# Dynamic Programming

Este es el paradigma más difícil de aplicar, este paradigma está enfocado en recursiones y relaciones recurrencia.

La clave y habilidad a adquirir deberá ser con respecto a la identificación de los estados de los problemas y determinar las relaciones o transiciones entre los problemas actuales y sus sub problemas, y precisamente estas habilidades fueron empleadas para desarrollar soluciones con backtracking recursivo, es por esto mismo que algunos problemas de DP con restricciones de entrada son pequeñas (la entrada es pequeña) es posible resolverlo con bactracking aun siendo un problema de DP, es por esto mismo que complete search (en especial backtracking) tienen mucha relación con DP.

De lo anterior, hay dos enfoques de Dp: top down y bottom up, y lo primero a abordar será top down

## Top down

Se puede considerar que DP (top down) es la versión inteligente o rápida del backtracking recursivo debido a que muchos datos a calcular ya fueron calculados antes así que solo se debe de acceder a ellos. Por ende, para este tipo de soluciones es muy conveniente el uso de un árbol de decisiones

DP es primordialmente usado para resolver problemas de optimización o contar las soluciones existentes, por lo cual al encontrar alguna de las siguientes palabras en un problema se puede intuir que su solución será usando DP:

1. Minimiza esto
2. Maximiza esto
3. Cuenta las maneras de hacer esto

Si estas palabras se encuentran en un problema no significa que automáticamente se trata de un problema de DP, pero si tiene muchas oportunidades de que sea un problema que se resuelve con DP

En los concursos se suele preguntar por el valor optimo/total y no la solución óptima por sí mismo, los cuales al no tener que producir una solución optima se estaría eliminando la necesidad de usar backtracking

No obstante, los problemas más difíciles se requiere que se devuelva la solución optima de alguna manera.

## UVa 11450 - Wedding Shopping

Este problema se trata que son unos amigos el cual uno de ellos ya se va casar e invitó a sus demás amigos, el problema es que ellos no tienen ropa formal como para asistir a la boda (algo raro, ¿no?), en fin. Deben de visitar a una tienda ropa en donde le ofrecen una cantidad ‘c’ de diferentes tipos de prendas, y se tiene M de presupuesto, pero necesitan comprar 1 de cada prenda.

El chiste es el mismo que el ejercicio de CD – 624, se tiene que comprar una pieza de cada prenda de tal modo que el presupuesto restante se el menor posible.

Se puede ver la similitud con el ejercicio de CD, y también se puede apreciar como un cierto problema de la mochila modificado. Por ende, se procede a ver que si aplicamos un algoritmo greedy como el de las monedas se puede tener un momento en donde escoger el número mayor haga que no hay solución y por ende genera un WA, por otro lado complete search sí lo puede resolver empleando backtracking recursivo pero el tiempo es demasiado.

Para visualizar la cantidad de operaciones se podría ver con un árbol, aunque se observa que simplemente en cada decisión habrá dos posibilidades que será tomar la prenda o no, entonces hay dos posibilidades y esto mismo sucederá por cada modelo de cada prenda.

Se debe de tener en cuenta que puede existir C diferentes tipos de prendas, y habrá k modelos de cada prenda. Los límites de cada es hasta 20, o sea que se puede tener como máximo 20 prendas diferentes y en cada prenda se puede tener 20 diferentes tipos de modelos de cada prenda, o sea multiplicar 20 veces 20 para la cantidad más extrema de datos que se puede llegar alcanzar en un test case

Entonces la cantidad de tiempo en un máximo será de $20^{20}$.

Analizando “no solution”

Esto sucede cuando el presupuesto es muy pequeño y el precio de las cada modelo de las prendas son muy elevadas:

However, suppose we have this test case B with $M = \mathbf{9}$ **(limited budget)**, $C = 3$:
Price of the 3 models of garment g = 0 → 6 4 8
Price of the 2 models of garment g = 1 → 5 10
Price of the 4 models of garment g = 2 → 1 5 3 5

En este test case no hay forma alguna en el que se pueda comprar un modelo de cada prenda sin pasarse del presupuesto, entonces se dice que es “no solution”

**Complete Search - solución**

Lo primero a ver es que DP será como la solución mejorada para este ejercicio, pero primero se empieza usando un backtracking. Entonces, se observa que se debe de estar preparado cuando el precio de cualquier subconjunto de modelos sobrepasa el presupuesto, o sea que llega a una solución negativa, por ende se podría podar esta parte y no seguir realizando cálculos o simplemente regresar un valor muy grande negativo para que no se tome en cuenta esa solución

Lo otro es sobre la formula a seguir, como en el caso de los CD donde la formula fue usando un mínimo

**Aplicando el concepto de DP (top down)**

Para aplicar esto se debe de asegurar de que se tiene lo siguiente

1. Sub estructuras o sub problemas óptimos, o sea que la resolución de los sub problemas arman a la solución del problema original, o sea que nuestra elección final deberá ser óptima.

2. Se tiene superposición de sub problemas, esto hace referencia a que se repite el cálculo de los sub problemas para armar la solución de otro subproblema.

Estas son las claves, pero en especial el segundo punto debido a que en realidad la complejidad máxima no sería $20^{20}$ debido a que hay muchos cálculos que se repiten, o sea que fácilmente se podría guardar esos valores para posteriormente consultarlos cuando se solicite para armar otra solución de un sub problema. Esto mismo es la idea de la técnica de DP memoization.

Para observar que en efecto hay sub problemas que se repiten su cálculo, se debe de pensar en que hay precios que se repite por ende el mismo subproblema será calculado más de una vez, además de que se debe de calcular por cada elección

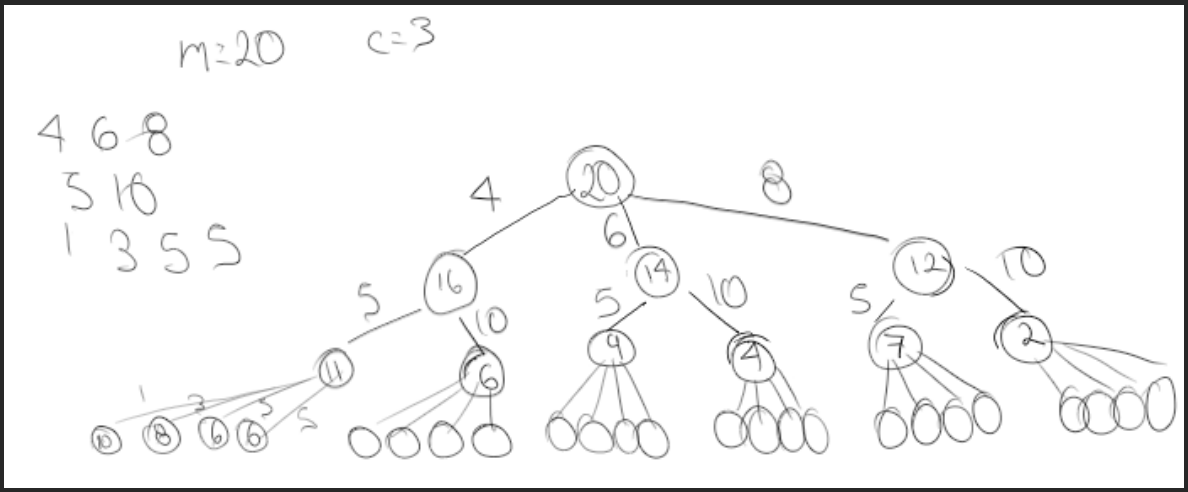
Entonces, en realidad se estaría teniendo 201 (presupuesto) * 20 (prendas) y además otros 20 por cada modelo. Si nos aseguramos que cada sub problema sea calculado solo una vez el programa será mucho más rápido

Entonces, se podría iniciar primero con la solución por medio de backtracking y se toma en cuenta que se debe de escoger uno de cada uno a fuerzas, así que se tomará y se tendrá que verificar si en caso que la suma de los precios sobrepase el presupuesto entonces no se podrá tener una solución.

Al revisar bien, en realidad esto no se trata de tener dos opciones de tomar o no, en caso de que una prenda haga que se sobrepase del presupuesto se tendrá que podar toda esa solución y no tomarla.

```
int minimo=10000; //Me va guardar la cantidad restante después de las compras

bool comprar(int gasto, int valor)
    {
        if(gasto-valor<0)   return false;
        else    return true;
    }

int backtrack(int indice, int gasto)
    {
        // Caso base:
        if(indice == c) // Ya se recorrió todas las prendas
            {
                return gasto;
            }

        for(int modelo=1;modelo<=prendas[indice][0];modelo++)
            {
                if( comprar(gasto,prendas[indice][modelo]) )
                    {
                        minimo = min(minimo,backtrack(indice+1,gasto-prendas[indice][modelo]) );
                    }

            }
        return minimo;  // Al final se retorna la respuesta, el más mínimo dinero disponible.
    }
```

Se tiene la versión de backtracking, por lo cual para introducir el algoritmo de DP por medio del acercamiento de memoization será simplemente agregar una tabla que vaya guardando esos datos, entonces si se repite ese subproblema tan solo se busca en la tabla de memoization para devolver ese valor que ya fue calculado




En este caso se puede ver que en caso que exista un valor del inicio repetido, o en cualquier prenda, si se repite el precio de un modelo de la **misma prenda** entonces se observa que se tendría que hacer las mismas operaciones que anteriormente ya se habían hecho.

Es importante ver que debe ser de la misma prenda debido a la naturaleza del problema, o sea que al ser el mismo precio en la misma prenda se tendrá que hacer igual las mismas operaciones que antes fueron hechos en la primera aparición de dicho precio del modelo de esa prenda.

Entonces, en nuestra tabla de memoization debe de localizar el valor necesario por medio de la prenda y el gasto que se lleva actualmente, esto se debe de ver que es para generalizar la tabla. Puesto que si tenemos un precio repetido en la primera prenda se observa que es fácil observar que los índices serán

Modelo k

En prenda g

La cuestión es que si accedemos de este modo ¿cómo podríamos hacerle si un precio se repite en la segunda, tercera, cuarto, etc prenda? Para esto nos vamos a valer del precio puesto que al ser los mismos precios de los modelos va a generar el mismo resultado al restar el precio con presupuesto restante.

La implementación es realmente sencilla en este caso y queda de la siguiente manera

```
int backtrack(int indice, int gasto)
    {
        // Caso base:
        if(indice == c) // Ya se recorrió todas las prendas
            {
                return gasto;
            }

        // Verificando si el subproblema (state) ya fue solucionado anteriormente
        if( memorizacion[indice][gasto] != -1)
            return memorizacion[indice][gasto];

        // En caso contrario se tendrá que computar.
        for(int modelo=1;modelo<=prendas[indice][0];modelo++)
            {
                if( comprar(gasto,prendas[indice][modelo]) )
                    {
                        minimo = min(minimo,backtrack(indice+1,gasto-prendas[indice][modelo]) );
                    }

            }
        // También se pude hacer;
        //memorizacion[indice][gasto]=minimo;
        // return memorizacion[indice][gasto];
        return memorizacion[indice][gasto]=minimo;
        // Al final se retorna la respuesta, el más mínimo dinero disponible.
    }
```

El chiste es observar que efectivamente tenga dichos sub problemas que se repiten para que esto sea mucho más rápido

De aquí se puede observar que este acercamiento o modo de implementar DP por medio de memoization es literal tomar la forma nativa de complete search (bactracking recursivo) y encontrar en dónde o cómo se podría guardar dichas soluciones de sub problemas que se repiten constantemente, entonces este sería otra cosa que podría dificultarse.

Esta forma de realizar una solución por medio de DP es llamada: memoization o top-down, y se relaciona directamente con una implementación de un bactracking recursivo

Ahora la complejidad es menor

Los sub problemas a resolver son llamados “states” en términos de DP

Las recurrencias son llamadas “transiciones” en términos de DP usando un acercamiento de Complete Search

**Complejidad del algoritmo**

La complejidad espacial será de $O(M*g + k^2)$

Debido a que se estará guardando los precios de los modelos deuna tabla de dos dimensiones, además de que se ocupará espacio para la tabla de memoization

Time complexity: es de O(M*k) por cada caso, debido a que se tiene M estados producidos por el gasto y k precios de cada modelo.

Entonces, en si se tendría O(M*k*g) en total, M estados, k modelos y g prendas.

**Extra**

Es posible usar la dirección de memoria del variable mínimo para evitar mandar a llamar la tabla dos veces.

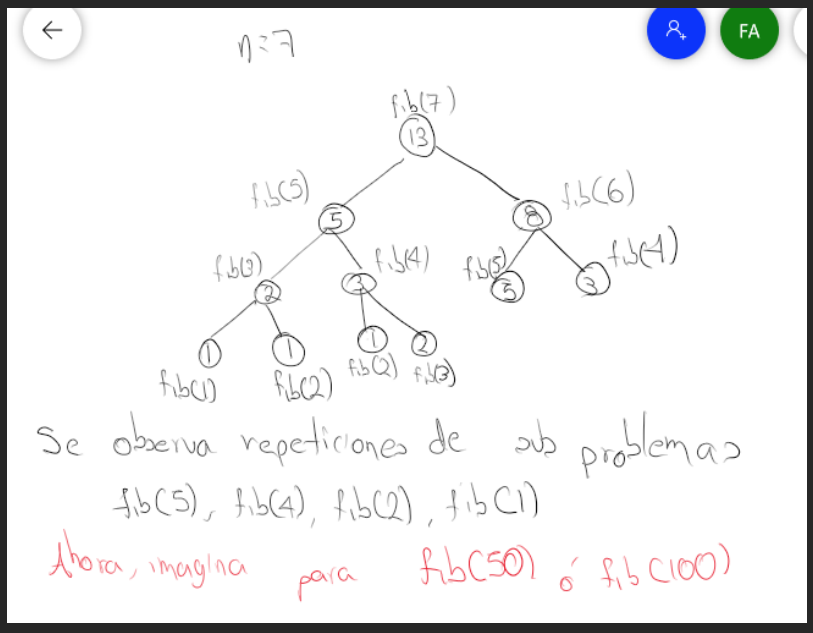
```
int &ans = memo[money][g];                    // remember the memory address
if (ans != -1) return ans;
for (int model = 1; model <= price[g][0]; model++)
  ans = max(ans, shop(money - price[g][model], g + 1));
return ans;                  // ans (or memo[money][g]) is directly updated
```

Acercamiento por Fibonacci

```
int fibonacci(int n)
    {
        if(n==0)    return 0;
        if(n==1)    return 1;

        return fibonacci(n-1) + fibonacci(n-2);
    }

int main()
    {
        cout<<fibonacci(7);

        return 0;
    }
```

Este es realizar la solución por medio de un backtracking, de tal modo que habrá dos decisiones siempre por la naturaleza del return final.

Su árbol será



La complejidad del programa es exponencial $O(2^n)$ y por ende al querer calcular el número de fibonacci 50 simplemente se tendrá que realizar una barbaridad de operaciones.

Se observa que los sub problemas (states) son cuando se debe de calcular los dos números de Fibonacci anteriores al valor que deseamos calcular, y de estos se despreden otras 4 sub problemas en total para cacular esos valores anteriores de fibonacci que se necesita para calcular el número de Fibonacci solicitado

Se observa entonces que estos son los sub problemas que se debe de solucionar.

También es posible ver que la repetición computar dichos sub problemas existe y en ocasiones se pide computar el mismo state (sub problema) más de una vez.

Entonces se concluye que es posible una solución por DP y se usará memorización, entonces lo que se debe de hacer es memorizar esos resultados que se retona y de forma explícita se tiene lo siguiente

```cpp
unsigned long long memo[100];

long long fibonacci(unsigned long long n)
    {
        if(n==0)    return 0;
        if(n==1)    return 1;

        if(memo[n]!=-1) return memo[n];

        // En caso contrario
        memo[n] = fibonacci(n-1) + fibonacci(n-2);

        return memo[n];
    }

int main()
    {
        int modulo = 1e9+7;
        memset(memo,-1,sizeof(memo));
        cout<<(fibonacci(50));

        return 0;
    }
```

**Bottom-Up DP – método de tabulación**

Se trata de otro enfoque que se le puede dar al DP y es implementar mediante una tabulación.

Este es la verdadera forma de DP, DP fue originalmente conocido como método de tabular, es una técnica de la computación la cual implica el uso de una tabla.

Pasos para una solución

1. Determina el conjunto requerido de parámetros únicos que describen al problema, el cual es un paso similar al descrito bactracking recursivo y topdown DP.

2. Si tenemos N parámetros que se requieren para representar los states (sub problemas) se debe de preparar una tabla DP de dimensión N con una entrada por estado. Esto podría tener cierta equivalencia con la tabla de memoization de DP top down, pero hay diferencias

    a. En la tabla de DP necesitamos inicializar algunos valores que ya conocemos gracias a los casos base. Por otro lado, en la tabla de memoization toda la tabla estará inicializada con valores sin significados (usualmente el -1) los cuales me estarán indicando qué estados no han sido computados todavía.

3. Al tener ya listos y reconocidos las celdas con valores de los estados base, la tabla DP estará lista para ser llenada por la siguiente transición. Se repetirá este proceso hasta que la tabla DP esté llena de forma completa, para este estilo de bottom up la parte de llenar la tabla DP se hace por medio de loops

De esta última característica se puede ver cierta similitud existente con top down, debido a que top down se implementa por medio de recursividad se debe de tener casos base para romper la recursividad y tener una respuesta para los sub problemas (states), mientras que aquí los casos bases que encontrados deben ser utilizados para inicializar la tabla y poder encontrar la siguiente solución de los sub problemas (states).

**Fibonacci – bottom up**

Primero, se debe de entender el cómo será llenada la tabla debido a que no se puede comenzar tal cual en un programa ya “base” como un top down ya que no está basado de un complete search, no obstante es posible es posible aplicar ciertos criterios como la búsqueda de dlos sub problemas (states) para ir armando la solución

Entonces, primero

1. Parámetros, sabemos que el parámetros a tener será el propio número, entonces la tabla será de una dimensión
2. Casos bases, se sabe que para 0,1,2 su número de Fibonacci son 0,1,1 estos son nuestros casos base
3. Colocar nuestros casos base en la tabla para rellenarla

Por ejemplo, calcular fib(6)

Esta es nuestra tabla actualmente

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | | | | |

Se podrá ver que en cada iteración del loop será posible mandar a llamar a los últimos dos celdas (con valores) para realizar la suma del nuevo valor

Entoncse quedaría celda 4 = dp(2) + dp (1) =>>  celda_4=1+1 =>> celda_4=2

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | | | |

Ahora, la celda_5

Celda_5 = dp (3)+ dp (2)  =>>  celda_5=2+1  =>> celda_5=3

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 3 | | |

Celda_6

Celda_6= dp (4)+fib(3)  =>>  celda_6=3+2  =>>  celda_6=5

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 3 | 5 | |

Celda_7

Celda_7= dp (5)+ dp (4)  =>> celda_7=5+8  =>> celda_7=8

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 3 | 6 | 8 |

Es muy importante hacer la observación que el resultado es el que se encuentra en la última posición. En cada problema de DP se debe de hacer un análisis para ver en dónde queda ubicado el resultado

Así quedaría implementada la tabla, entonces se puede deducir la formula usada para conocer estos valores

Dp[n] = dp (n-1)+ dp (n-2)

El cual va estar dentro de un arreglo de tamaño n

Es por esto que se puede decir que su complejidad de espacio es ‘n’

Space complexity = O(n)  Time complexity = O(n)

```
long long dp[100];

long long fibonacci(int n)
    {
            // inicializando con los casos base
        dp[0]=0;
        dp[1]=1;
        dp[2]=1;

        for(int i=3;i<=n;i++)
            {
                dp[i] = dp[i-1] + dp[i-2];
            }
        // al final devuelvo el resultado
        return dp[n]; // Porque está indexado en 0
    }

int main()
    {
        cout<<(fibonacci(50));
        return 0;
    }
```

La solución es más corta, y se necesitó de ese i<= para que llegue a dp[50] =

Estas observaciones pueden ser claves para algunos ejercicios

Ahora, algo importante a mencionar es que esta forma de implementar el bottom up es guardando los valores en las celdas del arreglo, no obstante existe otra manera y es usando un arreglo de booleanos en donde el resultado de cada sub problema será usado como índice y el valor en los índices solo serán true o false.

Esta manera es usar una matriz de booleanos, o sea manejando puro true o false

Entonces para el problema de la serie de Fibonacci se tendría algo como lo siguiente

Se va calcular fib(6)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

La primera columna es solo para ubicar los índices, no es otra dimensión

Entonces, como primer paso se debe de inicializar con true los casos base

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| true | True | true | | | | | | | |

Entonces, se va a tomar como índice de la tabla

Para fib(4)

dp( (3-1) + (3-2)) =>> dp[3] = true; -> Valor del índice indexado en 0 como en la tabla

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| true | True | true | true | | | | | | |

Para fib(5)

dp( (4-1) + (4-2)) =>> dp[5] = true;

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| true | True | true | true | | true | | | | |

Para fib(6)

dp( (5-1) + (5-2)) =>> dp[8] = true;

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| true | True | true | true | | true | | | true | |

Entonces, será el último true de la tabla el que tenga el resultado

El problema de esta forma de realizarlo es que tendría que ocupar mucha memoria para ser capaz de almacenar el resultado en esa posición, por ejemplo en caso de calcular el número de Fibonacci 50 tendríamos que tener un arreglo de tamaño 12586269025

Puesto que en dp[12586269025] esa posición contendrá la solución, en este caso este modo no es óptimo para nada, pero es posible aplicar esta forma de matriz de booleanos en otros problemas, un ejemplo sería en el problema de wedding shopping

**Bottom up con matriz de booleanos**

La aplicación de la matriz de booleanos es guardar la diferencia del presupuesto con los precios de cada modelo, entonces se estará computando renglón por renglón debido a que al tener dos parámetros se tendrá que ocupar una matriz de 2 dimensiones.

1. La dimensión será de dos dimensiones
2. Las celdas que deberán ser inicializados serán la resta del presupuesto m con el precio de cada modelo de la primera prenda g=0.
3. Resolver

Entonces, la tabla que se tendría es la siguiente

Ejemplo de test case

20 3

3 4 6 8

2 5 10

4 1 3 5 5

Nota: las primeras columnas no son datos, únicamente son los índices para guiarnos

Nota2: 0-false y 1-true

Entonces, como para comprar otro modelo de la siguiente prenda es obligatorio que se haya hecho la compra de un modelo anterior de una prenda anterior. Por ende, se debe de ver que el factor determinante para realizar una próxima compra es que debe ser true. Es por esto mismo que se debe de usar los posibles caminos a tomar que son las primeros precios de los modelos de la primera prenda.

20-4 = 16 20-6= 14 20-8=12

| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| **2** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Por ende, si se compró una prenda en ese mismo camino se puede continuar comprando otra modelo del siguiente garment, siempre y cuando el precio no sea negativo

Para 12:

12-5= 7 12-10=2

Para 14

14-5=9 14-10=4

Para 16

16-5=11 16-10=6

Estos son precios de los diferentes caminos posibles a tomar del árbol de decisiones, o sea que por los caminos diferentes tomados generan diferentes presupuestos restantes

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Ahora, el último garment

Se observa que solo nos interesa el renglón actual y el anterior, ya no me estaría interesando el renglón 1, solo el 2 y 3

Para 2:

2-1= 1 2-3=false 2-5=false 2-5=false

Para 7

7-1=6 7-3=4 7-5=2 7-5=2

Para 4

4-1=3 4-3=1 4-5=false 4-5=false

Para 9

9-1=8 9-3=6 9-5=4 9-5=4

Para 6

6-1=5 6-3=3 6-5=1 6-5=1

Para 11

11-1=10 11-3=8 11-5=6 11-5=6

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 1/1/1 | 1 | 1/1 | 1/1 | 1 | 1/1/1 | 0 | 1/1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

En este caso, esta es la respuesta, el mínimo presupuesto que queda después de comprar 1 prenda de cada uno se encuentran en el último renglón, de tal modo que al querer que el presupuesto sea el menor posible se escoge el primer 1 que aparece.

En este caso hay 3 posibles soluciones o caminos de comprar 1 prenda de cada uno para cumplir con lo solicitado

**Presupuesto restante después de las compras: 1**

Código

Inicializando con los casos base

```
// Inicializar con los casos base
for(int j=1;j<=precios[0][0];j++){
    if(m-precios[0][j] >= 0)
        dp[0][m-precios[0][j]]=true;
}
```

Parte del dp

```
    for(int a=1;a<c;a++)
        {
            //Verificando si son true esas celdas
            for(int presupuesto_restante=0;presupuesto_restante<=m;presupuesto_restante++)
                {
                    if(dp[a-1][presupuesto_restante]==true) // Si es true se continua ese camino
                        {
                            for(int c=1;c<=precios[a][0];c++)
                                {
                                    //Si la compra no se pasa del presupuesto_restante entonces se añade
                                    if(presupuesto_restante-precios[a][c]>=0)
                                        {
                                            dp[a][presupuesto_restante-precios[a][c]]=true;
                                        }
                                }
                        }

                }
        }

    //Recorrer para ver si en el último renglón hay algún 1, puesto que ese índice es la respuesta
    int presupuesto=0;  // El presupuesto sobrante, se escoge el menor
    for(presupuesto=0;presupuesto<m && dp[c-1][presupuesto]!=true;presupuesto++);

    // En donde pare estará el índice respuesta
    if(presupuesto==m)  printf("no solution\n");
    else                printf("%d\n",m-presupuesto);
}
```

En ocasiones suele ser un problema encontrar los casos base para soluciones DP por medio de bottom up.

**Uso y ventaja de bottom up**

La ventaja de usar este tipo de acercamiento a DP es cuando queremos la última actualización de las soluciones, o la última actualización de los states, entonces se puede ver que su uso va encaminado normalmente a eso

**Truco para salvar (guardar) memoria – optimizar el uso de memoria**

Este truco nace de la observación que hay ejercicios donde solo importa conocer el renglón actual y el anterior, como el ejercicio wedding shopping en donde necesitamos saber los presupuestos restantes de cada camino tomado para poder tomar la siguiente decisión, o sea el siguiente state.

Entonces, en lugar de tener 'n' renglones es posible reducir la idea a tener solo 2 renglones y m columnas.

El uso de este truco queda en referenciar a un renglón como el actual y el otro como el previo. Para que después de procesar el estado se haga un intercambio, es decir

1. Renglón 0 -> previo
2. Renglón 1 -> actual

*Después de procesar el state*

1. Renglón 0 -> actual
2. Renglón 1 -> previo

Entonces se estará computando renglón por renglón y se debe de notar que en realidad se estará perdiendo la información de los anteriores renglones, por ejemplo si son 3 renglones en total al final solo tendré la información de los últimos 2 renglones (1,2) mientras que la información del primer renglón se perderá

Implementación:

```
for(int a=1;a<c;a++)
    {
        if(a%2==0)
            {
                renglonA=1; // Renglón anterior
                renglonB=0; // Renglón actual
            }
        else
            {
                renglonA=0; // Renglón anterior
                renglonB=1; // Renglón actual
            }
```

Voy a escoger qué renglón funcionará como el actual y anterior, en este caso el nombre se conserva pero los valores de cada variable van variando.

El criterio fue haciendo la siguiente observando: después de un número par el siguiente número es un número impar, entonces usando este criterio será posible ir haciendo el intercambio de los renglones para cambiar y saber qué renglón será el actual y el previo

Lo demás de las iteraciones es igual, solo cambiando el nombre de las variables

```
//Verificando si son true esas celdas
for(int presupuesto_restante=0;presupuesto_restante<=m;presupuesto_restante++)
    {

        if(dp[renglonA][presupuesto_restante]==true) // Si es true se continua ese camino
            {
                for(int c=1;c<=precios[a][0];c++)
                    {
                        //Si la compra no se pasa del presupuesto_restante entonces se añade
                        if(presupuesto_restante-precios[a][c]>=0)
                            {
                                dp[renglonB][presupuesto_restante-precios[a][c]]=true;
                            }
                    }
            }
        // Una vez computado lo borro
        dp[renglonA][presupuesto_restante]=false;

    }
```

Una última observación es que al momento de tener entradas en donde solo sea una única prenda, o sea que de garments solo sea 1, entonces mi método será que me designe que el último renglón es aquel que tiene la respuesta, en este caso el último renglón debe ser en renglón actual, pero al ser solo 1 el conjunto entonces no hay ninguno otro, por ende el último renglón es el 0 el cual son los valores inicializados en la función principal

```
//Escoger el último renglon
int ultimorenglon; // El último renglón es el renglón actual y ese es el

if(c==1)    ultimorenglon=0;
else    ultimorenglon=renglonB;

//Recorrer para ver si en el último renglón hay algún 1, puesto que ese índice es la respuesta
int presupuesto=0;  // El presupuesto sobrante, se escoge el menor
for(presupuesto=0;presupuesto<m && dp[ultimorenglon][presupuesto]!=true;presupuesto++);

// En donde pare estará el índice respuesta
if(presupuesto==m)  printf("no solution\n");
else                printf("%d\n",m-presupuesto);
```

Así que al ser C==1 (un solo garment) pues se escoge aquel precio del modelo que sea más grande, por ende en la resta se observa como aquel que haga que la resta con el presupuesto sea el menor.

Nota:

En este ejercicio no es tan significativo el ahorro de memoria, pero podríamos imaginar casos en donde la cantidad de prendas puede llegar a ser hasta de 2000 y ahí sí podríamos generar problemas con el tema de la memoria. Guardar este truco

Por otro lado, existe también un cierto truco equivalente, no es igual pero estaría funcionando más o menos con el mismo objetivo para el DP memoization – top down

**Usar un árbol binario de búsqueda balanceado como tabla memoization**

Se recuerda que el BTS balanced en c++ es el mapa, STL map.

En este caso es simplemente asociar lo que iba en los corchetes como llaves para después poner como valor el resultado de ese sub problema.

No obstante, el problema de esto al parecer es que con un tema de las constantes donde por notación big O se suele dejar los valores más significativos y por ende las constantes no se toman en cuenta, pero dependiendo del problema se puede generar una complejidad de por ejemplo 100, entonces al usar el mapa con complejidad logarítmica en total estaríamos generando O(c*logk) o sea que al estar salvando la complejidad del espacio estaremos incrementando la complejidad del tiempo. Esto del tema de las constantes ya se debe al problema en sí.

**Top-Down versus Bottom-Up DP**

Ambos acercamientos usan tablas, solo que su llenado se hace de manera diferente, mientras que la versión top down usa de manera nativa la implementación de un backtracking recursiva para después detectar los sub problemas que se repiten y almacenarlos para acceder a sus soluciones de manera constante, el acercamiento de bottom up hace que los valores actuales se guarden una tabla para que el siguiente estado a calcular se use los valores anteriores para obtener el valor actual y esto se hace de manera iterativa

La manera de realizar el llenado de la tabla usando bottom up es en orden topológico del implícito DAG (grafo acíclico dirigido) en la estructura de recurrencia, para muchos problemas un orden topológico se puede simplemente con ciclos for anidados

Para muchos problemas, estos do estilos de implementar DP son igual de buenos, no obstante son en los problemas difíciles en donde escoger una u otra técnica puede llegar a ser algo muy significativo.

**TOP DOWN**

Ventajas:

Su principal ventaja es que es una transformación natural de una solución CompleteSearch recursivo, técnicamente solo tenemos que agregar la tabla en donde se originan las soluciones de los sub problemas y ya.

Se computan los problemas solo cuando son necesarios, esto fue viste anteriormente donde cuando se computan los sub problemas (estados) y cuando se vuelve a necesitar para calcular otro state entonces ya se accede a la solución del state de forma constante

Desventajas:

Puede llegar a ser lento si se revisan muchos sub problemas debido a la sobrecarga de llamadas a la función. De lo anterior se puede llegar a pensar que esto estaría sucediendo que muchos sub problemas no se estén repitiendo y es tengan que computar sub problemas diferentes de uno con el otro provocando que no se use del todo la tabla memo, pero creo

que esto más bien se refiere que para conocer un state pues debe si o si mandar a llamar otra vez la función de forma recursiva, entonces si para dar con una solución se generan demasiados sub problemas se debe de hacer llamadas recursivas para cada sub problema por ende es ahí en donde se genera una sobre carga de llamadas de forma recursiva.

Aunque en palabras del autor del libro, menciona que no suele ser penalizado en las competencias

Otro inconveniente es con respecto a la memoria, debido a que si es posible tener M estados entonces la tabla deberá ser de tamaño M para asegurarse que los sub problemas se guarden, lo que podría generar MLE (límite de memoria excedido). No obstante, esto puede impedirse si usamos el truco de usar un BST balanceado (C++ map) como tabla memo

**BOTTOM UP**

Ventajas

Sigue siendo rápido si muchos sub problemas son revisados debido a que no se tiene que mandar a llamar la función de forma recursiva, sino que se hace de forma iterativa. Entonces esto es una ventaja sobre el top down, si para nuestro problema se debe de visitar muchos sub problemas entonces usar una solución bottom up

Se puede salvar el espacio de memoria usando el truco presentado anteriormente donde solo se va guardando las soluciones anteriores en un renglón para ocupar el otro renglón restante como el renglón actual y después rotarlo. De aquí se recuerda que también existe una contraparte en top down que es usar un balanced BTS

Desventajas

No es intuitivo hacer una solución bottom up para usuarios que se inclinaron por soluciones recursivas

Si hay M estados se va a tener que visitar y llenar los valores de todos esos M estados, lo cual creo que sucede en ambas soluciones

**Conclusión**

Se escoge según sea la situación, si hay muchos sub problemas que se deben de mandar a llamar entonces escoger una solución bottom up, de lo contrario escoger una solución top down. En ambas se puede usar trucos para el ahorro de memoria.

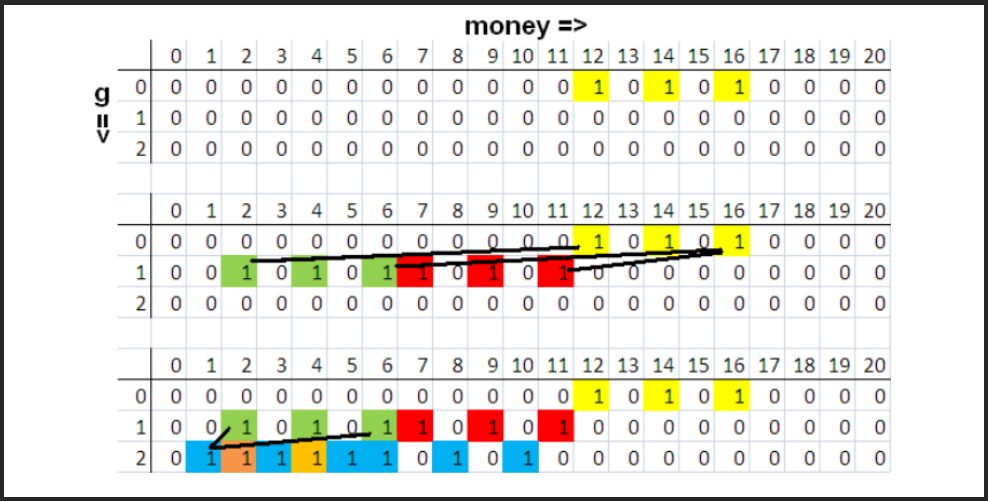
**Cómo imprimir la solución óptima en pantalla**

La necesidad de imprimir la solución óptima se refiere a imprimir los números o los valores con los cuales se logró la solución, esto es debido a que muchos de los problemas en realidad lo que piden es el valor optimo, o sea el valor de la última actualización, y no los valores con los cuales se llegaron a dicha solución.

Un ejemplo de esto sería el problema de CD en donde además de pedirnos el espacio que se estaría cubriendo (este es el valor optimo) nos piden el tamaño de los tracks usados para llenar dicho espacio, y esto último es la solución óptima.

La primera forma es principalmente aplicado en una solución Bottom up, pero también podría usarse en una solución top down

La idea principal es que una vez llegado al último state (el más optimo) de ahí se deberá de retroceder para ir guardando cada solución en una lista y después imprimirla

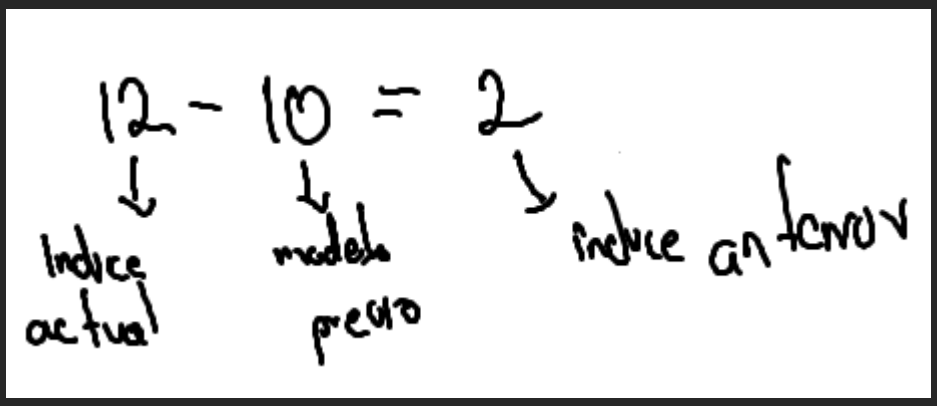



La forma de lograr esto será por medio de un backtracking recursivo donde debemos de llegar al state final, y de ahí aplicar el backtracking, por ejemplo en el ejercicio de Wedding Shopping la solución se encuentra dp[2][1] entonces al retroceder tenemos dos opciones, así que se deberá escoger solo una en caso de que se necesite imprimir una sola respuesta, si se quiere imprimir todas las posibles respuestas entonces se escoge todas.

En fin, el chiste es retroceder por medio de la posición del último state para que una vez llegando a ese state previo se vuelva aplicar el backtrack para retroceder. En cada retroceso estaremos imprimiendo el valor usado para llegar ahí, pero en caso de que sean varias soluciones se deberá de guardar los valores en una lista.

Esto puede llegar a ser un poco difícil de verlo, más que nada en cada ejercicio pues sería diferente el modo de hacer esto

Es necesario ver que en caso de tener que mostrar la solución óptima no se podrá usar el truco de salvar memoria usando solo dos renglones por obvios motivos




**Impresión de solución en top down**

La manera de relación la impresión de la solución óptima es explotando la fuerza o el concepto de backtracking y memoization.

La cuestión es que analizando la tabla memo este va a almacenar valores óptimos, los cuales serán los resultados de los states del problema, por ende se deberá de recorrer los precios de los modelos de cada prenda para ver cuál es el óptimo, la manera de saber si este es óptimo o no será introduciendo ese valor dentro de la función de tal forma que si el valor devuelto es distinto al valor guardado en esa posición en la tabla memo entonces significa que ese precio no es óptimo y no se debe de imprimir

Cuando encontremos el modelo óptimo hacemos una llamada recursiva para buscar en la siguiente prenda revisando el precio de sus modelos.

Al parecer en la tabla memo se va llenando desde el último garment, o sea el último renglón que se alcanza con la entrada, y de ahí hasta el renglón cero. Es por esto mismo que el renglón 0 contendrá un único elemento el cual será el presupuesto restante, o sea el valor optimo en este caso

Entonces, si seguimos el camino desde el último renglón, el último garment, vamos a llegar a un único elemento de la tabla puesto que será el valor óptimo.

En cada renglón se va reduciendo el número de elementos que aparecen

```
void impresion(int presupuesto, int g)
    {
        // Caso base
        if(presupuesto<0 || g==c)
            {
                return;
            }

        //Recorrer los precios de los modelos en busca del óptimo
        for(int i=1;i<=prendas[g][0];i++)
            {
                // Verificar si es óptimo
                if(backtrack(g+1,presupuesto-prendas[g][i]) == memorizacion[g][presupuesto])
                    {
                        printf("%d%c",prendas[g][i], g == c-1 ? '\n' : '-'); // Impresión
                        //Busco el sigueinte precio que sea óptimo
                        impresion(presupuesto-prendas[g][i],g+1);
                        break;// Para no buscar otros
                    }
            }
    }
```

**Impresión usando un entero como conjuntos**

Esta idea es la hablada anteriormente donde tenemos una variable de tipo entero y al momento de guardar un número tan solo hacemos un número de ‘n’ recorridos puesto que en ese punto estaría representando un número, para agregar otro simplemente sería

variable |= (i<<numero);

Entonces, esta misma idea es posible de usar para usarla para imprimir aquellos números que fueron usados en nuestra solución óptima.

La idea principal de esta técnica es que usará para índices, esto con la idea de que los valores para hacer las distintas soluciones o representar los states sean almacenados en un vector o un arreglo en sí, por ende al conocer sus índices será posible tan solo recorrer la variable para ir obteniendo qué índices corresponden a los valores usados para generar la solución óptima.

Esto fue empleado para soluciones recursivas, o sea top down, para que en cada llamada a la función se meta el índice del número actual tomado y así hasta llegar a una posible solución y escoger si es indicada o no:

sol|(1<<indice)

Se hace énfasis que no se usa |= debido que ahí se estaría asignando el valor ya a la variable y eso no lo queremos, únicamente queremos pasarle el valor para que en caso de que no se tome ese elemento entonces tan solo volvemos y ya no se deberá de quitar ni nada, o sea solo es volver y como no se asignó ese elemento.

Al final, cuando llegamos al caso base se hará una actualización del conjunto de índices

```
// Caso base
if(indice == tracks.size())
    {
        if(gasto<menor)
            {
                menor=gasto;
                b2=sol;
            }
        return gasto;
    }
```

Ambas variables al ser declaradas se inicializan en 0.

Para identificar los índices es por medio de un recorrido

```
    for(int i=0;i<n_tracks;i++)
        {
            if((b2>>i) %2)
                cout<<tracks[i]<<" ";
        }

    cout<<"sum:"<<n-minimo<<endl;
    tracks.clear();
    b2=0;    menor=1e8;
}
```

Esta técnica es muy elegante, no obstante como ya se pudo ver este método no es posible de aplicarlo para todo problema debido a que un entero mantiene 32 bits y un long long 64 bits, podría usarse un bitset pero en sí yo recomendaría su uso cuando la cantidad de elementos es máximo 64, no hay muchos problemas así pero sí existen como el problema UVA 624 - CD

**EJEMPLOS CLÁSICOS**

El ejercicio visto en todo lo anterior llamado wedding shopping no es un problema clásico, sino que es non-clasical problem.

Los ejemplos clásicos son aquellos cuyos states DP y sus transiciones ya se tienen bien conocidos, es por esto que los problemas clásicos y sus soluciones deben estar dominados

1. **Max 1D Range Sum**

UVA Jill Rides Again:

El problema de trata de una persona quien anda por bicicleta, el punto es que también está involucrado en su jornada andar en un autobús, de tal modo que hay paradas y debe de andar en bicicleta, ella con su tiempo de experiencia ha asignado varios números (positivos y negativos) para ilustrar que partes de la ruta lo considera buenos haciendo que los números positivos indique esa zona de la ruta es agradable mientras que los números negativos indican que dicha zona no lo es, por ende se debe de encontrar aquel intervalo dentro de la ruta que haga que se pase por más partes buenas, o sea que se debe de encontrar en qué rango de la ruta resulta ser el mejor y esto se logra conociendo

la suma de los valores de dicho rango, se debe de maximizar esa suma, entonces esto se resume en encontrar el rango de la ruta que produzca la suma más grande.

Es necesario ver que el primer elemento dado de la ruta es un número que índica el número de paradas (stops) de tal forma que en cada número de la ruta habrá tantos atrás como entre paradas y por medio de esos stops se va a definir el rango que produzca mayor suma

3

-1

6

O sea, esto en realidad sería

3

Parada 1

-1

Parada 2

6

Parada 3

De tal forma que el rango que hace esto es en el rango de la parada 2 y el de la parada 3

Podría resolverse por medio de un segment tree o Fenwick Tree, pero problemas relacionados con el rango de la entrada estática son posibles de resolver usando bottom up DP

Una solución tener en consideración primero que con complete search sería generar los subconjuntos pero teniendo en cuenta que hay 2n subconjuntos pues no sería nada optimo hacer esto por este medio.

Otra solución sería observar lo siguiente y es que es posible realizar una pre computación, o sea pre computar la suma del array de tal modo que en cada posición vamos a saber cuál es la cantidad de sumada en la posición ‘i’ así que puedo ya computar cualquier suma de 0 a ‘i’, el problema es que necesito saber la suma de un rango entonces ese 0 a ‘i’ se debe de incorporar otro índice para cortar, en otras palabras necesito computar la suma en una posición ‘j’ para entonces realizar la resta y conocer la suma del rango en ‘i’ y ‘j’. Esto de manera gráfica se podría ver como:

| | |
|---|---|
| -1 | 6 |

Entonces, sumando:

| | |
|---|---|
| -1 | 5 |

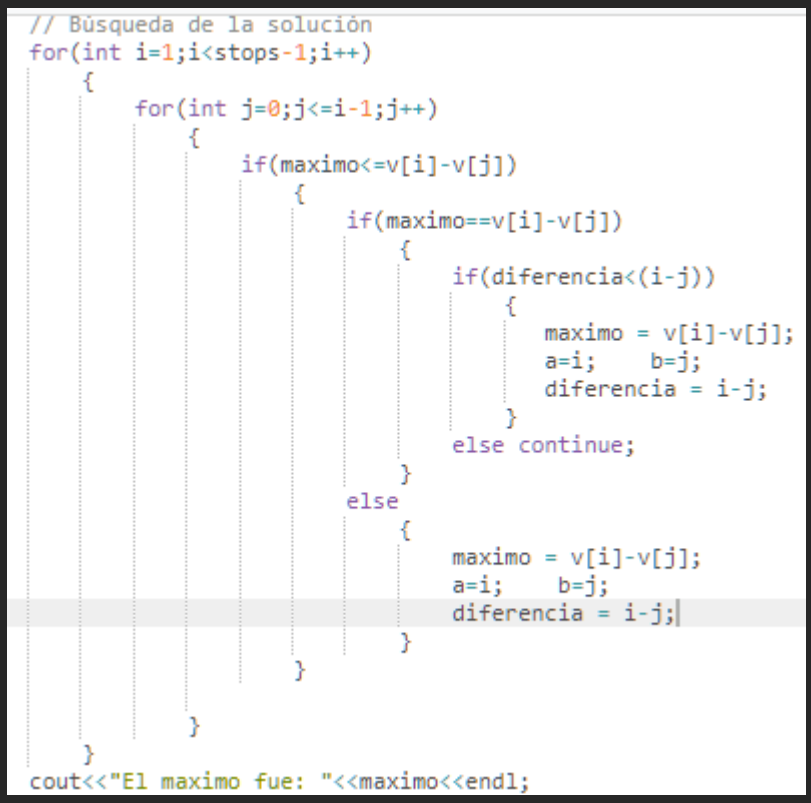
Por otro lado, con el índice ‘j’

| | |
|---|---|
| -1 | 5 |

Por ende, ya se tiene dos índices, j=0 e ‘i=1, entonces de aquí al querer calcular la suma en ese rango de j-i tan solo se debe hacer la resta de los valores en esos índices

B[i] – B[j] = B[1] – B[0] = 5 – (-1) = 5+1 = 6

Entonces la suma en ese rango es 6 y aquí fácilmente se encuentra la solución más óptima, el problema es que obviamente las longitudes de arreglos dados no serán tan pequeños y las condiciones del problema nos dice que será hasta 20k elementos que podremos tener, entonces para encontrar la mayor suma dentro de un rango ‘i’ y ‘j’ será por medio de un ciclo anidado



```
// Búsqueda de la solución
for(int i=1;i<stops-1;i++)
    {
        for(int j=0;j<=i-1;j++)
            {
                if(maximo<=v[i]-v[j])
                    {
                        if(maximo==v[i]-v[j])
                            {
                                if(diferencia<(i-j))
                                    {
                                        maximo = v[i]-v[j];
                                        a=i;    b=j;
                                        diferencia = i-j;
                                    }
                                else continue;
                            }
                        else
                            {
                                maximo = v[i]-v[j];
                                a=i;    b=j;
                                diferencia = i-j;
                            }
                    }

            }
    }
cout<<"El maximo fue: "<<maximo<<endl;
```

En sí esta sería la solución por el lado de búsqueda completa, donde adentro tiene unas ciertas condiciones debido al formato de salida del ejercicio, no obstante la esencia son los for anidados, de tal modo que vamos comparando rango por rango y esto genera una complejidad cuadrática lo cual significa que para una entrada de hasta 20k simplemente esto sea algo ineficiente debido a que solo se computa 1e8 operaciones por segundo.

Al parecer tendríamos que necesitar aproximadamente 4 segundos para que esto pudiera correr sin generar un TLE, según mis cálculos debido a que se tiene 3 segundos en el problema.

No obstante, esta solución puede funcionar para otro caso pero por mientras se deja así

Entonces, la solución por medio de complete search no es óptimo así que se debe de buscar otro algoritmo capaz de cumplir con la respuesta en un buen tiempo, y no hay necesidad de pensar en uno debido a que ya existe el algoritmo de Jay Kadane's el cual tiene de complejidad de tiempo lineal.

La idea del algoritmo se puede ver tanto como una solución greedy o DP, pero la idea es la siguiente, se debe de ver la existencia de una tabla DP (esto en caso de que se quiera atacar el problema desde el punto de vista DP) la cual por la cantidad de parámetros que se debe de tener para formular un state sería de dimensión 1.

Entonces, a medida que se vayan introduciendo los datos se deberá realizando la suma como si fuera la matriz de suma acumulativa anteriormente, no obstante se debe de observar lo siguiente y es que al momento de sumar y toparnos con 0 la suma se va a reiniciar ahí, en caso de que sea positivo se continuará suma y la idea de DP está presente ya que al ser una suma acumulativa se debe de tener presente el anterior para poder sumarlo en la posición actual junto con el valor actual positivo.

Pero lo interesante y la clave del algoritmo tienen que ver con los números negativos y es que al momento de hacer la suma se debe de ver que siempre es mejor reiniciar en cero a sumar a un número negativo. En otras palabras, si la suma que llevamos actualmente ya no son números negativos se debe de reiniciar ya que llevar la suma ahí no será óptimo.

De este modo se ve la solución DP, el caso base será el número 0 ya que por sí solo ya es 0 al menos que se cambie, pero si sigue siendo cero entonces no habrá solución para este problema.

La solución se resume como: dp[i] += (array[i] + dp[i-1])

Pero en caso de que dp[i]<0, o sea que la suma acumulada ya de 0 se reinicia en 0 dp[i]=0

Se observar que esto funciona y sería una solución bottom up, no obstante esto puede optimizarse donde en lugar de usar una tabla dp simplemente se use una variable para ir guardando la suma, lo cual al hacer esto en sí se estaría haciendo el empleo del truco de salvar espacio al usar solo dos renglones, en este caso al ser de 1 dimensión pues solo se estará usando el mismo renglón así que no tendría caso realmente usar una tabla.

Hay unas cuentas cosas que se deben de considerar con este ejercicio y es que hay ocasiones en donde distintos rangos generan la misma suma, la cual es mayor, en ese caso el criterio que se toma en cuenta para ver cuál escoger es viendo la longitud del rango, por ende se debe de escoger aquellos rangos que abarquen más y en caso de que sean iguales lo rangos también no se debe de hacer nada.

```
for(int i=0;i<stops-1;i++)
    {

        int aux; cin>>aux;
        dp+= aux;

        if(maximo<=dp)
            {
                if(maximo==dp)
                    {
                        if(diferencia<(i-indice2))
                            {
                                indice=i;
                                inicio = indice2;
                                diferencia=(i-indice2);
                            }
                    }
                else
                    {
                        maximo=dp;
                        indice=i;
                        inicio = indice2;
                        diferencia=(i-indice2);
                    }

            }

        if(dp<0)
            {
                dp=0;
                if(i+1<stops-1)
                    {
                        indice2=i+1;
                    }
            }
```

Este es el código, se observa que en esencia es solo imprimir el valor máximo, pero como en este caso pide los stops que serían los rangos, la solución es esta

Entonces, este problema se reduce a encontrar los rangos cuya suma sea la mayor y ese rango sea el más grande al mismo tiempo.

**Max 2D Range Sum**

Ahora tenemos el problema extendido a dos dimensiones, entonces podríamos ver similitudes y es que sabemos que es posible pre computar un arreglo de suma acumulativa, o sea que en ese arreglo en la posición ‘i’ podría conocer la suma de los elementos hasta esa posición ‘i’, por ende para conocer la suma entre un cierto rango es simplemente restar con el valor de la posición del índice ‘j’

Entonces, si podemos computar un arreglo de 1D lo mismo podríamos hacer para una matriz de 2D. Para lograr se debe de observar que aquí ya se tendría que tener cuidado con valores debido a que hay sumas que se repite y esto se observa en lo siguiente:

| | | | |
|---|---|---|---|
| 0 | -2 | -7 | 0 |
| 9 | 2 | -6 | 2 |
| -4 | 1 | -4 | 1 |
| -1 | 8 | 0 | -2 |

Por ejemplo esto sería suma y lo primero a sumar será el primer renglón y columna

| | | | |
|---|---|---|---|
| 0 | -2 | -9 | -9 |
| 9 | 2 | -6 | 2 |
| 5 | 1 | -4 | 1 |
| 4 | 8 | 0 | -2 |

Por ende ya puedo computar por ejemplo el cuadro tabla[1][1] cuyo valor actual es 0, y la forma de hacer esto es:

| | | | |
|---|---|---|---|
| 0 | -2 | -9 | -9 |
| 9 | 2 | -6 | 2 |
| 5 | 1 | -4 | 1 |
| 4 | 8 | 0 | -2 |

Entonces, se puede ver que hay un valor que se repite y es el 0, en este caso no hay problema pero si fuera otro número se tendría problemas, pero en este caso en ambos renglones se considera ese número debido a que se topan como cruz por lo cual al ser esto se debe de pues quitar esa repetición y para esto e tendría que restar ese cero. Esto mismo se repetirá para los demás casos

$$2 + 9 + (-2) – 0 = 9$$

Esto es igual

Tabla[1][1] += (tabla[1][0] + tabla[0][1]) – tabla[0][0]

Esto será la formula con la cual se realizará la suma y obtener la matriz de la suma acumulativa

Una vez obtenida la matriz suma ahora falta calcular un índice de tal forma que ayude a limitar para encontrar una sub matriz cuya suma sea la mayor.

$$\begin{matrix} 0 & -2 & -7 & 0 \\ 9 & 2 & -6 & 2 \\ -4 & 1 & -4 & 1 \\ -1 & 8 & 0 & -2 \end{matrix}$$

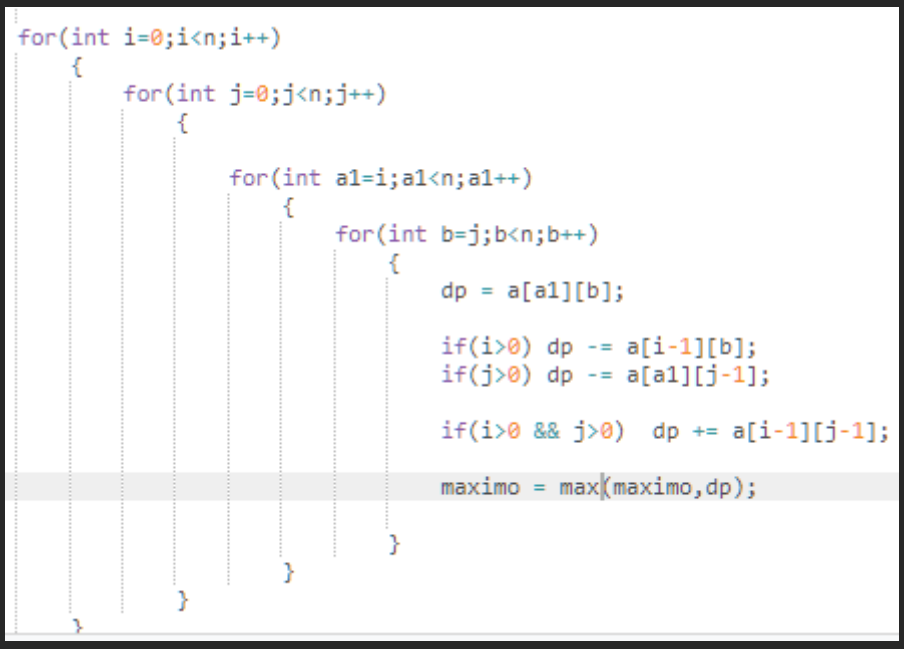
Entonces, la suma sería tendría que efectuarse como fue descrito arriba

| | | | |
|---|---|---|---|
| 0 | -2 | -9 | -9 |
| 9 | 9 | -4 | 2 |
| 5 | 6 | -11 | -8 |
| 4 | 13 | -4 | -3 |

Ahora, lo otro ya es encontrar un índice el cual me esté limitando pues el sub matrix de tal forma que sería necesario

```
for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
            {
                cin>>a[i][j];

                if(i>0) a[i][j] += a[i-1][j];
                if(j>0) a[i][j] += a[i][j-1];

                if(i>0 && j>0)  a[i][j] -= a[i-1][j-1];
            }
    }
```

Suma



```
for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
            {

                for(int a1=i;a1<n;a1++)
                    {
                        for(int b=j;b<n;b++)
                            {
                                dp = a[a1][b];

                                if(i>0) dp -= a[i-1][b];
                                if(j>0) dp -= a[a1][j-1];

                                if(i>0 && j>0)  dp += a[i-1][j-1];

                                maximo = max(maximo,dp);

                            }
                    }
            }
    }
```

Encontrando las sub matrix con suma máxima

Para el problema funciona, pero al ser una complejidad de n a la cuarta para entradas más grandes nos dará TLE.

**Longest Increasing Subsequence (LIS)**

Este tipo de problema se trata de encontrar una sub secuencia, pesto que se encuentra contenida en un arreglo mayor, que sea la más larga posible.

Se debe de notar que la sub secuencia dentro del arreglo no debe de estar forzosamente de forma contigua o sea que se debe de encontrar una sub secuencia -1 63 2 34643 6

Por ejemplo sería -1 2 6 y no necesariamente se encontraban seguidos

La solución por búsqueda completa se resume en hallar todos los posibles sub conjuntos y de ahí ver cuál conjunto sería el mayor y al mismo tiempo sea posible definirlo como una secuencia de menor a mayor. Esto genera que el algoritmo sea muy ineficiente puesto que existen $2^n$ sub conjunto de un conjunto original entonces generar todos es simplemente TLE.

Hay tres formas de resolverlo:

1. Usando un lower bound de tal forma que debemos aprovechar su funcionamiento en un arreglo no ordenado
2. Usando un segment tree
3. Viendo el problema como un DAG y de ahí buscar el camino más largo

**Usando búsqueda binaria**

Para resolverlo por medio de este camino se usa la variante de búsqueda binaria la cual es lower bound y se encuentra en la librería STL y esta función, recordando, me va a encontrar un número que sea igual o mayor al número buscado.

8

7 3 5 3 6 2 9 8

Se puede ver que es un arreglo de longitud 8

Entonces, teniendo la idea del uso de lower bound se podría comenzar colocando el primer número dentro de un arreglo nuevo

7

Entonces, podríamos buscar con el lower bound el siguiente valor que sería 3, de tal forma que lo que hace lower bound es regresarnos un iterador apuntando a un valor dentro del arreglo que sea mayor o igual al buscado entonces en este caso me estaría regresando el valor 7 puesto que es mayor que 3

Dentro de c+ referencias se puede ver que

```
std::cout << "lower_bound at position " << (low- v.begin())
```

Esta resta nos dice que en valor está en esa posición o debería de estar en esa posición, ocupando esto deberíamos de ver que 3 no está en esa posición puesto que ahí está 7, por ende se podría primero preguntar si:

3==7 en este caso no lo, por ende nos preguntamos

7>3 lo cual es cierto así que se intercambia el valor de la posición actual.

Array actual

3

Seguimos y ahora buscamos en nuestro arreglo el siguiente valor el cual es 5 y nos hacemos las preguntas

3==5 no

3>5 no, entonces lo agregamos como otro elemento del arreglo

Array actual

3 5

El siguiente valor es 3, así que todo se conserva igual

El siguiente valor es 6 por ende si es mayor a 5 así que se agrega

Array actual: 3 5 6

El siguiente valor es 2 y aquí la función obtiene un uso más importante debido a que nos va a regresar que dentro de nuestro arreglo ese 2 está o debería de estar en la posición 0 pero al hacer la comparación

2==3 no son iguales

Entonces

3>2 sí, por ende como pasó anteriormente con el 7 se debe de sustituir ahora el 3 por el 2

Array actual: 2 5 6

El siguiente valor es 9 y se ve que sí es más grande que 6 por ende se agrega

Array actual: 2 5 6 9

El siguiente valor es 8 por ende debería estar en la última posición pero se observa que ahí está el 9 el cual es mayor a 8 así que se reemplaza

Array actual: 2 5 6 8

Este es el sub arreglo ascendente más grande que se puede encontrar

Se observa que al colocar ese

```
std::cout << "lower_bound at position " << (low- v.begin())
```

Se obtiene que la función lower_bound está funcionando como una búsqueda binaria realmente, lo más parecido que se puede encontrar debido a que al hacer la resta nos devuelve la posición en donde presuntamente está nuestro número o debería de estar nuestro número. Al parecer no hay como tal una búsqueda binaria ya implementada sino que existen estas dos versiones: lower y upper bound por ende se hace uso de lower para imitar el comportamiento de la búsqueda binaria

Nota importante, la resta del iterador de vuelto con v.begin() se puede almacenar en un entero haciendo que ya se manipule que nos devolvió un índice, una posición, un entero.

Para el problema como ejemplo vamos a suponer que nos dan una longitud del arreglo y sus enteros.

Es importante que la técnica DP se tendrá que ocupar para ver que para representar un state es usando un solo parámetro, de tal modo que dp[i] nos dice el último número agregado, entonces en sí dp nos contendrá el arreglo creciente más largo encontrado hasta el momento

Para computar todos los estados será pues recorriendo cada número del arreglo lo cual ya nos genera una complejidad lineal, pero al emplear el lower bound se debe de considerar una complejidad logarítmica, por ende la complejidad final es O(n*logk)

Se señala que ‘k’ será la longitud de dp, puesto que es en ese array en donde se va a emplear la búsqueda. Dp.size = k

**Nota sobre los iteradores**

```
posicion = lower_bound(dp.begin(),dp.end(),aux);
```

Si lo colocamos solo así nos arroja un error y es que nos dice que no se puede convertir un iterator int a un int

Pero al hacer la resta sí es posible

```
posicion = lower_bound(dp.begin(),dp.end(),aux) - dp.begin();
```

Finalmente el código se resume en:

```
int aux; cin>>aux;
dp.push_back(aux);

for(int i=1;i<n;i++)
    {
        cin>>aux;
        posicion = lower_bound(dp.begin(),dp.end(),aux) - dp.begin();

        if(dp[posicion] == aux)
            {
                continue;
            }
        else    // Si no son iguales hacemos las comparaciones para decidir si añadir o sustituir.
            {
                if(dp[posicion] > aux)  // Si resulta mayor se sustituye
                    {
                        dp[posicion] = aux;
                    }
                else    // Si resulta menor entonces se agrega
                    {
                        dp.push_back(aux);
                    }
            }

    }

cout<<"Tamaño del sub arreglo más largo de forma creciente: "<<dp.size()<<endl;
```

Ya después podemos imprimir el tamaño, los números que conforman la solución o lo que sea

**Input**

```
8
7 3 5 3 6 2 9 8
```

**Output**

```
Tamaño del sub arreglo más largo de forma creciente: 4
El arreglo es:
2 5 6 8
```

Lo mismo se llegó de forma escrita arriba,

Se nota que también se podría considerar el uso de una técnica greedy debido a que en el arreglo dp siempre queremos considerar los números más pequeños conforme se avanza en el arreglo original, por ejemplo lo que ocurrió con el 3 en donde fue intercambiado por el 2, siempre nos aseguramos a colocar los números más pequeños

Solución con Segment Tree

**0-1 Knapsack (Subset Sum)**

Este problema deriva del problema de la mochila, en este caso vamos a tener una lista de los objetos con su respectivo peso pero se agrega a que cada objeto tiene un cierto valor (beneficio). El problema podría pensarse como un como una mochila con una capacidad máxima de 'n' y tenemos 'a' objetos con sus pesos y además cada objeto tiene un valor o beneficio, entonces de aquí se resume que se debe de encontrar cuál es la cantidad máxima o beneficio máximo que podemos obtener al agregar algunos objetos y de esto mismo se desprende el nombre 0-1 debido a que es posible agregar unos objetos e ignorar otros objetos con el fin de que los objetos agregados me den el máximo valor que se puede conseguir con esa lista de objetos y sin pasarse del peso que puede soportar la mochila

Capacidad = 6

Objetos:

| | | | | |
|---|---|---|---|---|
| W | 4 | 3 | 2 | 1 |
| V | 5 | 4 | 3 | 2 |

Vemos que aquí se puede escoger como mínimo respuestas:

Los índices: 0,2 lo que me da un beneficio de 8

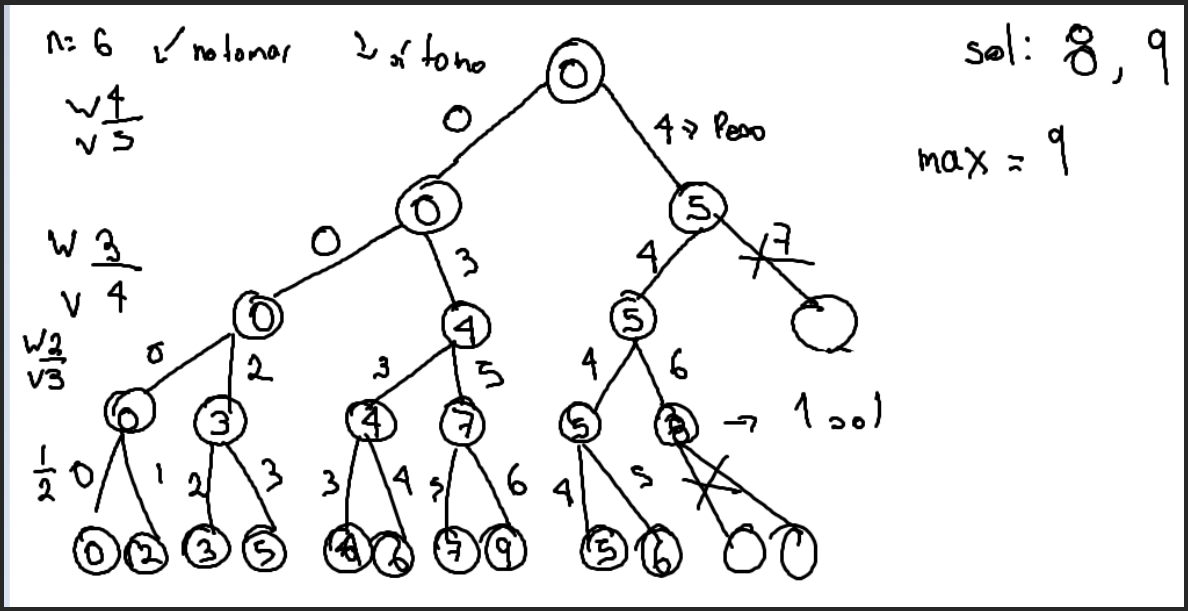
Otra solución sería

Los índices: 1,2,3 lo que me da un beneficio de 9

Entonces la solución más óptima fue la segunda, de esta forma queda explicito que una solución greedy no sería buena idea puesto que ese algoritmo tendría que escoger el primero el 4 y después escoge el 2, lo mismo que la primera solución debido a que toma el valor más grande y esto no da el máximo beneficio que podemos obtener.

Entonces, para esto se podría ir primero por el lado de buscar la relación de recurrencia de complete search, lo que es lo mismo que buscar los states de DP o simplemente plantear una solución Backtracking search

Lo primero sería considerar siempre ilustrar el problema con un árbol de decisiones y esto pone en evidencia que existe 2 caminos en cada elección lo que sería tomar el elemento o no tomar dicho objeto




Se observa estas dos decisiones y en efecto me llevan a la solución, se debe de observar algunas cosas:

1. Los parámetros para representar un estado será que necesitamos el peso actual y el beneficio que llevamos actualmente
   Backtrack(peso, indice)

2. Se debe escoger entre el máximo de escoger o añadir dicho objeto
   Max(Backtrack(peso+peso_Objeto, índice+1), Backtrack(peso, índice+1) )

3. Se debe de contemplar siguientes condiciones
   a. Peso>capacidad_mochila: si esto ocurre debemos de podar toda esa rama puesto que no es posible así que esas soluciones ya no me interesan
   b. peso+peso_Objeto>capacidad_mochila: si esto ocurre entonces no puedo meter ese objeto y forzosamente tendré ignorarlo y probar otro

Esta solución es empleando desde un punto de vista de backtracking recursivo ya se puede los sub problemas que se repiten y son cuando no se toma dicho elemento entonces al tomarlo se debe de computar de nuevo ese para avanzar en los índices y tomar otra elección, también podría ver el caso en donde los valores se repiten por ende se tendría que computar todo eso.

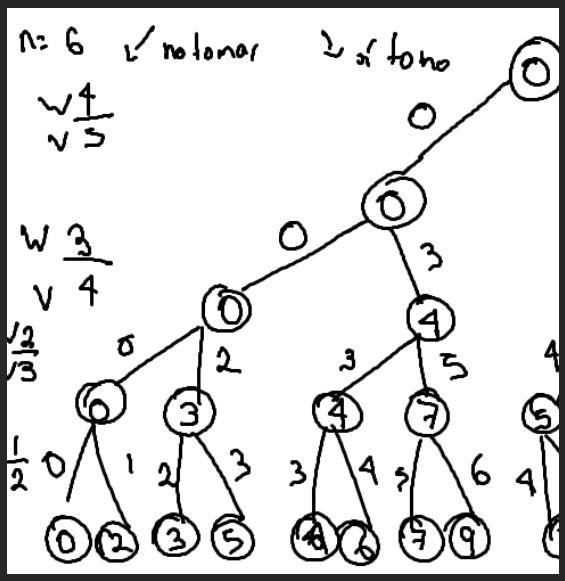
Entonces, sería posible implementar simplemente una tabla de memo y ya, usando el truco de balanced BST aunque al poder computar los estados en tiempo constate ya con el BST se tendría que computar en tiempo logarítmico

De igual forma se podría solucionar usando el acercamiento de bottom up ya que como se comentó anteriormente los estados del backtracking pueden usarse para los state de un bottom up entonces la solución sería tener en cuenta esto

dp[peso][índice] = beneficio

dp[peso][índice] = max( dp[peso][índice]+valor_objeto, dp[peso][índice] )

Aquí además se debe de pensar sobre los casos base los cuales debe ser puro ceros, o sea al momento de que hacemos el árbol se comienza en cero



Se puede ver que toda una rama lo ilustra cuando no tomamos ninguno entonces para poder cumplir con esto se debería de inicializar el reglón y la columna 0 con (irónicamente) con ceros, 0.

RECORDAR QUE UNA SOLUCIÓN BOTTOM UP PASA POR TODAS LOS DISTINTOS ESTADOS MIENTRAS QUE UNA SOLUCIÓN TOP DOWN SOLO PASA POR LOS QUE COMPUTA EN ESE MOMENTO

Debido a lo anterior se puede entender el por qué es necesario inicializar toda la columna y renglón, ya que al computar todos los estados también tendremos que computar aquel estado en donde se escoge no tomar ninguno

Entonces, ya se puede programar ambas soluciones

```
bool se_Puede_Colocar(int peso, int siguiente)
    {
        if(peso-siguiente >=0)
            return true;
        return false;
    }

int backtrack(int peso, int indice)
    {
        // Caso base
        if(indice==n || peso==0)
            {
                return 0;   //Se retorna cero para no figurar en la suma
            }
        if(memo[peso][indice]!=-1)  return memo[peso][indice];
        else
            {
                if(se_Puede_Colocar(peso,mochila[0][indice]) )
                    return memo[peso][indice] = max( backtrack(peso-mochila[0][indice],indice+1)+mochila[1][indice], backtrack(peso,indice+1));
                else
                    return memo[peso][indice] =  backtrack(peso,indice+1);
            }
    }
```

Se debe de recordar el uso del índice y es que aquí vamos avanzando y tomando uno por uno, o sea cada valor está asociado a otro número por ende esto se puede tratar como un

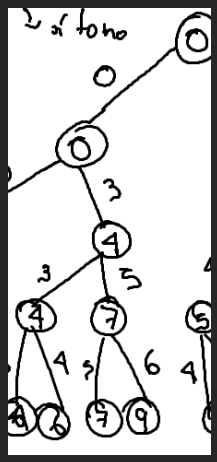
arreglo de 1 dimensión realmente, o sea no hay necesidad de recorrer con un for puesto que el índice cumple la función en posicionarnos en el siguiente valor

**Ahora, lo más importante aquí es ese return 0 y sumarle a la función backtrack el peso del objeto que se está añadiendo**

Lo que se está haciendo realmente es que si estamos profundizando el árbol y al momento de añadir un objeto se debe de sumar y como es recursivo la suma se estará acumulando hasta llegar al final para ver qué ruta obtuvo un mayor beneficio, en sí esto se pudo resolver usando un tercer parámetro para al final devolver el beneficio recaudado no obstante esto queda más elegante y óptimo de ver

Y como en cada llamada recursiva se va acumulando los beneficios se debe de regresar cero cuando ya se haya recorrido todo el arreglo o se haya ocupado todo el espacio de la mochila, la razón de por qué se retorna cero es debido a que cero es un número neutro en la suma entonces cada vez que se vaya llamando se va sumando los beneficios y al final al regresar el cero en señal de que ya se acabó esa rama del árbol se regresa cero para que no figure en la suma y únicamente el return 0 tenga la finalidad de indicar que la rama o ruta se acabó.

Con rama del árbol me refiero a:



En este caso hay 4 rutas aquí en la imagen

Su contra parte de bottom up es emplear la tabla, tabulación y lo descrito anteriormente así que se debe de inicializar y ocupar ciclos

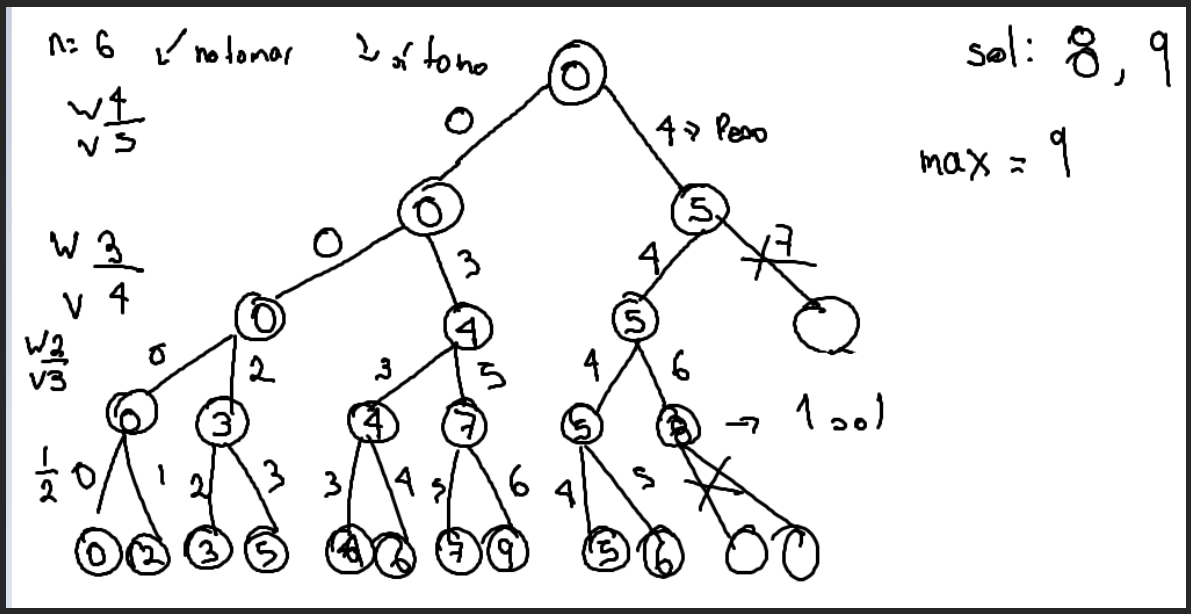
La idea es como el uso de una matriz de booleanos, no obstante aquí no sería tan recomendable debido a que necesito saber el precio reunido en cada state, por ende lo mejor es guardar los beneficios en las posiciones de las posiciones y técnicamente solo por ese motivo lo demás se tendría que ver y resolver teniendo la misma idea que una matriz de booleanos, los índices de las columnas van a funcionar realmente como el peso y en cada posición el vez de colocar un true vamos a guardar los beneficios, el valor de cada objeto

Entonces se debe de checar la posiciones anteriores

En cada iteración se va consultando la capacidad restante de tal modo que en consulta se hará la resta y se buscará en renglón anterior debido a que así se podrá saber si ya fue colocado o no un objeto anteriormente o el beneficio que quedó

```
6 4
4 3 2 1
5 4 3 2
```

```
0 0 0 5 5 5
0 0 4 5 5 5
0 3 4 5 7 8
2 3 5 6 7 9
9
```




```
// Solución
for(int i=1;i<=n;i++)    // Indices de cada objeto
    {
        for(int espacio_restante=1;espacio_restante<=capacidad;espacio_restante++)
            {
                if(espacio_restante-elementos[0][i]>=0)
                    {
                        // Se ve si es mejor añadirlo o no
                        dp[i][espacio_restante] = max(dp[i-1][espacio_restante-elementos[0][i]]+elementos[1][i]
                        , dp[i-1][espacio_restante]);
                    }
                else
                    {
                        // No añado el elemento
                        dp[i][espacio_restante]=dp[i-1][espacio_restante];
                    }
            }
    }
cout<<dp[n][capacidad];
```

Este sería una forma más fácil de ver

**Coin Change (CC) - The General Version**

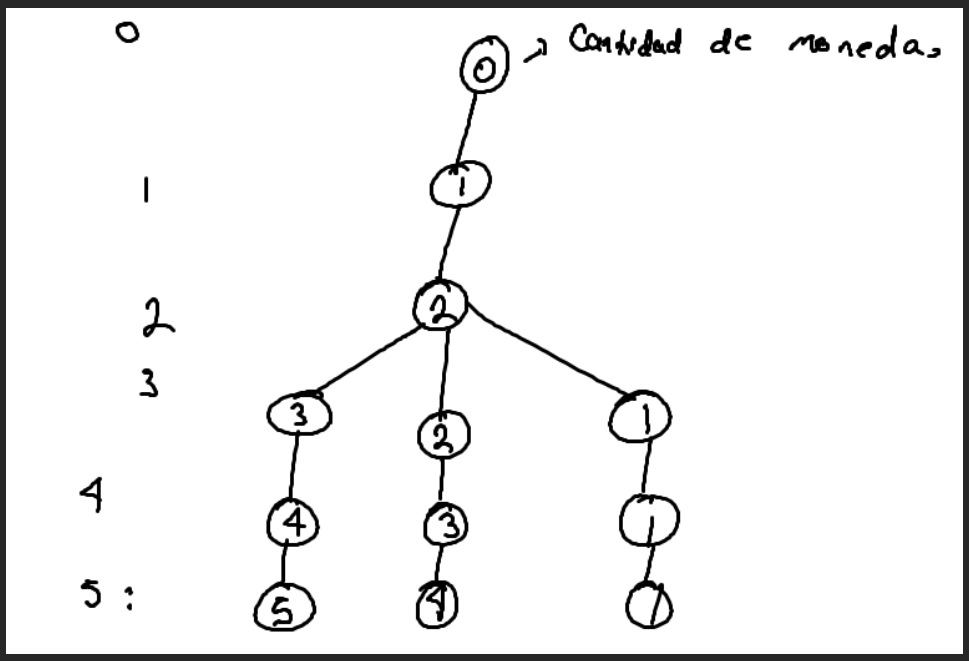
El famoso problema de las monedas, fue el discutido en temas anteriores y este problema ahora se aborda de manera general, se generaliza el problema para cualquier conjunto de monedas y cualquier monto a reunir.

El problema es que se tiene un conjunto de monedas de diferentes denominaciones y se debe de suponer que la disponibilidad que se tiene de cada moneda es simplemente ilimitada, entonces dado un valor V ¿cuántas son la mínima cantidad de monedas necesarias para representar el valor V?

Se observa que el problema se trata de optimización ya que se necesita la mínima cantidad de monedas.

Entonces, suponemos los siguientes datos para dar una solución

$$V = 7, n = 4, \texttt{coinValue} = \{1, 3, 4, 5\}$$




Sería algo así el árbol, ya tan solo lo extendemos para saber cuántas monedas son necesarias para representar el 6 y 7

Entonces, con esta observación es posible pensar en una función:

Función(moneda_representar) = monedas_necesarias_para_representarla

Se observa que el uso de una solución con backtracking no sería tan conveniente y resalta un uno uso más bottom, parece que podría resultar más fácil entender por este medio la solución del problema

Se puede decir que los states o los sub problemas es saber cuántas monedas son necesarias para representar la moneda ‘i’

Entonces, tan solo sería necesario tener un parámetros para los state y se ve que el caso base a inicializar la tabla sería que sabemos que para todo caso es posible reunir 0 monedas y pues para esto se necesita de cero monedas

Dp[0] = 0;

Entonces, lo otro es considerar los inconvenientes y uno de ellos sería cuando no se puede representar una cantidad de monedas y esto nace debido a que la formula que podríamos ocupar, y para esto se debe de deducir por medio de la construcción de la tabla

$$V = 7, n = 4, \texttt{coinValue} = \{1, 3, 4, 5\}$$

Dp[0] =0

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | | | | | | |

Probando el primer valor 1

Dp[1] solo se necesita de una moneda para representarlo

Visitando los demás tipos de monedas se observa que ninguna del conjunto puede satisfacerlo sin pasarse

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | | | | | | |

Dp[2] se comprueba con todas las posibles que tenemos

2= 1+1

Solo se puede ese, debido a que el 3 ya resulta mayor, el 4 y 5 igual entonces no entran

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | | | | | |

Dp[3]

Revisando todas las formas

3=1+1+1

3=3

El mínimo sería el 3

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 1 | | | | |

Aquí se observa que lo anterior se pudo ver como

Dp[3] = min( 1+dp[3-valor[0]], 1+dp[3-valor[1]], 1+dp[3-valor[2]], 1+dp[3-valor[3]] )

Lo que es igual a

Dp[3] = min(1+dp[3-1], 1+dp[3-3], 1+dp[3-4], 1+dp[3-5] )

Dp[3] = min(1+dp[2], 1+dp[0], 1+dp[-1], 1+dp[-2] )

Aquí vemos otra cosa que se debe de tener en cuenta y es cuando la resta del número que se quiere representar y el valor que se está usando para tratar de satisfacerlo resulta en un número negativo, ahí debemos ya sea regresar un número muy grande o simplemente no computar esto

Dp[3] = min( 1+2,1+0, no se puede, no se puede )

Dp[3] = 1

Entonces, se ve la importancia que tiene el árbol para representar estas soluciones y es que todo esto nace de las recurrencias que se encuentran por medio de un enfoque de búsqueda completa

Otra cosa que se debe de tener en cuenta es que obviamente no se puede tener realizar la comparación del valor a representar con todas las monedas que se necesitan al mismo tiempo, a lo máximo se podría solo realizar la comparación con dos elementos entonces lo mejor sería realizar la comparación de cada valor de las monedas con un valor mínimo por defecto que se vaya actualizando en cada comparación, por ende se podría tener

Int minimo = 1e8

Dp[i] = min( minimo, dp[i-valores[j])

Minimo = dp[i]

Así hasta pasar por todas las monedas, esta sería realmente la forma de solucionar el problema teniendo en cuenta que la cantidad de monedas es ilimitada

```
int valor; cin>>valor;
int n; cin>>n;
vector<int> monedas;
vector<int> dp(valor+1,0);

for(int i=0;i<n;i++)
    {
        int aux; cin>>aux;
        monedas.push_back(aux);
    }

dp.push_back(0);    // Caso base.

for(int i=1;i<=valor;i++)
    {
        int minimo=1e9;
        for(int j=0;j<n;j++)
            {
                if(i-monedas[j]>=0)
                    {
                        dp[i] = min(minimo,1+dp[i-monedas[j]]);
                        minimo = dp[i];
                    }
            }
    }

cout<<dp[valor];
```

Se observa que la cantidad de for anidados no radica en cuenta a la dimensión que debe de tener la tabla dp.

Otra cosa aclarar es que en este caso usé un vector debido a que la cantidad de datos de entrada pues no lo conozco así que tendría que usar esto

Las soluciones DP en sí podrían ser usadas en cualquier problema (en donde pueda aplicarse) no obstante hay un factor que limita su uso y es el costo de la memoria, esto queda al descubierto si por ejemplo hay nV=1M la solución DP ya no es viable. Esto es debido a que se observa que los costos o peso, el valor de las parámetros son usualmente usados en los índices de la tabla dp entonces usar tantos datos llega un momento en donde se puede desbordar la tabla según entiendo.

Es importante ver que las soluciones que involucra DP usan como índices los parámetros, o un juego con ellos y dentro de los posiciones usualmente se coloca información en este caso se usó las posiciones para guardar la cantidad de monedas necesarias para representar el índice 'i', en el problema de la mochila 01 se usó para poder guardar el beneficio que se lleva hasta el momento, no obstante hay otros problemas donde no resulta necesarias guardar esto y ahí es donde se puede usar una matrix de booleanos como el caso de Wedding shopping

**Coin Change (CC) – Formas posibles de conseguir el monto V**

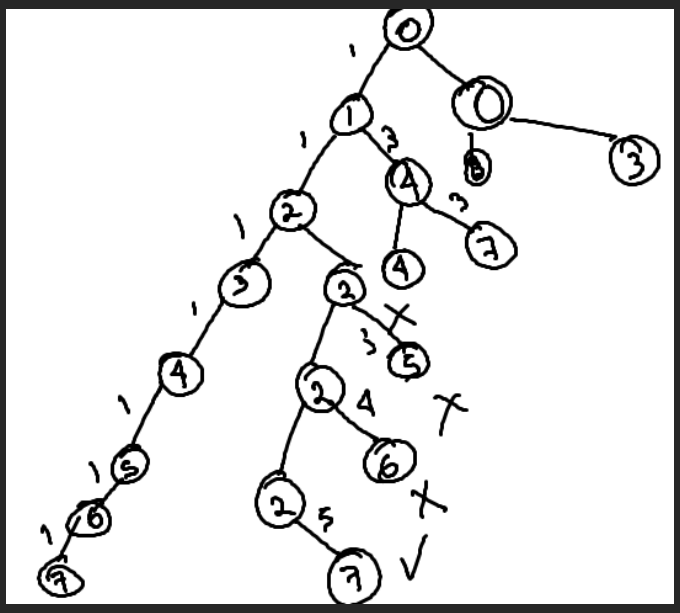
Este es una variante del problema original en donde se solicita contar todas las formas posibles de conseguir el número V, el monto que se desea reunir. En este caso ya la pregunta del problema es diferente, anteriormente se solicitaba ¿cuál es la mínima cantidad de monedas necesarias para representar el número V? ahora se tiene, ¿De cuántas formas es posible formar al número V?

Por ende, como ya cambió la pregunta es necesario considerar que ya lo que interesa es formar el número y no me interesa con cuántas monedas puedo lograr hacer esto, entonces un ejemplo sería el siguiente:

$$V = 7, n = 4, \texttt{coinValue} = \{1, 3, 4, 5\}$$

1. 1+1+1+1+1+1+1
2. 1+1+1+1+3
3. 1+3+3
4. 3+4
5. 1+1+1+4
6. 1+1+5

Con esto la relación de recurrencia cambia y se llega al ya bien conocido tomar o no el número actual y esto sí podría fácilmente resolverse con backtracking



La cantidad de opciones que se tiene para escoger son muchas y se observa que hay sub árboles o sub problemas en donde se debe de computar de nuevo para probar con otro valor, un ejemplo de ello es al estar probando con el 2 en donde se debe de consultar el dos para poder probar con las denominaciones siguientes.

Entonces, se puede ver que podemos escoger uno y otro además de que se puede observar algo importante y es que podemos definir un índice que sirva para irme moviendo a través de las diferentes denominaciones, esto nace debido a que al cambiar de número 1 para consultar otras denominaciones ya no vamos a volver a usar ese número.

Un estado podríamos definirlo con el número a representar y el índice que me dice qué denominación se tiene

El caso base es formar el número cero lo cual puede realizarse de una única manera la cual es con 0 monedas

Dp[índice][0] = 0 monedas

Podriamos entonces definir una inicialización con ceros en el renglón y columna cero

Dp[0][0]=0

Las condiciones a tomar en cuenta es que cuando la valor actual y la denominación actual haga que sea negativo entonces no se toma en cuenta dicho valor

```
if(i-monedas[j]>=0)
```

En este caso la solución podría mejor modelarse por medio de una tabla memo, o sea una solución top down

```
int backtrack(int valor, int indice)
    {
        // Caso base
        if(valor==0)
            {
                return 1;
            }
        int sol=0;      // Se cuenta por cada rama.
        for(int denominacion=indice;denominacion<n;denominacion++)
            {
                if(valor-monedas[denominacion]>=0)
                sol+= backtrack(valor-monedas[denominacion],denominacion);
            }

        return sol;
    }
```

Agregando la tabla memo

```
int memo[2000][2000];
int backtrack(int valor, int indice)
    {
        // Caso base
        if(valor==0)
            {
                return 1;
            }

        if(memo[indice][valor]!=-1) return memo[indice][valor];    // Guardando el estado

        int sol=0;      // Se cuenta por cada rama.
        for(int denominacion=indice;denominacion<n;denominacion++)
            {
                if(valor-monedas[denominacion]>=0)
                sol+= backtrack(valor-monedas[denominacion],denominacion);
            }

        return memo[indice][valor]=sol;
    }
```

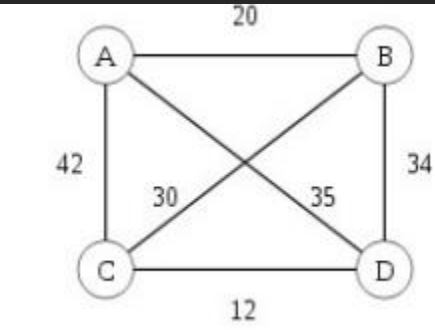
El único tema sería el por qué reiniciar el que va marcando las soluciones en cero en cada llamada recursiva, esto podría ser debido a que sería la suma de las ramas que dieron una solución, por ende si no reiniciara esa suma se estaría sumando con las anteriores así en cada llamada recursiva.

La complejidad es de O(nV) de tiempo

No obstante, si ya no se computa los estados pues la complejidad DP se reduce a constante, entonces al momento de tener varias solicitudes (queries) en el mismo conjunto de monedas entonces la tabla memo deberá conservarse y no se tendrá que reiniciar por ende los cálculos se reducen a tiempo constante

**Traveling Salesman Problem (TSP)**

Este problema se trata que dado ‘n’ ciudades con sus caminos y cada uno tiene un determinado costo, se debe de obtener la ruta que dé el costo más corto tal que se visite todas las ciudades una sola vez y al final se regresar a la ciudad donde se inició el recorrido




| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 20 | 42 | 35 |
| B | 20 | 0 | 30 | 34 |
| C | 42 | 30 | 0 | 12 |
| D | 35 | 34 | 12 | 0 |

Hay 4 vértices, por ende hay 4!=24 caminos donde mínimo uno de ellos se obtiene un mínimo costo y se recorre todas las ciudades, obviamente se puede obtener más de una solución óptima

Se observa que en la matriz el cómo está dada al insertar un vértice en los renglones y el otro en las columnas se tiene en esa posición el costo de ir al vertice1 al vertice2, esto se observa al momento de que se tiene tabla[A][B]=20 y esto es cierto, al ver la gráfica se observa que el costo de ir al vértice A al vértice B es de 20, de este modo sucede para los demás vértices

Si se resuelve con fuerza bruta se tendría que checar todas los posibles permutaciones de los vértices de tal modo que se cumpla con lo comentado anteriormente, para esto se tendría una complejidad de (n-1)! debido a que esa es la cantidad de subconjuntos que se podrían formar y de ahí escoger el que dé el costo del camino más pequeño.

Se tendría que usar permutaciones debido a que un conjunto sería generar también conjunto de 1 solo vértice y nosotros queremos a los 4 vértices, por ende son permutaciones y además el orden importa puesto que no sería lo mismo viajar de ABC a viajar de ACB ya que hay un costo diferente

Segú el libro, al hacer esta solución podría funcionar aproximadamente para 12 vértices como máximo, y si se necesita computar varios queries entonces como máximo se tendría 11 vértices. Esto se ve debido a que la creación de todas las permutaciones es de complejidad factorial entonces 12! Ya es algo muy grande

Algo que podemos ver es que existe sub problemas repetidos, podemos suponer que tenemos los dos siguientes caminos:

A-B-C- (¿?)

A-C-B – (¿?)

Por ende, se ve que al poder visitar solo una vez cada camino y además de que se tiene que al final regresar al vértice donde se partió, entonces el único vértice que en ambos casos debe de computarse sería D y al final ese D tendría que regresar a A.

Entonces, se ve que ese caso de computar D y este ya esté conectado a A sería un sub problema el cual en este ejemplo se ver que se repite 2 veces, por ende en lugar de computarlo de nuevo podríamos guardarlo y así cuando se necesite acceder a él lo cual al hacer esto se estaría ahorrando mucho tiempo debido a que solo accedemos al resultado y no se tiene que computar todo de nuevo

Lo otro a analizar sería el cómo se podría representar estos sub problemas, los states, para esto se puede ver que se tendría que tener en cuenta e último vértice que fue visitado, o sea la ciudad, y un set en donde se tenga guardado las ciudades (vértices) que ya fueron visitadas para evitar visitarlas de nuevo.

De lo anterior se puede generar la duda de cómo representar este subcojunto y pues tan solo se debe de recordar el truco usado anteriormente con un bits (bitmask) el cual era usar una variable entero como un conjunto de tal manera que se para añadir cada número o lo que sea se verá representado por re corrimientos a la izquierda para que en esa posición dentro del entero se prenda ese bit.

Cabe recordar que los enteros en su representación de bits tiene 36 por ende es límitado, por lo cual al atacar el problema usando este set es sabiendo que la cantidad de ciudades no va a sobrepasar de esa cantidad de bits disponibles en el entero

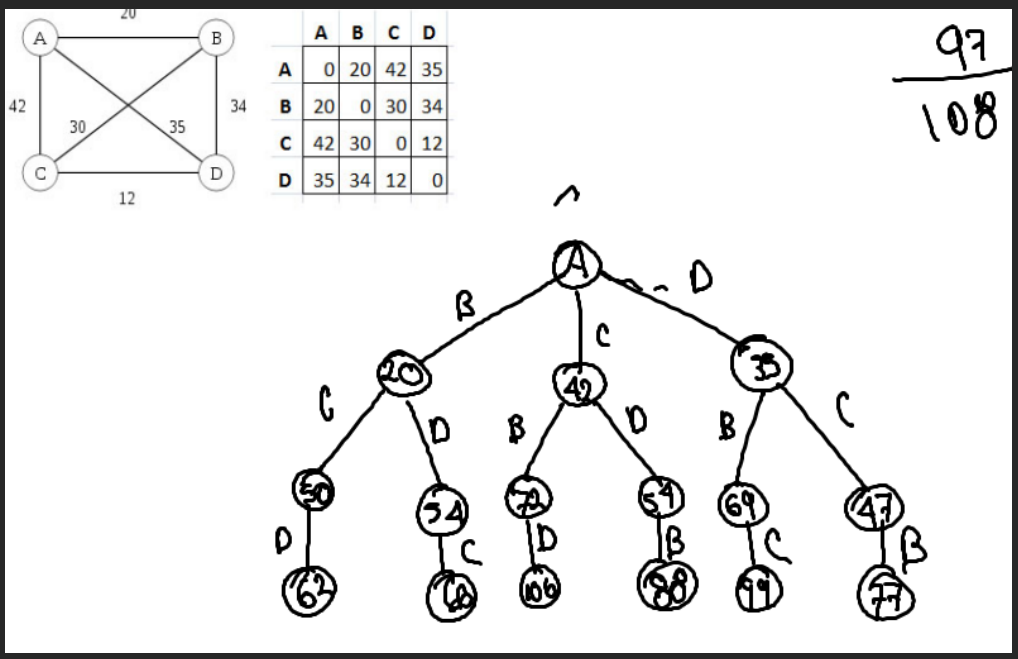
Operaciones que se usarán con este set de bitmask:

Checar si la ciudad ‘i’ está: set&(1<<i)

Añadir la ciudad al set: set |= (1<<i)

De este último se podría quitar el ‘=’ debido a que en la práctica al realizar las llamadas recursivas no es conveniente esto ya que al regresar se tendría que quitar nuevamente para buscar otro de manera manual

Como siempre, se debe de deducir la relación de concurrencia apoyándonos delo acercamiento por búsqueda completa, al parecer esto sería como la generación de permutaciones pero guardando los sub problemas o lo que se tenga para después acceder a ellos en tiempo constante




La ayuda de Dp será evitar re computar las longitudes de todos los estados, y es lo normalmente ayuda en los problemas de dp esos resultados de los sub problemas.

Al final, cuando ya estemos haciendo el caso base se va a retornar el valor sumando la longitud de camino al vértice A, de tal modo que de ahí ya se escoge el menor.

La relación de concurrencia sería obtener el mínimo, donde se deberá empezar por todos los caminos en realidad, o sea que primero tendremos que explorar una rama y después regresar para explorar los anteriores, se observa que se debe de escoger todas entonces se hará con un ciclo for de tal forma para saber qué escoger será con ayuda del subconjunto y de esto se deduce que A,B,C,D serán representados con sus índices 0,1,2,3

Voy a poner mi entrada así para poder tener algo qué resolver

```
4 0
0 20 42 35
20 0 30 34
42 30 0 12
35 34 12 0
```

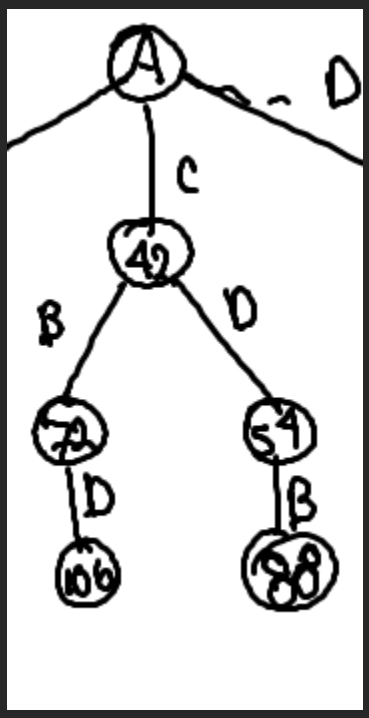
4 es la cantidad de ciudades y hay 6 caminos entre ellos (no es necesario)

Posteriormente se pasa la matriz

```
int backtrack(int ciudad, int subSet)
    {
        int sol=0;
        // Caso base
        if(subSet==pow(2,ciudades)-1)   // Todas las ciudades fueron visitadas
            {
                cout<<"Ultima ciudad: "<<ciudad<<endl;
                return caminos[ciudad][inicio];

            }

        // Caso general
        int minimo=1e9;
        for(int c=0;c<ciudades;c++)
            {
                if(subSet&(1<<c))
                    {
                        continue;       // La ciudad ya fue visitada.
                    }
                else    // Si no está esa ciudad se añade
                    {
                        minimo = min(minimo,caminos[ciudad][c] + backtrack(c,subSet|(1<<c)) );

                    }
            }
        return minimo;
    }
```

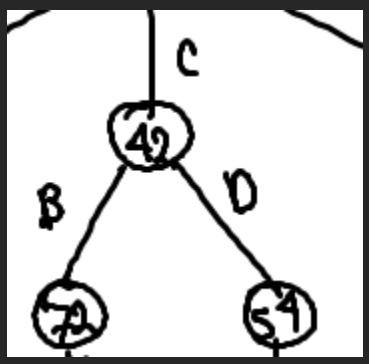
Esto es el código que resuelve el ejercicio por medio de backtracking recursivo, o sea complete search. Lo más importante de aquí es fijarse sobre la inicializar la variable mínimo antes del for, al menos a mí me causaba confusión esto debido a que pensaba que la inicializarlo como global pues se estaría haciendo la comparación ya con los anteriores o sea que se estaría llevando el control de quién es el mejor hasta ese momento, pero la realidad es que para hacer eso se debe de inicializarlo antes del ciclo for.

Para entender lo anterior nos tenemos que regresar al árbol, y observamos que en cada rama, a partir del primer vértice de ahí se parte a más soluciones, y en este caso esa rama abre a dos sub problemas más dando en total que esa rama nos de dos soluciones




A lo que me refiero es esto, se abre a dos soluciones más

Esto es obvio debido a que al explorar primero la rama de la izquierda pues ya finalmente se acaba de explorar toda esa sección para después regresar y verificar si el anterior for (der la anterior llamada de backtraking) puede avanzar a otro vértice que no fue visitado, no obstante no podemos y regresamos hasta el vértice C, y es ahí en donde se abren los dos caminos




Una vez en este estado se deberá de tener en cuenta que cada llamada termina al recorrer todo el for, o sea cada llamada a la función tendrá su fin al revisar si ya todas las ciudades fueron visitadas, entonces para terminar se retorna el mínimo, o sea el costo del camino hastra ese entonces:

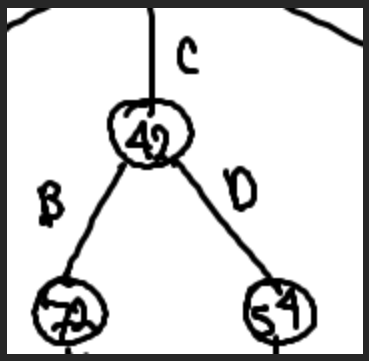
```
for(int c=0;c<ciudades;c++)
    {
        if(subSet&(1<<c))
            {
                continue;       // La ciudad ya fue visitada.
            }
        else    // Si no está esa ciudad se añade
            {
                minimo = min(minimo,caminos[ciudad][c] + backtrack(c,subSet|(1<<c)) );

            }
    }
return minimo;
```

Por ende, se puede pensar que al recorrer primero la rama de lzquierda en realidad esa comparación sí se estaría haciendo así

Minimo = min(1e9,backtrack-..)

Entonces se va a retornar en cada termino de la llamada de backtrack el minimo, provocando que hasta ese entonces entonces del vértice C también se retorne el valor del mínimo, y ahora será ya en ese vértice C que tiene ahora ya el valor minimo para la variable mínimo

Una vez que ya tiene esto ya puede avanzar su ciclo for para poder explorar la rama de la derecha




Y ahora al realizar la comparación con el min() se estará haciendo con el valor que nos arroje las llamadas y con el valor retornado de la llamada anterior que fue el minimo anterior cxonseguido

Es por esto que se coloca esa inicialización del min en cada llamada puesto que existen diversas ramas, diferentes caminos

Finalmente, solo agregamos la tabla memo para realizar la solución dp top down, para esto se debe de tomar en cuenta la cantidad de vértices que se tendrán, puesto que es evidente que el estado a guardar va a depender directamente de la ciudad actual que se tenga como parámetro y del subset por ende se debe de conocer la cantidad de vértices como máximo que se tendrá. En este caso se considera tener 11 vértices (igual que el problema de abajo)

Nota importante: es posible designar el espacio de la matriz por medio de la siguiente operación de a nivel de bits

int memo[11][1<<11];

Entonces, todo se reduce a solo insertar y ya

```
int backtrack(int ciudad, int subSet)
    {
        int sol=0;
        // Caso base
        if(subSet==pow(2,ciudades)-1)   // Todas las ciudades fueron visitadas
            {
                return caminos[ciudad][inicio];
            }

        if(memo[ciudad][subSet]!=-1)    return memo[ciudad][subSet];

        // Caso general
        int minimo=1e9;
        for(int c=0;c<ciudades;c++)
            {

                if(subSet&(1<<c))
                    {
                        continue;       // La ciudad ya fue visitada.
                    }
                else    // Si no está esa ciudad se añade
                    {
                        minimo = min(minimo,caminos[ciudad][c] + backtrack(c,subSet|(1<<c)) );

                    }
            }
        memo[ciudad][subSet] = minimo;
        return memo[ciudad][subSet];
```

**10496 - Collecting Beepers** **Se verá este problema para ilustrar el código**

El problema se trata de que se tiene un robot que vive en un sistema de coordenadas las cuales son rectangulares y se tiene una cierta cantidad de beepers que se deben de colocar o llegar a ellos (no entendí esa parte) en su mundo, o sea en su sistema de coordenadas. El chiste es que se debe de conseguir la longitud del camino más corto para llegar a cada coordenadas desde su inicio, o sea se visita todos los beepers y se regresa al inicio.

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

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Inicio | - | - | - | - | - | - | - | - | | |
| - | | | | | | | | - | | |
| - | X | | | | | | | - | | |
| | - | | | | | | | X | | |
| | - | - | - | X | X | - | - | - | | |
| | | | | | | | | | | |
| | | | | | | | | | | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | |
| | | | | | | | | | | |

Esta es mi ilustración y es la forma en que se estaría resolviendo, como se observó dan coordenadas y no vértices, en la descripción se específica que cada movimiento que se realice se realiza con un costo de 1, entonces a partir de esta información es posible ver este problema como uno de TSP y no un recorrido a través de un grid de forma pura (por decirlo de una forma) entonces como se tiene las coordenadas y se supone que sería como lo ilustrado arriba, podemos ser capaces de obtener una matriz con esas distancias entre los beepers, de tal forma que podríamos designar que la primera coordenada que se coloca podría ser el beeper 0, y así consecutivamente

Entonces, lo anterior se lleva a realizar una matriz con las distancias de cada vértice o beeper

Viendo la tabla podemos considerar que la distancia necesaria a recorrer entre cada beeper podría ser la suma de la diferencia de las coordenadas entre cada beepers, esto es:

Para el beeper 0: coor = 2 3

Para el beeper 1: coor = 5 5

Entonces al realizar la resta entre las coordenadas se obtiene: 3 2 y al sumarlas da 5, esto quiere decir que la distancia entre el beeper 0 y 1 es de 5, lo cual podemos comprobar

| | | | |
|---|---|---|---|
| X | | | |
| - | | | |
| - | - | - | X |

Esto mismo deberíamos hacer para el beeper 0 con el beeper 1,2,3,n, y después se repite lo mismo para 2 con cada beeper, esta operación deberá ser cuadrática.

```
cin>>inicio_x;  cin>>inicio_y;
coordenadas.push_back(make_pair(inicio_x,inicio_y));// La ciudad donde inicia siempre es 0
cin>>ciudades;

for(int a=0;a<ciudades;a++)
    {
        int x,y; cin>>x>>y;
        coordenadas.push_back(make_pair(x,y));
    }

for(int a=0;a<ciudades+1;a++)
    {
        for(int b=0;b<ciudades+1;b++)
            {
                caminos[a][b] = abs(coordenadas[a].first - coordenadas[b].first) + abs(coordenadas[a]
            }
    }
```

Se traduce en esto donde podemos hacer que la coordenada de inicio se traduzca a una ciudad, o sea que la ciudad en donde iniciemos siempre sea 0 lo cual puede facilitar mucho las cosas.

Lo siguiente es un ciclo for en donde obtendré las coordenadas en el siguiente ciclo for obtenga la matriz con sus distancias, de la misma forma que fue descrita anteriormente

Dentro de la función de dp en sí solo se hicieron dos cambios y fue debido a que se coinsideró el inicio como una nueva ciudad

```
// Caso base
if(subSet==pow(2,(ciudades+1))-1)   // Todas las ciudades fueron visitadas
```

```
for(int c=0;c<=ciudades;c++)
```

Finalmente, se reset las estructuras en donde se colocó la información

```
// Reinicio de las estructuras que contienen los valores.
memset(caminos,-1,sizeof(caminos));
coordenadas.clear();
```

**PROBLEMAS NO CLÁSICOS**

Se aborda el tema en donde pues es obvio que problemas clásicos en su forma pura no suelen ser colocados en los concursos modernos, no obstante se pueden encontrar eso sí problemas que deriven de ellos. No obstante, el verdadero reto nace en aquellos problemas que no derivan de ninguno que sean clásicos, y estos son los problemas no clásicos los cuales serán de gran ayuda para seguir desarrollando habilidades de DP, por lo cual lo siguiente tratará de ellos.

Un problema no clásico fue el abordado anteriormente y era Wedding Shopping, entonces al observar se puede seguir usando las mismas relaciones de concurrencia para poder encontrar una relación para una solución DP, entonces podría seguir apoyando del acercamiento por Complete Search, o sea por un backtracking recursivo.

**10943 - How do you add?**

El problema se trata de dos personas que se quedan varados en una isla, entonces para pasar el tiempo se propusieron un problema como pasa tiempo y con consecuencias fatales, o algo así.

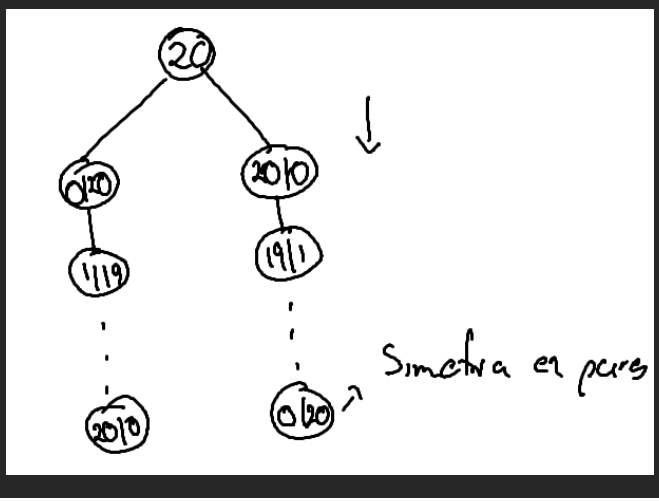
El punto es que se entre un número n y un número k, donde debemos de ver las formas en que es posible realizar la suma ‘k’ número no negativos ni cero para formar el número ‘n’, y el ejemplo es el siguiente

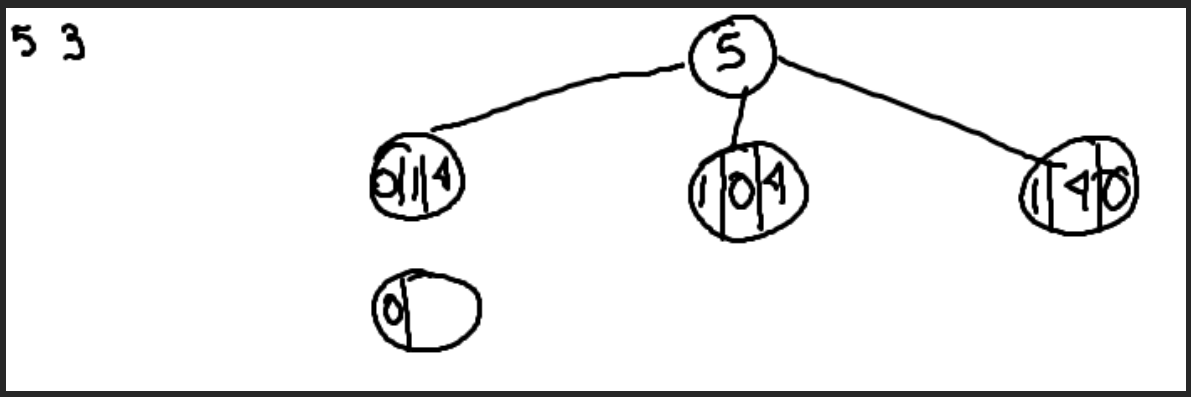
20 2

Entonces, aquí nos pide de cuántas formas o maneras hay para formar 20 con dos números y lo cual significa:

1+19 = 20

Esta es una forma, se usan dos números y se está creando el 20.




Para tres se observa esto entonces el comienzo de cada uno es técnicamente ir considerar siempre que al menos uno será 0 mientras que los otros quedan como complementados

También debería existir una cuarta rama donde no haya ceros sino que todos tengan un número y podría comenzar como 1 1 3

Este es el código

```
int memo[105][105];
int modulo=1000000;
int n;
int backtrack(int numero, int tipo_numero)
    {
        // Caso base
        if( tipo_numero==1 )
            {
                return 1;
            }

        if(memo[numero][tipo_numero]!=-1)    return memo[numero][tipo_numero];

        int sol=0;
        for(int i=0;i<=numero;i++)
            {
                sol= (sol + backtrack(numero-i, tipo_numero-1)) % modulo;
            }

        return memo[numero][tipo_numero]=sol;
    }
```

La forma en que se solucionó fue pensando en lo siguiente y es que ‘k’ nos va a decir con cuantos número se debe de reconstruir la solución por ende ‘n’ podrá realizarse las restas y como se observa se puede haciendo la resta usando un ciclo for

Se usa la variable que está sumando donde se inicializa dentro de la función lo cual ya fue explicado anteriormente y fue por el tema de las ramas.

Entonces, al estar haciendo esta sunna en realidad no se estará emulando al 100% el proceso de realizar todas las soluciones, no obstante sí podrá usarse como base, o sea que el chiste es pensar sobre el caso base.

El caso base es cuando k=1, esto se debe a que cuando k=1 significa que tenemos que construir a un número ‘n’ con 1 número y la única forma que se puede hacer esto es usando el propio n, no se podría hacer desde 0,1,2 ya que al no estar sumando con nada no se puede hacer cumplir que sea ‘n’ la construcción, entonces en pocas palabras solo hay una manera de armar ‘n’ cuando k=1

Lo siguiente es ya pensar sobre los casos en donde k>1 y es ahí en donde se debe de pensar en que se va a tener ‘k’ números que sumados den ‘n’ y por lo tanto se deberá decremento este número en cada recursión y dando la oportunidad en cada llamada se esté restando a ‘n’ el número recorrido con for ya que como se vio en el ejemplo se es posible ver que siempre se va tener un decremento de n a 0, y esto se puede apreciar más con el ejemplo del problema en donde k=2, entonces teniendo esto en cuenta y al ver sobre pensar el caso donde k=3 pues se va a repetir este mismo comportamiento en cada rama solo que una posición diferente o ya sea que se vaya de 0 a n o de n a 0

En otras palabras

$$
\begin{array}{l}
0+20 \\
1+19 \\
2+18 \\
3+17 \\
4+16 \\
5+15 \\
\ldots \\
18+2 \\
19+1 \\
20+0
\end{array}
$$

Se observa lo visto anteriormente, y se repite en ambas posiciones de los número el mismo comportamiento, siempre hay incremente o decremento de 1, se va de ‘n’ hasta 0

Y este mismo comportamiento se aprecia con k=3, y hasta k=cualquier número, entonces generalizando esto se puede deducir que al no pedirnos todas las respuestas posibles entonces es posible realizar la resta del número ‘n’ a todos por igual.

Lo que significa es que en sí en el ejemplo se ve que inicia en 0 20 y termina en 20 0 y esto es que simplemente está volteado, entonces en lugar de realizar este volteo es posible generalizar para que las restas se haga siempre de 20 a 0, o sea de ‘n’ a 0, y esto es lo mismo que se estará haciendo en el programa

El siguiente tema es ver, ¿Por qué se usa como condición parámetro en el for?

```
for(int i=0;i<=numero;i++)
```

Esto es debido a que se hace la siguiente observación, cuando se hace las restas pues no tendría sentido llegar en todos los números para armar a ‘n’ el mismo número debido a que se va a sobrepasar del número entonces por eso se usa este como parámetro para evitar sobre sumar estos números

Estas son observaciones que deben de realizarse para poder entender bien los problemas no clásicos.