre la etapa correspondiente.
- Un elemento adicional imprescindible es la **función de poda** o **test de factibilidad**, que permite determinar cuando una solución parcial puede conducir a una solución satisfactoria. *Ya que habrá soluciones que no cumplen la función de criterio específica del problema que estemos resolviendo.*
- De tal manera, si un *nodo* no satisface la función de poda es inútil continuar la búsqueda por esa rama del árbol. La **función de poda** permite pues reducir la búsqueda en el árbol de exploración.

la poda

Reduce el número de posibilidades de un problema.

vuelta atrás.

Si no me equivoco la función de poda sirve para que se cumplan las restricciones implícitas del problema.

- Una vez definido el árbol de exploración, el algoritmo para resolver el problema consistirá en realizar un **recorrido del árbol** en cierto orden.
- Durante el proceso, para cada nodo se irán generando sus sucesores; así denominaremos **nodos vivos** a aquellos para los cuales todavía no se han generado todos sus hijos; **nodos en expansión** a aquellos cuyos hijos están siendo generados; y **nodos muertos** a aquellos que no pueden ser expandidos, bien porque no hayan superado el test de factibilidad, o bien porque todos sus hijos ya han sido generados.

- Una vez definido el árbol de exploración, el algoritmo para resolver el problema consistirá en realizar un **recorrido del árbol** en cierto orden.
- Durante el proceso, para cada nodo se irán generando sus sucesores; así denominaremos **nodos vivos** a aquellos para los cuales todavía no se han generado todos sus hijos; **nodos en expansión** a aquellos cuyos hijos están siendo generados; y **nodos muertos** a aquellos que no pueden ser expandidos, bien porque no hayan superado el test de factibilidad, o bien porque todos sus hijos ya han sido generados.

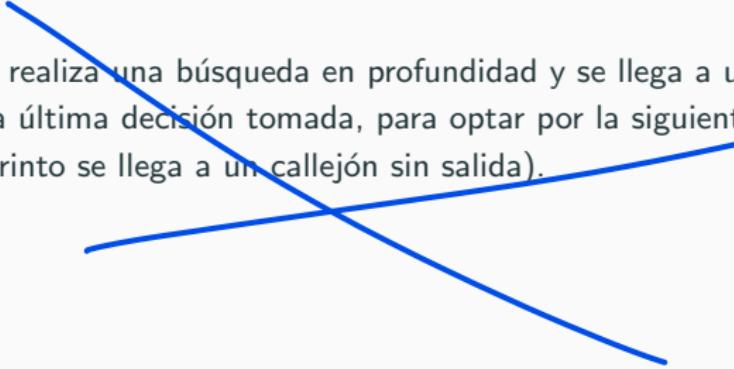
- Se pueden estudiar dos recorridos:

Vuelta atrás: el recorrido se realiza en profundidad, de forma que los nodos vivos se gestionan mediante una pila. El método resulta sencillo y eficiente en espacio.

Ramificación y poda: corresponde a una búsqueda más “inteligente”, donde en cada momento se expande el nodo vivo más “prometedor”, de forma que los nodos vivos se gestionan mediante una cola con prioridad.

Esquema general de vuelta atrás

Cuando se realiza una búsqueda en profundidad y se llega a un nodo muerto, hay que deshacer la última decisión tomada, para optar por la siguiente alternativa (como cuando en un laberinto se llega a un callejón sin salida).



Esquema general de vuelta atrás

Cuando se realiza una búsqueda en profundidad y se llega a un nodo muerto, hay que deshacer la última decisión tomada, para optar por la siguiente alternativa (como cuando en un laberinto se llega a un callejón sin salida).

```
vueltaAtras (Tupla & sol, int k) {  
    prepararRecorridoNivel(k);  
    while (!ultimoHijoNivel(k)){  
        sol[k] = siguienteHijoNivel(k); // comprobamos el siguiente nivel de la rama  
        if (esValida(sol, k)){ // si es valida entonces ver si es solución posible  
            if (esSolucion(sol, k))  
                { tratarSolucion(sol); }  
            else  
                { vueltaAtras(sol, k + 1); }  
        }  
    }  
}
```

Mientras que no llegemos a una solución

si es valida entonces ver si es solución posible

si no volver atrás

Esquema vuelta atrás

Coloreado de un mapa

Cómo realizar el problema de coloreado de un mapa

Teniendo las siguientes restricciones:

Hay n colores con $n > 0$. Explícita

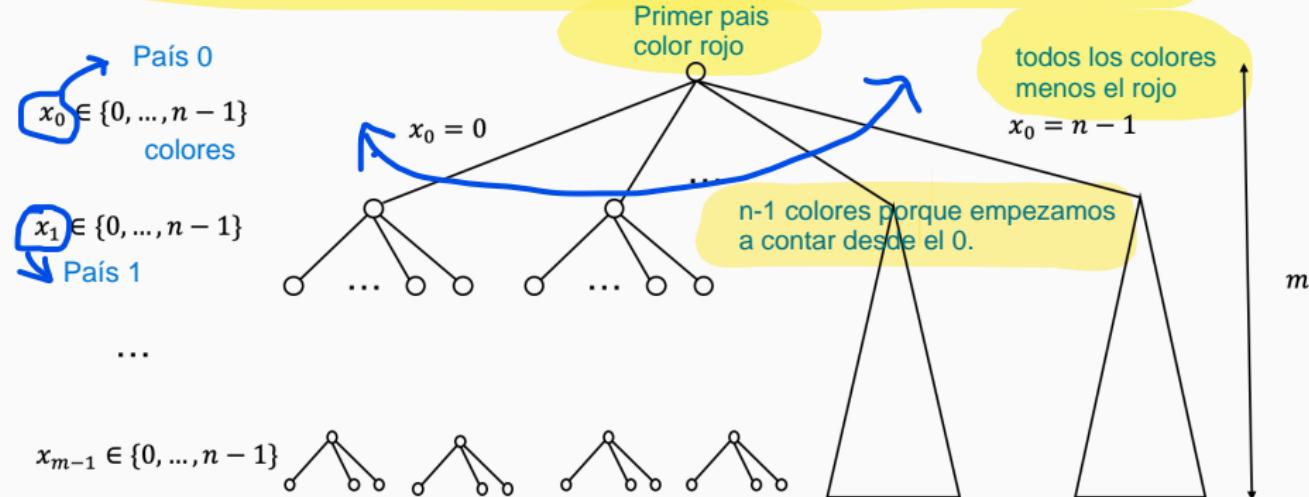
Hay m países. Explícita

Dos países contiguos no se pueden colorear del mismo color (Imp)

Coloreado de un mapa

Consistía en colorear un mapa de M países con un total de n colores de tal forma que dos países contiguos no pueden ser del mismo color.

A cada país m le asignaremos un color n . Al primer país, por ejemplo le asignamos el primer color.



Por ejemplo, el primer país es España, y lo coloreamos de rojo. Entonces Portugal lo podremos colorear de todos los colores menos el rojo, ya que no puede haber dos países del mismo color.

El árbol de exploración tiene profundidad m y rama n, de forma que en cada etapa tomamos la decisión de asignarle un color factible a cada país.

Coloreado de un mapa

Función que comprueba si la solución que estamos explorando es posible

```
bool esValida(bool mapa[MAX][MAX], int sol[], int k){  
    int j=0;           mapa  
    while (j < k && (!mapa[j][k] || sol[j] != sol[k])) {  
        j = j + 1; }  Recorre las regiones ya coloreadas(j<k)  
    return j==k;      verifica si son adyacentes  
}  
  
void colorear(bool mapa[MAX][MAX], int sol[], int k, int n, int m){  
    for(int c = 0; c < n; c++){  
        sol[k] = c;      pais que comprobamos n colores  
        if(esValida(mapa, sol, k)) {  m paises  
            if(k == m-1) Si es el último país.  
                { tratarSolucion(sol, m); }  
            else  
                { colorear(sol, k + 1, n, m); }  
        }  
    }  
}
```

Al país k le asignamos el color c.

Aún no realizamos marcaje ni desmarcaje.

Si no es valido asigna a ese país otro color hasta que sea válido.

Esquema vuelta atrás

Cadenas de caracteres

Creo que es otro problema distinto al de los países.

EFECTIVAMENTE.

Cadenas de caracteres

Dadas n letras, todas ellas diferentes, y $m \leq n$, queremos calcular todas las palabras con m letras diferentes escogidas entre las dadas.

n = abecedario que se utilice

m = el número de letras que podrán tener las palabras que formemos.

Cuántas palabras se pueden formar.

Yo diría que las restricciones en este problema son:

El número de letras del abecedario es n .

El número de letras de cada palabra es m .

En esas m letras de la palabra, todas han de ser distintas.

Estas dos son explícitas

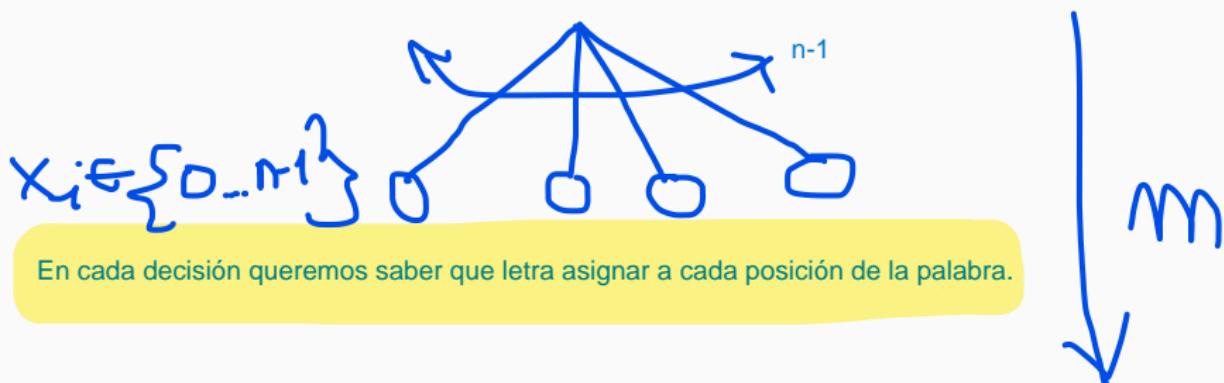
Esta es implícita.

Cadenas de caracteres

Dadas n letras, todas ellas diferentes, y $m \leq n$, queremos calcular todas las palabras con m letras diferentes escogidas entre las dadas.

tenemos n letras diferentes todas. Calcular cuantas palabras con m letras diferentes podemos formar (5, 6, 4 letras).

- Supongamos que las letras son $\{0, \dots, n - 1\}$. Representamos las soluciones mediante tuplas $(x_0, x_1, \dots, x_{m-1})$, donde x_i es la i -ésima letra de la palabra.



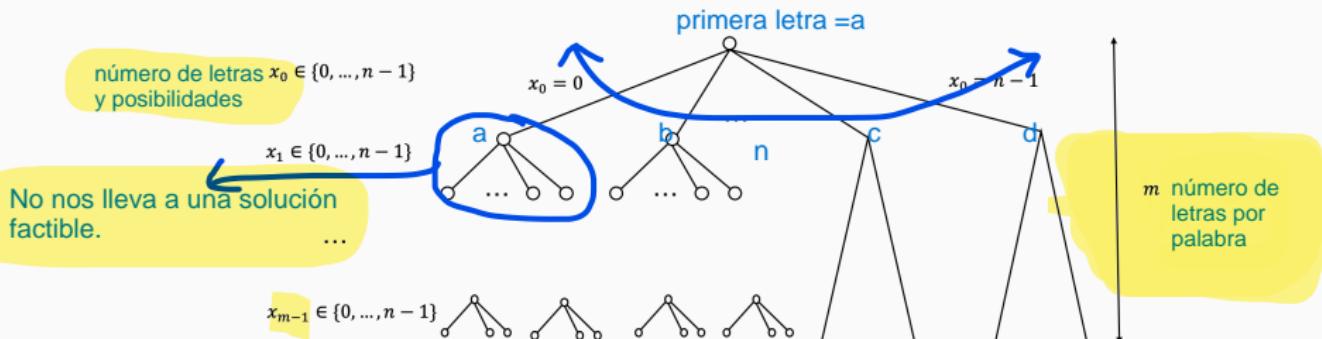
En cada decisión queremos saber que letra asignar a cada posición de la palabra.

Cadenas de caracteres

Dadas n letras, todas ellas diferentes, y $m \leq n$, queremos calcular todas las palabras con m letras diferentes escogidas entre las dadas.

tenemos un alfabeto de n letras (todas las letras diferentes entre si). Queremos saber cuántas palabras de m letras distintas podemos formar.

- Supongamos que las letras son $\{0, \dots, n-1\}$. Representamos las soluciones mediante tuplas $(x_0, x_1, \dots, x_{m-1})$, donde x_i es la i -ésima letra de la palabra.



- Las soluciones tienen que cumplir como restricciones explícitas que se utilicen letras válidas, y como restricciones implícitas que no haya letras repetidas.

Importante aclarar ambos tipos de restricciones en cada problema.

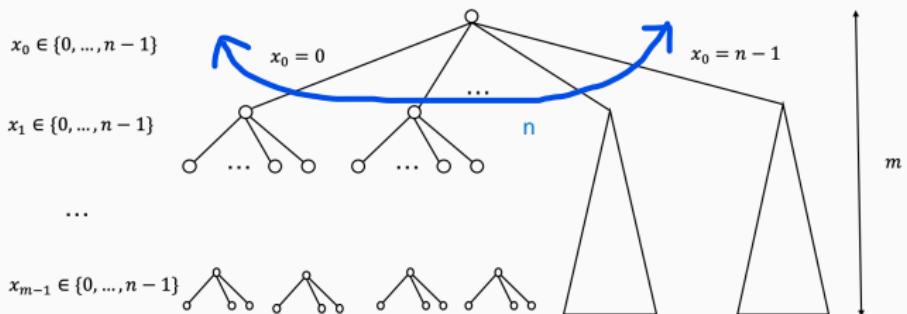
Tanto el árbol de exploración como las restricciones tanto explícitas como implícitas las tenemos que aclarar en nuestros problemas. Y el coste como consecuencia del árbol

Cadenas de caracteres

Dadas n letras, todas ellas diferentes, y $m \leq n$, queremos calcular todas las palabras con m letras diferentes escogidas entre las dadas.

Tenemos un alfabeto de n letras y queremos formar palabras de m letras, todas ellas diferentes.

- Supongamos que las letras son $\{0, \dots, n - 1\}$. Representamos las soluciones mediante tuplas $(x_0, x_1, \dots, x_{m-1})$, donde x_i es la i -ésima letra de la palabra.



Árbol de ramas n y de profundidad m .

- Las soluciones tienen que cumplir como restricciones explícitas que se utilicen letras válidas, y como restricciones implícitas que no haya letras repetidas. Si consideramos palabras de 5 letras sobre un alfabeto de 27 letras distintas, el número de posibilidades se reduce de $27^5 = 14.348.907$ a $\frac{27!}{22!} = 9.687.600$.

$$27 \cdot 26 \cdot 25 \cdot 24 \cdot 23$$

Es como las variaciones sin repetición. Si por ejemplo elegimos la a como letra, ya no la podemos volver a utilizar.

Cadenas de caracteres

```
void variaciones(int n, int m){  
    int solucion[m];  
    variaciones(solucion, 0, n, m);  
}
```

Número de letras que tiene que tener nuestra solución.

Esto es la llamada inicial a la función recursiva de vuelta atrás.

```
void variaciones(int solucion[], int k, int n, int m){  
    for(int letra = 0; letra < n; letra++){ recorremos las n's, la rama  
        solucion[k] = letra; Empezamos con la primera letra  
        if(esValida(solucion, k)){ Si cumple que no se repite y que se encuentra en el alfabeto:  
            if(esSolucion(k, m)) Mirar si es ya la última letra  
                { tratarSolucion(solucion, m); }  
            else  
                { variaciones(solucion, k + 1, n, m); } Y si no seguir con la  
                    siguiente  
                    Llamada recursiva con la siguiente posición de letra.  
    }  
}
```

Sería algo así como if($k == m - 1$)

Aquí estamos recorriendo la m

Tiene que pertenecer al abecedario que te digan Y además no puede estar repetida. Por tanto yo comprobaría si la letra está en las posiciones anteriores. Que es lo que se ve después. Creo que esta función es la poda de factibilidad

Cadenas de caracteres

La función esValida es la encargada de comprobar que la nueva letra que hemos incorporado a la solución no está repetida con las anteriores.

```
bool esValida(int solucion[], int k) {  
    int i = 0;  
    while(i < k && solucion[i] != solucion[k]) i++;  
    return i == k;
```

- } Aquí comprueba si cada letra que hemos puesto no coincide con la nueva letra (es decir, si son todas distintas).

Cadenas de caracteres

La función esValida es la encargada de comprobar que la nueva letra que hemos incorporado a la solución no está repetida con las anteriores.

```
bool esValida(int solucion[], int k) {  
    int i = 0;  
    while(i < k && solucion[i] != solucion[k]) i++;  
    return i == k;  
}
```

La función esSolucion simplemente comprueba que hemos colocado todas las letras.

```
bool esSolucion(int k, int m){  
    return k == (m - 1);
```

Comprueba que si nos dicen que la palabra tiene 5 letras, comprueba que haya 5 letras, 1 por cada posición.

Al ser algo tan pequeño lo podríamos poner directamente en el algoritmo de vuelta atrás.

Cadenas de caracteres

La función esValida es la encargada de comprobar que la nueva letra que hemos incorporado a la solución no está repetida con las anteriores.

Comprueba que se cumplen lo que dicen las restricciones.

```
bool esValida(int solucion[], int k) {  
    int i = 0;  
    while(i < k && solucion[i] != solucion[k]) i++;  
    return i == k;  
}
```

Comprueba para cada letra ya escrita
Que esa letra no se haya puesto ya en la palabra.

La función esSolucion simplemente comprueba que hemos colocado todas las letras.

```
bool esSolucion(int k, int m){  
    return k == (m - 1);  
}
```

Aquí comprueba que, si nos han pedido palabras de 5 letras, que la palabra tenga 5 letras. Si no, colocará otra letra.

Cuando encontramos una solución, podemos simplemente escribirla por la salida estándar:

```
void tratarSolucion(int solucion[], int m){  
    cout << "Solucion: ";  
    for(int i = 0; i < m; i++)  
        cout << solucion[i] << " ";  
    cout << endl;  
}
```

Creo que muestra por pantalla cada letra de manera individual. Muestra la palabra. Todas las posibles.

Vuelta atrás con marcaje

Es muy importante ya que me acuerdo de que lo utiliza en muchos de los ejercicios. Hay que realizar marcas, y además, acordarnos de desmarcarlas. Además el marcaje y el desmarcaje debe ser al mismo nivel.

Importante marcar y desmarcar SIEMPRE al mismo nivel.

Vuelta atrás con marcaje

Se puede optimizar el test de factibilidad por el método de asociar a cada nodo cierta cantidad de información que corresponde a "cálculos parciales" de dichos tests.

El objetivo principal del marcaje es optimizar el test de factibilidad para hacerlo más eficiente ya que en algunos casos el coste es lineal. Y lo podemos hacer constante.

Vuelta atrás con marcaje

Se puede optimizar el test de factibilidad por el método de asociar a cada nodo cierta cantidad de información que corresponde a "cálculos parciales" de dichos tests.

Se utilizan parámetros adicionales (**marcadores**) de entrada/salida que equivalen a variables globales y por tanto suponen un pequeño incremento del coste en espacio.

Vuelta atrás con marcaje

Se puede optimizar el test de factibilidad por el método de asociar a cada nodo cierta cantidad de información que corresponde a "cálculos parciales" de dichos tests.

También veremos que se utilizan vectores de marcadores

Se utilizan parámetros adicionales (**marcadores**) de entrada/salida que equivalen a variables globales y por tanto suponen un pequeño incremento del coste en espacio.

```
vueltaAtrasConMarcaje (Tupla & sol, int k, Marca & marcas) {  
    prepararRecorridoNivel(k);  
    while (!ultimoHijoNivel(k)){  
        sol[k] = siguienteHijoNivel(k);  
        if (esValida(sol, k, marcas)){  
            if (esSolucion(sol, k))  
                { tratarSolucion(sol); }  
            else{  
                marcar(marcas, sol, k); //Marcaje  
                vueltaAtrasConMarcaje(sol, k + 1, marcas);  
                desmarcar(marcas, sol, k); //Desmarcaje  
            }  
        }  
    }  
}
```

Como vemos
se realiza al
mismo nivel

Cadenas de caracteres con marcaje

podemos utilizar marcaje para evitar el bucle de comprobación cada vez que aumentamos la solución. Para ello utilizamos un vector de booleanos de tamaño el número de letras del alfabeto considerado. Cada posición del vector indica si la letra correspondiente ha sido ya utilizada.

Vamos, que en este caso (no se si en todos) el desmarcaje nos ayuda para eliminar el bucle de comprobación de si hemos utilizado la letra o no.

```
void variaciones(int solucion[], int k, int n, int m, bool marcas[]){
    for(int letra = 0; letra < n; letra++){
        if(!marcas[letra]){
            solucion[k] = letra;
            if(k == m - 1)
                { tratarSolucion(solucion, m); }
            else {
                marcas[letra] = true; //marcar
                variaciones(solucion, k + 1, n, m, marcas);
                marcas[letra] = false; //desmarcar
            }
        }
    }
}
```

letra actual alfabeto Número de letras de cada palabra

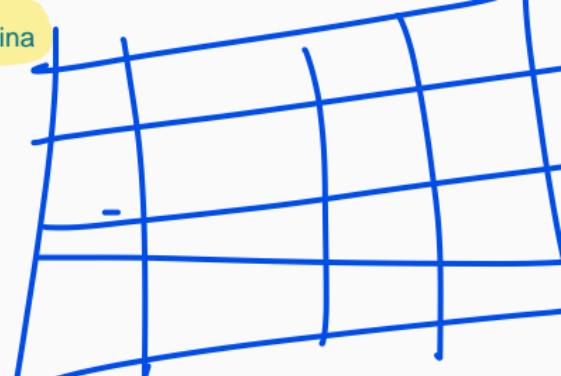
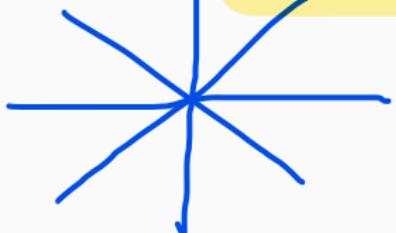
en teoría se encarga de crear todas las posibles soluciones.

nos indica si el elemento está marcado o no para poder utilizarlo.

Si lo estuviera no entramos porque no nos permiten repetir letras.

Lo que hace este algoritmo es que, si por ejemplo tenemos el alfabeto nuestro (a,b,c,...) y nos piden palabras de 3 letras sin repetir letras entonces el flujo será el siguiente: para la letra a, como no está marcada entra en el primer if. Como la palabra aún no es de 3 letras (y por tanto $k \neq m-1$) no lo trata como solución y va a seguir rellenando hasta tener tres letras; por tanto entra en el else y marca dicha letra y hace la llamada recursiva. Prueba con la a, y como ya está marcada no entra y pasa a la b. Como está desmarcada entra... y así sucesivamente.

Direcciones en las que se puede mover la reina



Vuelta atrás con marcaje

Problema de las n reinas

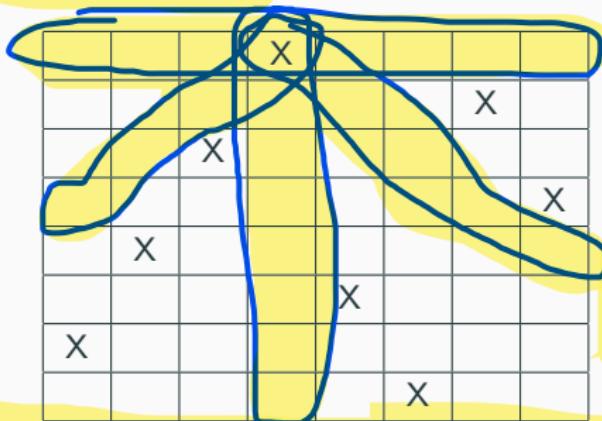
En un tablero

En un tablero de $n \times n$ posiciones queremos colocar n reinas sin que se amenacen entre si. Para ello debemos de conocer las reglas del ajedrez y es que la reina se mueve en todas las direcciones posibles. Hacia arriba, hacia abajo derecha izquierda, y en cualquiera de sus diagonales. Por tanto ninguna reina debe poder colocarse en cualquiera de las posiciones anteriores nombradas.

Problema de las n reinas

- El clásico problema de las 8 reinas consiste en colocar 8 reinas en un tablero de ajedrez sin que se amenacen. Dos reinas se amenazan si comparten la misma fila, columna o diagonal.

Tablero ha de ser 8x8



Vamos, que tenemos que poner, en este caso, 8 reinas (n nos dice arriba, porque se puede hacer con más o con menos) de forma que si colocamos una no puede haber otra en su fila, su columna o en cualquiera de sus diagonales

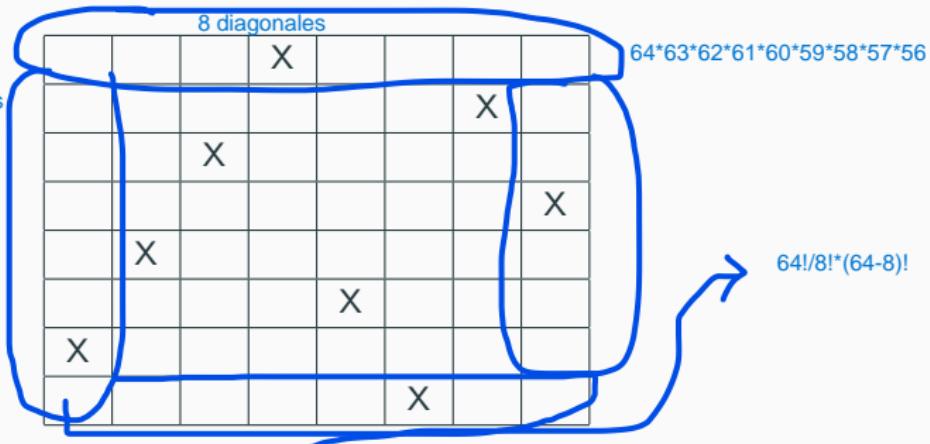
Vemos la que hemos rodeado que no le amenaza ninguna en ningún sitio. Ni en la fila ni en la columna ni en las diagonales.

Problema de las n reinas

- El clásico problema de las 8 reinas consiste en colocar 8 reinas en un tablero de ajedrez sin que se amenacen. Dos reinas se amenazan si comparten la misma fila, columna o diagonal.

El número de resultados posibles depende de las dimensiones del tablero. Si es un tablero de 8x8 podrás colocar como MÁXIMO 8 reinas, si es un tablero de 11x11 podrás colocar como máximo 11 reinas. En ningún caso podremos colocar una reina si hay otra en su misma fila, columna o en alguna de sus diagonales.

7 diagonales



- El espacio de búsqueda teórico es de $\binom{64}{8}$, que representan todas las combinaciones en las que podemos poner 8 reinas en un tablero de 64 casillas. Muchas soluciones posibles.
- Entonces, el número de soluciones potenciales a evaluar sería de 4.426.165.368.
- Realizar una búsqueda exhaustiva en este espacio es impracticable, siendo todavía más problemático si ampliamos el tamaño de la entrada, por ejemplo tratando de colocar 11 reinas en un tablero de 11×11 (743.595.781.824 soluciones potenciales).

Problema de las n reinas

- No podemos poner dos reinas en la misma fila, así que vamos a replantear el problema. Trataremos de colocar una reina en cada fila del tablero, de forma que no se amenacen.
- De esta manera, toda solución del problema se puede representar como una 8-tupla (x_0, \dots, x_7) en la que x_i representa la columna en la que se coloca la reina que está en la fila i del tablero.

En cada fila, rellenamos con la columna en la que ponemos la reina

vector de filas:



para cada fila marcamos la columna que marcamos. Lo llamo n también porque el número de filas y de columnas a de ser el mismo

Es decir, el marcaje lo vamos a realizar sobre un vector de n posiciones (desde 0 hasta $n-1$) que hace referencia a las n filas del tablero. En cada fila, en cada posición del vector por tanto, incluiremos un número desde 0 hasta $n-1$ que nos dirá la columna asignada a la fila correspondiente.

Para este vector yo pondría como comprobación que podemos poner la reina si la colocamos

Problema de las n reinas

- No podemos poner dos reinas en la misma fila, así que vamos a replantear el problema. Trataremos de colocar una reina en cada fila del tablero, de forma que no se amenacen.
- De esta manera, toda solución del problema se puede representar como una 8-tupla (x_0, \dots, x_7) en la que x_i representa la columna en la que se coloca la reina que está en la fila i del tablero.
- Restricciones explícitas:
 - $S_i = \{0, \dots, 7\}, 0 \leq i \leq 7$. Es decir, cada columna tiene que estar dentro del tablero.
 - Esta representación hace que el espacio de soluciones potenciales se reduzca a 8^8 posibilidades (16.777.216 valores).

Sin tener en cuenta restricciones explícitas ni implícitas, es decir, si pudiéramos colocar a las reinas en cualquier posición del tablero SIN tener en cuenta que se puedan amenazar.

Problema de las n reinas

- No podemos poner dos reinas en la misma fila, así que vamos a replantear el problema. Trataremos de colocar una reina **en cada fila del tablero**, de forma que no se amenacen.
- De esta manera, toda solución del problema se puede representar como una 8-tupla (x_0, \dots, x_7) en la que x_i representa la columna en la que se coloca la reina que está en la fila i del tablero.

Restricciones explícitas:

- $S_i = \{0, \dots, 7\}, 0 \leq i \leq 7$. Es decir, cada columna tiene que estar dentro del tablero.
- Esta representación hace que el espacio de soluciones potenciales se reduzca a 8^8 posibilidades (16.777.216 valores).

Restricciones implícitas: Como las que nos dicen en el problema. Que podrían ser distintas

- No puede haber dos reinas en la misma columna, ni en la misma diagonal.
- Al no poder haber dos reinas en las misma columna, se deduce que todas las soluciones son permutaciones de la 8-tupla $(1, 2, 3, 4, 5, 6, 7, 8)$. Por lo tanto el espacio de soluciones potenciales se reduce a $8!$ (40.320 valores diferentes).

Estas son las que tenemos que especificar en el examen de una manera más clara.

Problema de las n reinas (sin marcaje)

```
void nReinas(int solucion[], int k, int n){  
    for(int i = 0; i < n; i++){ En teoría itera sobre las 8 columnas del tablero  
        solucion[k] = i; Coloca una reina  
        if (esValida(solucion, k)){ Debe comprobar que no hay ninguna reina en su COLUMNA y  
                                         DIAGONAL  
            if(k == n - 1) Comprueba si se han colocado las n reinas (es decir todas).  
                { tratarSolucion(k, n); }  
            else  
                { nReinas(solucion, k + 1, n); } Si no se han colocado todas las  
                                         reinas coloca una reina  
    }} a la fila k le asignamos la columna i
```

La función `esValida` comprueba que la nueva reina no está en la misma columna ni diagonal que las anteriores:

```
bool esValida(int solucion[], int k) {  
    int i = 0;  
    while (i < k && solucion[i] != solucion[k]  
          && abs(solucion[k] - solucion[i]) != k - i)  
        { i = i + 1; }  
    return i==k;  
}
```

Que no hemos colocado una reina en+ la misma columna que otra.

Comprueba para todas las reinas colocadas en el tablero que no haya otra ni en su columna ni en su diagonal.

Esto comprueba si están en la misma diagonal

Comprueba que no esté en la misma columna que el resto de reinas colocadas

Problema de las n reinas con marcaje

Ahora vamos a aplicar los marcajes

- Para controlar las columnas llevaremos un marcador cols que indica las columnas que ya están ocupadas. Ahora veremos si se utiliza un vector o que.

Problema de las n reinas con marcaje

Llevaremos un marcador de columnas y otro de diagonales.

- Para controlar las columnas llevaremos un marcador cols que indica las columnas que ya están ocupadas.
- Para controlar las diagonales llevaremos un marcado diags que indica las diagonales que ya están amenazadas.

Marcadores

Problema de las n reinas con marcaje

- Para controlar las columnas llevaremos un marcador cols que indica las columnas que ya están ocupadas. En este caso hace uso de vectores de marcas para ambos casos.
- Para controlar las diagonales llevaremos un marcado diags que indica las diagonales que ya están amenazadas. Quedarnos con que este es el número de diagonales. En un tablero $n \times n$ hay $4n - 2$ diagonales:
 - Diagonales descendentes (paralelas a la principal) numeradas de 0 a $2n-2$: una coordenada (i,j) pertenece a la diagonal descendente $j - i + n - 1$.
 - Diagonales ascendentes numeradas de $2n-1$ a $4n-3$: una coordenada (i,j) pertenece a la diagonal ascendente $i + j + 2n - 1$

```
bool segura(vector<int> const& sol, int k, int n,
            vector<bool> const& cols, vector<bool> const& diags)
{
    return !cols[sol[k]] && !diags[k+sol[k]+2n-1] && !diags[sol[k]-k+n-1];
}
```

Comprueba si columnas o diagonales están marcadas

Vector de booleanos que recoge las diagonales que ya están marcadas.

Vector de booleanos que recoge las columnas que han sido marcadas

diagonales descendentes.

diagonales ascendentes.

Problema de las n reinas con marcaje

```
void nReinas(vector<int> &sol, int k, int n,
             vector<bool> &cols, vector<bool> &diags) {
    for (int col = 0; col < n; col++) { Recorremos todas las columnas del tablero y para cada una..
        sol[k] = col; le asignamos la columna
        if (segura(sol, k, n, cols, diags)) { Si es valida (es decir si no está atacada y por tanto si no esta marcada, ni en diagonales ni en columnas)
            if (k==n-1) Comprueba si es la última reina a colocar en el tablero. Si es la última fila.
                { imprimir(sol); } Si es la última reina imprime la solución.
            else { Si no es la última reina entonces
                cols[sol[k]] = true; marcamos la columna // Marcar
                diags[sol[k]-k+n-1] = true;
                diags[k+sol[k]+2n-1] = true; } marcamos las diagonales superior e inferior
                nReinas(sol, k+1, n, cols, diags); Buscamos donde colocar la siguiente reina en la siguiente columna. Sabemos que en cada columna habrá a lo sumo 1 reina.
                cols[sol[k]] = false; // Desmarcar
                diags[sol[k]-k+n-1] = false;
                diags[k+sol[k]+2n-1] = false; }
            }
        }
    }
}
```

sol[k] = para la fila k le asignamos la columna col que estemos recorriendo en ese momento.

Encontrar una solución

Si no me equivoco, hasta el momento en todos los problemas que hemos visto se manejan todas las soluciones posibles. Sin embargo, nosotros podemos querer que únicamente nos encuentre una la primera, la mejor, la peor.. única solución

MAXIMIZACIÓN

MINIMIZACIÓN.

Encontrar una solución

Dominó

Que nos imprima una única solución, en vez de que nos imprima todas las posibles soluciones.

Dominó: una solución

- Se trata de encontrar una cadena circular de fichas de dominó, teniendo en cuenta:
 - Cada cadena tiene que utilizar las 28 fichas diferentes que contiene el juego de dominó.
El dominó tiene 28 fichas, y como condición debemos de ponerlas todas y no repetir ninguna.
 - No se puede repetir ninguna ficha.
- Las cadenas tienen que ser correctas, es decir, cada ficha tiene que ser compatible con la siguiente y la cadena tiene que cerrar (el valor de un extremo de la última ficha tiene que coincidir con el otro extremo de la primera). Por ejemplo:

6|3 → 3|4 → 4|1 → 1|0 → ... → 5|6

es una cadena correcta.

Si 3/4 entonces también marcar 4/3 porque son la misma ficha leída del derecho y del revés.

En el dominó las reglas son que cada pieza debe compartir el número con la pieza que conectes. Si por ejemplo pones una pieza que sea 5|3 por la derecha solo podrás poner otra ficha que tenga un 3 y por la izquierda otra ficha que tenga un 5 y conectarlo por ese 5. Solo hay una ficha 5|3 es decir, la 5|3 y la 3|5 son la misma ficha. Además, está la ficha 6|6, 5|5, 4|4...0|0

Dominó: una solución

- Se trata de encontrar una cadena circular de fichas de dominó, teniendo en cuenta:

- Cada cadena tiene que utilizar las 28 fichas diferentes que contiene el juego de dominó.

El domino tiene 28 fichas y todas ellas distintas.

- No se puede repetir ninguna ficha.

- Las cadenas tienen que ser correctas, es decir, cada ficha tiene que ser compatible con la siguiente y la cadena tiene que cerrar (el valor de un extremo de la última ficha tiene que coincidir con el otro extremo de la primera). Por ejemplo:

$$6|3 \rightarrow 3|4 \rightarrow 4|1 \rightarrow 1|0 \rightarrow \dots \rightarrow 5|6$$

es una cadena correcta.

29 valores porque para la primera ficha ponemos sus dos nums, para el resto solo 1.

- La solución va a ser una tupla de 29 valores (x_0, \dots, x_{28}) ; cada x_i es un número del 0 al 6.

Buscar foto donde se ve, pero básicamente si hay un 6|5 no hay otra 5|6 ya que es la misma

- En la solución no guardaremos las fichas, sino los valores de uno de los extremos. En el ejemplo anterior, la solución tendría la siguiente forma: 6, 3, 4, 1, 0, ..., 5, 6.

La primera ficha guardas ambos números, en este caso el 6 y el 3. En los siguientes solo el de la derecha.

- No necesitamos guardar explícitamente los dos extremos de las fichas, ya que cada ficha tiene que coincidir con la siguiente.

Únicamente de la primera guardamos ambos extremos.

En el domino las reglas son que cada pieza debe compartir el número con la pieza que conectes. Si por ejemplo pones una pieza que sea 5|3 por la derecha solo podrás poner otra ficha que tenga un 3 y por la izquierda otra ficha que tenga un 5 y conectarlo por ese 5. Guardamos el primer extremo izquierdo y del resto guardamos todos los extremos derechos

Dominó: una solución

Matriz de booleanos con las fichas que ya hemos utilizado.

- Para evitar fichas repetidas utilizaremos una matriz (7×7) donde marcaremos las fichas usadas.
- Hay que tener en cuenta que si marcamos la casilla (i, j) habrá que marcar la simétrica (j, i) , ya que se trata de la misma ficha. Si marcamos la 5|6 marcar también la 6|5
- El problema pide que se encuentre una sola solución, no todas las que existan, así que vamos a tener que abortar la búsqueda en el momento que aparezca la primera.
- Para ello utilizaremos una variable de control `exito` que se hace cierta al encontrar una solución.

Si marcamos 3|0, marcamos 0|3 porque es la misma ficha usado de forma diferente.

0	1	2	3	4	5	6
0			X			
1						
2						
3	X		X			
4						
5						
6						

Bool `exito`. si `exito` salir.
No queremos la mejor ni la peor ni nada.
Simplemente la primera que encuentre nos sobra y nos 25

Dominó: una solución

Vector de 29 valores.

k empieza siendo 2.

```
void domino(int sol[], int k, int n, bool marcas[NUM_VAL][NUM_VAL],  
           bool &exito){  
    int i = 0; int m = (n * n + n) / 2;  
    while (i < n && !exito){  
        if (!marcas[sol[k - 1]][i]) {  
            sol[k] = i;  
            if (k == m) {  
                if (sol[0] == sol[k]) {  
                    tratarSolucion(sol, m);  
                    exito = true; } Aquí saldría de todo el bucle  
                }  
            else {  
                marcas[sol[k - 1]][i] = true; Se marca la ficha y la simétrica (si es la 5|6 se  
                marcas[i][sol[k - 1]] = true; marca también la 6|5) ya que son la misma  
                domino(sol, k + 1, n, marcas, exito);  
                marcas[sol[k - 1]][i] = false; Se desmarca la ficha y la simétrica (si es la  
                marcas[i][sol[k - 1]] = false; 5|6 se marca también la 6|5) ya que son la  
            }  
        }  
        i++; Aquí usamos while para que podamos hacer que, en cuanto encuentre una solución, entonces salga del  
    }  
}
```

$m = (n \cdot n + n) / 2$ es para saber el número de piezas.

Dominó: una solución

- Las cadenas son circulares y no queremos que nuestro algoritmo repita soluciones simétricas innecesariamente, así que en la llamada inicial fijaremos la primera ficha.
- Por tradición en el juego del dominó, siempre se empieza por el doble 6, así que pondremos los dos primeros valores como 6 (aunque podríamos haber utilizado cualquier par de valores).

```
int main()
{
    const int NUM_VAL = 7;
    const int TAM_SOL = (NUM_VAL*NUM_VAL+NUM_VAL)/2+1; Cuantas números tiene
    int sol[TAM_SOL];
    bool marcas[NUM_VAL][NUM_VAL]; = vector<vector<int>>marcas(n...)
    for(int i = 0; i < NUM_VAL; i++)
        for(int j = 0; j < NUM_VAL; j++)
            marcas[i][j] = false; rellenamos la matriz de marcas con falses porque ninguna está
    sol[0] = NUM_VAL-1; sol[1] = NUM_VAL-1; Doble 6 como primera sol
    marcas[NUM_VAL-1][NUM_VAL-1] = true; empezamos por tradición con el doble 6
    bool exito = false;
    domino(sol, 2, NUM_VAL, marcas, exito); Llamada inicial.
    return 0;
}
```

Ya que k normalmente si no me equivoco, en la llamada inicial empieza siendo 0. Pero al poner la primera ficha tenemos ya 2 soluciones porque la primera ficha tenemos que poner ambos números (en el resto solo 1)

Problemas de optimización

Que la solución que encontremos sea la mejor entre todas las soluciones reales.

Tenemos los problemas de:

- º Maximización
- º Minimización.

Problema del viajante

Un vendedor ambulante de alfombras persas tiene que recorrer n ciudades volviendo tras ello al punto de partida. El buen señor se ha informado sobre las posibles conexiones directas por ferrocarril entre las ciudades y sobre las tarifas correspondientes.

El vendedor desea conocer un circuito en tren que recorra cada ciudad exactamente una vez y regrese a la ciudad de partida, y cuya tarifa total sea mínima.

Hay ciudades que no conectan entre si, pero queremos recorrer todas exactamente una vez y regresar a la ciudad de partida. Además tenemos que conseguir el menor coste posible.

MINIMIZACIÓN

Restricciones explícitas:
 n ciudades

Restricciones implícitas:
Volver al punto de partida, recorrer cada ciudad exactamente una vez
Tarifa total sea MÍNIMA-> PROBLEMA DE MINIMIZACIÓN.

Problemas de optimización



- Modificar los esquemas vistos de forma que se almacene la mejor solución encontrada hasta el momento.
- Al tratar una nueva solución, se comparará con la que se tiene almacenada.

Iremos almacenando la solución y compararemos cuales son mejor

Para estos problemas realizábamos una poda de optimalidad.

- Modificar los esquemas vistos de forma que se almacene la mejor solución encontrada hasta el momento.
- Al tratar una nueva solución, se comparará con la que se tiene almacenada.
- Para realizar esta comparación de la forma más eficiente posible, es preciso almacenar junto con la mejor solución su valor asociado.
- Además, para facilitar el cálculo del valor de cada solución alcanzada se incorpora como marcador el valor (parcial) de la tupla parcial.

También es importante en los problemas, además de señalar el árbol de exploración y las restricciones (tanto implícitas como explícitas), es importante señalar los marcadores que utilizamos en cada problema. Porque cuenta siempre ponerlos.

Problemas de optimización

Maximización

Minimización

- Modificar los esquemas vistos de forma que se almacene la mejor solución encontrada hasta el momento.
- Al tratar una nueva solución, se comparará con la que se tiene almacenada.
- Para realizar esta comparación de la forma más eficiente posible, es preciso almacenar junto con la mejor solución su valor asociado.
- Además, para facilitar el cálculo del valor de cada solución alcanzada se incorpora como marcador el valor (parcial) de la tupla parcial.
- Los problemas de optimización admiten un **mecanismo adicional de poda** que se realiza cuando se puede asegurar que ninguno de los descendientes del nodo a expandir puede llegar a alcanzar una solución mejor que la mejor encontrada hasta ese momento.

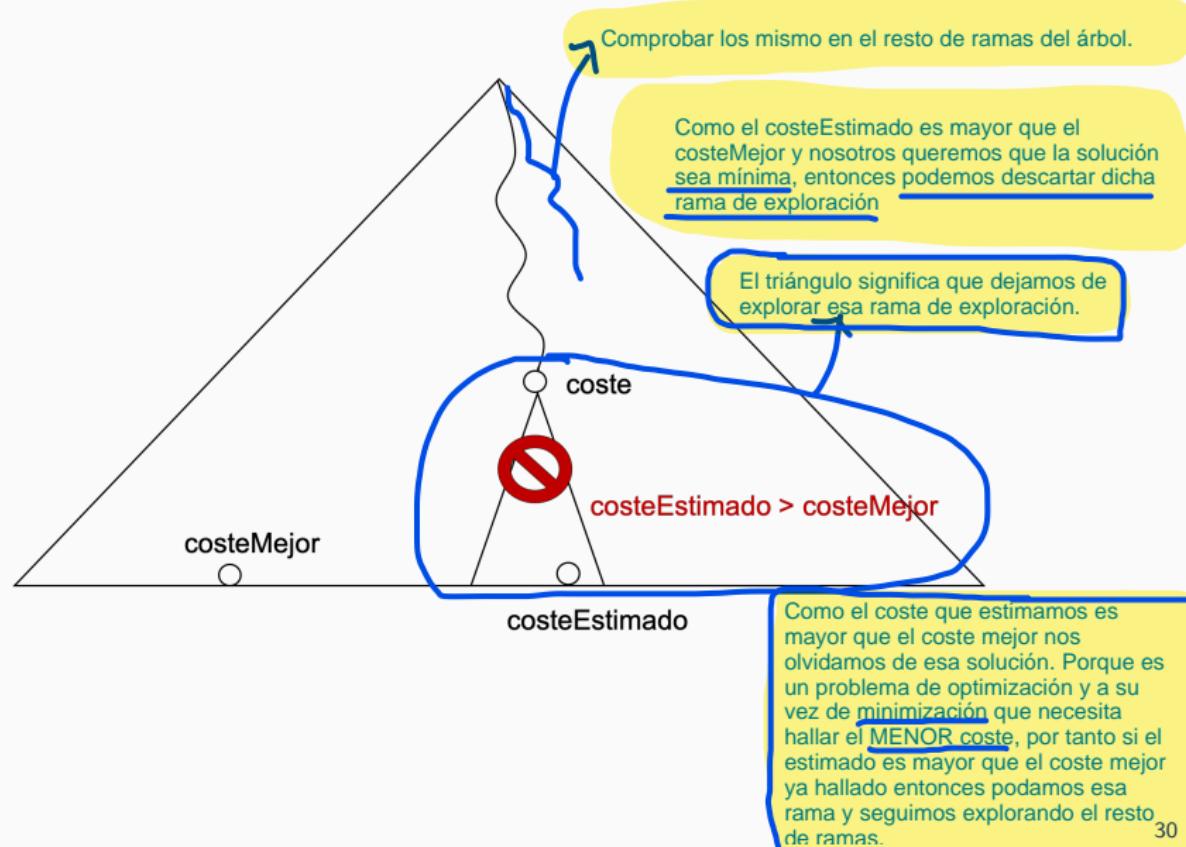
Habrá un mecanismo para asegurar que ninguna de las siguientes ramas de exploración posee una solución mejor.

Esta solución es la estimada

Hay unas estimaciones más claras, o más específicas y eficientes que otras; y por tanto hay estimaciones mejores que otras.

- Modificar los esquemas vistos de forma que se almacene la mejor solución encontrada hasta el momento.
- Al tratar una nueva solución, se comparará con la que se tiene almacenada.
- Para realizar esta comparación de la forma más eficiente posible, es preciso almacenar junto con la mejor solución su valor asociado.
- Además, para facilitar el cálculo del valor de cada solución alcanzada se incorpora como marcador el valor (parcial) de la tupla parcial.
- Los problemas de optimización admiten un **mecanismo adicional de poda**, que se realiza cuando se puede asegurar que ninguno de los descendientes del nodo a expandir puede llegar a alcanzar una solución mejor que la mejor encontrada hasta ese momento.
- En un problema de minimización, una cota inferior o estimación de la mejor solución alcanzable desde un nodo sirve para podarlo si la estimación es ya mayor que el valor asociado a la mejor solución encontrada hasta el momento.

Poda de optimalidad



Esquema de optimización

```
vueltaAtrasOptimizacion (Tupla & sol, Valor valor, int k,  
                         Tupla & solMejor, Valor valorMejor) {  
    prepararRecorridoNivel(k);  
    while (!ultimoHijoNivel(k)) {  
        sol[k] = siguienteHijoNivel(k);  
        if (esValida(sol, k)) {  
            if (esSolucion(sol, k)) {  
                if (mejor(valor, valorMejor)) {  
                    solMejor = sol;  
                    valorMejor = valor;  
                }  
            }  
            else if (esPrometedor(sol, valor, valorMejor))  
                { vueltaAtras(sol, k + 1); }  
        }  
    }  
}
```

estos son marcadores

Si es solución factible tendremos que comprobar si es solución mejor.

vemos cual es mejor

Comprueba / hace una estimación de las siguientes.

Problemas de optimización

Problema del viajante

Veremos como hallar el mínimo coste de recorrido y volver a nuestra ciudad.

Problema del viajante

- Hay que encontrar una permutación del conjunto de ciudades tal que la suma de las distancias entre una ciudad y la siguiente sea mínimo. La distancia entre dos ciudades viene dada en una matriz de distancias.
- El espacio de soluciones por tanto tiene tamaño $(n - 1)!$, ya que corresponde a todas las posibles permutaciones, teniendo en cuenta que el principio y el final es el mismo y que para evitar soluciones repetidas podemos fijar como comienzo del recorrido cualquiera de las ciudades, por ejemplo la 0.

Problema del viajante

- Hay que encontrar una permutación del conjunto de ciudades tal que la suma de las distancias entre una ciudad y la siguiente sea mínimo. La distancia entre dos ciudades viene dada en una matriz de *distancias*.
- El espacio de soluciones por tanto tiene tamaño $(n - 1)!$, ya que corresponde a todas las posibles permutaciones, teniendo en cuenta que el principio y el final es el mismo y que para evitar soluciones repetidas podemos fijar como comienzo del recorrido cualquiera de las ciudades, por ejemplo la 0.
- Como marcador usamos un vector usadas de n componentes, donde el valor de cada componente indica si la ciudad correspondiente ha sido visitada.

```
bool esValida(int distancias[][], int solucion[], int k,  
              bool usadas[]){  
    return (hayArista(distancias, solucion[k-1], solucion[k])  
           && !usadas[solucion[k]]); }  
    Comprueba que ambas ciudades estén conectadas  
    Verifica que la ciudad NO haya sido visitada
```

si están esas dos conectadas

- Para calcular una estimación optimista es suficiente con encontrar la mínima distancia entre cualquier par de ciudades, y considerar que todos los desplazamientos van a tener esa distancia.

Se pueden hacer otras posibles estimaciones, pero esto también es correcto.

Problema del viajante

```
matriz que contiene la distancia entre las ciudades Almacena la secuencia parcial de solus Coste actual
void viajante(int distancias[MAX][MAX], int solucion[], int &coste,
int k, int n, int solucionMejor[], int &costeMejor, bool m usadas[]){
    for(int i = 1; i < n; i++){
        Posición actual solucion[k] = i; Almacenar la mejor solución
        Número total de ciudades                                         encontrada
        if(esValida(distancias, solucion, k, usadas)){ Si la solución es válida
            entonces
                coste += distancias[solucion[k-1]][solucion[k]]; Sumamos el
                usadas[i]=true; marca la ciudad y aumenta el coste
                if(k==n-1){ Si es la última ciudad Debemos comprobar que hay recorrido de la
                    primera a la última
                    if(hayArista(distancias, solucion[k], solucion[0])){
                        if(coste +distancias[k][0] < costeMejor){ Si es menor que el
                            que ya llevamos
                                costeMejor = coste+distancias[k][0]; cambiamos la solución
                                copiarSolucion(solucion, solucionMejor);
                            }
                        } else { si no se ha completado el recorrido Hacer un coste estimado de las restantes
                            costeEstimado = coste + (n - k + 1) * costeMinimo;
                            if(costeEstimado < costeMejor) Si el coste estimado es menor se llama a
                                recursividad
                                viajante(distancias, solucion, coste, k+1, n,
                                solucionMejor, costeMejor); }
                            usadas[i]=false;
                            coste -= distancias[solucion[k-1]][solucion[k]];
                        }
                    }
                }
            }
        }
    }
}
```

Veremos 4 formas distintas de hacerlo en el problema de los funcionarios.

DESMARCAJE AL MISMO NIVEL

Problema del viajante

Coste mínimo encontrado entre todas las aristas del grafo

```
costeMinimo=calcularMinimo(distancias);
```

solucion[0]=0; usadas[0]=true; Marcamos la solución 0 y establecemos la primera ciudad como inicial

```
for (int i=1;i<n;i++) {usadas[i]=false;};
```

coste=0; inicializamos coste del recorrido a 0
Marcamos el resto de ciudades a false (no visitadas).

```
costeMejor = ...
```

```
//una cota superior
```

//1- la suma de todas las aristas del grafo Sumar todas directamente

//2- maximaArista * n Coger la máxima distancia y multiplicarla por el número de ciudades

```
viajante(distancias,solucion,coste,1,n,solucionMejor,costeMejor,  
usadas,costeMinimo);
```

mínimo de todas las distancias, se puede hacer en otra función, luego se utiliza en el coste estimado.



Usando este bucle for

k numero ciudades

Importante entender bien estos problemas

Problema de la mochila

Problema de maximización. El anterior es de minimización.

Cuando Alí-Babá consigue por fin entrar en la Cueva de los Cuarenta Ladrones encuentra allí gran cantidad de objetos muy valiosos. A pesar de su pobreza, Alí-Babá conoce muy bien el peso y valor de cada uno de los objetos en la cueva.

La mochila solo permite un peso.,, tendrá que llevar la mochila de más valor en función del peso.

Debido a los peligros que tiene que afrontar en su camino de vuelta, solo puede llevar consigo aquellas riquezas que quepan en su pequeña mochila, que soporta un peso máximo conocido. La mochila tiene un máximo de peso que puede llevar (cota).

Determinar qué objetos debe elegir Alí-Babá para maximizar el valor total de lo que pueda llevase en su mochila.

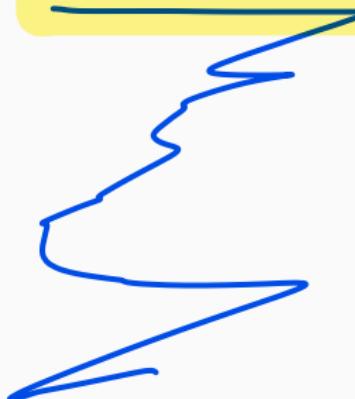
También se da en 3ero. Meter objetos en una mochila de forma que se maximice el valor de los objetos.

Restricciones: La mochila podrá llevar una cantidad de objetos que no superen el peso máximo m que puede cargar.
El valor de todos los objetos que lleve DEBE SER MÁXIMO.

Problemas de optimización

Problema de la mochila

Problema de maximización



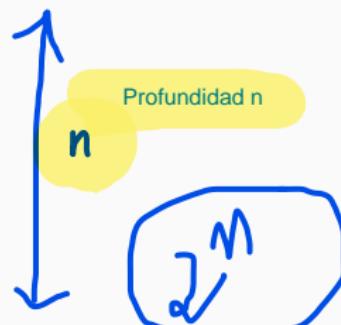
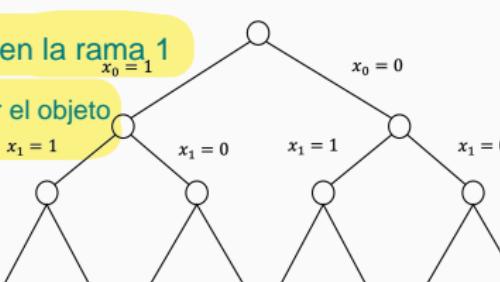
Problema de la mochila

En otras versiones se pueden partir los objetos. Partir = cortar. PERO NOSOTROS NO DEBEMOS CONOCER ESA OTRA VERSIÓN

- Tenemos n objetos no fraccionables de valores (v_0, \dots, v_{n-1}) y pesos (p_0, \dots, p_{n-1}) , y tenemos que determinar qué distribución de objetos óptima podemos transportar en la mochila sin superar su capacidad máxima m.
La cueva tiene n objetos no fraccionables y la mochila es de capacidad m
- En cada nivel del árbol de búsqueda vamos a decidir si cogemos o no el i -ésimo objeto. La solución será una tupla de (x_0, \dots, x_{n-1}) de booleanos.

Conviene empezar a buscar en la rama 1

En cada decisión debemos decidir el objeto a coger.



- El árbol de búsqueda es un árbol binario de profundidad n con las siguientes restricciones:
 - Deberemos de maximizar el valor de lo que nos llevamos $\sum_{i=0}^{n-1} b_i v_i$.
 - El peso no debe exceder el máximo permitido $\sum_{i=0}^{n-1} b_i p_i \leq m$.

Problema de la mochila

```
array con los pesos de objetos      array con el valor   Sol parcial (objetos que hemos metido)
void mochila(float P[], float V[], bool solucion[], int k, int n,
peso max                           float m, float &peso, float &beneficio, objeto actual
capacidad mochila, peso actual de objetos colocados   beneficio actual de los colocados en la mochila
int solucionMejor[], int &valorMejor){total de objetos
estos almacenan la mejor solución encontrada           m
// hijo izquierdo [cogémos el objeto]               mejor solución y cuanto vale.

solucion[k] = true;
peso = peso + P[k]; //marcamos peso y beneficio
beneficio = beneficio + V[k];
if(peso <= m){
    if(k == n-1){
        if(valorMejor < beneficio){
            valorMejor = beneficio;
            copiarSolucion(solucion, solucionMejor);
        }
    } else {la estimación la incluiríamos aquí
        mochila(P,V,solucion, k+1,n, m, peso, beneficio,
                solucionMejor, valorMejor);
    }
}
peso = peso - P[k]; //desmarcamos peso y beneficio
beneficio = beneficio - V[k];
```

MARCAJE
Y
DESMARCA

AL MISMO
NIVEL

si hemos cogido el
objeto o no(true si
false no)...

k, int n,

objeto actual

total de objetos

beneficio actual de los colocados en la mochila

mejor solución y cuanto vale.

Recordar que es vital decir el tamaño del árbol de exploración las restricciones tanto implícitas como explícitas y también los marcadores que utilizamos. Para los problemas de VUELTA ATRÁS.

```

// hijo derecho [no cogemos el objeto]
solucion[k] = false;
if(k == n-1){
    if(valorMejor < beneficio){
        valorMejor = beneficio;
        copiarSolucion(solucion, solucionMejor); }
} else {
    mochila(P,V,solucion, k+1,n, m, peso, beneficio,
            solucionMejor, valorMejor);
}

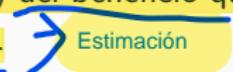
```

No cogemos nada luego no hay que marcar nada

Tampoco debemos desmarcar nada.

Es distinto este problema. Debemos de hacer de forma individual para la rama de exploración en la cual se coge el objeto y para la rama de exploración en la que no se coge el objeto. Importante porque el problema que salió en el examen de este año se realizaba como este de aquí, con esta estructura.

Problema de la mochila: cota optimista

- Podemos calcular una cota superior (una evaluación optimista) del beneficio que podemos obtener con lo que nos resta para llenar la mochila.  **Estimación**
- La versión del problema de la mochila en que los objetos se pueden fraccionar tiene una estrategia de resolución muy eficiente que consiste en ordenar de mayor a menor los objetos por la densidad del valor por unidad de peso v_i/p_i y a continuación ir metiendo los objetos enteros hasta que haya uno que no quepa por peso. Este último objeto se fracciona y se suma el valor de lo que quepa en la mochila hasta completarla.
- Este algoritmo proporciona siempre una cota superior a cualquier solución donde no se permita fraccionamiento, por lo que vamos a utilizarlo para calcular una cota optimista: suma del beneficio actual mas el beneficio conseguido cogiendo los objetos que quepan en el orden indicado desde el $k+1$ hasta el $n-1$.
- Para poder aplicar esta cota es necesario tener los objetos ordenados de mayor a menor valor por unidad de peso, lo cual se lleva a cabo antes de la llamada a la vuelta atrás.

Tomamos en cuenta el espacio restante en la mochila (HUECO) y los objetos que podrían caber en ese espacio segun su peso y su valor.

Problema de la mochila: cota optimista

```
float calculoEstimacion(float P[], float V[], float m, int k,  
    Total menos lo que llevamos hasta ahora float peso, float beneficio){  
    float hueco = m - peso; float estimacion = beneficio; Estimación inicial del  
    int j = k + 1; objetos restantes Peso límite- el peso acumulado  
    iteración sobre restantes beneficio  
    while (j < n && P[j] <= hueco){ No hemos acabado con los objetos y todavía queda  
        hueco = hueco - P[j]; hueco  
        estimacion = estimacion + V[j];  
        j = j + 1; Coger una fracción del objeto. Es una estimación, esto no quiere decir que  
    }; se quiera partir  
    if (j < n) { estimacion = estimacion + (hueco/P[j])*V[j]; } Hacer esto es más realista que hallar de los que quedan el máximo y  
    return estimacion; multiplicar.  
}
```

Situación en la que el objeto no cabe en la mochila.

Modificamos el algoritmo: Calcular bien la cota

```
...  
} else {  
    float estimacion = calculoEstimacion(P, V, m, k, peso, beneficio);  
    if (estimacion > beneficioMejor)  
        mochila(P, V, solucion, k+1, n, m, peso, beneficio,  
                solucionMejor, valorMejor);  
}
```

Problemas de optimización

Funcionarios con trabajo

Este problema está en el campus, y en este problema se explican 4 formas de estimaciones para el mismo problema (en realidad son 5 pero la 5a no es muy eficiente.)

Fucionarios con trabajo

Nos lo va a poner en el juez

Cada fucionario tiene trabajos

Nos lo ha puesto con y sin estimación

Está hecho, he copiado y pegado lo que pone en estos ejercicios.

El Ministro de Desinformación y Decencia se ha propuesto hacer trabajar en firme a sus n fucionarios, para lo que se ha sacado de la manga n trabajos. n trabajos y n fucionarios.

Habrá tantos trabajos como fucionarios

A pesar de su ineefacia, todos los fucionarios son capaces de hacer cualquier trabajo, aunque unos tardan más que otros. Cualquier fucionario es capaz de hacer cualquier trabajo.

Para cada trabajo uno tarda más o menos que otros. Pero todos son capaces de hacer cualquiera de los trabajos

La información al respecto se recoge en la tabla T , donde $T[i][j]$ representa el tiempo que el fucionario i tarda en realizar el trabajo j .

Hay una matriz de tiempos donde las filas representan a los fucionarios y las columnas los trabajos. La intersección cuánto tardan.

Para justificar su puesto, Su Excelencia el Sr. Ministro desea conocer la asignación óptima de trabajos a fucionarios de modo que la suma total de tiempos sea mínima.

Para cada fucionario un trabajo

Para cada trabajo un fucionario

Problema de minimización.

Funcionarios con trabajo

- Representamos la solución en tuplas de la forma $(x_0, x_2, \dots, x_{n-1})$ donde x_i es el trabajo asignado al funcionario i .

Funcionarios con trabajo

No se puede asignar el mismo trabajo a personas distintas

iteraremos sobre los trabajos, llevaremos un vector de trabajos asignados y la k recorrerá los funcionarios.

- Representamos la solución en tuplas de la forma $(x_0, x_1, \dots, x_{n-1})$ donde x_i es el trabajo asignado al funcionario i .
- Ya que cada trabajo solo puede ser asignado a un funcionario, llevaremos cuenta de los trabajos ya asignados en un vector marcador *asignado* de booleanos.
- Para la solución parcial (x_0, \dots, x_k) , el tiempo hasta el momento es $\text{tiempo} = \sum_{i=0}^k T[i][x_i]$, y tenemos que estimar el tiempo del resto de la solución:

Tendremos un vector donde recogeremos qué trabajos han sido asignados con un vector de booleanos. La solución parcial será el sumatorio del tiempo que tarda un funcionario en realizar cierto trabajo y además tendremos que realizar una estimación del resto de la solución.

Ahora, para aumentar la poda veremos las distintas opciones que tenemos para realizar la estimación.

Hay 4 opciones

Funcionarios con trabajo

No se puede asignar el mismo trabajo a personas distintas

- Representamos la solución en tuplas de la forma $(x_0, x_1, \dots, x_{n-1})$ donde x_i es el trabajo asignado al funcionario i .
- Ya que cada trabajo solo puede ser asignado a un funcionario, llevaremos cuenta de los trabajos ya asignados en un vector marcador *asignado* de booleanos.
- Para la solución parcial (x_0, \dots, x_k) , el tiempo hasta el momento es $\text{tiempo} = \sum_{i=0}^k T[i][x_i]$, y tenemos que estimar el tiempo del resto de la solución:

En este problema tenemos muchas opciones de estimación

- ① La opción más sencilla (y más optimista) es aproximar con 0 este tiempo, y utilizar tiempo como estimación. Que no se tarde nada
estimacion = tiempo; todos tardan nada en hacer el siguiente, demasiado optimista
Coste en cada llamada: $\Theta(1)$.
No es lo más realista
coste de hallar la cota.

Vamos a tener que llevar un vector de booleanos que me dice que trabajos han sido asignados.
Para cada trabajo (T/F) si ya ha sido asignado o no.

Hay una matriz de tiempos
funcionario

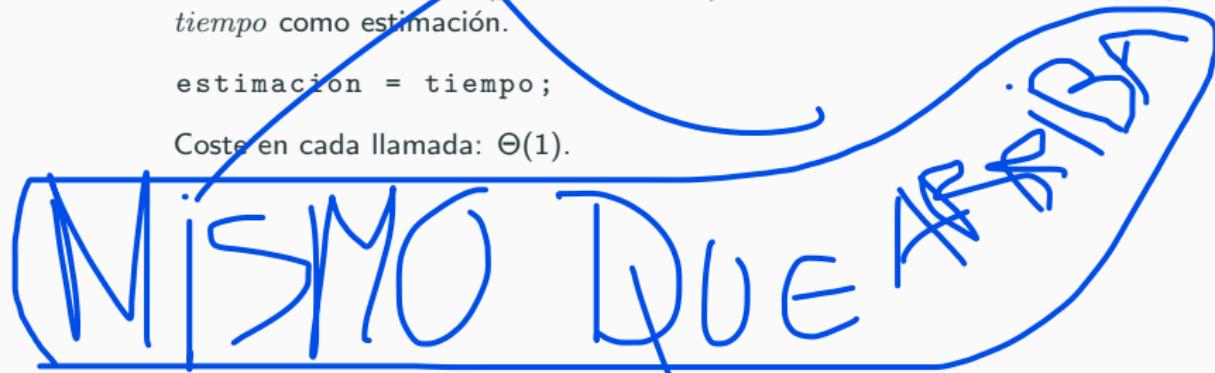
TIEMPO

- Representamos la solución en tuplas de la forma $(x_0, x_1, \dots, x_{n-1})$ donde x_i es el trabajo asignado al funcionario i .
- Ya que cada trabajo solo puede ser asignado a un funcionario, llevaremos cuenta de los trabajos ya asignados en un vector marcador *asignado* de booleanos.
- Para la solución parcial (x_0, \dots, x_k) , el tiempo hasta el momento es $tiempo = \sum_{i=0}^k T[i][x_i]$, y tenemos que estimar el tiempo del resto de la solución:

- ① La opción más sencilla (y más optimista) es aproximar con 0 este tiempo, y utilizar *tiempo* como estimación.

```
estimacion = tiempo;
```

Coste en cada llamada: $\Theta(1)$.



Funcionarios con trabajo

calculamos el mínimo y para los funcionarios que quedan ($n-k-1$) decimos que todos tardan ese mínimo.

Otra opción

Considerar que todos son tan rápidos como el más rápido

- ② Otra posibilidad es calcular un mínimo global de la matriz T ,

$$\min T = \min\{T[i][j] \mid 0 \leq i \leq n-1 \wedge 0 \leq j \leq n-1\},$$

que sirva como cota inferior al tiempo de realización de cada trabajo. Desde $k+1$ hasta $n-1$ quedan $n-k-1$ funcionarios a los que les falta trabajo por asignar:

estimacion = tiempo + (n-k-1)*minT; desde k+1 hasta el final

El mínimo de la matriz T se calcula antes de la llamada inicial a la vuelta atrás y se pasa como parámetro.

Ya que recorremos la matriz y vamos obteniendo cuál es el mínimo

- Coste de cálculo inicial: $\Theta(n^2)$.
- Coste en cada llamada: $\Theta(1)$.

Esto también es demasiado optimista. No es nada realista que todos los restantes vayan a tardar tanto como el que menos tarda.

Funcionarios con trabajo

Versión menos optimista: poner el mejor tiempo de cada funcionario. Esto va a ser más realista. Nos va a PODAR MÁS

- ③ Tener calculado para cada funcionario k el mínimo de los tiempos de los funcionarios que quedan por asignar: vector de mínimos que almacena para cada funcionario el menor de sus trabajos restantes

$$\minimo[k] = \min\{ T[i][j] \mid k + 1 \leq i \leq n - 1, 0 \leq j \leq n - 1\}.$$

Coges la matriz inicialmente y calcular el mínimo de cada fila.

Funcionarios con trabajo

Creo que es que calculamos para cada funcionario, cuanto tarda cada uno de los siguientes en realizar el trabajo más rápido.

- ③ Tener calculado para cada funcionario k el mínimo de los tiempos de los funcionarios que quedan por asignar:

$$\text{minimo}[k] = \min\{T[i][j] \mid k+1 \leq i \leq n-1, 0 \leq j \leq n-1\}.$$

De esta manera podemos calcular la estimación del tiempo restante como:

`estimacion = tiempo + (n-k-1)*minimo[k];` El mínimo de los funcionarios restantes

Para que sea eficiente, el vector `minimo` se calcula antes de la llamada inicial a la vuelta atrás y se pasa como parámetro:

- Coste de cálculo inicial: $\Theta(n^2)$.
- Coste en cada llamada: $\Theta(1)$.

Calculamos la suma de los mínimos

Busca el tiempo mínimo restante para cada funcionario k

Funcionarios con trabajo

Pero si tienes en cuenta que ya hay trabajos asignados hay otros mínimos.

()

- ④ Tener calculado un mínimo por cada fila: para cada funcionario, cuánto tarda en realizar el trabajo que realiza más rápidamente:

$$\text{rapido}[i] = \min\{T[i][j] \mid 0 \leq j \leq n - 1\}.$$

De esta manera podemos calcular la estimación del tiempo restante como:

$$\text{sumaRapido}[k] = \sum_{i=k+1}^{n-1} \text{rapido}[i].$$

Funcionarios con trabajo

Consiste en hallar cual es la opción mejor, cual es el menor tiempo de los no asignados e ir sumando esas opciones

- ④ Tener calculado un mínimo por cada fila: para cada funcionario, cuánto tarda en realizar el trabajo que realiza más rápidamente:

$$\text{rapido}[i] = \min\{T[i][j] \mid 0 \leq j \leq n - 1\}.$$

De esta manera podemos calcular la estimación del tiempo restante como:

$$\text{sumaRapido}[k] = \sum_{i=k+1}^{n-1} \text{rapido}[i].$$

```
estimacion = tiempo + sumaRapido [k];
```

Para que sea eficiente, el vector sumaRapido se calcula antes de la llamada inicial a la vuelta atrás y se pasa como parámetro:

- Coste de cálculo inicial: $\Theta(n^2)$.
- Coste en cada llamada: $\Theta(1)$.

ES EL MÁS REALISTA DE TODOS LOS QUE HEMOS VISTO.

En este enfoque se busca el tiempo mínimo de trabajo más rápido

- ⑤ Calcular el vector `sumaRapido` dinámicamente entre los trabajos no repartidos ya:

$$\text{rapido-din}[j] = \min\{ T[i][j] \mid 0 \leq j \leq n - 1 \wedge j \text{ no ha sido asignado}\}.$$

$$\text{sumaRapido}[k] = \sum_{i=k+1}^{n-1} \text{rapido-din}[i].$$

Coste en cada llamada: $\Theta(n^2)$.

Es necesario valorar experimentalmente si compensa el coste de determinar si se poda o no respecto a la cantidad de nodos podados.

Funcionarios con trabajo

- ⑤ Calcular el vector `sumaRapido` dinámicamente entre los trabajos no repartidos ya:

$$\text{rapido-din}[i] = \min\{\Gamma[i][j] \mid 0 \leq j \leq n - 1 \wedge j \text{ no ha sido asignado}\}.$$

$$\text{sumaRapido}[k] = \sum_{i=k+1}^{n-1} \text{rapido-din}[i].$$

Coste en cada llamada: $\Theta(n^2)$.

Es necesario valorar experimentalmente si compensa el coste de determinar si se poda o no respecto a la cantidad de nodos podados.

Tardo demasiado tiempo en calcular la poda

Funcionarios con trabajo

Las sumas acumuladas no hacerlo cuadrático

matriz de tiempos

$k = 1$

Coste = 10

0

2	3	7	5	6
6	1	2	4	3
9	5	3	7	5
10	7	4	5	9
6	6	6	6	5

minimo	rapido	sumaRapido
1	2	13
3	1	12
4	3	9
5	4	5
0	5	0

Para los funcionarios 0 y 1. Ya lo tenemos

A medida que aumentamos la opción aumenta la poda, luego mejor

OPCION

Que todos tarden 0

opcion1

que los restantes tarden el mínimo de la matriz (1 en este caso)

opcion2

que tarden como el mínimo de los restantes. El mínimo a partir del 2 fun es 3.

$10 + 3 * 3 = 19$ opcion3

que tarden como mínimo la suma de los mínimos de los restantes.

$10 + 12 = 22$ opcion4

$3+4+5 = 12$ opcion5

$10 + (5+5+5) = 26$ opcion5

Gastas tiempo cuadrático en calcularla

Las distintas cotas

Funcionarios con trabajo

precálculo de la estimación

Número de funcionarios y, por tanto, de trabajos

```
void preCalculoEst(int n, vector<vector<int>> const& tiempos,
                    vector<int> & sumaRapido) {
    // mínimo de cada fila
    vector<int> rapido(n);
    for (int i = 0; i < n; ++i) {
        rapido[i] = tiempos[i][0];
        for (int j = 1; j < n; ++j) {
            rapido[i] = std::min(rapido[i], tiempos[i][j]);
        }
    }
    CÁLCULO DEL MÍNIMO DE CADA FILA. Y LO GUARDAMOS
    // calculo de las estimaciones
    sumaRapido[n-1] = 0; No hay trabajadores restantes después del último
    for (int i = n-2; i >= 0; --i) { Desde n-2 hasta i=0 (penúltimo hasta el primero)
        sumaRapido[i] = sumaRapido[i+1] + rapido[i+1];
    }
}
```

SUMARAPIDO SUMA EL MÍNIMO DE CADA FUNCIONARIO.

matriz con los tiempos

Opción 4

para cada posición calculas el mínimo de cada posición

calcula el mínimo entre el primero y el siguiente y lo guarda en un vector de rápidos.

Ir sumando de abajo arriba acumulando los valores

Llevas la suma de los mínimos de la siguiente k en adelante

Nos pondrá dos ejercicios. Procedimiento: 1 versión sin poda. 2 probar con las podas. En el examen habrá que proceder de manera parecida. Primero una versión sin poda y otra con podas. Es decir comenzar el ejercicio y hacer una versión sin poda y si está bien realizarla con poda.

Funcionarios con trabajo

número de funcionarios y trabajos

```
void funcionarios(int n, vector<vector<int>> const& tiempos,
vector<int> const& sumaRapido, vector<int>& sol, int k, int& tiempo,
vector<bool>& asignado, vector<int>& sol_mejor, int& tiempo_mejor) {
    for (int t = 0; t < n; ++t) {
        if (!asignado[t]) {
            sol[k] = t; sol_parcial
            asignado[t] = true; //Marcamos la solución
            tiempo += tiempos[k][t]; tiempo acumulado
            int tiempo_estimado = tiempo + sumaRapido[k];
            if (tiempo estimado < tiempo_mejor) {
                if (k == n-1) {
                    sol_mejor = sol;
                    tiempo_mejor = tiempo;
                } else {
                    funcionarios_va(n, tiempos, sumaRapido, sol, k+1,
                                    tiempo, asignado, sol_mejor, tiempo_mejor);
                }
            }
            asignado[t] = false; desasignar
            tiempo -= tiempos[k][t]; }
    }
}
```

La k recorre los funcionarios y recorremos los trabajos con el bucle.

matriz de tiempos

almacena la solución parcial

tarea actual

tiempo acumulado

Almacenan la mejor solución y el mejor tiempo

si NO está
asignado

Iteramos sobre los funcionarios disponibles.

MARCAJE

si es menor entonces
si es el último el tiempo
mejor será el acumulado

calculamos el tiempo
estimado

si el tiempo estimado es menor que el tiempo mejor.

DESMARCAJE

desmarcaje

y le restamos el tiempo que habíamos añadido

Funcionarios con trabajo

```
//Lectura de datos
//...
// cálculo de las estimaciones
vector<int> sumaRapido(n);
preCalculoEst(n, tiempos, sumaRapido); Precálculo inicial

int tiempo = 0;
vector<bool> asignado(n, false); asignar todos los funcionarios a false
vector<int> sol(n);
int tiempo_mejor = 1<<30;
vector<int> sol_mejor(n);
funcionarios_va(n, tiempos, est, sol, 0, tiempo, asignado,
                sol_mejor, tiempo_mejor); Llamada inicial

cout << tiempo_mejor << "\n";
//imprimir(sol_mejor);
```

Coste cuadrático

IMPORTANTE LA ESTRATEGIA DE LA ELABORACIÓN DEL EXAMEN.