

# 75.29 - Teoría de Algoritmos I



## Trabajo Práctico N°2 Programación dinámica

Apellidos y nombres	Padrón	Correo electrónico
Sueiro, Ignacio	96817	iasueiro@gmail.com
Graffe, Fabricio	93158	zebas.grafe@gmail.com
Battan, Manuel Victoriano	95851	mvbattan@gmail.com
Reyes, Ezequiel	92119	ezequielmreyes@gmail.com

**2° Cuatrimestre 2016**

## Introducción teórica

La programación Dinámica (PD) resuelve problemas a través de combinar soluciones a subproblemas. Comienza resolviendo las instancias más simples de los problemas, y guardando sus resultados en alguna estructura de datos especial, para construir soluciones de instancias más complejas, se divide la instancia en subproblemas más simples y se recuperan los resultados ya calculados de la estructura de datos. Esta se aplica cuando los subproblemas no son independientes entre sí, es decir los subproblemas tienen subproblemas en común. Esto se denomina superposición de subproblemas.

**Top-down:** Es la caída directa de la formulación recursiva de cualquier problema. Si la solución a cualquier problema se puede formular recursivamente usando la solución a sus sub-problemas, y si sus sub-problemas se solapan, entonces uno puede fácilmente memorizar o almacenar las soluciones a los subproblemas en una tabla. Siempre que intentamos resolver un nuevo subproblema, primero verificamos la tabla para ver si ya está resuelto. Si una solución ha sido registrada, podemos usarla directamente, de lo contrario resolvemos el sub-problema y agregamos su solución a la tabla.

**Bottom-up:** Una vez que formulamos la solución a un problema recursivamente como en términos de sus sub-problemas, podemos intentar reformular el problema de una manera ascendente: trate de resolver primero los sub-problemas y use sus soluciones para construir- Y llegar a soluciones a sub-problemas más grandes. Esto también se hace generalmente en una forma tabular iterativamente generando soluciones a los sub-problemas más grandes y más grandes usando las soluciones a los sub-problemas pequeños. Por ejemplo en el caso de Fibonacci, si ya conocemos los valores de  $F(41)$  y  $F(40)$ , podemos calcular directamente el valor de  $F(42)$ .

## Problema de la mochila

Para evitar la utilización de la recursividad en la solución (recordando que con este enfoque, para problemas con N muy grande puede traer problemas de tiempos y espacio en la ejecución debido al crecimiento de la pila) se optó por la utilización de **bottom-up**. Se almacenan los valores dentro de una tabla para que el algoritmo pueda ir consultando los mismos a medida que avanza la ejecución con el fin de no re-calcularlos.

### Resultados de test:

Se testeó contra algunos archivos provenientes de [smallcoeff\\_pisinger.tgz](#) y [hardinstances.tgz](#) las corridas arrojaron los siguientes valores (los archivos completos se encuentran adjuntos al código fuente, para el análisis actual se pone sólo 1 caso de cada archivo corrido):

#### Smallcoeff

knapPI\_1\_50\_1000\_results:

```
"Listado numero 1, peso maximo: 1008"
"Id:7, Valor: 457, Peso: 43"
"Id:11, Valor: 791, Peso: 9"
"Id:24, Valor: 700, Peso: 72"
"Id:26, Valor: 874, Peso: 138"
"Id:33, Valor: 908, Peso: 97"
"Id:38, Valor: 931, Peso: 70"
"Id:39, Valor: 726, Peso: 98"
"Id:49, Valor: 724, Peso: 29"
"Id:54, Valor: 641, Peso: 46"
"Id:61, Valor: 800, Peso: 90"
"Id:122, Valor: 770, Peso: 10"
"Id:135, Valor: 569, Peso: 30"
"Id:138, Valor: 609, Peso: 100"
"Id:147, Valor: 889, Peso: 28"
"Id:148, Valor: 443, Peso: 61"
"Id:152, Valor: 406, Peso: 66"
```

El valor optimo conseguido por Pisinger es 11238 y el tiempo 0.0 mientras que los valores conseguidos por el algoritmo fueron 11238 y el tiempo 0.0369229316711

-----

knapPI\_1\_200\_1000\_results:

```
"Listado numero 1, peso maximo: 1008"
"Id:7, Valor: 457, Peso: 43"
"Id:11, Valor: 791, Peso: 9"
"Id:24, Valor: 700, Peso: 72"
"Id:26, Valor: 874, Peso: 138"
"Id:33, Valor: 908, Peso: 97"
"Id:38, Valor: 931, Peso: 70"
"Id:39, Valor: 726, Peso: 98"
"Id:49, Valor: 724, Peso: 29"
"Id:54, Valor: 641, Peso: 46"
```

"Id:61, Valor: 800, Peso: 90"  
"Id:122, Valor: 770, Peso: 10"  
"Id:135, Valor: 569, Peso: 30"  
"Id:138, Valor: 609, Peso: 100"  
"Id:147, Valor: 889, Peso: 28"  
"Id:148, Valor: 443, Peso: 61"  
"Id:152, Valor: 406, Peso: 66"

**El valor optimo conseguido por Pisinger es 11238 y el tiempo 0.0 mientras que los valores conseguidos por el algoritmo fueron 11238 y el tiempo 0.0369229316711**

-----

knapPI\_1\_500\_1000\_results:

"Listado numero 1, peso maximo: 2543"  
"Id:7, Valor: 457, Peso: 43"  
"Id:11, Valor: 791, Peso: 9"  
"Id:13, Valor: 667, Peso: 122"  
"Id:14, Valor: 598, Peso: 94"  
"Id:24, Valor: 700, Peso: 72"  
"Id:26, Valor: 874, Peso: 138"  
"Id:33, Valor: 908, Peso: 97"  
"Id:38, Valor: 931, Peso: 70"  
"Id:39, Valor: 726, Peso: 98"  
"Id:49, Valor: 724, Peso: 29"  
"Id:54, Valor: 641, Peso: 46"  
"Id:61, Valor: 800, Peso: 90"  
"Id:122, Valor: 770, Peso: 10"  
"Id:135, Valor: 569, Peso: 30"  
"Id:138, Valor: 609, Peso: 100"  
"Id:147, Valor: 889, Peso: 28"  
"Id:148, Valor: 443, Peso: 61"  
"Id:152, Valor: 406, Peso: 66"  
"Id:216, Valor: 998, Peso: 181"  
"Id:217, Valor: 859, Peso: 10"  
"Id:250, Valor: 477, Peso: 34"  
"Id:255, Valor: 660, Peso: 105"  
"Id:270, Valor: 253, Peso: 33"  
"Id:274, Valor: 888, Peso: 40"  
"Id:282, Valor: 943, Peso: 52"  
"Id:335, Valor: 128, Peso: 15"  
"Id:348, Valor: 911, Peso: 19"  
"Id:363, Valor: 859, Peso: 74"  
"Id:374, Valor: 933, Peso: 144"  
"Id:380, Valor: 954, Peso: 26"  
"Id:383, Valor: 446, Peso: 26"  
"Id:420, Valor: 770, Peso: 21"  
"Id:422, Valor: 542, Peso: 44"  
"Id:427, Valor: 282, Peso: 8"  
"Id:447, Valor: 764, Peso: 98"  
"Id:464, Valor: 126, Peso: 16"  
"Id:470, Valor: 856, Peso: 71"  
"Id:474, Valor: 195, Peso: 5"  
"Id:477, Valor: 952, Peso: 64"  
"Id:481, Valor: 961, Peso: 157"

"Id:494, Valor: 666, Peso: 45"

"Id:495, Valor: 931, Peso: 52"

El valor optimo conseguido por **Pisinger** es **28857** y el tiempo **0.0** mientras que los valores conseguidos por el algoritmo fueron **28857** y el tiempo **0.24060177803**

-----

"Listado numero 1, peso maximo: 25016"

"Id:7, Valor: 457, Peso: 43"

"Id:11, Valor: 791, Peso: 9"

"Id:14, Valor: 598, Peso: 94"

"Id:24, Valor: 700, Peso: 72"

"Id:26, Valor: 874, Peso: 138"

"Id:33, Valor: 908, Peso: 97"

"Id:38, Valor: 931, Peso: 70"

"Id:39, Valor: 726, Peso: 98"

"Id:49, Valor: 724, Peso: 29"

"Id:54, Valor: 641, Peso: 46"

"Id:61, Valor: 800, Peso: 90"

Se omiten el resto de los objetos pues no le aportan datos al análisis posterior.

El valor optimo conseguido por **Pisinger** es **276457** y el tiempo **0.0** mientras que los valores conseguidos por el algoritmo fueron **276457** y el tiempo **36.7090628147**

-----

## Hard instances

### knapPI\_11\_20\_1000\_results

"Listado numero 1, peso maximo: 970"

"Id:7, Valor: 918, Peso: 594"

"Id:10, Valor: 510, Peso: 330"

El valor optimo conseguido por **Pisinger** es **1428** y el tiempo **0.0** mientras que los valores conseguidos por el algoritmo fueron **1428** y el tiempo **0.0105829238892**

-----

### knapPI\_11\_50\_1000\_results

"Listado numero 21, peso maximo: 3964"

"Id:3, Valor: 35, Peso: 840"

"Id:5, Valor: 370, Peso: 8"

"Id:6, Valor: 35, Peso: 840"

"Id:7, Valor: 1295, Peso: 28"

"Id:8, Valor: 1110, Peso: 24"

"Id:9, Valor: 555, Peso: 12"

"Id:10, Valor: 555, Peso: 12"

"Id:12, Valor: 70, Peso: 1680"

"Id:14, Valor: 1110, Peso: 24"

"Id:15, Valor: 370, Peso: 8"

"Id:16, Valor: 1110, Peso: 24"

"Id:18, Valor: 1295, Peso: 28"

"Id:20, Valor: 925, Peso: 20"

"Id:23, Valor: 925, Peso: 20"

"Id:25, Valor: 740, Peso: 16"

"Id:26, Valor: 1295, Peso: 28"

"Id:27, Valor: 555, Peso: 12"

"Id:29, Valor: 1110, Peso: 24"  
"Id:30, Valor: 925, Peso: 20"  
"Id:33, Valor: 1295, Peso: 28"  
"Id:35, Valor: 925, Peso: 20"  
"Id:36, Valor: 1295, Peso: 28"  
"Id:37, Valor: 370, Peso: 8"  
"Id:38, Valor: 1110, Peso: 24"  
"Id:39, Valor: 1295, Peso: 28"  
"Id:40, Valor: 1480, Peso: 32"  
"Id:41, Valor: 925, Peso: 20"  
"Id:42, Valor: 1295, Peso: 28"  
"Id:43, Valor: 185, Peso: 4"  
"Id:47, Valor: 185, Peso: 4"  
"Id:49, Valor: 1480, Peso: 32"  
"Id:50, Valor: 1110, Peso: 24"

El valor optimo conseguido por Pisinger es 27335 y el tiempo 0.0 mientras que los valores conseguidos por el algoritmo fueron 27335 y el tiempo 0.0533170700073

-----

#### knapPI\_11\_200\_1000\_results

"Listado numero 11, peso maximo: 16133"  
"Id:5, Valor: 339, Peso: 294"  
"Id:7, Valor: 904, Peso: 784"  
"Id:8, Valor: 113, Peso: 98"  
"Id:9, Valor: 678, Peso: 588"  
"Id:10, Valor: 452, Peso: 392"  
"Id:14, Valor: 565, Peso: 490"  
"Id:15, Valor: 791, Peso: 686"  
"Id:16, Valor: 113, Peso: 98"  
"Id:18, Valor: 1130, Peso: 980"  
"Id:20, Valor: 1130, Peso: 980"  
"Id:23, Valor: 226, Peso: 196"  
"Id:25, Valor: 113, Peso: 98"  
"Id:26, Valor: 678, Peso: 588"  
"Id:27, Valor: 452, Peso: 392"  
"Id:30, Valor: 678, Peso: 588"  
"Id:33, Valor: 226, Peso: 196"  
"Id:35, Valor: 452, Peso: 392"  
"Id:36, Valor: 452, Peso: 392"  
"Id:37, Valor: 565, Peso: 490"  
"Id:38, Valor: 1017, Peso: 882"  
"Id:39, Valor: 452, Peso: 392"  
"Id:40, Valor: 1017, Peso: 882"  
"Id:41, Valor: 226, Peso: 196"  
"Id:42, Valor: 226, Peso: 196"  
"Id:43, Valor: 1130, Peso: 980"  
"Id:47, Valor: 1130, Peso: 980"  
"Id:49, Valor: 791, Peso: 686"  
"Id:50, Valor: 565, Peso: 490"  
"Id:51, Valor: 904, Peso: 784"  
"Id:57, Valor: 1017, Peso: 882"

El valor optimo conseguido por Pisinger es 18532 y el tiempo 0.0 mientras que los valores conseguidos por el algoritmo fueron 18532 y el tiempo 0.714709997177

-----

knapPI\_11\_5000\_1000\_results

"Listado numero 1, peso maximo: 22887"

"Id:5, Valor: 612, Peso: 396"

"Id:7, Valor: 918, Peso: 594"

"Id:8, Valor: 408, Peso: 264"

"Id:9, Valor: 714, Peso: 462"

Se omiten el resto de los objetos pues no le aportan datos al análisis posterior.

El valor optimo conseguido por Pisinger es 35292 y el tiempo 0.01 mientras que los valores conseguidos por el algoritmo fueron 35292 y el tiempo 25.7434761524

-----

En esta tanda de casos se encontró que el algoritmo realizado respondió en cuanto a resultados de manera similar (se pudo verificar que llegó a mismo resultado total pero utilizando distintos items con mismo valor-peso) y óptima. En cuanto a los tiempos de ejecución, se pudo notar que para  $n < 500$  los tiempos se mantuvieron dentro del parámetro establecido (en su mayoría con valores que no pasan 1 segundo) , pero a medida que se fue aumentando el  $n$ , el tiempo comenzó a incrementar, por ejemplo con  $n = 5000$  el tiempo fue de 36 segundos contra 0.0 obtenido por Pisinger, claramente se puede notar una deficiencia en el algoritmo implementado.

## Problema del viajante

Se resolvió el problema del viajante utilizando una solución que hace uso de la programación dinámica, más específicamente el algoritmo Bellman-Held-Karp.

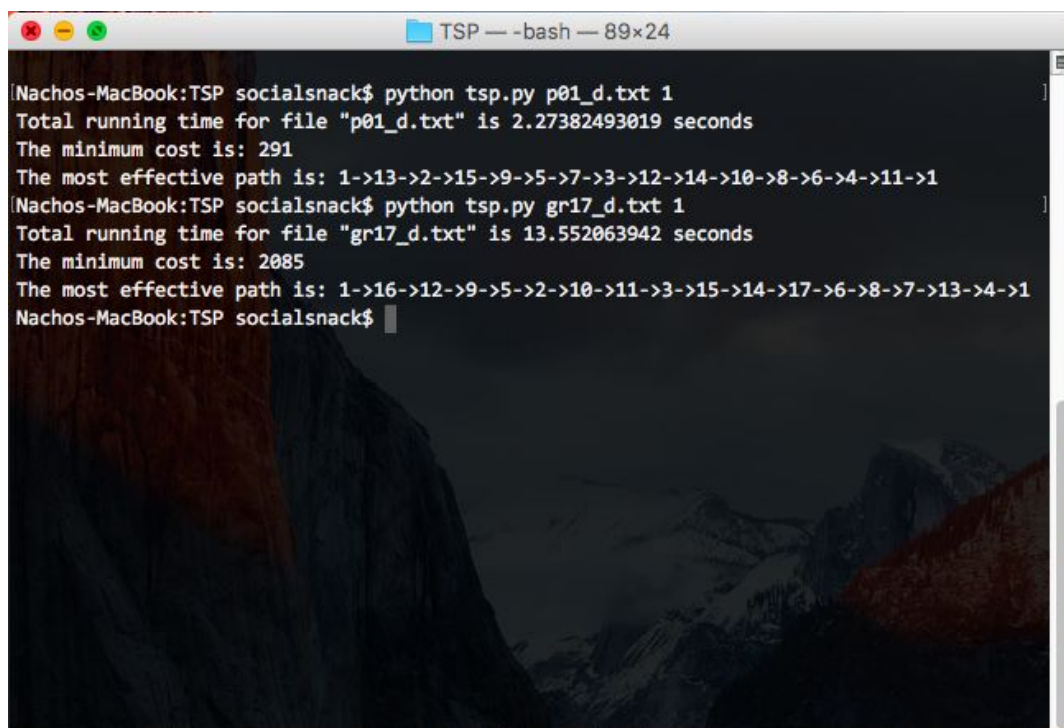
Este consiste en calcular todas las combinaciones posibles de caminos que inicien en el origen, culminen en un destino especificado (en el último paso será el mismo origen) y pasen por 0,1,...,n ciudades.

En cada paso se calcula el camino mínimo que pase por todos los vértices especificados, utilizando las soluciones del paso anterior para evitar calcular recursivamente estas.

Esta solución tiene una complejidad en tiempo de  $O(2^n * n^2)$ , ya que  $2^n$  es la cantidad de combinaciones posibles de caminos para n pasos y en cada una de estas combinaciones se recorre cada posibilidad para ver desde qué vértice es más efectivo llegar.

### Ejemplos

Para testear la implementación del algoritmo se utilizó el set de datos recopilado por John Burkardt. Más específicamente los sets "p01" y "gr17". Los resultados obtenidos fueron los siguientes



```
Nachos-MacBook:TSP socialsnack$ python tsp.py p01_d.txt 1
Total running time for file "p01_d.txt" is 2.27382493019 seconds
The minimum cost is: 291
The most effective path is: 1->13->2->15->9->5->7->3->12->14->10->8->6->4->11->1
Nachos-MacBook:TSP socialsnack$ python tsp.py gr17_d.txt 1
Total running time for file "gr17_d.txt" is 13.552063942 seconds
The minimum cost is: 2085
The most effective path is: 1->16->12->9->5->2->10->11->3->15->14->17->6->8->7->13->4->1
Nachos-MacBook:TSP socialsnack$
```

Total running time for file "p01\_d.txt" is 2.27382493019 seconds

The minimum cost is: 291

The most effective path is: 1->13->2->15->9->5->7->3->12->14->10->8->6->4->11->1

Total running time for file "gr17\_d.txt" is 13.552063942 seconds

The minimum cost is: 2085

The most effective path is: 1->16->12->9->5->2->10->11->3->15->14->17->6->8->7->13->4->1



Ambos resultados coinciden con las salidas esperadas.

Los tiempos de ejecución hacen sentido ya que:

- $O(2^{15} * 15^2) = 7.472.800$
- $O(2^{17} * 17^2) = 37.879.808$
- $(2^{17} * 17^2 - 2^{15} * 15^2) / (2^{17} * 17^2) = 0.80$

Con este cálculo podemos ver que, para 15 ciudades el algoritmo debería tardar una quinta parte de lo que tarda para 17 ciudades, lo que, haciendo una estimación burda, daría aproximadamente 11,66 segundos, lo que es bastante similar. Además podemos asignar la discrepancia a tiempo que fue utilizado por otras tareas del CPU, ya que la función utilizada para el control del tiempo de ejecución mide tiempo de reloj y no tiempo de uso del CPU.

Finalmente, no se incluyen tiempos de ejecución para los sets “fri26” y “att48” ya que, luego de lo discutido en una consulta, los tiempos de ejecución escalan muchísimo.

- $(2^{26} * 26^2 - 2^{17} * 17^2) / (2^{26} * 17^{26}) = 0.99916$
- $(2^{48} * 48^2 - 2^{17} * 17^2) / (2^{48} * 48^2) = 0.999999999994$

Haciendo la misma simplificación que antes para dar un estimado, para 26 ciudades tardaría **4.5 horas** y para 48 ciudades alrededor de **7356 años**

Apéndice:

### ***Mochila:***

#### **mochila.py**

```
import csv
import sys

import calculador_mochila
import time

CONST_NOMBRE = 0
CONST_N = 1
CONST_C = 2
CONST_Z = 3
CONST_TIEMPO = 4
CONST_PRIMERA_LINEA = 5
CONST_ID = 0
CONST_VALOR = 1
CONST_PESO = 2
CONST_SEPARADOR = "-----\n"
if __name__ == '__main__':
    if len(sys.argv) != 3:
        print('parametro 1: archivo de entrada, parametro 2: archivo de salida')
        sys.exit()
    archivoEntrada = sys.argv[1]
    archivoSalida = sys.argv[2]
    with open(archivoSalida, 'w') as csvfile:
        spamwriter = csv.writer(csvfile)
        with open(archivoEntrada, 'rb') as archivo:
            lineas = archivo.readlines()
            cantLineas = len(lineas)
            i = 0
            listado = 0
            while(i < cantLineas):
                listado = listado + 1
                items = []
                pesoMaximo = int(lineas[CONST_C + i].split(" ")[1])
                tiempoTotalPisinger = float(lineas[CONST_TIEMPO + i].split(" ")[1])
                valorMaxPisinger = int(lineas[CONST_Z + i].split(" ")[1])
                i = i + 5
                while(lineas[i] != CONST_SEPARADOR):
                    separada = lineas[i].split(",")
                    id = int(separada[CONST_ID])
                    valor = int(separada[CONST_VALOR])
                    peso = int(separada[CONST_PESO])
                    items.append([valor, peso, id])
                    i = i + 1
                inicio = time.time()
                mejorValorAlgoritmo, objetos = calculador_mochila.calcular_mochila(items,
                pesoMaximo)
                tiempoAlgoritmo = time.time() - inicio
```

```

        spamwriter.writerow(["Listado numero {0}, peso maximo: {1}".format(listado,
pesoMaximo)])
        for valor, peso, id in objetos:
            spamwriter.writerow(["Id:{2}, Valor: {0}, Peso: {1}".format(valor, peso,
id)])
        i = i + 2
        spamwriter.writerow(["El valor optimo conseguido por Pisinger es {0} y el tiempo
{1} mientras que los valores conseguidos por el algoritmo fueron {2} y el tiempo
{3}".format(valorMaxPisinger, tiempoTotalPisinger, mejorValorAlgoritmo, tiempoAlgoritmo)])
        spamwriter.writerow(["-----"])

```

## calculador\_mochila.py

```

def calcular_mochila(items, pesoMax):
    matrizDevolucion = []
    valores = [[0] * (pesoMax + 1)
for i in xrange(len(items) + 1)]
for i, (valor, peso, id) in enumerate(items):
    i += 1
    for capacidad in xrange(pesoMax + 1):
        if peso > capacidad:
            valores[i][capacidad] = valores[i - 1][capacidad]
        else:
            primerCandidato = valores[i - 1][capacidad]
            segundoCandidato = valores[i - 1][capacidad - peso] + valor
            valores[i][capacidad] = max(primerCandidato, segundoCandidato)
longItems = len(items)
j = pesoMax
while longItems > 0:
    if valores[longItems][j] != valores[longItems - 1][j]:
        matrizDevolucion.append(items[longItems - 1])
        j -= items[longItems - 1][1]
    longItems -= 1
matrizDevolucion.reverse()
return valores[len(items)][pesoMax], matrizDevolucion

```

## Flujo de Redes

El problema expuesto es una variación del problema de selección de proyectos que se comenta a continuación.

### Introducción teórica: El Problema de Selección de proyectos

En el problema de selección de proyectos, hay  $n$  proyectos y  $m$  areas. Cada proyecto  $p_i$  dá ingresos  $r(p_i)$  y cada experto en el area  $q_j$  cuesta  $c(q_j)$  para la su contratación. Cada proyecto requiere una serie de expertos en determinadas áreas y cada experto puede ser compartido por varios proyectos. El problema es determinar qué proyectos y expertos deben ser seleccionados y contratados respectivamente, de modo que el beneficio se maximice.

Sea  $P$  el conjunto de proyectos no seleccionados y  $Q$  el conjunto de expertos contratados, entonces el problema puede ser formulado como,

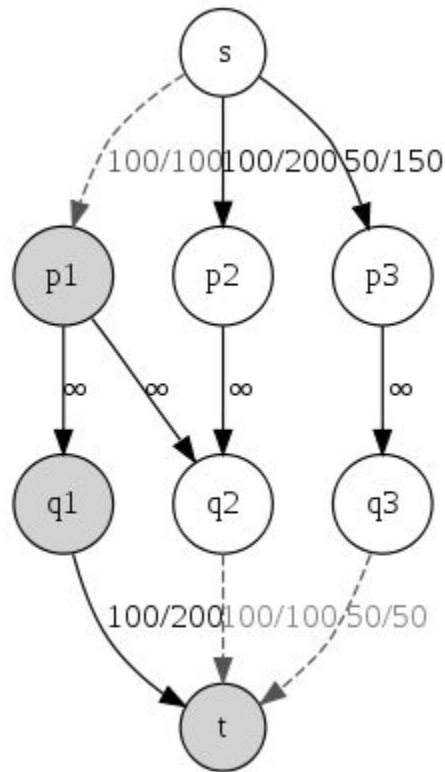
$$\max\{g\} = \sum_i r(p_i) - \sum_{p_i \in P} r(p_i) - \sum_{q_j \in Q} c(q_j).$$

Dado que el primer término no depende de la selección de  $P$  y  $Q$ , este problema de maximización puede ser formulado como un problema de minimización en su lugar, es decir,

$$\min\{g'\} = \sum_{p_i \in P} r(p_i) + \sum_{q_j \in Q} c(q_j).$$

El problema de minimización anterior puede entonces formularse como un problema de corte mínimo construyendo una red, donde la fuente está conectada a los proyectos con capacidad  $r(p_i)$ , y el sumidero está conectado por los expertos con capacidad  $c(q_j)$ . Se añade una arista  $(p_i, q_j)$  con capacidad infinita si el proyecto  $p_i$  requiere experto de area  $q_j$ . El conjunto de corte s-t representa los proyectos y expertos en  $P$  y  $Q$ , respectivamente. Mediante el teorema de corte mínimo de flujo máximo, se puede resolver el problema como un problema de flujo máximo.

La siguiente figura muestra el modelo de red del siguiente problema de selección de proyecto:



	Proyecto $r(p_i)$	Experto $c(q_j)$	
1	100	200	El Proyecto 1 requiere experto en áreas 1 y 2
2	200	100	El Proyecto 2 requiere experto en áreas 2
3	150	50	El Proyecto 3 requiere experto en áreas 3

La capacidad mínima de un corte s-t es de 250 y la suma de los ingresos de cada proyecto es de 450; Por lo tanto el beneficio máximo  $g$  es  $450 - 250 = 200$ , seleccionando los proyectos  $p_2$  y  $p_3$ .

La idea aquí es hacer "fluir" las ganancias del proyecto a través de los "pipes" del experto. Si no podemos llenar el pipe, el retorno de la contratación del experto es menor que su costo, y el algoritmo de corte mínimo lo encontrará más "barato" para cortar la arista del beneficio del proyecto en lugar de la arista del costo de la contratación del experto.

## Complejidad

Cuando las capacidades de las aristas del grafo son enteros, el algoritmo de Ford-Fulkerson tiene un orden de complejidad de  $O(E * f)$ , donde  $E$  es el número de aristas y  $f$  es el flujo máximo en el grafo. Esto se debe a que cada ruta de aumento se puede encontrar en el tiempo  $O(E)$  y aumenta el flujo en una cantidad entera de al menos 1.

Siendo esto así, se tiene que el algoritmo utilizado para resolver el problema de selección de proyectos usa primero ford-fulkerson para encontrar el flujo máximo y el grafo residual, y luego un recorrido (en nuestro caso DFS) para encontrar todos los vertices dentro del primer grupo del corte mínimo.

Esto da como orden total:

$$O(E * f) + O(V + E)$$

Lo cual queda acotado por:

$$O(V + E * f)$$

Luego, llevando las variables  $V$ ,  $E$  y  $f$  al dominio del problema presentado:

$$O((m + n) + (r * f))$$

Siendo:

$m$ : cantidad de proyectos

$n$ : cantidad de áreas

$r$ : cantidad de requerimientos totales

$$r = \sum_{i=1}^m |R_i|$$

## Programa

Para correr el programa solicitado por el enunciado ejecutar

```
python main.py < input
```

Siendo input un archivo con los datos de entrada en el formato solicitado.

# Código

## Modulo Main

```
from MaxFlow import FlowNetwork
INF = 99999
area_count = int(raw_input())
project_count = int(raw_input())

areas = {}
for i in xrange(1, area_count+1):
    cost = int(raw_input())
    areas["req_" + str(i)] = cost

projects_revenues = {}
projects_reqs = {}

for i in xrange(1, project_count+1):
    project_str = raw_input()
    project = project_str.split()
    proj_id = "proy_" + str(i)
    projects_revenues[proj_id] = int(project[0])
    projects_reqs[proj_id] = map(lambda x: "req_" + str(x), project[1:])

g = FlowNetwork()
g.add_vertex('s')
g.add_vertex('t')
[g.add_vertex(v) for v in projects_revenues.keys()]
[g.add_vertex(v) for v in areas.keys()]

for proj in projects_revenues.keys():
    print "arista", "s -> " + proj
    g.add_edge('s', proj, projects_revenues[proj])

for proj in projects_reqs.keys():
    reqs = projects_reqs[proj]
    for area in reqs:
        print "arista", proj + " -> " + area
        g.add_edge(proj, area, INF)

for area in areas.keys():
    print "arista", area + " -> t"
    g.add_edge(area, 't', areas[area])

print "MAX_FLOW", (g.max_flow('s', 't'))
print "MIN_CUT", g.find_min_cut('s', [])
print "POYECTOS REALIZADOS"
for result in g.find_min_cut('s', []):
    if result.sink in projects_revenues:
        print "\t", result.sink + ":", projects_revenues[result.sink]
```

## Modulo MaxFlow

```
class Edge(object):
    def __init__(self, u, v, w):
        self.source = u
        self.sink = v
        self.capacity = w
    def __repr__(self):
        return "%s->%s:%s" % (self.source, self.sink, self.capacity)

class FlowNetwork(object):
    def __init__(self):
        self.adj = {}
        self.flow = {}

    def add_vertex(self, vertex):
        self.adj[vertex] = []

    def get_edges(self, v):
        return self.adj[v]

    def add_edge(self, u, v, w=0):
        if u == v:
            raise ValueError("u == v")
        edge = Edge(u,v,w)
        redge = Edge(v,u,0)
        edge.redge = redge
        redge.redge = edge
        self.adj[u].append(edge)
        self.adj[v].append(redge)
        self.flow[edge] = 0
        self.flow[redge] = 0

    def find_path(self, source, sink, path):
        if source == sink:
            return path
        for edge in self.get_edges(source):
            residual = edge.capacity - self.flow[edge]
            if residual > 0 and edge not in path:
                result = self.find_path( edge.sink, sink, path + [edge])
                if result != None:
                    return result

    def max_flow(self, source, sink):
        path = self.find_path(source, sink, [])
        while path != None:
            residuals = [edge.capacity - self.flow[edge] for edge in path]
            flow = min(residuals)
            for edge in path:
                self.flow[edge] += flow
                self.flow[edge.redge] -= flow
            path = self.find_path(source, sink, [])
        return sum(self.flow[edge] for edge in self.get_edges(source))
```



```

def find_min_cut(self, source, path):
    non_saturated = filter(lambda edge: ((edge.capacity - self.flow[edge]) > 0) and
(edge not in path), self.get_edges(source))
    if (len(non_saturated) == 0):
        return path

    for edge in self.get_edges(source):
        residual = edge.capacity - self.flow[edge]
        if residual > 0 and edge not in path:
            result = self.find_min_cut(edge.sink, path + [edge])
            if result != None:
                return result

```