Análisis y diseño de algoritmos

Sesión 09

Logro de la sesión

Al finalizar la sesión, el estudiante realiza un análisis y desarrollo de algoritmos voraces o golosos (Greedy) utilizando un lenguaje de programación

Agenda

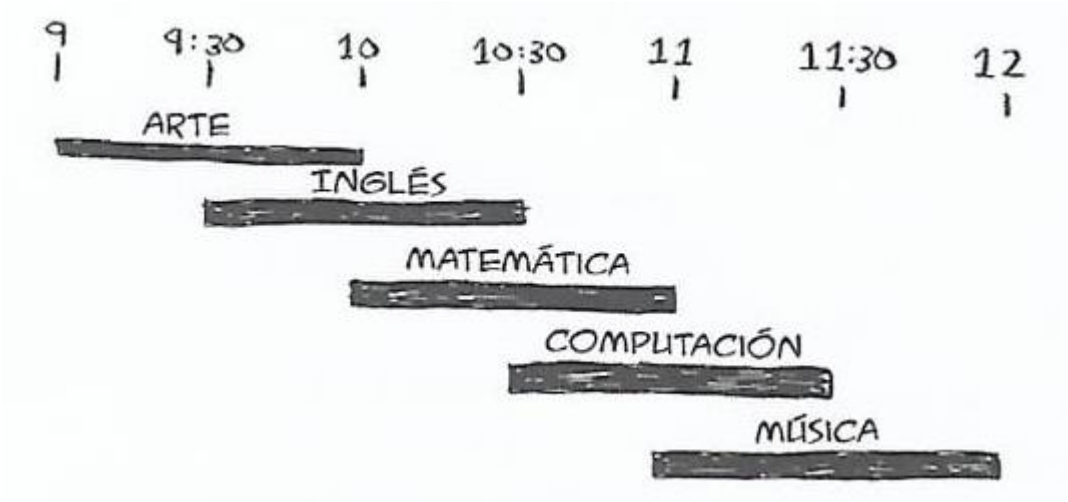
- Algoritmos voraces - Ejemplos
- Algoritmo voraces - Introducción
- Problema de la mochila
- Problema del cambio de moneda
- Problema del conjunto de cobertura
- Problemas de Planificación de Tareas

Algoritmos voraces – El problema de los horarios

CLASE	INICIO	FINAL
ARTE	9AM	10AM
INGLÉS	9:30AM	10:30AM
MATEMÁTICA	10AM	11AM
COMPUTACIÓN	10:30AM	11:30AM
MÚSICA	11AM	12PM

Supón que tienes un aula y quiere utilizarla para impartir la mayor cantidad de clases posible. Te dan una lista de clases

No puedes dar todas las clases en la misma aula porque alguna de ellas coinciden



Si quieres dar la mayor cantidad de clases posibles en el aula ¿Cómo escoges qué conjunto de clases impartir, de forma tal que puedas crear el mayor conjunto posible?

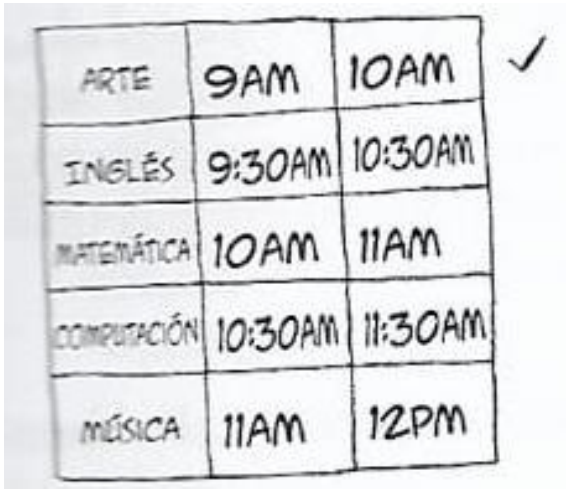
Suena como un problema difícil ¿cierto? Sin embargo, el algoritmo es tan sencillo que te pudiera sorprender. A continuación se explica como funciona

1. Escoge la clase que termina primero. Esa será la clase inicial para impartir en el aula

2. Ahora, tienes que escoger cuál comienza luego de la primera clase. De nuevo escogerás la clase que termina más pronto. Esta será la próxima clase que impartir

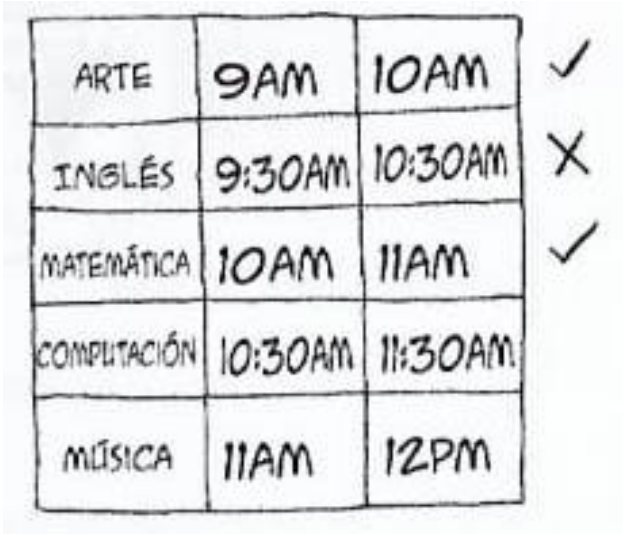
Algoritmos voraces – El problema de los horarios

Mantenemos este proceso y se encontrará la respuesta



La clase de Arte termina antes que ninguna otra, a las 10:00 am, así que la escogerás

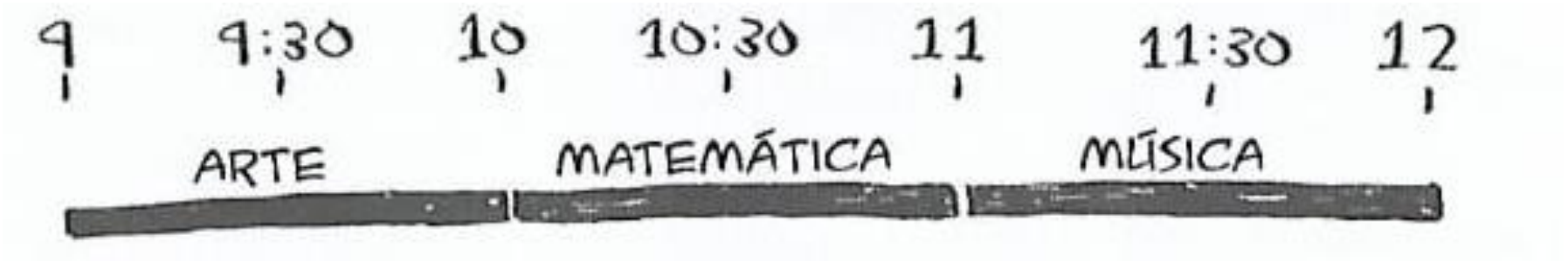
Ahora tienes que escoger la siguiente que comience después de las 10:00 am y termine más rápido



Inglés está descartada porque coincide con Arte, pero Matemáticas se puede escoger. Finalmente Computación coincide con Matemáticas, pero música no coincide con Matemáticas

Algoritmos voraces – El problema de los horarios

Entonces estas son las tres clases que impartirás en el aula



1. Un algoritmo voraz es simple: En cada paso se escoge el movimiento óptimo

En este caso cada vez que escoges una clase, seleccionas aquella que termina primero

2. En cada paso escoges la solución que corresponde al óptimo local, y al final obtienes como solución completa el óptimo global

En este caso este algoritmo encuentra la solución óptima para este problema de planificación.
Sin embargo los algoritmos voraces no siempre funcionan ! Pero son sencillos de escribir!

Algoritmos voraces – Problema del Cambio de monedas

Problema del cambio de monedas

Se dispone de *n* monedas de soles con valores de 1,2,5,10,20 y 50 céntimos de sol, 1 y 2 soles.

Dada una cantidad *X* de soles, devolver dicha cantidad con el menor número posible de monedas

Nuevamente, la estrategia golosa o voraz es muy sencilla:

1. Tomaremos cada vez el billete de mayor denominación, siempre y cuando la suma acumulada no sea mayor que la cantidad a devolver.

Ejemplo: devolver 2.24 soles

Solución: 4 (2 + 0,20 + 0,02 + 0,02)

Y si se dispone de *n* monedas de soles con valores de 1,2,5,10,**12**,20 y 50 céntimos de sol, 1 y 2 soles. ¿Cuál es la solución? ¿Es óptima?

Elementos

Conjunto de candidatos *C*

- La solución se construirá con un subconjunto de estos candidatos

Función de selección

- Selecciona el candidato “local” más idóneo

Función de factibilidad

- Comprueba si un candidato es factible

Función objetivo

- Determina el valor de la solución (función a optimizar)

Función solución

- Determina si el subconjunto de candidatos ha alcanzado una solución

Algoritmos voraces – El problema de la mochila

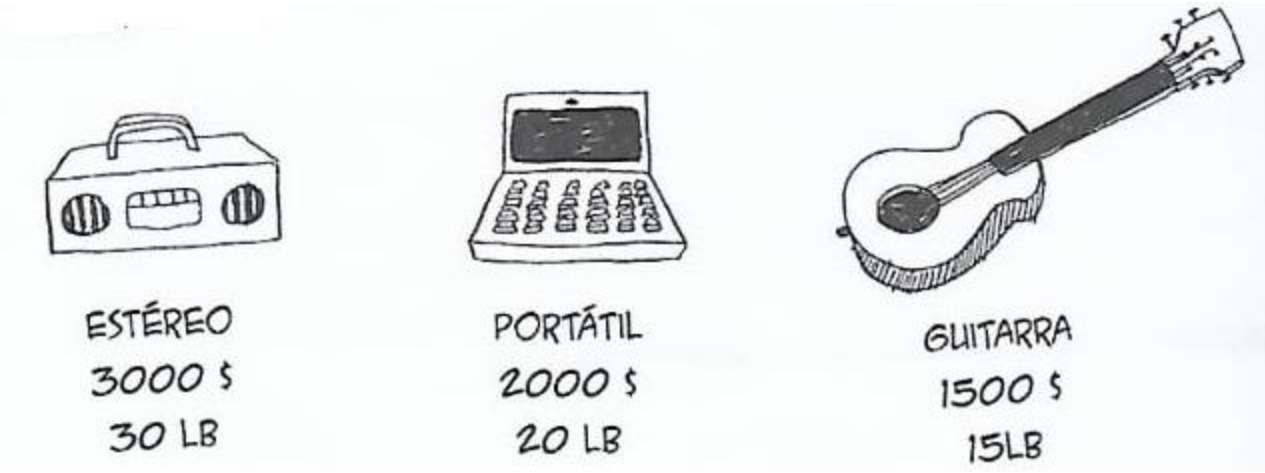
Supón que eres un ladrón ambicioso. Estás en una tienda con una mochila rodeado de objetos que puedes robar. Sólo puedes llevarte lo que cabe en la mochila. Tu mochila soporta hasta 35 libras

Estás intentando maximizar el valor de los objetos que pones en la mochila. ¿Qué algoritmo usarías?. Nuevamente la estrategia golosa o voraz es muy sencilla:

1. Escoge el objeto más valioso que quepa en la mochila

2. Escoge el próximo objeto de mayor valor, que quepa en la mochila y así sucesivamente.

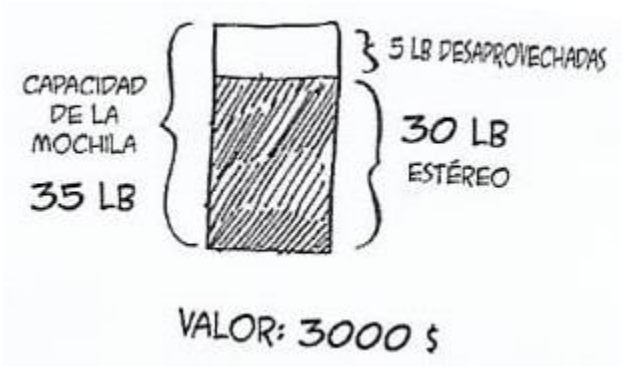
¡Pero esta vez no funciona! Por ejemplo, supón que hay 3 objetos que puedes robar.



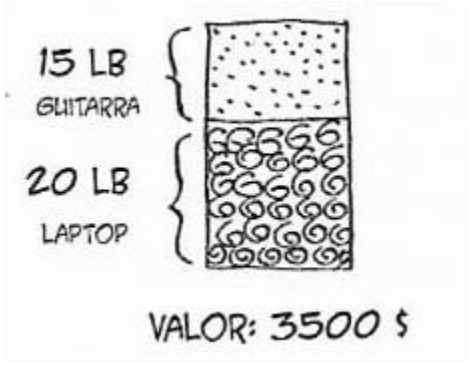
Algoritmos voraces – El problema de la mochila

Mantenemos este proceso y se encontrará la respuesta

Tu mochila puede llevar hasta 35 libras de objetos. EL estéreo es el más caro, así que seleccionas ese. Ahora no tienes espacio disponible



Obtienes 3000 $ de valor, pero si hubieras escogido la laptop y la guitarra hubieras llevado un botín de 3500 $



Claramente la estrategia voraz o golosa no conduce a la solución óptima, pero te acerca bastante.

A veces lo que se necesita es sólo un algoritmo que resuelva el problema lo suficientemente bien. Los algoritmos voraces son por lo general sencillos de escribir y te acercan lo suficiente a una buena solución

Ejercicios

Caso 1

Trabajas para una empresa de muebles y tienes que enviar muebles a todo el país. Necesitas transportar cajas en tu camión. Todas las cajas son de diferentes tamaños, y estas tratando de maximizar el espacio que utilizas en cada camión. ¿Cómo elegirías cajas para maximizar el espacio? Piensa en una estrategia golosa. ¿Eso te dará la solución óptima?

Madrid 3 10
Paris 2 8
Venecia 2 6
Londres 2 4
Berlin 1 2

Caso 2

Vas a Europa y tienes 7 días para ver todo lo que puedas. Asigna valores de puntos a cada elemento (cuánto deseas verlo) y estima cuánto tiempo toma. ¿Cómo puedes maximizar el total de puntos (ver todas las cosas que realmente quieres ver) durante tu estadía? Idea una estrategia golosa o voraz. ¿Eso te dará la solución óptima?

Algoritmos voraces

- Los algoritmos voraces se suelen aplicar a problemas de optimización
  - Maximizar o minimizar una función objetivo
- Suelen ser rápidos y fáciles de implementar
- Exploran soluciones “locales”
- No siempre garantizan la solución óptima.

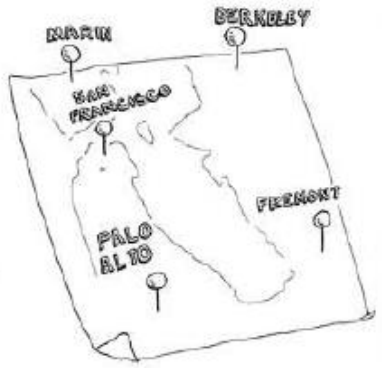
Algoritmos voraces

- Son algoritmos que siguen una heurística mediante la cual toman **"decisiones óptimas locales"** en cada paso, de manera muy eficiente, con la esperanza de poder encontrar un óptimo global tras una serie de pasos

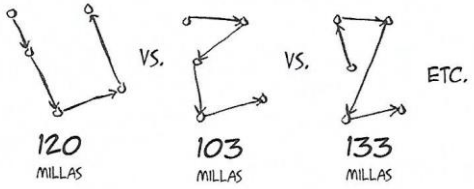
- Se aplican, sobre todo, a problemas duros, desde un punto de vista computacional
- Ejemplo: problema del viajante de comercio
- Heurística: "escoge la ciudad más próxima no visitada aún"

El vendedor tiene que visitar 5 ciudades

Este vendedor, quien se llamará Opus, quiere ir a todas las ciudades recorriendo la menor distancia posible.



Aquí hay una forma de hacerlo: buscar cada posible orden en el que pudiera viajar a las ciudades y seleccionar la menor.



CIUDADES	OPERACIONES
6	720
7	5040
8	40 320
...	...
15	1 307 674 368 000
...	...
30	2 652 52 0 859 812 191 058 636 308 480 000 000

En general, para n elementos, se necesitará n! (n factorial) operaciones para calcular el resultado. Entonces este es el tiempo O (n!).

- Para ciertos problemas se puede demostrar que algunas estrategias voraces sí que logran hallar un óptimo global de manera eficiente

Algoritmos Voraces – Ventajas y desventajas

Ventajas

- Son fáciles de implementar
- Producen soluciones eficientes
- A veces encuentran la solución óptima

Desventajas

- No todos los problemas de optimización son resolubles con algoritmos voraces
- La búsqueda de un óptimo local no implica encontrar un óptimo global
- Dificultad de encontrar la función de selección que garantice la elección óptima

Esquema general

- La técnica voraz funciona por pasos:
  - Partimos de una solución vacía y de un conjunto de candidatos a formar parte de la solución.
  - En cada paso se intenta añadir el mejor de los candidatos restantes a la solución parcial
  - Una vez tomada la decisión, no se puede deshacer
  - Si la solución ampliada es válida ⇒ candidato incorporado
  - Si la solución ampliada no es válida ⇒ candidato desechado

- El algoritmo acabará cuando el conjunto de elementos seleccionados constituya una solución o cuando no queden elementos sin considerar

Esquema general en Pseudocódigo

```
función voraz(C)      //C es el conjunto de candidatos//

    S ← ∅
    mientras C ≠ ∅ y no solución(S) hacer
        x ← seleccionar(C)
        C ← C\{x}
        si factible(S ∪ {x}) entonces
            S ← S ∪ {x}

    si solución(S) entonces
        devolver S      //S es una solución//
    si no
        devolver ∅      //No hay soluciones//
```

Esquema general en Pseudocódigo

```
función voraz ( ↓datos : conjunto) ⟶S : conjunto

  Variables
    candidatos : conjunto
    S : conjunto
    x : elemento
  Acciones
    S ⟵ ∅  // la solución se irá construyendo en el conjunto S
    candidatos ⟵ datos
    Mientras candidatos ≠ ∅ AND  NOT(solución(S)) hacer
      x ⟵ SeleccionVoraz( candidatos )
      candidatos ⟵ candidatos − {x}
      Si (factible(S ∪ {x}) entonces S⟵ S ∪ {x}
      Fin Si
    Fin Mientras

    Si solución(S) entonces
      regresar ( S )
    sino
        regresar ( ∅ )  // No encontró la solución

Fin voraz
```

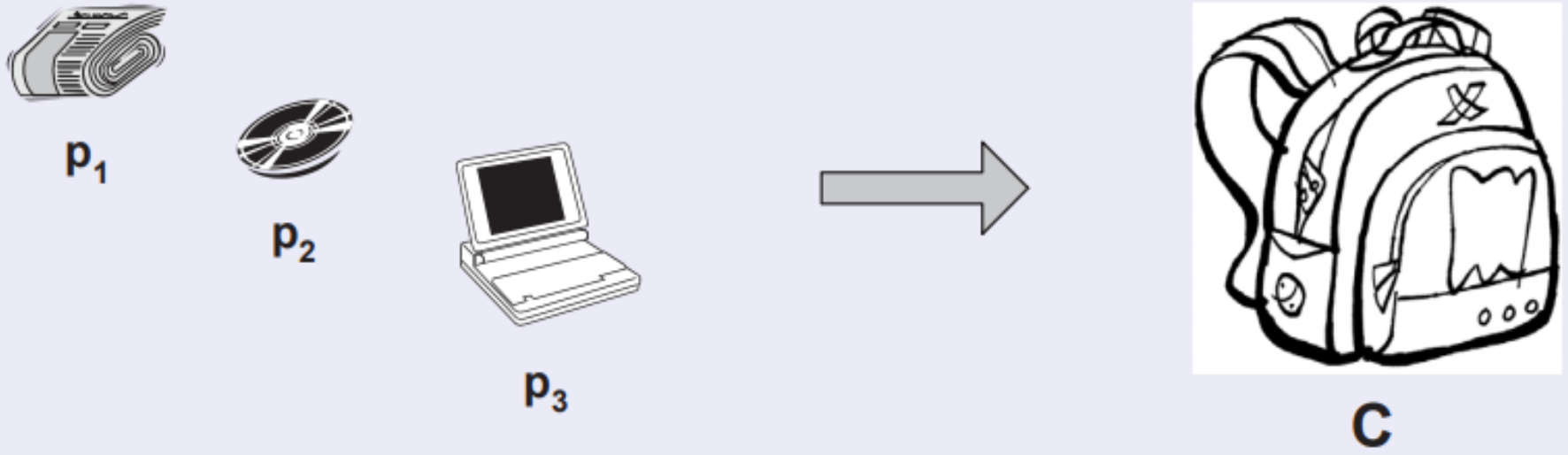
Esquema general en Phyton

```
def Algoritmo_Voraz(Conjunto_Entrada):
    Conjunto = Conjunto_Entrada
    Solucion = []
    encontrada = False
    while not EsVacio(Conjunto) and not encontrada:
        x = Seleccionar_Mejor_Candidato(Conjunto)
        Conjunto = Conjunto - [x]
        if EsFactible(Solucion + [x]):
            Solucion = Solucion + [x]
            if EsSolucion(Solucion):
                encontrada = True
    return Solucion
```

Problema de la mochila - Versión 1

Problema de la mochila - Versión 1

Se tiene un conjunto de n objetos, cada uno con un peso p_i, y una mochila con capacidad C.



- Maximizar el **número de objetos** que se pueden introducir en la mochila sin sobrepasar la capacidad C

Problema de la mochila - Versión 1

Problema de la mochila - Versión 1 - Formulación matemática

Dado un conjunto de n objetos, cada uno con un peso p_i, $i = 1, \ldots, n$, y una mochila con capacidad C. Maximizar el **número de objetos** que se pueden introducir en la mochila sin sobrepasar la capacidad C:

$$\underset{x}{\textit{maximizar}} \quad \sum_{i=1}^{n} x_i$$

$$\text{sujeto a} \quad x_i \in \{0, 1\} \qquad i = 1, \ldots, n$$

$$\sum_{i=1}^{n} x_i p_i \leq C$$

- Las variables x_i determinan si se introduce el objeto i en la mochila
- Es un problema de programación lineal entera

Problema de la mochila - Versión 1

- Ejemplo: $C = 15$, $p = (9, 6, 5)$
- ¿Función de selección?
  - Peso decreciente
    - Solución: $2\ (9 + 6)$
  - Peso creciente
    - Solución: $2\ (5 + 6)$

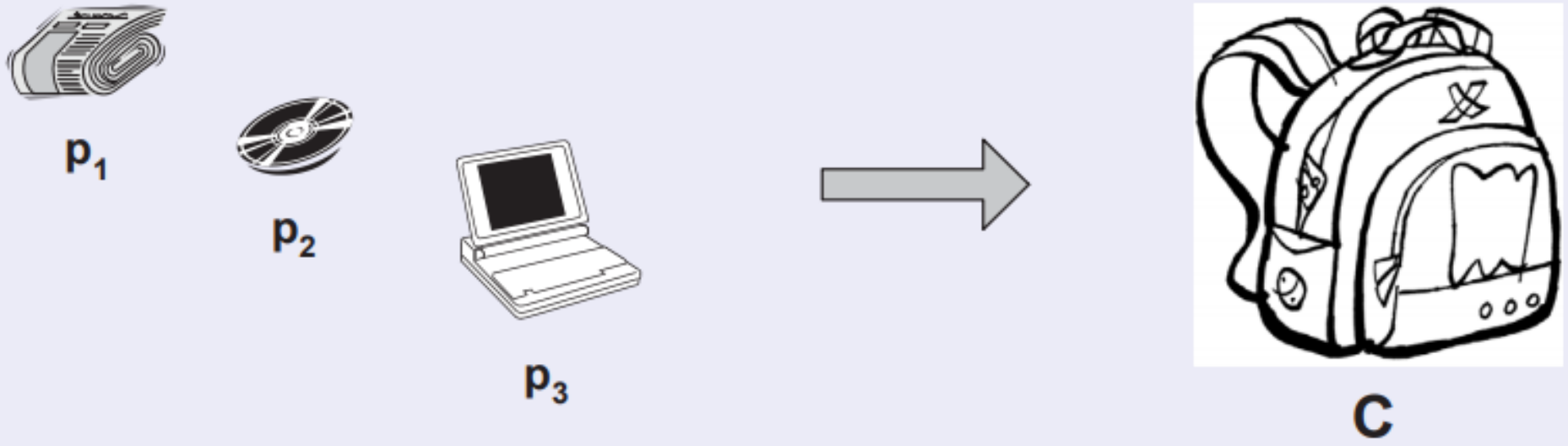
- Ejemplo: $C = 15$, $p = (9, 5, 6, 4)$
- ¿Función de selección?
  - Peso decreciente
    - Solución: $2\ (9 + 6)$
  - Peso creciente
    - Solución: $3\ (4 + 5 + 6)$

- La estrategia voraz escogiendo candidatos en orden creciente de peso es mejor (para este problema es óptima)

Problema de la mochila - Versión 2

Problema de la mochila - Versión 2

Se tiene un conjunto de n objetos, cada uno con un peso p_i, y una mochila con capacidad C.



- Maximizar el **peso de los objetos** que se introducen en la mochila sin sobrepasar la capacidad C

Problema de la mochila - Versión 2

Problema de la mochila - Versión 2 - Formulación matemática

Dado un conjunto de n objetos, cada uno con un peso p_i, $i = 1, \ldots, n$, y una mochila con capacidad C. Maximizar el **peso de los objetos** que se introducen en la mochila sin sobrepasar la capacidad C:

$$\begin{aligned} \underset{x}{\text{maximizar}} \quad & \sum_{i=1}^{n} x_i p_i \\ \text{sujeto a} \quad & x_i \in \{0,1\} \qquad i = 1, \ldots, n \\ & \sum_{i=1}^{n} x_i p_i \leq C \end{aligned}$$

- Las variables x_i determinan si se introduce el objeto i en la mochila
- Es un problema de programación lineal entera

Problema de la mochila - Versión 2

- Ejemplo: $C = 15$, $p = (9, 6, 5)$
- ¿Función de selección?
  - Peso decreciente
    - Solución: 15 $(9 + 6)$
  - Peso creciente
    - Solución: 11 $(5 + 6)$

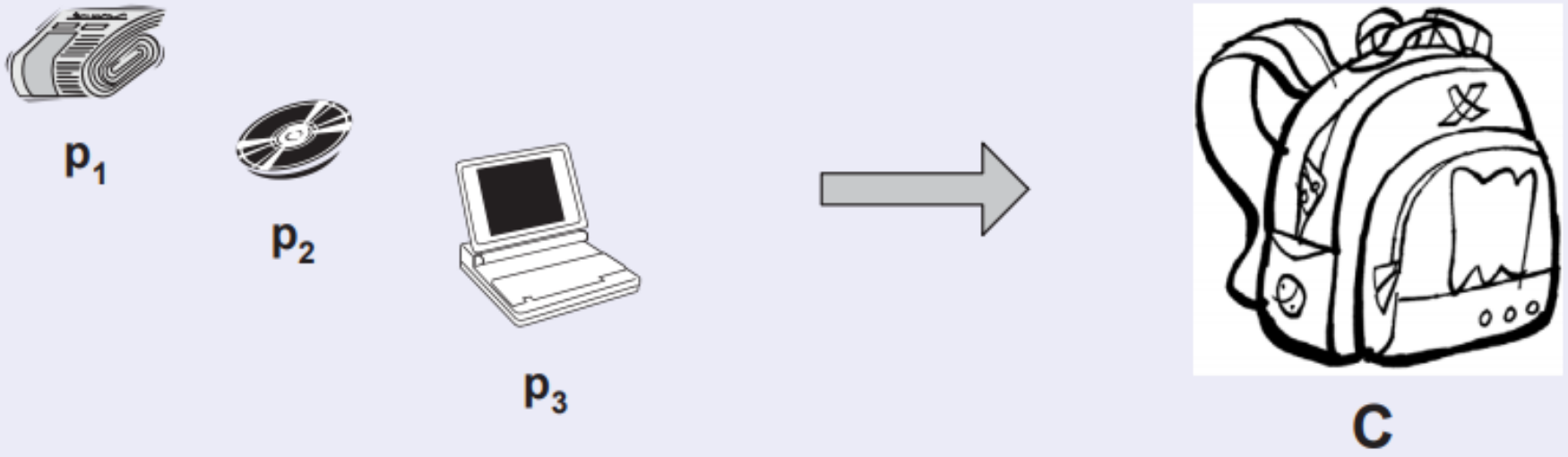
- Ejemplo: $C = 15$, $p = (10, 7, 6)$
- ¿Función de selección?
  - Peso decreciente
    - Solución: 10 (10)
  - Peso creciente
    - Solución: 13 $(6 + 7)$

- Ninguna de las estrategias de selección es óptima, ni mejor que la otra

Problema de la mochila - Versión 3

Problema de la mochila - Versión 3

Se tiene un conjunto de n objetos, cada uno con un peso p_i y un valor v_i, y una mochila con capacidad C. Los objetos pueden partirse en fracciones más pequeñas.



- Maximizar la **suma de los valores asociados a los objetos** que se introducen en la mochila, sin sobrepasar la capacidad C

Problema de la mochila - Versión 3

Problema de la mochila - Versión 3 - Formulación matemática

Dado un conjunto de n objetos, cada uno con un peso p_i y un valor v_i, $i = 1, \ldots, n$, y una mochila con capacidad C. Maximizar la suma de los valores asociados a los objetos que se introducen en la mochila, sin sobrepasar la capacidad C, sabiendo que los objetos pueden partirse en fracciones más pequeñas:

$$\underset{x}{\text{maximizar}} \quad \sum_{i=1}^{n} x_i v_i$$

$$\text{sujeto a} \quad 0 \leq x_i \leq 1 \qquad i = 1, \ldots, n$$

$$\sum_{i=1}^{n} x_i p_i \leq C$$

- Las variables x_i determinan la fracción del objeto i que se introduce en la mochila
- Es un problema de programación lineal

Problema de la mochila - Versión 3

- Conjunto de candidatos:
  - Todos los objetos
- Función de factibilidad:
  - $\sum_{i=1}^{n} x_i p_i \leq C$
- Función objetivo:
  - Maximizar $\sum_{i=1}^{n} x_i v_i$
- Funciones de selección posibles:
  - Seleccionar el objeto con mayor valor
  - Seleccionar el objeto con menor peso restante
  - Seleccionar el objeto cuyo valor por unidad de peso sea el mayor posible
- Función solución:
  - Cualquier conjunto de elementos es válido si no se ha sobrepasado C

Problema de la mochila - Versión 3

- Ejemplo: $C = 100$, $n = 5$, con:

i	1	2	3	4	5
p_i	10	20	30	40	50
v_i	20	30	66	40	60
v_i/p_i	2,0	1,5	2,2	1,0	1,2

seleccionar x_i	x_i					valor total
maximizar v_i	0	0	1	0,5	1	146
minimizar p_i	1	1	1	1	0	156
maximizar v_i/p_i	1	1	1	0	0,8	164

- Función de selección óptima es: maximizar v_i/p_i

Pseudocódigo del problema de la Mochila - Versión 3

Llenar una mochila que soporta un peso máximo W. Se tienen n objetos, cada uno de estos objetos tiene un peso w_i y un valor v_i (i = 1, 2, …, n).

El problema consiste en maximizar

$$\sum_{i=1}^{n} x_i v_i$$

Con la restricción:

$$\sum_{i=1}^{n} x_i w_i \leq W$$

Los objetos se pueden romper en trozos más pequeños al multiplicarlos por una fracción 0 ≤ xi ≤ 1, de tal manera que el peso del objeto i dentro de la mochila sea xiwi . De esta manera cada objeto i contribuye en xiwi al peso total de la mochila y en xivi al valor de la carga

```
Función mochila ( ↓pesos ∈ Real Vector[1...n], ↓valores ∈ Real Vector[1...n], ↓W∈ Entero,
                  ↓ ↑ x ∈ Real Vector[1...n])
 Constantes
    ø
 Variables
  i, suma ∈ Entero
 Acciones
  suma ← 0
  Para i←1 hasta n
   x[i] = 0
  Fin Para

  Mientras suma < W
    i ← SeleccionVoraz( x, pesos, valores )
    Si suma + pesos[i] ≤ W entonces
       x[i]← 1
       suma← suma + pesos[i]
    sino
       x[i]← (W-suma)/ pesos[i]
       suma ← W
    Fin Si
  Fin Mientras
  regresar ( x )
Fin Función mochila
```

Implementación de problema de la mochila - Versión 3 en C++

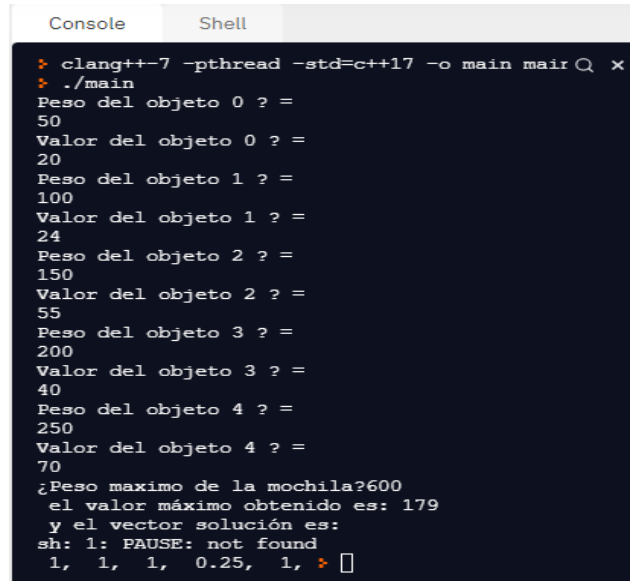
```
#include <cstdlib>
#include <iostream>
using namespace std;
struct objeto{
double peso;
double valor;
double valorPeso;
bool tomado;};
objeto objetos[5];
double solucion[]={0,0,0,0,0};

int SeleccionVoraz( ){
//estrategia del mayor valor/peso
double mayor = -1;
int indice=-1, j=0;
for(int j=0; j<5; j++){
if(!objetos[j].tomado && (mayor < objetos[j].valorPeso)){
mayor = objetos[j].valorPeso;
indice = j;};
};
return indice;
};
```

```
double valorTotal(){
double valor=0;
for( int i=0; i<5; i++)
valor = valor + objetos[i].valor * solucion[i];
return valor;
};
```

Implementación de problema de la mochila - Versión 3 en C++

```
int main(int argc, char *argv[]){
double PesoMax, suma=0;
int i;
for( i=0; i<5; i++){
cout<< "Peso del objeto "<< i << " ? = \n";
cin >> objetos[i].peso;
cout<< "Valor del objeto "<< i << " ? = \n";
cin >> objetos[i].valor;
objetos[i].valorPeso =
objetos[i].valor/objetos[i].peso;
objetos[i].tomado = false;
};
cout << "¿Peso maximo de la mochila?"; cin >>
PesoMax;
while(suma < PesoMax){
i= SeleccionVoraz();
if((suma + objetos[i].peso) < PesoMax ){
solucion[i] = 1;
suma = suma + objetos[i].peso;
objetos[i].tomado = true;
}
```

```
else{
solucion[i] = (PesoMax-suma)/
objetos[i].peso;
suma = PesoMax; // corregir error al
multiplicar por fracción
};
};
cout << " el valor máximo obtenido es: "
<< valorTotal() << endl;
cout << " y el vector solución es: \n";
for(i=0; i<5; i++)
cout<< " " << solucion[i] << ", ";
system("PAUSE");
return EXIT_SUCCESS;
}
```



Implementación de problema de la mochila - Versión 3 en Python

```
def Mochila(peso, beneficio, M):
    n=len(peso)
    solucion = [0.0 for i in range(n)]
    peso_actual = 0.0
    while peso_actual < M:
        i = mejor_objeto_restante(peso, beneficio)
        if peso[i] + peso_actual <= M:
            solucion[i] = 1
            peso_actual = peso_actual + peso[i]
        else:
            solucion[i] = (M - peso_actual) / peso[i]
            peso_actual = M
    return solucion
```

Ejercicios

Caso 1

¿Cuál sería la mejor función de selección voraz en el caso en el que todos los objetos tuvieran el mismo valor?

Caso 2

¿Cuál sería la mejor función de selección voraz en el caso en el que todos los objetos tuvieran el mismo peso?

Algoritmos voraces – Problema del Cambio de monedas

Problema del cambio de monedas

Se dispone de *n* monedas de soles con valores de 1,2,5,10,20 y 50 centimos de sol, 1 y 2 soles.

Dada una cantidad *X* de soles, devolver dicha cantidad con el menor número posible de monedas

Nuevamente, la estrategia golosa o voraz es muy sencilla:

1. Tomaremos cada vez el billete de mayor denominación, siempre y cuando la suma acumulada no sea mayor que la cantidad a devolver.

Ejemplo: devolver 2.24 soles

Solución: 4 (2 + 0,20 + 0,02 + 0,02)

Algoritmos voraces – Problema del Cambio de monedas

- Conjunto de candidatos:
  - Todos los tipos de monedas
- Función solución:
  - Conjunto de monedas que suman X
- Función de factibilidad:
  - Si $\sum_{i=1}^{8} v_i n_i > X$, el conjunto obtenido no podrá ser solución
    - $n_i =$ número de monedas de tipo i
    - $v_i =$ valor de una moneda de tipo i
- Función objetivo:
  - Minimizar la cardinalidad de las soluciones posibles
- Función de selección:
  - Moneda de valor más alto posible, que no supere el valor que queda por devolver

Algoritmos voraces – Problema del Cambio de monedas

- ¿Es óptima la función de selección propuesta?

Problema del cambio de monedas

Se dispone de *n* monedas de soles con valores de 1,2,5,10,**12**,20 y 50 céntimos de sol, 1 y 2 soles.

- Dada una cantidad *X* de soles, devolver dicha cantidad con el menor número posible de monedas

Ejemplo: devolver 2.24 soles

- Solución voraz: 4 (2 + 0,20 + 0,02 + 0,02)
- Solución óptima: 3 (2 + 0,12 + 0,12)

Pseudocódigo del problema del cambio de monedas

Dado un monto n y un conjunto de denominaciones de billetes, se desea elegir la mínima cantidad de billetes cuya suma sea n.

```
función cambio ( ↓monto ∈ Entero, ↑ numBilletes ∈ Vector[1...5] de Entero)
  Constantes
    denominaciones ←{ 100, 20, 10, 5, 1 }
  Variables
    suma, iDen ∈ Entero
    numBilletes←{ 0, 0, 0, 0, 0 }
  Acciones
    suma ← 0
    Mientras suma < monto
      iDen ← SeleccionVoraz( monto – suma )
      Si iDen > 5 entonces
          Mensaje "No se encontró solución. Cambio incompleto"
          regresar ( numBilletes )
      Fin Si
      numBilletes[ iDen ] ← numBilletes[ iDen ] + 1
      suma ← suma + denominaciones[ iDen ]
    Fin Mientras
    regresar ( numBilletes )
Fin Función
```

Para construir la solución con un algoritmo voraz tomaremos cada vez el billete de mayor denominación, siempre y cuando la suma acumulada no sea mayor a n. El algoritmo falla cuando se presenta el caso en el que la suma de los billetes no puede igualar a n, en este caso el algoritmo ávido no encuentra una solución

```
Funcion SeleccionVoraz (↓resto ∈ Entero )⟶ ∈ Entero
  Constantes
    denominaciones ←{ 100, 20, 10, 5, 1}
  Variables
    i ∈ Entero
  Acciones
    i←1
    Mientras ( i ≤ 5 ) AND ( denominaciones[i] > resto ) Hacer
        i← i + 1
    Fin Mientras
    regresar i
Fin SeleccionVoraz
```

Nótese que en este caso en particular no existe restricción en cuanto al número de billetes que se pueden tomar, es decir, siempre es posible elegir k billetes de una determinada denominación. Por lo tanto no es necesario tomar en cuenta si un billete ya se consideró o no.

Implementación del problema del cambio de monedas en Python

```
monedas=[500,200,100,50,25,5,1]
def Devolver_Cambio(cantidad, monedas):
    n = len(monedas)
    cambio = [0 for i in range(n)]
    for i in range(n):
        while monedas[i] < cantidad:
            cantidad = cantidad - monedas[i]
            cambio[i] = cambio[i] + 1
    return cambio
```

Este algoritmo es de complejidad lineal respecto a n: número de monedas del sistema monetario.

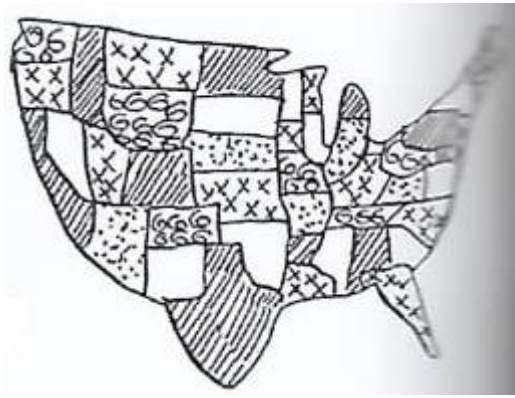
¿Se nos ocurre alguna mejora?

El problema del conjunto de cobertura

Supón que estas creando un programa de radio. Quieres llegar a oyentes en 50 estados de EEUU. Tienes que decidir en qué estaciones de radio pasarás tu programa para llegar a todos los oyentes. Cuesta dinero poner tu programa en cada estación, así que intentarás minimizar el número de estaciones

ESTACIÓN DE RADIO	DISPONIBLE EN
KUNO	ID, NV, UT
KDOS	WA, ID, MT
KTRES	OR, NV, CA
KCUATRO	NV, UT
KCINCO	CA, AZ

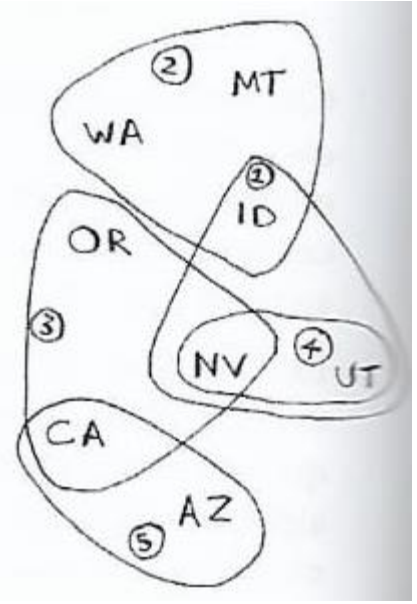
... ETC.



Cada estación cubre una región y hay cierto solapamiento. ¿Cómo encuentras el menor conjunto de estaciones con las cuales cubres los 50 estados. Parece fácil, pero no lo es. Aquí tienes cómo se hace:

1. Lista cada posible subconjunto de estaciones. Esto se conoce como conjunto potencia. Hay 2^n posibles subconjuntos

2. De estos, escoge el conjunto con la menor cantidad de estaciones que cubran los 50 estados.



Algoritmos de aproximación - El problema del conjunto de cobertura

El problema es que toma un largo tiempo calcular cada posible subconjunto de estaciones. Necesitas $O(2^n)$ porque tienes 2^n subconjuntos.

Es posible realizarlo si tienes un número pequeño de 5 a 10 estaciones. Pero que pasaría si tenemos muchos más elementos. Supón que puedes calcular 10 subconjuntos por segundo. !No existe un algoritmo que lo resuelva lo suficientemente rápido! ¿Qué hacemos?

NÚMERO DE ESTACIONES	TIEMPO QUE SE TOMA
5	3.2S
10	102.4S
32	13.6 AÑOS
100	$4*10^{21}$ AÑOS

¡Los algoritmos golosos o voraces llegan al rescate! Aquí tenemos uno que se acerca bastante

1. Escoge la estación que cubre la mayor cantidad de estados que no hayan sido cubiertos aún. Esta bien si dicha estación cubre algunos estados que fueron cubiertos anteriormente

2. Repite hasta que todos los estados se hayan cubierto.

Esto se conoce como un Algoritmo de aproximación. Cuando calcular la solución exacta implica demasiado tiempo, un algoritmo de aproximación funcionará. Estos algoritmos son evaluados según:

- *Cuán rápido resultan*
- *Cuán cerca se encuentran de la solución óptima*

Algoritmos de aproximación - El problema del conjunto de cobertura

Los algoritmos golosos son una buena elección porque no sólo resultan simples de entender sino que esa simplicidad usualmente significa que también son rápidos. En este caso, el algoritmo goloso corre en $O(n^2)$ donde n es el número de estaciones de radio

Supón que puedes calcular 10 subconjuntos por segundo.

NÚMERO DE ESTACIONES	$O(2^n)$ ALGORITMO EXACTO	$O(N^2)$ ALGORITMO GOLOSO
5	3.2S	2.5S
10	102.4S	10S
32	13.6 AÑOS	102.4S
100	$4*10^{21}$ AÑOS	16.67MIN

Ejercicio: Implementar el Código para este algoritmo

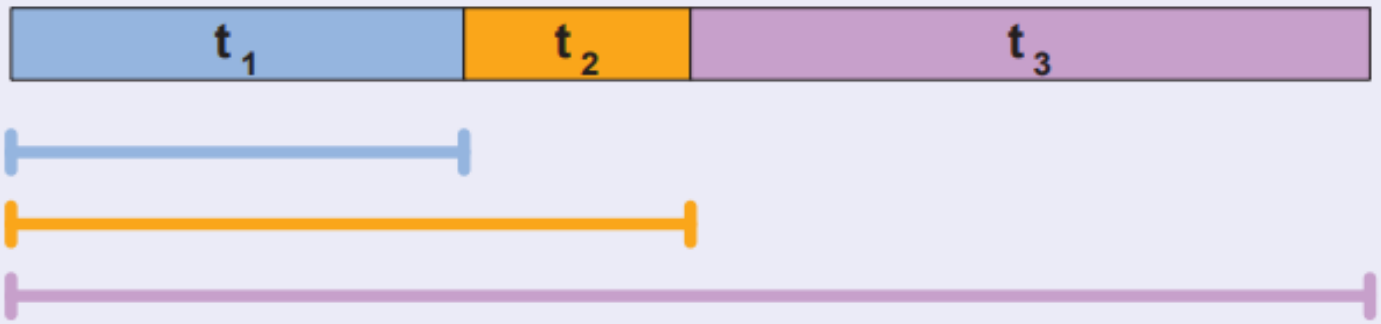
Algoritmos voraces – Problemas de planificación de tareas

- Minimización del tiempo en el sistema

Minimización del tiempo en el sistema

Considérese un servidor que tiene que dar servicio a n clientes, donde t_i, con $i = 1, \ldots, n$, es el tiempo requerido por el cliente i. Suponiendo que todos los clientes llegan al mismo tiempo al servidor pero solo uno puede usarlo, minimizar el tiempo T en el sistema para los n clientes:

$$T = \sum_{i=1}^{n} (\text{tiempo en el sistema para el cliente } i)$$



Algoritmos voraces – Problemas de planificación de tareas

- Ejemplo
  - Supongamos que tenemos 3 clientes con $t_1 = 5$, $t_2 = 10$, y $t_3 = 3$, hay varias posibilidades según el orden en el que sean tratados en el servidor

Orden	T	
123:	$5 + (5 + 10) + (5 + 10 + 3) = 38$	
132:	$5 + (5 + 3) + (5 + 3 + 10) = 31$	
213:	$10 + (10 + 5) + (10 + 5 + 3) = 43$	$\leftarrow$ peor planificación
231:	$10 + (10 + 3) + (10 + 3 + 5) = 41$	
312:	$3 + (3 + 5) + (3 + 5 + 10) = 29$	$\leftarrow$ mejor planificación
321:	$3 + (3 + 10) + (3 + 10 + 5) = 34$	

Algoritmos voraces – Problemas de planificación de tareas

- Estrategia Voraz
  - Dar servicio en orden creciente de tiempo t_i
  - Es una estrategia óptima:
    - Sea $P = \langle p_1, p_2, \ldots, p_n \rangle$ una permutación de los clientes (de enteros de 1 a n)
    - Sea $s_i = t_{p_i}$, entonces

$$\begin{aligned} T(P) &= s_1 + (s_1 + s_2) + (s_1 + s_2 + s_3) + \ldots \\ &= n s_1 + (n-1) s_2 + (n-2) s_3 + \ldots \\ &= \sum_{k=1}^{n} (n - k + 1) s_k \end{aligned}$$

    - Haciendo s_1 lo menor posible, luego s_2, etc. conseguimos minimizar T

Algoritmos voraces – Problemas de planificación de tareas

- Secuencia de tareas con plazo

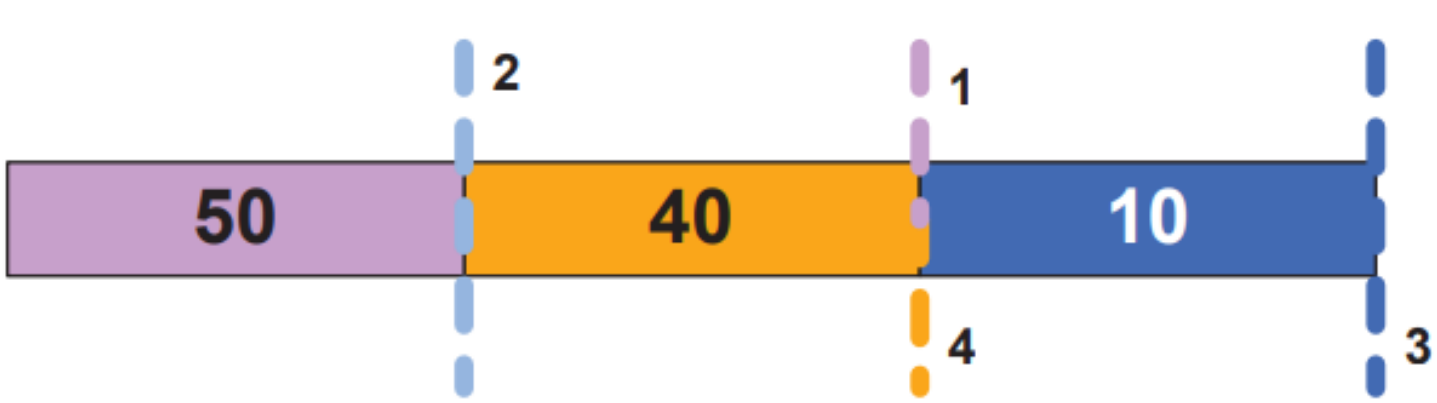
Secuencia de tareas con plazo

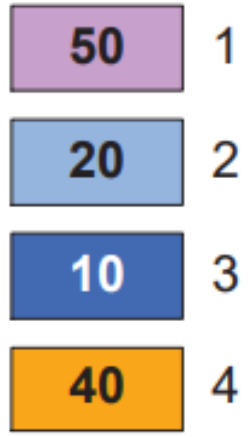
- Se van a ejecutar un subconjunto de n tareas que requieren una unidad de tiempo de ejecución
- Las tareas solo se ejecutan una vez
- En cualquier instante $T = 1, 2, \ldots$ se puede ejecutar una única tarea i, que aportará un beneficio g_i
- Dar una planificación de tareas que **maximice el beneficio**, sabiendo que la tarea i debe terminar de ejecutarse no más tarde que d_i (restricción de plazos)

Algoritmos voraces – Problemas de planificación de tareas

- Ejemplo

i	1	2	3	4
g_i	50	20	10	40
d_i	2	1	3	2





Estrategia Voraz – Problemas de planificación de tareas

- Conjuntos factibles: todos los que tienen 3 elementos o menos, excepto el $\{1, 2, 4\}$ (ya que la tercera tarea siempre ha de ser la 3)

conjuntos factibles	orden de proceso	beneficio
$\{1, 2, 3\}$	$2 - 1 - 3$	80
$\{1, 2\}$	$2 - 1$	70
$\{1, 3, 4\}$	$\boxed{1 - 4 - 3}$ y $4 - 1 - 3$	$\boxed{100}$
$\{1, 3\}$	$1 - 3$ y $3 - 1$	60
$\{1, 4\}$	$1 - 4$ y $4 - 1$	90
$\{1\}$	1	50
$\{2, 3, 4\}$	$2 - 4 - 3$	80
:	:	:

- **Estrategia óptima**: escoger la tarea con mayor beneficio g_i que no se haya elegido previamente, siempre que el resto de tareas en el subconjunto permanezcan factibles
  - Una vez escogida la tarea 1 como la óptima para la 1ª etapa, se obtiene un nuevo problema idéntico pero más pequeño (una tarea menos)

Algoritmos voraces – Problemas de planificación de tareas

Supongamos que disponemos de n trabajadores y n tareas. Sea ***cij > 0*** *el costo de asignarle el trabajo j al trabajador i (puede ser dinero o tiempo). Una asignación de tareas puede expresarse como una asignación de los valores 0 ó 1 a las variables aij, donde aij = 0 significa que al trabajador i no le han asignado la tarea j y aij= 1 indica que sí.*

Una asignación válida es aquella en la que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador.

Dada una asignación válida, definimos el costo total de dicha asignación como:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} a_i c_{ij}$$

Que es la expresión a minimizar. Para lograrlo podemos seguir una de las siguientes estrategias:

1. Asignar a cada trabajador la mejor tarea posible (por ejemplo, la que hace por menos dinero).

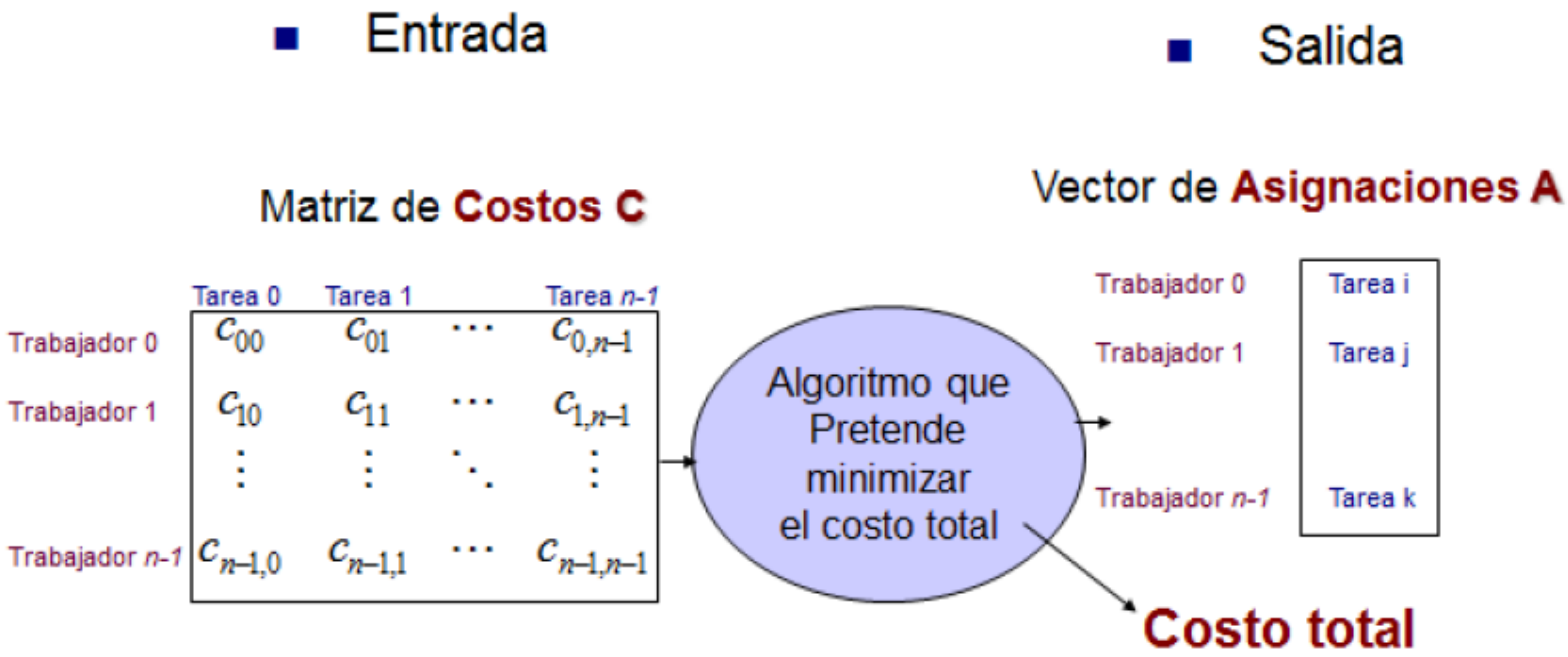
2. Asignar cada tarea al mejor trabajador disponible (por ejemplo, al que cobra menos).

Algoritmos voraces – Problemas de planificación de tareas

El costo de que el trabajador i realice la tarea j se indica en en una matriz de costos como mostrada a continuación, las tareas y los trabajadores se numeran del 0 a n-1, como se enumeran los índices de los arreglos en los lenguajes de programación

	Tarea 0	Tarea 1	⋯	Tarea *n-1*
Trabajador 0	c_{00}	c_{01}	⋯	$c_{0,n-1}$
Trabajador 1	c_{10}	c_{11}	⋯	$c_{1,n-1}$
⋮	⋮	⋮	⋱	⋮
Trabajador *n-1*	$c_{n-1,0}$	$c_{n-1,1}$	⋯	$c_{n-1,n-1}$

Suponiendo que conocemos n, el número de trabajadores y de tareas, el algoritmo para resolver el problema de la asignación de tareas tiene como entrada la matriz de costos, minimiza el costo total, y da como resultado un vector de asignaciones, tal como se ilustra



Este problema se aplica en la vida real a la asignación de máquinas-trabajadores, trabajadores-tareas, inversiones-ingresos, etc.

Algoritmos voraces – Problemas de planificación de tareas

¿Cómo asigna un algoritmo voraz la mejor tarea posible a cada trabajador? El algoritmo para resolver este problema se presenta a continuación. Se considera lo siguiente:

1. C es la matriz de costos, en este caso es de 5x5

2. A es el vector de asignación (su tamaño es el número de trabajadores) le llamaremos tareas

Primero se asigna la mejor tarea mediante el algoritmo de selección voraz AsignaTarea, la cual recibe como parámetro el índice del trabajador al que se le asignará la tarea y regresa el número de tarea que eligió. Posteriormente se despliega el costo mínimo encontrado en base a la tarea asignada a cada trabajador y su costo correspondiente.

```
Función tareasVoraz ( ↓Costos ∈ Real matriz[1...N, 1...N], ↑ costo ∈ Real )
    Constantes
        N=5
    Variables
      tareas ∈ Entero [1...N]
      trabajador, costoTotal ∈ Entero
    Acciones

     Para trabajador←1 hasta n
       tareas[trabajador] ← AsignaTarea(trabajador)
     Fin Para

     costoTotal ← calculaCosto()

    regresa costoTotal

Fin Función TareasVoraz
```

Algoritmos voraces – Problemas de planificación de tareas

El algoritmo que asigna a cada trabajador la tarea más barata dentro de las que aún no han sido asignadas es el siguiente.

```
Función AsignaTarea ( ↓trabajador ∈ Entero, ↑ tarea ∈ Entero )
    Constantes
      Maximo ← 1E10
    Variables
      tarea, mejorTarea ∈ Entero
      trabajador ∈ Entero
      min ∈ Real
    Acciones
     Min ← Maximo
     Para tarea←1 hasta N hacer
       Si (NOT YaAsignada(tarea)) entonces
         Si ( Costo[trabajador][tarea] < min ) entonces
             Min ← Costo[trabajador][tarea]
             mejorTarea ← tarea
          fin Si
       fin Si
     Fin Para

    regresa mejorTarea

Fin Función AsignaTarea
```

El siguiente algoritmo determina si la tarea ya fue o no asignada previamente a otro trabajador.

```
Función YaAsignada ( ↓tarea ∈ Entero, ↑ yaAsignada ∈ Booleano )
    Constantes

    Variables

    Acciones
     Para trabajador←1 hasta N hacer
       Si (tareas[trabajador] = tarea ) entonces
         Regresa TRUE
       fin Si
     Fin Para

    regresa FALSE

Fin Función YaAsignada
```

Algoritmos voraces – Problemas de planificación de tareas

La mejor tarea seleccionada, siempre y cuando no esté asignada, es la de menor costo. Una vez asignada una tarea a cada trabajador, solo resta calcular el costo mediante la siguiente función:

```
Función CalculaCosto ( ↑ suma ∈ Real )
    Constantes

    Variables
      suma ∈ Real
      tarea ∈ Entero

    Acciones
     suma⟵ 0
     Para trabajador⟵1 hasta N hacer
       tarea ⟵ tareas[trabajador]
       suma ⟵ Costo[trabajador][tarea]
     Fin Para

  regresa suma

Fin Función CalculaCosto
```

El vector tareas contiene el número de tarea que se le asignó a cada trabajador, el costo se obtiene de la matriz de costos usando como índices el trabajador con su correspondiente tarea asignada.

Tarea: Escribir el Código para los diferentes algoritmos golosos vistos en clase

FIN