Programación III

Apuntes de cátedra

Cuadrado Estrebou, María Fernanda
Trutner, Guillermo Hernán

agosto de 2013

Facultad de Ingeniería y Ciencias Exactas
Universidad Argentina de la Empresa

El objetivo de este apunte es el de ser una guía en el dictado de la materia, junto a las explicaciones de los docentes en las clases. Se encuentra en su primera versión y es por eso que agradeceremos todas las observaciones, comentarios y correcciones que nos puedan hacer llegar los alumnos para ir mejorándolo.

Resumen

El objetivo del curso de Programación III es continuar el camino comenzado en Programación II. En ésta materia nos centraremos en conocer técnicas de diseño de algoritmos para resolver diferentes problemas, utilizando los TDA vistos anteriormente. Se utilizará un lenguaje de algoritmos y el lenguaje de programación Java como lenguaje de soporte, remarcando que no es el objetivo del curso el lenguaje en sí mismo, sino sólo un medio para aplicar los contenidos. Para cada algoritmo planteado, buscaremos la complejidad del mismo medida en tiempo y en espacio. Esto pretende ser una herramienta para evaluar diferentes alternativas a la hora de resolver un problema y tomándolo en cuenta, además del entorno y las circunstancias, se podrá elegir la opción más acorde.

Índice general

Capítulo 1

Introducción

En los cursos anteriores hemos aprendido a resolver problemas utilizando algoritmos. Una vez planteados los algoritmos los tradujimos a un lenguaje de programación, para poder ejecutarlos en una computadora. Luego seguimos avanzando, llegamos a definir nuevos tipos de datos y el conjunto de operaciones básicas para poder operarlos, son los TDA o tipos de datos abstractos. Analizamos diferentes implementaciones para un mismo TDA para poder definir cuál de ellas es mejor que otra en qué contexto. Los análisis que hemos hecho tenían que ver con la eficiencia o con limitaciones de cada implementación. Ahora nos centraremos en resolver problemas utilizando diferentes técnicas de diseño de algoritmos junto a los TDA ya definidos, que nos permitirán evaluar, a priori, la eficiencia de una solución propuesta.

1.1. Definiciones preliminares

Vamos a definir y repasar algunos conceptos que venimos utilizando desde hace un tiempo para luego introducir algunos nuevos.
El primero de los conceptos a definir será *algoritmo*. Diremos que un *algoritmo* es una secuencia ordenada y finita de acciones que transforman un conjunto de datos de entrada en datos de salida y que resuelven un problema específico. Un algoritmo tiene cero o más datos de entrada, al menos una salida, cada paso está unívocamente determinado y debe terminar en un número finito de pasos. Existen excepciones a estas reglas, por ejemplo algoritmos que en algún punto toman decisiones aleatorias, estos son los algoritmos probabilistas.
Un *programa* es la traducción de un algoritmo a un lenguaje de programación para ser ejecutado en una computadora.
Una *instancia* de un problema es una combinación válida de los datos de entrada posibles. Un problema puede tener una o más instancias, incluso infinitas. Por ejemplo si el problema es multiplicar dos enteros, una instancia es multiplicar 38 por 47.

Un algoritmo será *correcto* o *eficaz* si funciona de la forma esperada para todas las instancias de un problema.
Un algoritmo será *eficiente* si resuelve un problema utilizando una cantidad acotada de recursos y tiempo.
La *algoritmia* se ocupa entonces del estudio de los algoritmos y abarca el análisis de la correctitud y de la eficiencia, es decir analiza si un algoritmo resuelve un problema y la cantidad de recursos necesarios para llegar a la solución.

1.2. Eficiencia de Algoritmos

Dijimos entonces que la eficiencia de un algoritmo es la cantidad de recursos y tiempo que utiliza para resolver un problema. Entonces conociendo la eficiencia de un algoritmo, nos permite compararlo con otro y decidir cuál nos conviene usar.
Podemos medir la eficiencia en forma teórica y en forma práctica. Durante el curso nos ocuparemos de conocer las herramientas para medir la eficiencia en forma teórica y luego comprobaremos estos resultados en forma práctica comparando los tiempos de ejecución en máquina. La ventaja de la evaluación teórica es que no necesitamos esperar el tiempo real de ejecución por lo tanto podremos calcularlo aún para entradas de tamaño muy grande.
La forma de medir los recursos que utiliza un algoritmo consiste en determinar la cantidad de datos elementales que dicho algoritmo debe almacenar sin contar la entrada y la salida.
La forma de medir el tiempo que utiliza un algoritmo consiste en medir la cantidad de operaciones elementales que ejecuta.
La definición de dato elemental y operación elemental es una discusión extensa. También es complicado definir el concepto de tamaño de la entrada. Estos conceptos los iremos aclarando para cada ejemplo, sin que esto genere dudas en los análisis que realizaremos.
En general utilizaremos tamaño de entrada como cantidad de elementos de un vector si la entrada es un vector, si la entrada es un número entero podremos utilizar el valor de dicho entero o la cantidad de dígitos según la característica del problema y si la entrada es un grafo podremos utilizar la cantidad de vértices, la cantidad de aristas o una combinación de ambas.
En general el costo de un algoritmo depende del tamaño de la entrada, por lo tanto lo podemos definir como una función de n donde n es el tamaño de la entrada. Por ejemplo podríamos tener un algoritmo cuyo costo sea $f(n) = 10n + 5$ y otro cuyo costo sea $f(n) = 3n^2$. En un caso así, cuál es la forma de compararlos? Si analizamos sus valores, podremos ver que para valores de n menores a 4, la segunda función es mejor, para todos los demás valores, la primera es mejor. Es decir que para valores de n grandes, la primera es mejor siempre.
La medida de eficiencia no deberá depender de la máquina en que se ejecute un

programa ni de su velocidad. Tampoco depende del lenguaje de programación ni de la implementación particular. El estudio de la eficiencia nos asegura lo que se conoce como principio de invarianza:

> *Dado un algoritmo A y dos implementaciones del mismo I_1 e I_2, donde la primera tarda $T_1(n)$ y la segunda $T_2(n)$ segundos, entonces existen un número real $c > 0$ y un entero n_0 tal que $T_1(n) \leq cT_2(n)$ para todo $n > n_0$.*

Esto significa que dos implementaciones distintas de un mismo algoritmo no difieren entre sí en más de una constante multiplicativa.

1.2.1. Cálculo del número de operaciones elementales

Para la mayoría de nuestros ejemplos, consideraremos una operación elemental a la asignación, la resolución de una expresión con operadores aritméticos y lógicos, acceso a componentes de un vector, llamadas a métodos. Todos estos casos suman 1 a nuestra cuenta.
Para una secuencia de operaciones, la cantidad de operaciones elementales es la suma de la cantidad de operaciones elementales de cada parte.
Para un condicional, la cantidad de operaciones elementales es el costo de la evaluación de la condición más el máximo entre la cantidad de operaciones de las dos ramas de la condición.
Para un bucle, la cantidad de operaciones elementales es N veces la cantidad de operaciones elementales del bloque de repetición, donde N es la cantidad de veces que se repite el bucle.
Para una llamada a un método, el número de operaciones elementales es 1 más el número de operaciones elementales de la ejecución del método.

1.2.2. Tipos de análisis

Existen diferentes tipos de análisis:

- **Peor caso**: Es el mayor de los costos de todas las instancias de un problema.
- **Caso promedio**: Es el promedio de los costos de todas las instancias de un problema.
- **Mejor caso**: Es el menor de los costos de todas las instancias de un problema.

Nos centraremos en el análisis del peor caso ya que es el que aporta una cota superior de todos los demás casos. En aquellos ejemplos en que sea necesario se hará referencia a otro tipo de análisis.

1.2.3. Notación asintótica

Como ya hemos dicho, es de nuestro interés estudiar el comportamiento de los algoritmos cuando el tamaño de la entrada crece mucho. Para ello, vamos a definir una serie de funciones base que tomaremos como funciones modelo para todo un conjunto de funciones. Estas funciones serán asíntotas del resto de las funciones. Esto nos permitirá comparar con facilidad diferentes algoritmos entre si.
Las funciones que tomaremos como base son $f : \mathbb{N} \to \mathbb{R}^{>0}$ con $k > 2$ y $b > 2$ las siguientes:

$f(n) = 1$	constante
$f(n) = \log(n)$	logarítmica
$f(n) = n$	lineal
$f(n) = n\log(n)$	
$f(n) = n^2$	cuadrática
$f(n) = n^k$	polinomial
$f(n) = 2^n$	exponencial
$f(n) = b^n$	exponencial
$f(n) = n!$	factorial

Existen varias formas de utilizar estas asíntotas. La primera y la que más utilizaremos es la medida de orden o notación $\mathcal{O}$. La definición es la siguiente:

Dada una función $g(n)$ definiremos $\mathcal{O}(g(n))$ al conjunto de funciones f para las que existe un número real $c > 0$ y número natural n_0 tal que para todo valor de $n > n_0$, $f(n) \leq cg(n)$. Diremos que $f(n) \in \mathcal{O}(g(n))$ o que f está en el orden de g. En este caso g es una cota superior para f.

Por ejemplo, $f(n) = 2n^2 + 5$ está en $\mathcal{O}(n^2)$, también de $\mathcal{O}(n^3)$ pero no está en $\mathcal{O}(n)$ ni $\mathcal{O}(\log(n))$.
Las siguientes propiedades son válidas para las funciones de orden:

- Si $f(n) \in \mathcal{O}(g(n))$ y $f(n) \in \mathcal{O}(h(n))$ entonces $f(n) \in \mathcal{O}(\text{mín}(g(n), h(n)))$.
- Si $f_1(n) \in \mathcal{O}(g(n))$ y $f_2(n) \in \mathcal{O}(h(n))$ entonces $f_1(n) \cdot f_2(n) \in \mathcal{O}(g(n) \cdot h(n))$.
- Si $f_1(n) \in \mathcal{O}(g(n))$ y $f_2(n) \in \mathcal{O}(h(n))$ entonces $f_1(n) + f_2(n) \in \mathcal{O}(\text{máx}(g(n), h(n)))$.

De las funciones asintóticas vistas, se da la siguiente relación:
$\mathcal{O}(1) \subset \mathcal{O}(\log(n)) \subset \mathcal{O}(n) \subset \mathcal{O}(n\log(n)) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^k) \subset \mathcal{O}(2^n) \subset \mathcal{O}(b^n) \subset \mathcal{O}(n!)$ con $k > 2$ y $b > 2$.
De forma análoga se puede definir la función omega (Ω) como una asíntota inferior:

Dada una función $g(n)$ definiremos $\Omega(g(n))$ al conjunto de funciones f para las que existe un número real $c > 0$ y número natural n_0 tal que para todo valor de $n > n_0$, $f(n) \geq cg(n)$. Diremos que $f(n) \in \Omega(g(n))$ o que f está en omega de g.

Incluso se puede definir que $f(n) \in \mathcal{O}(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$. También se pueden definir propiedades de está función.
Existe una tercera notación para medición de complejidad exacta, que es la notación Theta (Θ). La definición es

Dada una función $g(n)$ definiremos $\Theta(g(n))$ al conjunto de funciones f para las que existe un número real $c > 0$, un número real $d > 0$ y número natural n_0 tal que para todo valor de $n > n_0$, $cg(n) \leq f(n) \leq dg(n)$.

Diremos que $f(n) \in \Theta(g(n))$ o que f está en el orden exacto de g. Se puede demostrar que $f(n) \in \mathcal{O}(g(n)) \wedge f(n) \in \Omega(g(n)) \Leftrightarrow f(n) \in \Theta(g(n))$.

1.2.4. Resolución de recurrencias

Cuando analizamos el costo de un algoritmo recursivo, habitualmente obtenemos funciones de complejidad recurrentes, es decir expresadas en función de si mismas. Por ejemplo, si analizamos el factorial de un número:

```
int factorial(int n){
   if(n == 1){
      return 1;
   }else{
      return n*factoral(n-1);
   }
}
```

La función que representa el costo de está función será:

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ T(n-1) + c & \text{sino} \end{cases}$$

Si el valor de entrada es 1 entonces el tiempo es constante y si la entrada es mayor a 1, el tiempo está dado por el tiempo de ejecutar el algoritmo en el valor de la entrada disminuido en 1 más un costo constante de la multiplicación y la resta. Para resolver estas recurrencias vamos a ver dos métodos, aquellos que están basados en una sustracción, como el ejemplo anterior, donde la llamada recursiva se hace con una sustracción sobre n, osea de la forma $T(n-b)$ y otra forma que

se denomina resolución por división, donde la llamada se hace con una división sobre n, osea de la forma $T(n/b)$.
Para el primer caso en donde tenemos una función de tiempo como la siguiente:

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n-b) + p(n) & \text{si } n \geq b \end{cases}$$

donde $a, c \in \mathbb{R}^+$, $p(n)$ es un polinomio de grado k y $b \in \mathbb{N}$. a es la cantidad de llamadas recursivas que se ejecutan efectivamente para el peor caso, b es la cantidad de unidades en la que se disminuye el tamaño de la entrada y k es el grado del polinomio que representa el tiempo de ejecutar las sentencias fuera de la llamada recursiva, entonces se tiene que el orden es

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \; div \; b}) & \text{si } a > 1 \end{cases}$$

Para el caso de resolución por división, tenemos lo siguientes:

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n/b) + f(n) & \text{si } n \geq b \end{cases}$$

donde $a, c \in \mathbb{R}^+$, $f(n) \in \Theta(n^k)$ y $b \in \mathbb{N}$, $b > 1$. a es la cantidad de llamadas recursivas que se ejecutan efectivamente para el peor caso, b es la cantidad de unidades en la que se divide el tamaño de la entrada y k es el grado del polinomio que representa el tiempo de ejecutar las sentencias fuera de la llamada recursiva, entonces se tiene que el orden es

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log(n)) & \text{si } a = b^k \\ \Theta(n^{\log_b(a)}) & \text{si } a > b^k \end{cases}$$

El ejemplo del factorial se puede resolver con el primero de los casos en donde $a = 1$, $b = 1$ y $k = 0$, como $a = 1$ podemos utilizar la opción del medio y el resultado es $\Theta(n^{k+1}) = \Theta(n)$. En el siguiente capítulo se verán ejemplos de aplicación de ambos casos.

Capítulo 2

Divide y conquista

Vamos a comenzar a presentar diferentes técnicas de diseño de algoritmos. Entenderemos por técnica la idea de un patrón o una estrategia para resolver un problema. Debemos tener en claro que nada de esto significa una receta, es decir, no podríamos definir un algoritmo determinístico para escribir algoritmos, ni aún siguiendo una técnica específica. Iremos analizando diferentes técnicas, sus características básicas y algunos ejemplos significativos que responden a la misma además de algunos que no responden.
La primera de las técnicas que vamos a analizar es la técnica conocida en inglés como *Divide&Conquer*.

2.1. Características generales

El concepto de dividir y conquistar probablemente no sea novedoso ya que la idea se utiliza desde las primeras materias de programación para modularizar, que no es otra cosa que dividir un problema grande en problemas menores, de resolución más sencilla y luego combinar las pequeñas soluciones para resolver el problema original. Esta técnica consiste entonces en dividir un problema en subproblemas con varias características, una es que los subproblemas en los que se divide el problema original son de igual naturaleza entre si y de igual naturaleza del problema original. Además los subproblemas son de menor tamaño que el problema original y de tamaños similares entre sí. Existen casos en donde el problema original no se divide en varios subproblemas sino en un único subproblema. Esta división se realiza en forma recursiva hasta que el tamaño del subproblema a tratar sea de un tamaño "suficientemente pequeño" para poder resolverlo de una "forma trivial". El concepto de "suficientemente pequeño" y "forma trivial" lo iremos discutiendo en cada caso para entender qué es lo mejor.
El ejemplo más sencillo y conocido de esta técnica es el que resuelve el problema de indicar si un elemento x dado está presente en una secuencia ordenada de números S también dada. La solución que utiliza esta técnica es la llamada *Búsqueda binaria* y consiste en seleccionar el elemento que está en la posición del

medio de la secuencia S. Si ese elemento coincide con x estamos en condiciones de afirmar que x pertenece a S. Sino, como la lista está ordenada si el elemento del medio es mayor a x existen dos opciones, o que el x no está en S o que está a la izquierda del elemento del medio, procedemos a aplicar la misma técnica, pero ahora la secuencia pasa a ser la primera mitad de S. Si en cambio el elemento del medio es menor a x, procedemos a aplicar la misma técnica sobre la segunda mitad de S. En este caso procedemos recursivamente salvo que alcancemos un problema suficientemente pequeño, que podría ser que el tamaño de la subsecuencia sea 1. En ese caso, la solución trivial consiste en ver si el único elemento de la subsecuencia alcanzada coincide o no con el elemento x buscado. El algoritmo tendría una forma como la siguiente:

ALGORITMO BUSQUEDABIN
Entrada: S: Vector<entero>, x: entero
Salida: **verdadero** o **falso**
si $longitud(S) = 1$
devolver $S[0] = x$
sino
$y \leftarrow S[longitud(S)/2]$
si $x = y$
devolver **verdadero**
sino
$mitad \leftarrow longitud(S)/2$
si $x < y$
devolver $BusquedaBin(S[0, mitad - 1], x)$
sino
devolver $BusquedaBin(S[mitad..longitud(S) - 1], x)$
fin si
fin si
fin si

Por ejemplo, para un valor de $x = 3$ la siguiente secuencia

1	3	7	10	25	106	121	130	145

la secuencia de ejecución sería:

1	3	7	10	25

1	3

3

En este caso el resultado es verdadero y sale por la condición de longitud uno y valor coincide.
Si quisiéramos analizar la complejidad de este algoritmo, en forma intuitiva, podríamos ver que en cada llamada, el tamaño de la entrada se reduce a la mitad, hasta llegar a tamaño 1, con lo que estaríamos pensando en $\mathcal{O}(\log(n))$.
Si quisiéramos analizarlo con los métodos de resolución de recurrencias vistos, deberíamos plantear la recurrencia como:

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ 1T(n/2) + c & \text{si } n > 1 \end{cases}$$

y encontrando los valores de a, b y k que son necesarios para resolver la recurrencia con el método de la división, tenemos que $a = 1$, $b = 2$ y $k = 0$, entonces tenemos que $a = b^k$ ya que $1 = 2^0$ con lo que el resultado es $\Theta(n^k \log(n))$ que es $\Theta(n^0 \log(n) = \Theta(\log(n))$ que es lo que habíamos encontrado en forma intuitiva.
El código Java de la búsqueda binaria es el siguiente:

```
public static boolean busquedaBin(VectorTDA<Integer>
    valores, int inicio, int fin, int valor) throws
    Exception{
    if(inicio==fin){
        return (valores.recuperarElemento(inicio) == valor
            );
    }else{
        int mitad = (fin+inicio)/2;
        if(valor <= valores.recuperarElemento(mitad)){
            return busquedaBin(valores,inicio,mitad,valor);
        }else{
            return busquedaBin(valores,mitad+1,fin,valor);
        }
    }
}
```

En general, podemos decir que la técnica de Divide y Conquista tiene la siguiente forma:

ALGORITMO DYC
Entrada: x
  si $esPequeno(x)$
    devolver $solucion\_trivial(x)$
  sino
    $\langle x_1, ..., x_n \rangle \leftarrow descomponer(x)$
    para $i = 1$ **hasta** n
      $y_i \leftarrow DyV(x_i)$

```
    fin para
    devolver combinar(y_1, ..., y_n)
  fin si
```

y se debe cumplir que el problema es divisible en subproblemas más pequeños que el problema orignal, estos subproblemas son independientes entre si y debe existir una forma directa para resolver el problema suficientemente pequeño. Se espera que los costos de descomponer y de combinar sean lo suficienemente bajos para no alterar la ventaja de la técnica. En general se peuden encontrar los costos analizándolos con la técnica de resolución de recurrencias por división.

2.2. Métodos de ordenamiento

Ya conocemos métodos para ordenar los elementos de un array por ejemplo el ordenamiento por selección y por inserción. Vamos a plantear cada uno de ellos para analizar su costo y ver si podemos encontrar algoritmos más eficientes.
El método de ordenamiento por selección consiste en ir buscando el menor de los elementos, colocarlo en la primera posición, luego buscar el menor de los restantes, colocarlo en la segunda posición y así siguiente hasta ordenar todos los elementos. El algoritmo tiene la siguiente forma:

```
Algoritmo Seleccion
Entrada: S: Vector<entero>
  n ← longitud(S)
  para i = 0 hasta n − 1
    m ← i
    para j = i + 1 hasta n − 1
      si S[j] < S[m]
        m ← j
      fin si
    fin para
    aux ← S[i];
    S[i] ← S[m];
    S[m] ← aux;
  fin para
```

Partiendo del siguiente arreglo

5	3	7	10	2	1

y pasando por los siguientes pasos

1	3	7	10	2	5

1	**2**	7	10	3	5

1	**2**	**3**	10	7	5

1	**2**	**3**	**5**	7	10

1	**2**	**3**	**5**	**7**	10

se obtiene la solución

1	**2**	**3**	**5**	**7**	**10**

Si analizamos el costo de este algoritmo podemos ver claramente que está en $\mathcal{O}(n^2)$. Además se puede ver que sea cual fuere la entrada, el algoritmo no varía en su orden.
El código Java sería el siguiente:

```
public static void selectionSort(VectorTDA<Integer>
   valores,int inicio, int fin) throws Exception{
   int i,j,minj,minx;
   for(i = inicio; i < fin; i++){
      minj = i;
      minx = valores.recuperarElemento(i);
      for(j = i+1; j <= fin; j++){
         if(valores.recuperarElemento(j)<minx){
            minj = j;
            minx = valores.recuperarElemento(j);
         }
      }
      valores.agregarElemento(minj,valores.
         recuperarElemento(i));
      valores.agregarElemento(i, minx);
   }
}
```

El algoritmo de inserción consiste en ir ubicando cada elemento en su posición. Es decir que en cada iteración i se obtiene una secuencia ordenada de $i-1$ elementos en la primera parte del arreglo y se inserta el elemento de la posición i para que ahora la secuencia de i elementos quede ordenada. El algoritmo es el siguiente:

Algoritmo Insercion

Entrada: S: Vector<entero>
  $n \leftarrow longitud(S)$
  para $i = 1$ **hasta** $n - 1$
    para $j = i$ **hasta** 1 **decrementando** 1
      si $S[j] < S[j-1]$
        $aux \leftarrow S[j-1]$;
        $S[j-1] \leftarrow S[j]$;
        $S[j] \leftarrow aux$;
      fin si
    fin para
  fin para

Partiendo del siguiente arreglo

5	3	7	10	2	1

y pasando por los siguientes pasos

3	**5**	7	10	2	1

3	**5**	**7**	10	2	1

3	**5**	**7**	**10**	2	1

2	**3**	**5**	**7**	**10**	1

se obtiene la solución

1	**2**	**3**	**5**	**7**	**10**

Si analizamos el costo de este algoritmo podemos ver claramente que está en $\mathcal{O}(n^2)$. Además se puede ver que sea cual fuere la entrada, el algoritmo no varía en su tiempo. Haciéndole algunas mejoras, aunque se mejora el tiempo, no se altera el orden.
El código Java del método es:

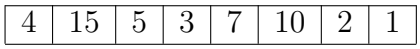
```
public static void insertionSort(VectorTDA<Integer>
   valores,int inicio, int fin) throws Exception{

   int j;
   int aux;
   for(int i = inicio+1; i <= fin; i++){
      aux = valores.recuperarElemento(i);
```

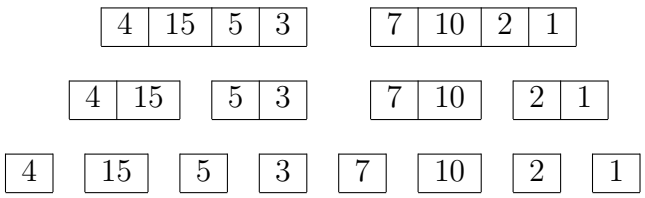
```
            j = i - 1;
            while(j>=inicio && aux < valores.recuperarElemento
               (j)){
               valores.agregarElemento(j+1,valores.
                  recuperarElemento(j));
               j--;
            }
            valores.agregarElemento(j+1,aux);
         }
      }
```

2.2.1. MergeSort

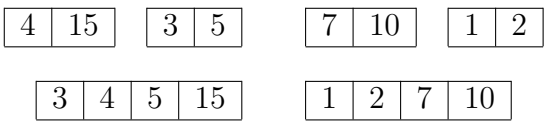
Otra técnica de ordenamiento conocida es la llamada *MergeSort* y consiste en dividir la secuencia de entrada en dos mitades, ordenar cada una de las mitades y luego mezclar las mitades ordenadas en una nueva secuencia ordenada. Como tamaño pequeño y solución trivial podríamos considerar una secuencia con un solo elemento, que está trivialmente ordenada. Podría ser una secuencia de tamaño 2, que se ordena en forma trivial con un condicional y tres asignaciones.
Por ejemplo, partiendo del siguiente arreglo



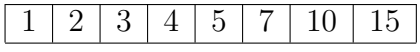
diviendo



combinando



obtenemos



El algoritmo tiene la siguiente estructura:

ALGORITMO MERGESORT
Entrada: S: Vector<entero>, *inicio*: entero, fin: entero
  si $inicio < fin$
    entero $medio \leftarrow (fin + inicio)/2$
    $MergeSort(S, inicio, medio)$
    $MergeSort(S, medio + 1, fin)$
    $Merge(S, inicio, fin)$
  fin si

ALGORITMO MERGE
Entrada: S: Vector<entero>, *inicio*:entero, fin: entero
  Vector $R \leftarrow inicializarVector(fin - inicio + 1)$
  entero $medio \leftarrow (inicio + fin)/2$
  $i \leftarrow inicio$
  $j \leftarrow medio + 1$
  para entero $k = 0$ **hasta** $fin - inicio$
    si ($j > fin$ **O** $S[i] \leq S[j]$
      $S[k] \leftarrow S[i]$
      $i \leftarrow i + 1$
    sino
      $S[k] \leftarrow S[j]$
      $j \leftarrow j + 1$
    fin si
  fin para
  para entero $k = 0$ **hasta** $fin - inicio$
    $S[inicio + k] \leftarrow R[k]$
  fin para

Para implementarlo en Java, en lugar de crear vectores auxiliares en el algoritmo principal, conviene trabajar con el mismo vector y subíndices que indican con qué parte del vector se está trabajando. Una posible implementación es la que sigue:

```
void mergeSort(VectorTDA<Integer> valores, int inicio,
   int fin) throws Exception{
   if(inicio<fin){
      int medio = (fin+inicio)/2;
      mergeSort(valores,inicio,medio);
      mergeSort(valores,medio+1,fin);
      merge(valores,inicio,fin);
```

```
    }
}

void merge(VectorTDA<Integer> valores,int inicio,int fin
    ) throws Exception{

    //utilizo un vector auxiliar, paso todo allí y luego
       lo vuelvo al vector de origen
    VectorTDA<Integer> resultado = new Vector<Integer>();
    resultado.inicializarVector(1+fin-inicio);
    int medio = (fin+inicio)/2;
    int i = inicio;
    int j = medio + 1;

    for(int k = 0; k <= fin-inicio; k++){
        if(j> fin || (i<=medio &&valores.recuperarElemento
            (i) <= valores.recuperarElemento(j))){
            resultado.agregarElemento(k,valores.
               recuperarElemento(i));
            i++;
        }else{
            resultado.agregarElemento(k,valores.
               recuperarElemento(j));
            j++;
        }
    }

    for(int k =0; k <= fin - inicio; k++){
        valores.agregarElemento(inicio+k,resultado.
           recuperarElemento(k));
    }
}
```

Si analizamos la complejidad del algoritmo, tenemos que el algoritmo principal tiene dos llamadas recursivas con un vector de la mitad de tamaño del vector de entrada, el cálculo de la mitad y la mezcla que a simple vista se ve que tiene $\mathcal{O}(n)$. Con lo cual podemos plantear la siguiente recurrencia:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + P^1(n) & \text{si } n > 1 \end{cases}$$

y encontrando los valores de a, b y k que son necesarios para resolver la recurrencia con el método de la división, tenemos que $a = 2$, $b = 2$ y $k = 1$, entonces tenemos que $a = b^k$ ya que $2 = 2^1$ con lo que el resultado es $\Theta(n^k \log(n))$ que es $\Theta(n^1 \log(n)) = \Theta(n \log(n))$. Es decir que este método de ordenamiento es mejor

que los encontrados hasta el momento y funciona siempre en el mismo tiempo independientemente de la forma de la entrada.

2.2.2. QuickSort

Existe también otro algoritmo de ordenamiento llamado *QuickSort*. El mismo tiene como idea base elegir un elemento al azar, llamado pivot, a partir de ese elemento dividir el arreglo en dos partes, a la izquierda del pivot todos los elementos menores y a la derecha del pivot todos los elementos mayores. Una vez hecho esto, el pivot estará ubicado en su posición definitiva en el vector y se aplicará en forma recursiva el método en las dos partes resultantes.
Existen diferentes técnicas para seleccionar el pivot, algunas más sencillas y otras más sofisticadas, ya que la buena elección del pivot redundará en una solución más eficiente. Cuanto más cercano al elemento mediano se encuentre el pivot, más parejas serán las dos partes del vector y mejor la solución, cuanto más cercano a alguno de los extremos se encuentre el vector, más desparejas las dos partes y menos eficiente la solución. La solución trivial podrá ser para un arreglo de una o dos posiciones.
Veamos un ejemplo seleccionando como pivot el primer elemento:

15	17	7	4	21	9	3	18	24	26	1

pivoteando en el elemento en negrita

7	4	9	3	1	**15**	17	21	18	24	26

4	3	1	**7**	9	15	**17**	21	18	24	26

3	1	**4**	7	9	15	17	18	**21**	24	26

1	**3**	4	7	9	15	17	18	21	**24**	26

El algoritmo luce como sigue:

ALGORITMO QUICKSORT
Entrada: S: Vector<entero>, *inicio*:entero, fin:entero
  si $inicio < fin$
    $p \leftarrow pivot(S, inicio, fin)$
    $QuickSort(S, inicio, p-1)$
    $QuickSort(S, p+1, fin)$
  fin si

ALGORITMO PIVOT
Entrada: S: Vector<entero>, $inicio$:entero, fin:entero
Salida: p: entero
entero $p \leftarrow S[inicio]$
entero $k \leftarrow inicio + 1$
entero $l \leftarrow fin$
mientras $S[k] \leq p$ **Y** $k < fin$
$k \leftarrow k + 1$
fin mientras
mientras $S[l] > p$
$l \leftarrow l - 1$
fin mientras
mientras $k < l$
$aux \leftarrow S[k]$
$S[k] \leftarrow S[l]$
$S[l] \leftarrow aux$
mientras $S[k] \leq p$
$k \leftarrow k + 1$
fin mientras
mientras $S[l] > p$
$l \leftarrow l - 1$
fin mientras
fin mientras
$aux \leftarrow S[inicio]$
$S[inicio] \leftarrow S[l]$
$S[l] \leftarrow aux$
devolver l

y la implementación, utilizando la misma salvedad de no usar arreglos auxiliares sino índices sobre el arreglo original, podría ser así:

```
void quickSort(VectorTDA<Integer> valores, int inicio,
   int fin) throws Exception{

   if(inicio < fin){
      int p = pivot(valores,inicio,fin);
      quickSort(valores,inicio,p-1);
      quickSort(valores,p+1,fin);
   }
}

int pivot(VectorTDA<Integer> valores, int inicio, int
   fin) throws Exception{
```

```
        int l;
        int aux;

        int p = valores.recuperarElemento(inicio);
        int k = inicio + 1;
        l = fin;

        while(valores.recuperarElemento(k) <= p && k < fin){
            k++;
        }

        while(valores.recuperarElemento(l)>p){
            l--;
        }

        while(k<l){
            aux = valores.recuperarElemento(k);
            valores.agregarElemento(k,valores.
                recuperarElemento(l));
            valores.agregarElemento(l, aux);
            while(valores.recuperarElemento(k) <= p){
                k++;
            }
            while(valores.recuperarElemento(l) > p){
                l--;
            }
        }

        aux = valores.recuperarElemento(inicio);
        valores.agregarElemento(inicio, valores.
            recuperarElemento(l));
        valores.agregarElemento(l,aux);

        return l;
    }
```

Si analizamos el costo de este algoritmo, vemos que el método pivot tiene un costo de $\mathcal{O}(n)$ ya que los while anidados siempre tienen como restricción el while principal. Y el método principal tiene dos llamadas recursivas. El mejor de los casos es cuando el pivot cae cerca del centro del vector, en cuyo caso en análisis de costos es el siguiente suponiendo que n es el tamaño del vector:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + P^1(n) & \text{si } n > 1 \end{cases}$$

y encontrando los valores de a, b y k que son necesarios para resolver la recurrencia con el método de la división, tenemos que $a = 2$, $b = 2$ y $k = 1$, entonces tenemos que $a = b^k$ ya que $2 = 2^1$ con lo que el resultado es $\Theta(n^k \log(n))$ que es $\Theta(n^1 \log(n)) = \Theta(n \log(n))$.
Sin embargo, para el peor caso, que es cuando el pivot cae cerca de uno de los extremos del vector, tendríamos:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ T(n-1) + P^1(n) & \text{si } n > 1 \end{cases}$$

y encontrando los valores de a, b y k que son necesarios para resolver la recurrencia con el método de la sustracción, tenemos que $a = 1$, $b = 1$ y $k = 1$, entonces tenemos que $a = 1$ con lo que el resultado es $\Theta(n^{k+1})$ que es $\Theta(n^{1+1}) = \Theta(n^2)$.
Esto significa que para el análisis que habitualmente hacemos, para el peor caso, este algoritmo es peor que el *MergeSort*, pero en las situaciones reales, donde los datos tienen una distribución donde cualquiera de los ordenamientos es igualmente probable, y haciendo un análisis más minucioso se puede demostrar que si bien el orden es igual para ambos algoritmos en el caso promedio, el tiempo de ejecución es menor para el *QuickSort*.

2.3. Otros ejemplos

Analizaremos algunos ejemplos más de problemas que se pueden resolver con esta técnica.

2.3.1. Elemento Mayoritario

Dado un vector A de números enteros, calcular elemento mayoritario. Si se tiene un vector A de n enteros, un elemento x se denomina mayoritario de A si x aparece en el vector A más de $n/2$ veces. Considerar que no puede haber más de un elemento mayoritario.
La forma natural y más sencilla de resolverlo es verificar para cada elemento del vector, si aparece más de $n/2$ veces. Este algoritmo tiene la forma de dos bucles anidados con lo que podríamos afirmar que esta estrategia tiene un costo de $\mathcal{O}(n^2)$.
Claramente con verificar sólo los elementos de la primera mitad, podríamos bajar el tiempo pero no el orden del algoritmo.
Un algoritmo más eficiente podría ser ordenar el vector ($\mathcal{O}(n \log(n))$). Si existe un elemento mayoritario, éste se encuentra en la posición media del vector, entonces se toma el elemento de la posición del medio y se cuenta si aparece más de $n/2$ veces($\mathcal{O}(n)$). Con lo que esta estrategia está marcada por el ordenamiento y la solución queda en $\mathcal{O}(n \log(n))$.
Una estrategia aplicando la técnica de divide y conquista consiste en buscar elementos mayoritario en la primera mitad del vector. Si se encuentra, se verifica si

ese elemento es mayoritario en el vector completo. Si no se encuentra, se busca en la segunda mitad y se procede de la misma forma. Esto se basa en la idea que si un elemento es mayoritario en un vector debe ser mayoritario en alguna de las dos mitades. Si se analizan los costos se puede ver que cae también en $\mathcal{O}(n \log(n))$.
Podríamos plantear otra estrategia basada también en la técnica de divide y conquista y en la idea que si un elemento x es mayoritario y dividimos el vector original en subvectores de tamaño 2 (salvo el último que si son n es impar quedaría de tamaño 1), debe aparecer x repetido en alguno de los subvectores o debe ser el elemento que está en vector de un sólo elemento para el caso de n impar. Si nos quedamos con todos los elementos que cumplen con esta condición, podremos reducir nuestro vector original a un vector que tiene a lo sumo $n/2$ elementos. Si esto lo repetimos hasta obtener un único candidato, luego nos resta verificar si ese candidato es elemento mayoritario o no.
Por ejemplo, para el siguiente vector:

1	1	3	3	1	3	1	4	1	1

El método que busca el candidato, en primera instancia reduce el vector a:

1	3	1

luego vuelve a reducir, como no hay ningún par de valores consecutivos iguales y el tamaño del vector es impar, devuelve como candidato al 1. Verificando en el vector original si el 1 cumple con la propiedad buscada vemos que es la solución. Entonces lo que haremos será buscar un candidato que cumpla con estas características, de la siguiente manera:

ALGORITMO BUSCARCANDIDATO
Entrada: V: Vector<entero>, *inicio*:entero, fin:entero
Salida: c: entero
  si $fin < inicio$
    devolver No hay cadidato
  sino
    si $inicio = fin$
      devolver $V[inicio]$
    sino
      $j \leftarrow inicio$
      si $esPar(fin - inicio + 1)$
        $i \leftarrow inicio + 1$
        mientras $i \leq fin$
          si $V[i-1] = V[i]$
            $V[j] \leftarrow V[i]$
            $j \leftarrow j + 1$

```
            fin si
            i ← i + 2
          fin mientras
          devolver BuscarCandidato(V, inicio, j)
        sino
          i ← inicio + 1
          mientras i < fin
            si V[i − 1] = V[i]
              V[j] ← V[i]
              j ← j + 1
            fin si
            i ← i + 2
          fin mientras
          si tieneSolucion(BuscarCandidato(V, inicio, j))
            devolver BuscarCandidato(V, inicio, j)
          sino
            devolver V[fin]
          fin si
        fin si
      fin si
    fin si
```

Entonces el algoritmo final tiene esta forma:

```
Algoritmo ElementoMayoritario
Entrada: V: Vector<entero>
Salida: x: entero
  entero x ← BuscarCandidato(V, 0, longitud(S) − 1)
  si existe(x)
    suma ← 0
    para entero i = 0 hasta longitud(V) − 1
      si x = V[i]
        suma ← suma + 1
      fin si
    fin para
    si suma > longitud(V)/2
      devolver x
    sino
      devolver No existe elemento mayoritario
    fin si
  sino
```

devolver No existe elemento mayoritario
fin si

Si analizamos el costo de este algoritmo, el método que busca el candidato, tiene

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ T(n/2) + P^1(n) & \text{si } n > 1 \end{cases}$$

y encontrando los valores de a, b y k que son necesarios para resolver la recurrencia con el método de la sustracción, tenemos que $a = 1$, $b = 2$ y $k = 1$, entonces tenemos que $a < b^k$ ya que $1 < 2^1$ con lo que el resultado es $\Theta(n^k)$ que es $\Theta(n^1) = \Theta(n)$ y el método principal también tiene $\mathcal{O}(n)$, es decir que encontramos un método para encontrar el elemento mayoritario en $\mathcal{O}(n)$.
El código Java para estos dos métodos es el siguiente:

```
public static Integer elementoMayoritario(VectorTDA<
   Integer> valores, int inicio, int fin) throws
   Exception{

   int i;
   VectorTDA<Integer> B = new Vector<Integer>();
   B.inicializarVector(1+fin-inicio);
   for(i = 0; i < 1+fin-inicio; i++){
      B.agregarElemento(i, valores.recuperarElemento(i+
         inicio));
   }
   Integer candidato = buscar_candidato(B,0,fin-inicio);
   if(candidato != null){
      int suma = 0;
      for(i = inicio; i < fin+1; i++){
         if(candidato.equals(valores.recuperarElemento(i
            ))){
            suma++;
         }
      }
      if(suma > (1+fin-inicio)/2){
         return candidato;
      }else{
         return null;
      }
   }else{
      return null;
   }
}
```

```
private static Integer buscar_candidato(VectorTDA<
   Integer> valores, int inicio, int fin) throws
   Exception{

   if(inicio > fin){
      return null;
   }else{
      if(inicio == fin){
         return valores.recuperarElemento(inicio);
      }else{
         int j = inicio;
         if((1+fin-inicio)% 2 == 0){
            for(int i= inicio+1; i <= fin; i=i+2){
               if( valores.recuperarElemento(i-1) ==
                  valores.recuperarElemento(i)){
                  valores.agregarElemento(j,valores.
                     recuperarElemento(i));
                  j++;
               }
            }
            return buscar_candidato(valores,inicio,j-1);
         }else{
            for(int i = inicio+1; i < fin; i=i+2){
               if(valores.recuperarElemento(i-1) ==
                  valores.recuperarElemento(i)){
                  valores.agregarElemento(j, valores.
                     recuperarElemento(i));
                  j++;
               }
            }
            Integer candidato = buscar_candidato(valores
               ,inicio,j-1);
            if(candidato != null){
               return candidato;
            }else{
               return valores.recuperarElemento(fin);
            }
         }
      }
   }
}
```

2.3.2. Multiplicación de matrices cuadradas

El problema consiste en encontrar, dadas dos matrices N_1 y N_2 de $N \times N$, el producto de ambas matrices que será una nueva matriz de $N \times N$.
El algoritmo tradicional, consiste en multiplicar para cada una de las N^2 posiciones a_{ij}de la matriz resultado, los N elementos de la fila i de N_1 por los N elementos de la columna j de N_2. Esto significa que para resolver el problema se realizarán N^3 multiplicaciones, lo que equivale decir que esta estrategia de resolución es $\mathcal{O}(n^3)$. Esto es a diferencia de la suma y resta de matrices que tienen $\mathcal{O}(n^2)$.
Existe una técnica llamada *Multiplicación de Strassen* que permite multiplicar dos matrices de una forma algo mejor que cúbica. La técnica aplicada a matrices de 2×2 dice que resolver

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

es igual a calcular

$$\begin{pmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{pmatrix}$$

donde
$m_1 = (a_{11} + a_{22})(b_{11} + b_{22})$
$m_2 = (a_{21} + a_{22})b_{11}$
$m_3 = a_{11}(b_{12} - b_{22})$
$m_4 = a_{22}(b_{21} - b_{11})$
$m_5 = (a_{11} + a_{12})b_{22}$
$m_6 = (a_{21} - a_{11})(b_{11} + b_{12})$
$m_7 = (a_{12} - a_{22})(b_{21} + b_{22})$

Por ejemplo, para multiplicar

$$\begin{pmatrix} 3 & 5 \\ 7 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 4 \\ 3 & 1 \end{pmatrix} = \begin{pmatrix} 21 & 17 \\ 17 & 29 \end{pmatrix}$$

$m_1 = (3 + 1)(2 + 1) = 12$
$m_2 = (7 + 1)2 = 16$
$m_3 = 3(4 - 1) = 9$
$m_4 = 1(3 - 2) = 1$
$m_5 = (3 + 5)1 = 8$
$m_6 = (7 - 3)(2 + 4) = 24$
$m_7 = (5 - 1)(3 + 1) = 16$

$$\begin{pmatrix} 12 + 1 - 8 + 16 & 9 + 8 \\ 16 + 1 & 12 + 9 - 16 + 24 \end{pmatrix} = \begin{pmatrix} 21 & 17 \\ 17 & 29 \end{pmatrix}$$

Como podemos ver, en lugar de hacer ocho multiplicaciones, sólo realiza siete pero en lugar de hacer cuatro sumas hace doce sumas y seis restas. Es decir que para este caso, no hubo mejora de costos si suponemos que la multiplicación de enteros tiene el mismo costo que la suma y la resta. Pero si en lugar de enteros fueran matrices la cosa cambia ya que para valores de n suficientemente grandes $7n^3+12n^2+6n^2 < 8n^3$ que es verdad para $n > 18$. Esto significa que si aplicamos la técnica de divide y conquista sobre matrices donde N es una potencia de dos, lo podemos resolver multiplicando y sumando sus submatrices, de la misma forma que resolvimos lo anterior, donde cada a_{ij} es una matriz de tamaño $\frac{N}{2} \times \frac{N}{2}$. Entonces, para multiplicar dos matrices cuadradas de dimensión igual a una potencia de dos en lugar de las $64 = 4^3$ multiplicaciones necesarias, sólo necesitaríamos $52 \simeq 4^{2.8}$. Por ejemplo, multiplicar

$$\begin{pmatrix} 2 & 3 & 1 & 2 \\ 1 & 4 & 1 & 8 \\ 1 & 2 & 6 & 2 \\ 5 & 3 & 1 & 7 \end{pmatrix} \cdot \begin{pmatrix} 4 & 2 & 2 & 5 \\ 2 & 1 & 2 & 7 \\ 2 & 1 & 4 & 3 \\ 1 & 4 & 3 & 5 \end{pmatrix} = \begin{pmatrix} 18 & 16 & 20 & 44 \\ 22 & 39 & 38 & 76 \\ 22 & 18 & 36 & 47 \\ 35 & 42 & 41 & 84 \end{pmatrix}$$

Aplicando el método visto, quedaría

$$\begin{pmatrix} \begin{pmatrix} 2 & 3 \\ 1 & 4 \end{pmatrix} & \begin{pmatrix} 1 & 2 \\ 1 & 8 \end{pmatrix} \\ \begin{pmatrix} 1 & 2 \\ 5 & 3 \end{pmatrix} & \begin{pmatrix} 6 & 2 \\ 1 & 7 \end{pmatrix} \end{pmatrix} \cdot \begin{pmatrix} \begin{pmatrix} 4 & 2 \\ 2 & 1 \end{pmatrix} & \begin{pmatrix} 2 & 5 \\ 2 & 7 \end{pmatrix} \\ \begin{pmatrix} 2 & 1 \\ 1 & 4 \end{pmatrix} & \begin{pmatrix} 4 & 3 \\ 3 & 5 \end{pmatrix} \end{pmatrix}$$

Tenemos que

$$m_1 = \left[\begin{pmatrix} 2 & 3 \\ 1 & 4 \end{pmatrix} + \begin{pmatrix} 6 & 2 \\ 1 & 7 \end{pmatrix}\right]\left[\begin{pmatrix} 4 & 2 \\ 2 & 1 \end{pmatrix} + \begin{pmatrix} 4 & 3 \\ 3 & 5 \end{pmatrix}\right] = \begin{pmatrix} 89 & 70 \\ 71 & 76 \end{pmatrix}$$

$$m_2 = \left[\begin{pmatrix} 1 & 2 \\ 5 & 3 \end{pmatrix} + \begin{pmatrix} 6 & 2 \\ 1 & 7 \end{pmatrix}\right]\begin{pmatrix} 4 & 2 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} 36 & 18 \\ 44 & 22 \end{pmatrix}$$

$$m_3 = \begin{pmatrix} 2 & 3 \\ 1 & 4 \end{pmatrix}\left[\begin{pmatrix} 2 & 5 \\ 2 & 7 \end{pmatrix} - \begin{pmatrix} 4 & 3 \\ 3 & 5 \end{pmatrix}\right] = \begin{pmatrix} -7 & 10 \\ -6 & 10 \end{pmatrix}$$

$$m_4 = \begin{pmatrix} 6 & 2 \\ 1 & 7 \end{pmatrix}\left[\begin{pmatrix} 2 & 1 \\ 1 & 4 \end{pmatrix} - \begin{pmatrix} 4 & 2 \\ 2 & 1 \end{pmatrix}\right] = \begin{pmatrix} -14 & 0 \\ -9 & 20 \end{pmatrix}$$

$$m_5 = \left[\begin{pmatrix} 2 & 3 \\ 1 & 4 \end{pmatrix} + \begin{pmatrix} 1 & 2 \\ 1 & 8 \end{pmatrix}\right]\begin{pmatrix} 4 & 3 \\ 3 & 5 \end{pmatrix} = \begin{pmatrix} 27 & 34 \\ 44 & 66 \end{pmatrix}$$

$$m_6 = \left[\begin{pmatrix} 1 & 2 \\ 5 & 3 \end{pmatrix} - \begin{pmatrix} 2 & 3 \\ 1 & 4 \end{pmatrix}\right]\left[\begin{pmatrix} 4 & 2 \\ 2 & 1 \end{pmatrix} + \begin{pmatrix} 2 & 5 \\ 2 & 7 \end{pmatrix}\right] = \begin{pmatrix} -10 & -15 \\ 20 & 20 \end{pmatrix}$$

$$m_7 = \left[\begin{pmatrix} 1 & 2 \\ 1 & 8 \end{pmatrix} - \begin{pmatrix} 6 & 2 \\ 1 & 7 \end{pmatrix}\right]\left[\begin{pmatrix} 2 & 1 \\ 1 & 4 \end{pmatrix} + \begin{pmatrix} 4 & 3 \\ 3 & 5 \end{pmatrix}\right] = \begin{pmatrix} -30 & -20 \\ 4 & 9 \end{pmatrix}$$

Y con estos datos, el resultado queda armado por

$$\begin{pmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{pmatrix}$$

donde

$$R_{11} = \begin{pmatrix} 89 & 70 \\ 71 & 76 \end{pmatrix} + \begin{pmatrix} -14 & 0 \\ -9 & 20 \end{pmatrix} - \begin{pmatrix} 27 & 34 \\ 44 & 66 \end{pmatrix} + \begin{pmatrix} -30 & -20 \\ 4 & 9 \end{pmatrix} = \begin{pmatrix} 18 & 16 \\ 22 & 39 \end{pmatrix}$$

$$R_{12} = \begin{pmatrix} -7 & 10 \\ -6 & 10 \end{pmatrix} + \begin{pmatrix} 27 & 34 \\ 44 & 66 \end{pmatrix} = \begin{pmatrix} 20 & 44 \\ 38 & 76 \end{pmatrix}$$

$$R_{21} = \begin{pmatrix} 36 & 18 \\ 44 & 22 \end{pmatrix} + \begin{pmatrix} -14 & 0 \\ -9 & 20 \end{pmatrix} = \begin{pmatrix} 22 & 18 \\ 35 & 42 \end{pmatrix}$$

$$R_{22} = \begin{pmatrix} 89 & 70 \\ 71 & 76 \end{pmatrix} + \begin{pmatrix} -7 & 10 \\ -6 & 10 \end{pmatrix} - \begin{pmatrix} 36 & 18 \\ 44 & 22 \end{pmatrix} + \begin{pmatrix} -10 & -15 \\ 20 & 20 \end{pmatrix} = \begin{pmatrix} 36 & 47 \\ 41 & 84 \end{pmatrix}$$

Osea que multiplicar entre si dos matrices de $N \times N$ equivale a hacer siete multiplicaciones entre matrices de $\frac{N}{2} \times \frac{N}{2}$ y dieciocho sumas y restas de matrices de $\frac{N}{2} \times \frac{N}{2}$. Si planteamos la recurrencia para analizar el costo es

$$T(n) = \begin{cases} 8 & \text{si } n = 2 \\ 7T(n/2) + 12n^2 & \text{si } n > 2 \end{cases}$$

y encontrando los valores de a, b y k que son necesarios para resolver la recurrencia con el método de la sustracción, tenemos que $a = 7$, $b = 2$ y $k = 2$, entonces tenemos que $a > b^k$ ya que $7 > 2^2$ con lo que el resultado es $\Theta(n^{\log_b(a)})$ que es $\Theta(n^{\log_2(7)}) = \Theta(n^{2,8})$.
Para resolver el algoritmo principal, deberemos contar con métodos auxiliares para resolver sumas y restas de matrices de $N \times N$ y producto de matrices de 2×2 a los que llamaremos *SumaNxN*, *RestaNxN* y *Producto2x2* respectivamente.

ALGORITMO PRODUCTOMATRICES
Entrada: A, B: matrices de $N \times N$
Salida: C: matriz de $N \times N$ resultado de $A \times B$
  si $tamanio(A) = 2$
    devolver $Producto2x2(A, B)$

sino
$\langle A_{11}, A_{12}, A_{21}, A_{22} \rangle \leftarrow Dividir(A)$
$\langle B_{11}, B_{12}, B_{21}, B_{22} \rangle \leftarrow Dividir(B)$
$M_1 \leftarrow ProductoMatrices(SumaNxN(A_{11}, A_{22}), SumaNxN(B_{11}, B_{22}))$
$M_2 \leftarrow ProductoMatrices(SumaNxN(A_{21}, A_{22}), B_{11})$
$M_3 \leftarrow ProductoMatrices(A_{11}, RestaNxN(B_{12}, B_{22}))$
$M_4 \leftarrow ProductoMatrices(A_{22}, RestaNxN(B_{21}, B_{11}))$
$M_5 \leftarrow ProductoMatrices(SumaNxN(A_{11}, A_{12}), B_{22})$
$M_6 \leftarrow ProductoMatrices(RestaNxN(A_{21}, A_{11}), SumaNxN(B_{11}, B_{12}))$
$M_7 \leftarrow ProductoMatrices(RestaNxN(A_{12}, A_{22}), SumaNxN(B_{21}, B_{22}))$
$C \leftarrow InicializarMatriz(tamanio(A))$
$C_{11} \leftarrow SumaNxN(RestaNxN(SumaNxN(M_1, M_4), M_5), M_7)$
$C_{12} \leftarrow SumaNxN(M_3, M_5)$
$C_{21} \leftarrow SumaNxN(M_2, M_4)$
$C_{22} \leftarrow SumaNxN(RestaNxN(SumaNxN(M_1, M_3), M_2), M_6)$
$C \leftarrow Componer(C_{11}, C_{12}, C_{21}, C_{22})$
devolver C
fin si

El código Java para el método principal del algoritmo entonces quedaría como sigue:

```
public static MatrizTDA<Integer> productoMatrices(
   MatrizTDA<Integer> A, MatrizTDA<Integer> B, int ai,
   int aj, int bi, int bj, int n){

   if((n)== 2){
      return productoMatrices2x2(A,B,ai,aj,bi,bj);
   }else{
      MatrizTDA<Integer> m1 = productoMatricesRec(
         sumaMatrices(A,A,ai,aj,ai+n/2,aj+n/2,n/2),
         sumaMatrices(B,B,bi,bj,bi+n/2,bj+n/2,n/2)
         ,0,0,0,0,n/2);
      MatrizTDA<Integer> m2 = productoMatricesRec(
         sumaMatrices(A,A,ai+n/2,aj,ai+n/2,aj+n/2,n/2),B
         ,0,0,bi,bj,n/2);
      MatrizTDA<Integer> m3 = productoMatricesRec(A,
         restaMatrices(B,B,bi,bj+n/2,bi+n/2,bj+n/2,n/2),
         ai,aj,0,0,n/2);
      MatrizTDA<Integer> m4 = productoMatricesRec(A,
         restaMatrices(B,B,bi+n/2,bj,bi,bj,n/2),ai+n/2,
         aj+n/2,0,0,n/2);
      MatrizTDA<Integer> m5 = productoMatricesRec(
```

```
            sumaMatrices(A,A,ai,aj,ai,aj+n/2,n/2),B,0,0,bi+
            n/2,bj+n/2,n/2);
        MatrizTDA<Integer> m6 = productoMatricesRec(
            restaMatrices(A,A,ai+n/2,aj,ai,aj,n/2),
            sumaMatrices(B,B,bi,bj,bi,bj+n/2,n/2),0,0,0,0,n
            /2);
        MatrizTDA<Integer> m7 = productoMatricesRec(
            restaMatrices(A,A,ai,aj+n/2,ai+n/2,aj+n/2,n/2),
            sumaMatrices(B,B,bi+n/2,bj,bi+n/2,bj+n/2,n/2)
            ,0,0,0,0,n/2);

        MatrizTDA<Integer> resultado = new Matriz<Integer
            >();
        resultado.inicializarMatriz(n);
        int i;
        int j;
        for(i = 0; i < n/2; i++){
            for(j = 0; j<n/2;j++){
                resultado.setearValor(i, j, m1.obtenerValor(
                    i, j)+m4.obtenerValor(i, j)-m5.
                    obtenerValor(i, j)+m7.obtenerValor(i, j))
                    ;
                resultado.setearValor(i,j+n/2,m3.
                    obtenerValor(i, j)+m5.obtenerValor(i, j))
                    ;
                resultado.setearValor(i+n/2,j,m2.
                    obtenerValor(i, j)+m4.obtenerValor(i, j))
                    ;
                resultado.setearValor(i+n/2,j+n/2,m1.
                    obtenerValor(i, j)+m3.obtenerValor(i, j)-
                    m2.obtenerValor(i, j)+m6.obtenerValor(i,
                    j));
            }
        }
        return resultado;
    }
}
```

donde los parámetros A y B son las matrices sobre las que se trabaja, los valores ai, aj, bi y bj representan la fila y columna donde comienza la matriz a considerar y n el tamaño de dicha matriz en A y B respectivamente. Este tema de trabajar con subíndices para referenciar submatrices en lugar de armar las cuatro submatrices de cada matriz se realiza para mejorar los tiempos y bajar un poco el espacio de memoria necesario. Los métodos auxiliares de suma, resta y producto2x2 reciben los mismos parámetros y con el mismo objetivo.

Capítulo 3

Greddy

3.1. Características generales

La siguiente técnica de diseño de algoritmos es la denominada *Greedy* (o por sus traducciones de algoritmos voraces, golosos, etc.). Un algoritmo Greedy es aquel que va construyendo la solución a partir de decisiones parciales basadas en la información disponible en cada momento. No mira hacia adelante, es decir que no ve los efectos de las decisiones a futuro y nunca reconsidera una decisión ya tomada. En general se utilizan para resolver problemas de optimización. Suelen ser muy eficientes pero se debe validar su eficacia, es decir que se debe tener cuidado con su correctitud. Esto significa que se debe demostrar su correctitud, es decir que se debe demostrar que encuentran la solución óptima.
Se basa en un conjunto de candidatos a formar parte de la solución. En cada paso se toma uno de los candidatos, el más apropiado y se evalúa si sirve o no, si sirve se agrega a la solución y si no se descarta. Para ello se debe poder saber en todo momento dado un candidato si está pendiente de ser evaluado, si fue evaluado y agregado a la solución o si fue descartado. Para cumplir con esto se deben conocer cuatro funciones: la función selección, que es la que selecciona el mejor candidato dentro de los pendientes, la función factibilidad que evalúa si un candidato seleccionado es factible de formar parte de la solución, la función solución que evalúa si un conjunto solución propuesto conforma la solución al problema y por último la función objetivo que es la que se debe maximizar o minimizar.
El ejemplo más sencillo de esta técnica es el que consiste en encontrar la forma de devolver un vuelto de valor v con monedas de denominaciones $d_1, d_2, ..., d_n$ con una cantidad infinita de monedas de cada denominación. Por ejemplo, queremos devolver \$1.83 y disponemos de monedas de 1, 5, 10, 25, 50 centavos y 1 peso. La forma más sencilla e intuitiva de hacerlo es una moneda de 1 peso, una moneda de 50 centavos, una moneda de 25 centavos, una moneda de 5 centavos y tres monedas de 1 centavo. Lo que hacemos en forma intuitiva es

ALGORITMO CAMBIO
Entrada: v: entero
Salida: n: entero
  entero $n \leftarrow 0$
  entero $s \leftarrow 0$
  entero $i \leftarrow 0$
  Vector $monedas = [100, 50, 25, 10, 5, 1]$
  mientras $(s < v)$ **Y** $(i < longitud(monedas))$
    si $s + monedas[i] < v$
      $s \leftarrow s + monedas[i]$
      $n \leftarrow n + 1$
    sino
      $i \leftarrow i + 1$
    fin si
  fin mientras
  si $i < longitud(monedas)$
    devolver n
  sino
    devolver No hay solución
  fin si

En este caso el conjunto de candidatos es el conjunto de monedas disponibles y se dispone de infinitas monedas de cada denominación. El conjunto de candidatos pendientes está dado por las posiciones mayores o iguales a i en el vector *monedas*. La función de selección es la que elige el elemento de la posición i sumado a que las monedas están ordenadas de mayor a menor. La función de factibilidad es la que pregunta si $s + monedas[i] < v$, la función solución es la que evalúa $s < v$ y la función objetivo, que está implícita es la que minimiza la cantidad de monedas.
Si analizamos el costo de este algoritmo, podríamos decir que está en el $\mathcal{O}(n)$ donde n es el valor de entrada. Si definimos que la cantidad de denominaciones diferentes es un valor constante dentro del problema y hacemos algunos cambios, podríamos afirmar incluso que el algoritmo está en $\mathcal{O}(1)$. Esto es así porque consideramos que las monedas están ordenadas en orden decreciente. Si éste no fuera el caso, habría que ordenar las monedas en dicho orden y el costo estaría dado por el ordenamiento que sería $\mathcal{O}(n \log n)$.
El código Java es el siguiente:

```
public static ConjuntoRepTDA<Integer> cambio(VectorTDA<
    Integer> monedas, int valor) throws Exception{

    ConjuntoRepTDA<Integer> resultado = new ConjuntoRep<
        Integer>();
```

```
        resultado.InicializarConjunto();

        int suma = 0;
        int i = 0;

        while(suma < valor){
            if(suma+monedas.recuperarElemento(i)<=valor){
                resultado.Agregar(monedas.recuperarElemento(i))
                    ;
                suma += monedas.recuperarElemento(i);
            }else{
                i++;
            }
        }

        return resultado;
    }
```

Este algoritmo es correcto para las denominaciones de monedas del ejemplo y por eso se puede considerar esta solución. Para otras instancias del problema, el algoritmo deja de ser óptimo y veremos otras técnicas para resolverlo. Por ejemplo, para las denominaciones 1, 4 y 6 y un vuelto de 8 este algoritmo da como resultado que se deben entregar 3 monedas, una de 6 y dos de 1 mientras que la mejor solución es entregar sólo dos monedas de 4.
La forma que tiene un algoritmo Greedy en general es la siguiente:

ALGORITMO GREEDY
Entrada: C: conjunto de candidatos
Salida: S solución del problema
  mientras $C \neq \emptyset$ **Y NO** $esSolucion(S)$
    $x \leftarrow Seleccionar(C)$
    $C \leftarrow C \setminus \{x\}$
    si $esFactible(S \cup \{x\})$
      $S \leftarrow S \cup \{x\}$
    fin si
  fin mientras
  si $esSolucion(S)$
    devolver S
  sino
    devolver No hay solución
  fin si

3.2. Ejemplos

Vamos a ver algunos ejemplos de problemas que se pueden resolver utilizando esta técnica.

3.2.1. Mochila

Se tienen n objetos y una mochila. Para $i = 1, 2, ..n$ el objeto i tiene un peso positivo p_i y un valor positivo v_i. La mochila puede llevar un peso que no sobrepase P. El objetivo es llenar la mochila de tal manera que se maximice el valor de los objetos transportados, respetando la limitación de capacidad impuesta. Los objetos pueden ser fraccionados, si una fracción x_i $(0 \leq x_i \leq 1)$ del objeto i es ubicada en la mochila contribuye en $x_i * p_i$ al peso total de la mochila y en $x_i \cdot v_i$ al valor de la carga. Formalmente, el problema puede ser establecido como maximizar $\sum_{i=1}^{n} x_i \cdot v_i$, con la restricción $\sum_{i=1}^{n} x_i \cdot p_i \leq P$ donde $v_i > 0$, $p_i > 0$ y $0 \leq x_i \leq 1$ para $1 \leq i \leq n$.
Por ejemplo, para la instancia $n = 3$ y $P = 20$ con un vector de valores $(v_1, v_2, v_3) = (25, 24, 15)$ y un vector de pesos $(p_1, p_2, p_3) = (18, 15, 10)$ algunas soluciones posibles son:

(x_1, x_2, x_3)	$x_i \cdot p_i$	$x_i \cdot v_i$
(1/2, 1/3, 1/4)	16.5	24.25
(1, 2/15, 0)	20	28.2
(0, 2/3, 1)	20	31
(0, 1, 1/2)	20	31.5

puede observarse que $(0, 1, 1/2)$ produce el mayor beneficio. Está claro que si la suma de todos los pesos es menor al peso máximo de la mochila, la solución consiste en incluir a todos. En caso de que esto no suceda, habrá que seleccionar cuáles son los objetos a considerar y cuáles no. Según el ejemplo anterior, y haciendo la demostración matemática de optimalidad correspondiente, se obtiene que lo mejor es ordenar los objetos de mayor a menor según la relación valor/peso de cada objeto, luego ir tomando los objetos en ese orden, considerar lo máximo que se puede tomar de cada objeto sin pasarse del peso y continuar. Eso deriva en una solución óptima.

ALGORITMO MOCHILA
Entrada: O: Vector<Objeto>, p: entero
Salida: R: Vector<real>
  $Ordenar(O)$ `//`según la razón valor/peso
  para $i = 0$ **hasta** $n - 1$

$R[i] \leftarrow 0$
fin para
$suma \leftarrow 0$
$objeto \leftarrow 0$
mientras $suma < p$
$R[objeto] \leftarrow MIN(1, (p - suma)/O[objeto].peso)$
$suma \leftarrow suma + MIN(1, (p - suma)/O[objeto], peso) * O[objeto].peso$
$objeto \leftarrow objeto + 1$
fin mientras
devolver R

El costo de este algoritmo es $\mathcal{O}(n \cdot \log n)$ respecto a la cantidad de objetos a considerar y está marcado por el costo del ordenamiento. En la implementación en Java, si en lugar del ordenamiento se utiliza una cola de prioridades para ordenar, el costo pasaría a $\mathcal{O}(n^2)$ ya que la inserción en una cola de prioridades es $\mathcal{O}(n)$ y se ejecuta n veces, una para cada objeto. Esto significa que habría que dedicarle algún tiempo a pensar mejor la forma de ordenar.
El código Java es el siguiente:

```
public static VectorTDA<Double> mochila(VectorTDA<
   Integer> o_valores, VectorTDA<Integer> o_pesos, int
   o_cantidad, int m_peso) throws Exception{

   VectorTDA<Double> resultado = new Vector<Double>();
   resultado.inicializarVector(o_cantidad);

   ColaPrioridadTDA<Integer> cola = new ColaPrioridad<
      Integer>();
   cola.InicializarCola();

   for(int i = 0; i < o_cantidad; i++){
      cola.AgregarElemento(i, (double)o_valores.
         recuperarElemento(i)/o_pesos.recuperarElemento(
         i));
   }

   double suma = 0;
   while(suma < m_peso && !cola.ColaVacia()){
      resultado.agregarElemento(cola.
         RecuperarMaxElemento(), Math.min(1,(m_peso-suma
         )/o_pesos.recuperarElemento(cola.
         RecuperarMaxElemento())));
      suma += Math.min(1,(m_peso-suma)/o_pesos.
         recuperarElemento(cola.RecuperarMaxElemento()))
```

```
            *o_pesos.recuperarElemento(cola.
            RecuperarMaxElemento());
         cola.EliminarMaxPrioridad();
      }

      return resultado;
   }
```

3.2.2. Procesos

Maximizar ganancia

Se deben procesar n tareas en un único procesador. Cada tarea se procesa en una unidad de tiempo y debe ser ejecutada en un plazo no superior a t_i. La tarea i produce una ganancia $g_i > 0$ si se procesa en un instante anterior a t_i. Una solución es factible si existe al menos una secuencia S de tareas que se ejecuten antes de sus respectivos plazos. Una solución óptima es aquella que maximiza la ganancia G tal que $G = \sum_{s \in S} g_s$.

Por ejemplo, para la instancia $n = 4$ y los siguientes valores:

$$(g_1, g_2, g_3, g_4) = (50, 10, 15, 30)$$

$$(t_1, t_2, t_3, t_4) = (2, 1, 2, 1)$$

las planificaciones que hay que considerar y los beneficios correspondientes son:

Secuencia	Beneficio
1	50
2	10
3	15
4	30
1,3	65

Secuencia	Beneficio
2,1	60
2,3	25
3,1	65
4,1	80
4,3	45

Para maximizar el beneficio en este ejemplo, se debería ejecutar la secuencia $4, 1$. La forma de encarar el algoritmo consiste en partir de una secuencia vacía e ir agregando las tareas de mayor ganancia a menor, siempre que al agregar cada tarea a la secuencia, ésta siga siendo factible. Sino es así, no se considera dicha tarea. En el ejemplo se agrega primero la tarea 1, la solución $\{1\}$ es factible. Luego se agrega la tarea 4, la solución $\{1, 4\}$ sigue siendo factible ya que la secuencia $\langle 4, 1\rangle$ lo es. Luego se intenta con la tarea 3, pero la solución $\{1, 3, 4\}$ no es factible. Luego se intenta con la tarea 2 pero la solución $\{1, 2, 4\}$ tampoco es factible, por lo tanto la solución es $\{1, 4\}$. Para decidir si la solución es factible, es conveniente mantenerla ordenada de menor a mayor por tiempo máximo de ejecución.

ALGORITMO MAXIMIZARGANANCIA
Entrada: O: Vector<tarea>
Salida: R: Vector<tarea>
  $Ordenar(O)$ //según la ganancia, de mayor a menor
  $cantidad \leftarrow 0$
  $etapa \leftarrow 1$
  $R \leftarrow inicializarVector(longitud(O))$
  $R[0] \leftarrow 0$ //agrego el primero
  mientras $etapa < longitud(O)$
    $j \leftarrow cantidad$
    mientras $j >= 0$ **Y** $O[R[j]].tiempo > MAX(O[etapa].tiempo, j)$
      $j \leftarrow j - 1$
    fin mientras
    si $O[etapa].tiempo > j$
      para $m = cantidad$ **hasta** j
        $R[m+1] \leftarrow R[m]$
      fin para
      $R[j+1] \leftarrow etapa$
      $cantidad \leftarrow cantidad + 1$
    fin si
    $etapa \leftarrow etapa + 1$
  fin mientras
  devolver R

El costo del ordenamiento según el valor de la ganancia es $\mathcal{O}(n \cdot \log n)$ y el resto es $\mathcal{O}(n^2)$ que es el orden final del algoritmo.

```
public static int planificar(VectorTDA<Integer>
   ganancias, VectorTDA<Integer> tiempos, int cantidad,
   VectorTDA<Integer> resultado) throws Exception{

   int j;
   int cantidad_secuencia = 0;
   int m;

   ColaPrioridadTDA<Integer> cola = new ColaPrioridad<
      Integer>();
   cola.InicializarCola();

   for(int i = 0; i < cantidad; i++){
      cola.AgregarElemento(i, ganancias.
         recuperarElemento(i));
   }
```

```
        resultado.agregarElemento(0, cola.
            RecuperarMaxElemento());
        cola.EliminarMaxPrioridad();

        while(!cola.ColaVacia()){
            j = cantidad_secuencia;
            while(j >= 0 && tiempos.recuperarElemento(
                resultado.recuperarElemento(j))>Math.max(
                tiempos.recuperarElemento(cola.
                RecuperarMaxElemento()), j)){
                j--;
            }
            if(tiempos.recuperarElemento(cola.
                RecuperarMaxElemento())>j){
                for(m = cantidad_secuencia; m > j; m--){
                    resultado.agregarElemento(m+1, resultado.
                        recuperarElemento(m));
                }
                resultado.agregarElemento(j+1, cola.
                    RecuperarMaxElemento());
                cantidad_secuencia++;
            }
            cola.EliminarMaxPrioridad();
        }

        return cantidad_secuencia;
    }
```

Minimizar tiempo de espera

Un procesador debe atender n procesos. Se conoce de antemano el tiempo que necesita cada uno de ellos. Determinar en qué orden el procesador debe atender dichos procesos para minimizar la suma del tiempo que los procesos están en el sistema.
Por ejemplo, para $n = 3$ se tienen, los procesos (p_1, p_2, p_3) y tiempos de proceso $(5, 10, 3)$

Orden de atención	Tiempo de espera
p_1, p_2, p_3	5 + (5+10) + (5+10+3) = 38
p_1, p_3, p_2	5 + (5+3) + (5+3+10) = 31
p_2, p_1, p_3	10 + (10+5) + (10+5+3) = 43
p_2, p_3, p_1	10 + (10+3) + (10+3+5) = 41
p_3, p_1, p_2	3 + (3+5) + (3+5+10) = 29
p_3, p_2, p_1	3 + (3+10) + (3+10+5) = 34

El ordenamiento que produce el tiempo de espera mínimo es (p_3, p_1, p_2). Es el que corresponde a ir tomando las tareas según su tiempo de ejecución, de menor a mayor.

ALGORITMO MINIMIZARESPERA
Entrada: O: Vector<tarea>
Salida: R: Vector<tarea>
Ordenar(O) //según su duración de menor a mayor
devolver O

El costo de este algoritmo, al igual que el anterior, está dado por el ordenamiento según la duración en $\mathcal{O}(n \cdot \log n)$.

```
public static VectorTDA<Integer> minimizarEspera(
   VectorTDA<Integer> tareas, int cantidad) throws
   Exception{

   VectorTDA<Integer> resultado = new Vector<Integer>();
   resultado.inicializarVector(cantidad);

   ColaPrioridadTDA<Integer> cola = new ColaPrioridad<
      Integer>();
   cola.InicializarCola();
   for(int i = 0; i < cantidad; i++){
      cola.AgregarElemento(i, tareas.recuperarElemento(i
         ));
   }

   int i = 0;
   while(!cola.ColaVacia()){
      resultado.agregarElemento(i, cola.
         RecuperarMinElemento());
      cola.EliminarMinPrioridad();
      i++;
   }

   return resultado;
}
```

3.3. Algoritmos sobre grafos

Vamos a ver a continuación varios ejemplos de problemas sobre grafos que se pueden resolver utilizando la técnica que estamos viendo.

3.3.1. Caminos mínimos: Dijkstra

El primer algoritmo que vamos ver es el problema de caminos mínimos que dice: *dado un grafo y un nodo particular del mismo, buscar el costo del camino más corto entre ese nodo y todos los demás, siempre que exista un camino.* El resultado es un grafo con los mismos nodos que el original pero sólo hay aristas entre el nodo buscado y todos los demás y el peso de la arista es el costo del camino mínimo. No devuelve el camino sino el costo. Si quisiéramos además recuperar el camino, se debería modificar el algoritmo para que lo vaya almacenando. Por este mismo motivo, si hubiera más de un camino con igual costo, no afecta al algoritmo.
Para poder ejecutar el algoritmo es necesario contar con un grafo dirigido, con todas sus aristas con peso positivo. Esto no significa que el grafo deba ser conexo, es decir, puede haber pares de nodos que no estén unidos por una arista, pero si ésta existe, debe tener un peso positivo.
El conjunto de candidatos es el conjunto de vértices del grafo, sacando el nodo de origen. En cada iteración, el método de selección será el encargado de elegir de todos los nodos pendientes de visitar, aquel que tenga el camino de menor costo desde el origen y lo marca como visitado, además de confirmar como definitivo el costo del camino mínimo desde el origen hasta dicho nodo. Para lograr esto, el algoritmo tendrá actualizados los costos del camino mínimo encontrado hasta el momento entre el origen y todos los nodos pendientes. Para comenzar, sólo tendrá como información que el costo del camino mínimo entre el nodo origen y todos los demás es el peso de la arista directa que hay en el grafo de origen desde el origen a los demás nodos. Luego, cada vez que se seleccione un nodo, se analiza si utilizando como puente dicho nodo, se puede mejorar el camino entre el origen y el resto de los pendientes. Cuando se hayan visitado todos los nodos, el algoritmo habrá finalizado.
Vamos a presentar el pseudocódigo del algoritmo para facilitar su entendimiento.

ALGORITMO DIJKSTRA
Entrada: G: Grafo<entero>, v: entero
Salida: A Grafo<entero>
`//Paso 1 //Conjunto de Vértices ya visitados`
Conjunto<entero> $Visitados \leftarrow \{v\}$
`//Paso 2 //Grafo auxiliar`
Grafo<entero> $A \leftarrow inicializarGrafo()$;
para cada $w \in Vertices(G)$
  $agregarVertice(A, w)$
fin para
para cada $v' \in Adyacentes(G, v)$
  $agregarArista(A, v, v', Peso(G, v, v'))$

fin para
`//Conjunto de Vértices pendientes de calculo`
$Pendientes \leftarrow Vertices(G) \setminus Visitados$
`//Paso 3`
mientras $Pendientes \neq \emptyset$
$w \leftarrow n : PesoArista(A, v, n) = \text{mín}\{PesoArista(A, v, p)\} \forall n, p \in Pendientes$
$Visitados \leftarrow Visitados \cup \{w\}$
$Pendientes \leftarrow Pendientes \setminus \{w\}$
$auxPendientes \leftarrow Pendientes$
mientras $auxPendientes \neq \emptyset$
$p \leftarrow elegir(auxPendientes)$
$auxPendientes \leftarrow auxPendientes \setminus \{p\}$
si $existeArista(A, v, w)$ **Y** $existeArista(G, w, p)$
si $existeArista(A, v, p)$
si $pesoArista(A, v, w) + pesoArista(G, w, p) < pesoArista(A, v, p)$
$agregarArista(A, v, p, pesoArista(A, v, w) + pesoArista(G, w, p)$
fin si
sino
$agregarArista(A, v, p, pesoArista(A, v, w) + pesoArista(G, w, p)$
fin si
fin si
fin mientras
fin mientras
devolver A

El resultado de aplicar el algoritmo de *Dijkstra* sobre el grafo de ejemplo de la figura 3.1 es el que se ve en la figura 3.2.
Modificando el algoritmo para que devuelva los caminos mínimos en lugar del costo, el código es el siguiente:

ALGORITMO DIJKSTRA
Entrada: G: grafo de origen, v: nodo origen
Salida: A: grafo solución, R: grafo con los caminos mínimos
//Paso 1
//Conjunto de Vértices ya visitados
$Visitados \leftarrow \{v\}$
//Paso 2
//Grafo auxiliar

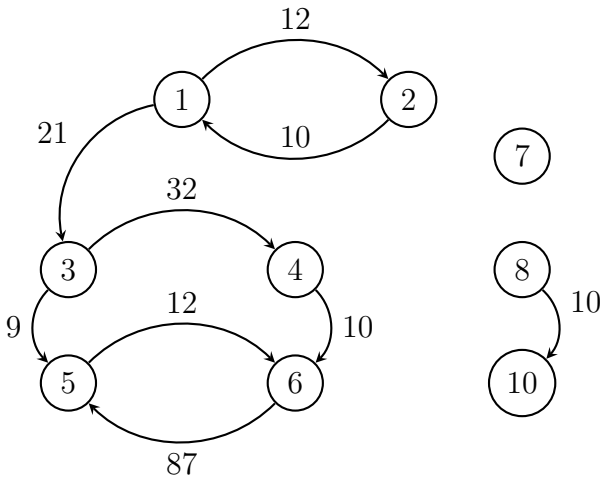


Figura 3.1: Ejemplo de grafo

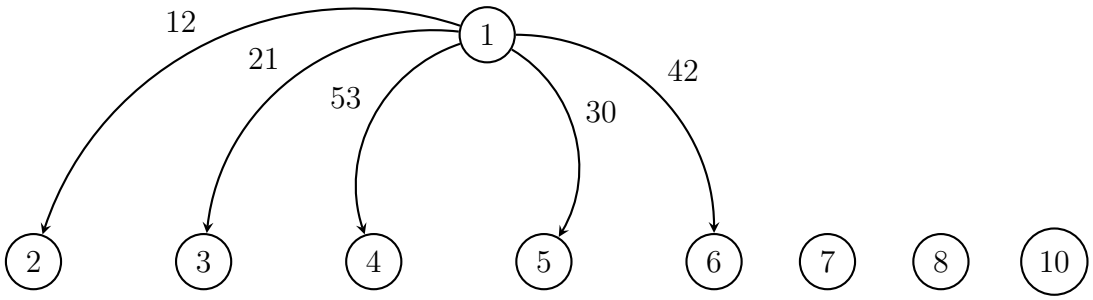


Figura 3.2: Resultado del algoritmo de Dijkstra

```
inicializar(A);
inicializar(R);
para cada w ∈ Vertices(G)
  agregarVertice(A, w)
  agregarVertice(R, w)
fin para
para cada v' ∈ Adyacentes(G, v)
  agregarArista(A, v, v', Peso(G, v, v'))
  agregarArista(R, v, v', Peso(G, v, v'))
fin para
//Conjunto de Vértices pendientes de calculo
Pendientes ← Vertices(G) \ Visitados
// Paso 3
mientras Pendientes ≠ ∅
  w ← n : PesoArista(A, v, n) = mín{PesoArista(A, v, p)}∀n, p ∈
  Pendientes
  Visitados ← Visitados ∪ {w}
  Pendientes ← Pendientes \ {w}
  auxPendientes ← Pendientes
  mientras auxPendientes ≠ ∅
    p ← elegir(auxPendientes)
    auxPendientes ← auxPendientes \ {p}
    si existeArista(A, v, w) Y existeArista(G, w, p)
      si existeArista(A, v, p)
        si pesoArista(A, v, w) + pesoArista(G, w, p) <
        pesoArista(A, v, p)
          agregarArista(A, v, p, pesoArista(A, v, w) +
          pesoArista(G, w, p)
          eliminarArista(R, x, p)∀x ∈ Vertices(R)
          agregarArista(R, w, p, pesoArista(G, w, p))
        fin si
      sino
        agregarArista(A, v, p, pesoArista(A, v, w) + pesoArista(G, w, p)
        agregarArista(R, w, p, pesoArista(G, w, p))
      fin si
    fin si
  fin mientras
fin mientras
devolver A
```

El resultado de aplicar el algoritmo de *Dijkstra* modificado sobre el grafo de ejemplo de la figura 3.1 es el que se ve en la figura 3.3.

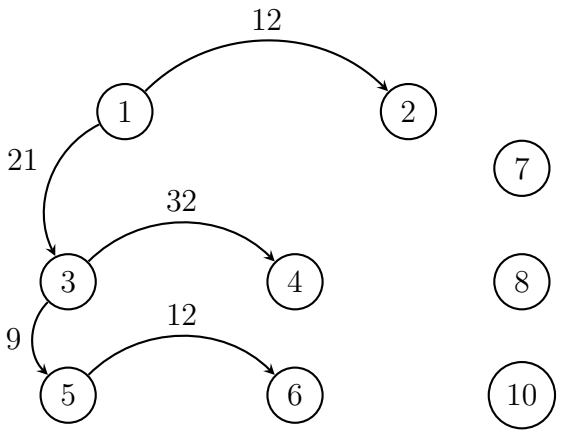


Figura 3.3: Resultado del algoritmo de Dijkstra modificado

El código resultante, utilizando nuestros TDA es el siguiente:

```
public static GrafoDirTDA<Integer> dijkstra(GrafoDirTDA<
    Integer> g, int origen){
    int vertice, aux_vertice, mejor_vertice,
        mejor_distancia;

    GrafoDirTDA<Integer> distanciasMinimas = new GrafoDir
        <Integer>();
    distanciasMinimas.InicializarGrafo();

    distanciasMinimas.AgregarVertice(origen);

    ConjuntoTDA<Integer> vertices = g.Vertices();
    vertices.sacar(origen);

    while(!vertices.conjuntoVacio()){
        vertice = vertices.elegir();
        vertices.sacar(vertice);
        distanciasMinimas.AgregarVertice(vertice);
        if(g.ExisteArista(origen, vertice)){
            distanciasMinimas.AgregarArista(origen, vertice
                , g.PesoArista(origen, vertice));
        }
    }

    ConjuntoTDA<Integer> pendientes = g.Vertices();
    pendientes.sacar(origen);

    ConjuntoTDA<Integer> aux_pendientes = new Conjunto<
```

```
        Integer>();
    aux_pendientes.inicializarConjunto();


    while(!pendientes.conjuntoVacio()){

        mejor_distancia = 0;
        mejor_vertice = 0;
        while(!pendientes.conjuntoVacio()){
            aux_vertice = pendientes.elegir();
            pendientes.sacar(aux_vertice);
            aux_pendientes.agregar(aux_vertice);
            if((distanciasMinimas.ExisteArista(origen,
                aux_vertice) && (mejor_distancia == 0 || (
                mejor_distancia > distanciasMinimas.
                PesoArista(origen, aux_vertice))))){
                mejor_distancia = distanciasMinimas.
                    PesoArista(origen, aux_vertice);
                mejor_vertice = aux_vertice;
            }
        }

        vertice = mejor_vertice;

        if(vertice!= 0){
            aux_pendientes.sacar(vertice);

            while(!aux_pendientes.conjuntoVacio()){
                aux_vertice = aux_pendientes.elegir();
                aux_pendientes.sacar(aux_vertice);
                pendientes.agregar(aux_vertice);

                if(g.ExisteArista(vertice, aux_vertice)){
                    if(!distanciasMinimas.ExisteArista(origen
                        , aux_vertice)){
                        distanciasMinimas.AgregarArista(origen
                            , aux_vertice, distanciasMinimas.
                            PesoArista(origen, vertice)+g.
                            PesoArista(vertice, aux_vertice));
                    }else{
                        if(distanciasMinimas.PesoArista(origen
                            ,aux_vertice)> distanciasMinimas.
                            PesoArista(origen, vertice)+g.
                            PesoArista(vertice, aux_vertice)){
                            distanciasMinimas.AgregarArista(
                                origen, aux_vertice,
```

```
                            distanciasMinimas.PesoArista(
                            origen, vertice)+g.PesoArista(
                            vertice, aux_vertice));
                    }
                }
            }
        }
    }

    return distanciasMinimas;
}
```

Si analizamos el costo de este algoritmo, vemos dos bucles anidados con lo cual podríamos estar hablando de un costo de $\mathcal{O}(n^2)$ salvo que el costo de trabajar con conjuntos para agregar o eliminar elementos tenemos un $\mathcal{O}(n)$, que dentro de los dos bucles, puede transformar el algoritmo en $\mathcal{O}(n^3)$. Se podrían buscar mejores estrategias y estructuras para representar los datos para poder evitar este problema y dejar el algoritmo en su costo original de $\mathcal{O}(n^2)$.

3.3.2. Arboles de cubrimiento mínimo

Vamos a introducir algunos conceptos nuevos sobre grafos para entender el próximo problema. En esta sección trabajaremos con grafos no dirigidos. Llamaremos *árbol de cubrimiento* de un grafo $G(V, A)$ a un grafo $G'(V, A')$ que contiene los mismos vértices que G y un subconjunto de las aristas que además forma un árbol, es decir que G' es un grafo conexo y acíclico. Llamaremos *árbol de cubrimiento mínimo* al árbol de cubrimiento que tenga el menor costo de todos, es decir que la suma de sus aristas sea la mínima. Para algunos grafos, puede existir más de un árbol de cubrimiento mínimo, lo que no puede ser más de uno es el costo de dicho árbol.
Por ejemplo, en la figura 3.4 se puede ver un grafo y a la derecha su árbol de cubrimiento mínimo correspondiente.
Buscaremos la forma de encontrar el árbol de cubrimiento mínimo de un grafo. Este problema se puede ver en la práctica para por ejemplo hacer un tendido de rutas entre ciudades de tal forma que queden todas conectadas entre sí y que la cantidad de kilómetros construida sea la mínima. O para extender algún tipo de cableado para intercomunicar un conjunto de ciudades.
Para resolver el problema tendremos que determinar cuál es el conjunto de candidatos que podrán ser los nodos o las aristas, la función de selección y la función de factibilidad que tendrá que definir en principio que no haya ciclos, además de la función solución que determina si el resultado obtenido hasta el momento es un árbol de cubrimiento. La función objetivo es minimizar la suma de los costos de las aristas de la solución.

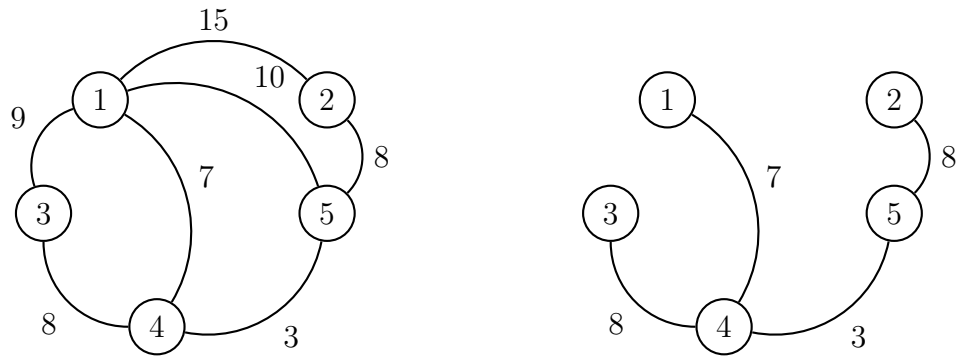


Figura 3.4: Arbol de cubrimiento mínimo

Prim

La primera estrategia para hallar el árbol de cubrimiento mínimo consiste en ir armando un árbol a partir de un vértice cualquiera del grafo, e ir agregando vértices hasta completar todos los vértices. La estrategia para ir agregando vértices es ir agregando el vértice que, no estando agregado aún, sea el que tenga la arista de costo mínimo con el otro extremo entre los vértices ya elegidos.
Por ejemplo, en la figura 3.5 se puede ver un grafo y la secuencia de decisiones tomadas por el algoritmo para llegar a la solución.

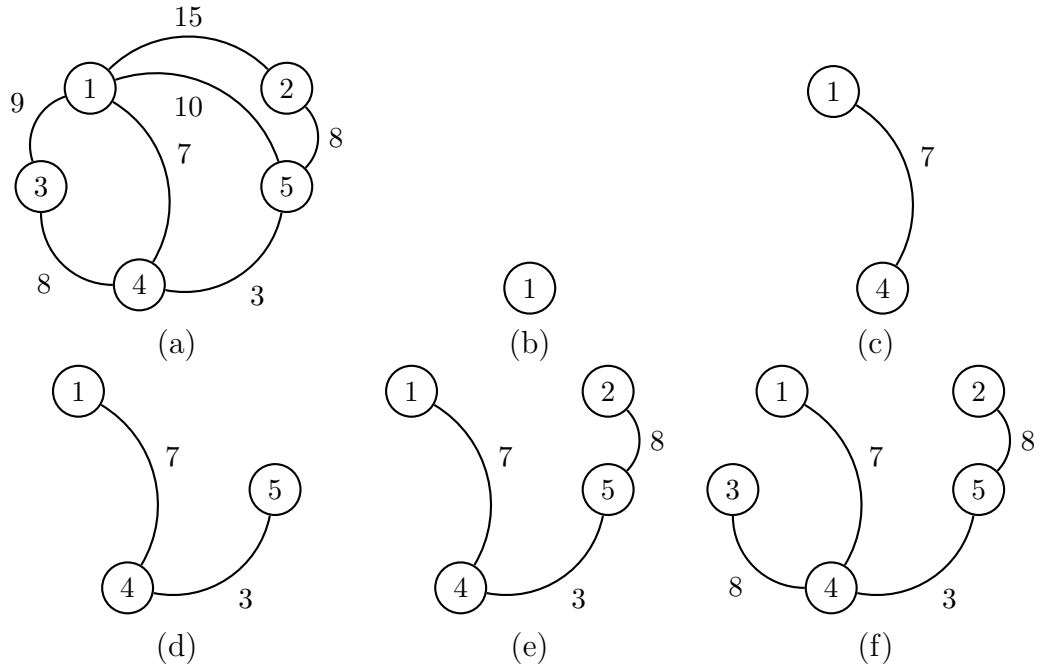


Figura 3.5: Algoritmo de Prim

Como estrategia de implementación, en el grafo resultado se mantendrá para los vértices ya visitados, las aristas correspondientes al resultado parcial, para los vértices no visitados, la menor de todas las aristas que lo unen a un vértice ya visitado. Con esta última mejora, evitamos recorrer todas las aristas para cada vértice pendiente, y esto hace bajar bastante el costo del algoritmo.

ALGORITMO PRIM
Entrada: G: grafo de origen
Salida: R: grafo solución
inicializar(R)
para cada $v \in Vertices(G)$
agregarVertice(R, v)
fin para
$vertice \leftarrow elegir(Vertices(R))$
$pendientes \leftarrow Vertices(R) \setminus \{vertice\}$
para cada $w \in Adyacentes(G, vertice)$
agregarArista$(R, vertice, w, pesoArista(G, vertice, w))$
fin para
mientras $pendientes \neq \emptyset$
$vertice = v : pesoArista(v, v_i) = \text{mín}(pesoArista(v_1, v_2) : \forall v_1 \in pendientes, v_2 \notin pendientes$
sacar$(pendientes, vertice)$
para cada $p \in pendientes$
si *existeArista*$(G, p, vertice)$ **Y NO** *existeArista*$(R, p, x) \forall x$
agregarArista$(R, p, vertice, pesoArista(G, p, vertice))$
sino
si *existeArista*$(G, p, vertice)$ **Y NO** $pesoArista(R, p, x) > pesoArista(G, p, vertice)$
eliminarArista(R, p, x)
agregarArista$(R, p, vertice, pesoArista(G, p, vertice))$
fin si
fin si
fin para
fin mientras
devolver R

El costo de este algoritmo está dado por el doble bucle por los vértices del grafo, es decir que estamos hablando de un $\mathcal{O}(n^2)$ sobre la cantidad de vértices.
El código Java es el siguiente:

```
public static GrafoTDA<Integer> prim(GrafoTDA<Integer> g
    ){
```

```
int vertice, aux_vertice, mejor_vertice,
   mejor_distancia;

GrafoTDA<Integer> resultado = new Grafo<Integer>();
resultado.InicializarGrafo();

ConjuntoTDA<Integer> vertices = g.Vertices();

vertice = vertices.elegir();
vertices.sacar(vertice);
resultado.AgregarVertice(vertice);

while(!vertices.conjuntoVacio()){
   aux_vertice = vertices.elegir();
   vertices.sacar(aux_vertice);
   resultado.AgregarVertice(aux_vertice);
   if(g.ExisteArista(aux_vertice, vertice)){
      resultado.AgregarArista(aux_vertice, vertice, g
         .PesoArista(aux_vertice, vertice));
   }
}

ConjuntoTDA<Integer> pendientes = g.Vertices();
pendientes.sacar(vertice);

ConjuntoTDA<Integer> aux_pendientes = new Conjunto<
   Integer>();
aux_pendientes.inicializarConjunto();

while(!pendientes.conjuntoVacio()){

   mejor_distancia = 0;
   mejor_vertice = 0;

   while(!pendientes.conjuntoVacio()){
      aux_vertice = pendientes.elegir();
      pendientes.sacar(aux_vertice);
      aux_pendientes.agregar(aux_vertice);
      if((!resultado.Adyacentes(aux_vertice).
         conjuntoVacio()) && (mejor_distancia == 0 ||
          (mejor_distancia > resultado.PesoArista(
         aux_vertice, resultado.Adyacentes(
         aux_vertice).elegir())))){
         mejor_distancia = resultado.PesoArista(
            aux_vertice, resultado.Adyacentes(
```

```
                aux_vertice).elegir());
            mejor_vertice = aux_vertice;
        }
    }

    vertice = mejor_vertice;
    aux_pendientes.sacar(vertice);

    // actualizo la mejor distancia de todos los
       pendientes
    while(!aux_pendientes.conjuntoVacio()){
        aux_vertice = aux_pendientes.elegir();
        aux_pendientes.sacar(aux_vertice);
        pendientes.agregar(aux_vertice);

        if(g.ExisteArista(aux_vertice, vertice)){
            if(resultado.Adyacentes(aux_vertice).
               conjuntoVacio()){
               resultado.AgregarArista(aux_vertice,
                  vertice, g.PesoArista(aux_vertice,
                  vertice));
            }else{
               if(resultado.PesoArista(aux_vertice,
                  resultado.Adyacentes(aux_vertice).
                  elegir())> g.PesoArista(aux_vertice,
                  vertice)){
                  resultado.EliminarArista(aux_vertice,
                     resultado.Adyacentes(aux_vertice).
                     elegir());
                  resultado.AgregarArista(aux_vertice,
                     vertice, g.PesoArista(aux_vertice,
                     vertice));
               }
            }
        }
    }

    return resultado;
}
```

Kruskal

El algoritmo de kruskal se basa en ir armando árboles pequeños e ir uniéndolos hasta obtener un único árbol. Se comienza armando un árbol con único nodo por cada vértice. Luego se toman las aristas del grafo original, ordenadas por su peso, de menor a mayor. Para cada arista, si tiene cada extremo en un árbol diferente, se agrega dicha arista al resultado y se unen los dos árboles que contienen sus extremos. Para mantener el conjunto de árboles, se deberá utilizar una estructura acorde, que su idea básica es mantener una estructura de conjuntos disjuntos. Para simplificar la implementación, mantendremos un hash que para cada vértice indica en qué conjunto se encuentra, comenzando con una inicialización en la que cada vértice está en su propio conjunto.

ALGORITMO KRUSKAL
Entrada: G: grafo de origen
Salida: R: grafo solución
  $inicializar(R)$
  $E \leftarrow inicializarConjuntos()$
  $cantidadVertices \leftarrow 0$
  para cada $v \in Vertices(G)$
    $agregarVertice(R, v)$
    $E[v] = v$
    $cantidadVertices \leftarrow cantidadVertices + 1$
  fin para
  $C \leftarrow inicializarColaPrioridad()$
  para cada $v1 \in Vertices(G)$
    para cada $v2 \in Vertices(G)$
      si $existeArista(G, v1, v2)$ **Y NO** $existe(C, Arista(v1, v2))$
        $agregar(C, Arista(v1, v2), pesoArista(G, v1, v2))$
      fin si
    fin para
  fin para
  mientras $cantidadVertices > 1$
    $a \leftarrow menor(C)$
    $eliminarMenor(C)$
    si $E[a.origen] <> E[a.destino]$
      $cantidadVertices \leftarrow cantidadVertices - 1$ //unifico los conjuntos
      para cada $v \in E$
        si $E[v] = E[a.destino]$
          $E[v] = E[a.origen]$
        fin si
      fin para

```
        agregarArista(R, a.origen, a.destino, a.peso)
    fin si
fin mientras
devolver R
```

Por ejemplo, en la figura 3.6 se puede ver un grafo y la secuencia de decisiones tomadas por el algoritmo para llegar a la solución. Se puede ver que a diferencia de Prim, que va agregando vértices, este algoritmo parte con todos los vértices y va agregando aristas.

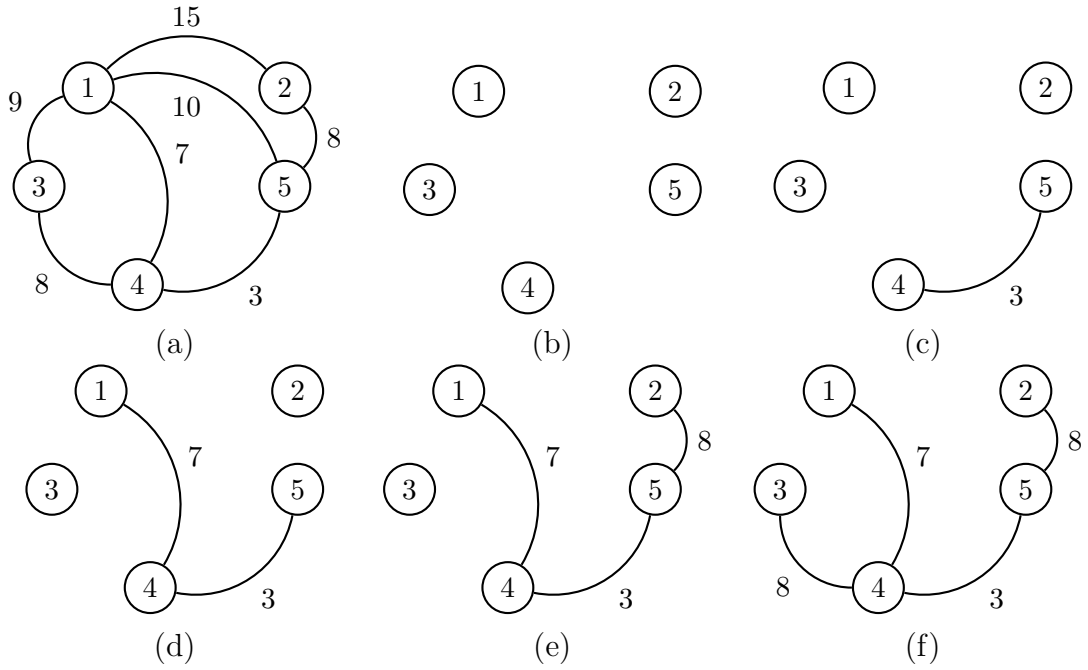


Figura 3.6: Algoritmo de Kruskal

El costo de este algoritmo está dado por el costo de ordenamiento de las aristas del grafo, es decir que estamos hablando de un $\mathcal{O}(a \cdot \log a)$ sobre la cantidad de aristas, ya que si bien existen un par de bucles anidados, estos en conjunto dan como resultado un $\mathcal{O}(n)$ ya que la cantidad de ejecuciones se complementan entre sí.
Si queremos comparar las dos soluciones a este problema, vemos que según la cantidad de aristas que haya en el grafo, una solución puede ser mejor que la otra. La cantidad de aristas puede variar entre la cantidad de vértices y la cantidad de vértices al cuadrado. Si la cantidad de aristas es cercana a la cantidad de vértices, podríamos decir que Kruskal está en $\mathcal{O}(n \cdot \log n)$ que es mejor que $\mathcal{O}(n^2)$ de Prim. Sin embargo, si la cantidad de aristas está más cercana a v^2, tenemos que Kruskal está cercano a $\mathcal{O}(n^2 \cdot \log n^2) = \mathcal{O}(n^2 \cdot \log n)$ que es peor que el $\mathcal{O}(n^2)$ de Prim.

El código Java es el siguiente:

```
public static GrafoTDA<Integer> kruskal(GrafoTDA<Integer
   > g) throws Exception{

   int vertice, cantidad = 0, i, aux_vertice, c1=0, c2
      =0, cantidad_pendientes;
   Par aux_par;
   ConjuntoTDA<Integer> aux_adyacentes;

   GrafoTDA<Integer> resultado = new Grafo<Integer>();
   resultado.InicializarGrafo();

   ConjuntoTDA<Integer> vertices = g.Vertices();

   while(!vertices.conjuntoVacio()){
      vertice = vertices.elegir();
      vertices.sacar(vertice);
      resultado.AgregarVertice(vertice);
      cantidad++;
   }

   // en el par, valor1 y valor2 son los vértices y
      prioridad es el peso
   ColaPrioridadTDA<Par> aristas = new ColaPrioridad<Par
      >();
   aristas.InicializarCola();

   // en el par, valor1 es el vertice y valor2 el
      conjunto
   VectorTDA<Par> conjuntos = new Vector<Par>();
   conjuntos.inicializarVector(cantidad);

   vertices = g.Vertices();
   i = 0;
   while(!vertices.conjuntoVacio()){
      vertice = vertices.elegir();
      vertices.sacar(vertice);
      aux_par = new Par();
      aux_par.valor1 = vertice;
      aux_par.valor2 = vertice;
      conjuntos.agregarElemento(i, aux_par);
      i++;
      aux_adyacentes = g.Adyacentes(vertice);
      while(!aux_adyacentes.conjuntoVacio()){
         aux_vertice = aux_adyacentes.elegir();
```

```
            aux_adyacentes.sacar(aux_vertice);
            if(aux_vertice > vertice){// este if es para no
                agregar (2,3) y (3,2)
               aux_par = new Par();
               aux_par.valor1 = vertice;
               aux_par.valor2 = aux_vertice;
               aristas.AgregarElemento(aux_par, g.
                  PesoArista(vertice, aux_vertice));
            }
         }
      }

      cantidad_pendientes = cantidad;
      // mientras haya más de un conjunto
      while(cantidad_pendientes>1){
         // tomo la arista más liviana
         aux_par = aristas.RecuperarMinElemento();

         // busco a qué conjunto pertenece cada vertice de
            la arista
         for(i = 0; i < cantidad; i++){
            if(conjuntos.recuperarElemento(i).valor1 ==
               aux_par.valor1){
               c1 = conjuntos.recuperarElemento(i).valor2;
            }
            if(conjuntos.recuperarElemento(i).valor1 ==
               aux_par.valor2){
               c2 = conjuntos.recuperarElemento(i).valor2;
            }
         }

         // si pertenecen a distintos conjuntos
         if(c1 != c2){
            //unifico los conjuntos
            for(i = 0;i<cantidad;i++){
               if(conjuntos.recuperarElemento(i).valor2 ==
                  c2){
                  conjuntos.recuperarElemento(i).valor2 =
                     c1;
               }
            }
            //agrego la arista al resultado
            resultado.AgregarArista(aux_par.valor1, aux_par
               .valor2, (int)aristas.RecuperarMinPrioridad
               ());
            cantidad_pendientes--;
```

```
            }
            aristas.EliminarMinPrioridad();
        }

        return resultado;
    }
```

Capítulo 4

Programación Dinámica

4.1. Características generales

En este capítulo nos centraremos en analizar la técnica de diseño conocida como *Programación Dinámica*. Esta técnica se basa en resolver problemas en función de subproblemas más pequeños del mismo. Ya hemos analizado una técnica con estas características, Divide y Conquista, que se basaba en la independencia de los subproblemas. Si los subproblemas no eran independientes, la eficiencia podía sufrir un aumento importante. En el caso de la programación dinámica, los problemas no deberían ser independientes, justamente se basa en subproblemas comunes para lo cual se resuelven los subproblemas y se almacena la solución para utilizarla nuevamente cuando sea necesario. Se utiliza en general para problemas de optimización.
Esta técnica basa su funcionamiento en la aplicación del *principio de optimalidad* que dice que *una solución puede ser óptima sólo si está basada en soluciones parciales también óptimas*. Si este principio no se puede aplicar para un determinado problema, no será posible entonces aplicar esta técnica.
Veamos un ejemplo sencillo para entender de qué estamos hablando. La función matemática de Fibonacci es $f : \mathbb{N} \cup \{0\} \to \mathbb{N}$ tal que

$$F(n) = \begin{cases} 1 & \text{si } n < 2 \\ F(n-1) + F(n-2) & \text{sino} \end{cases}$$

Si escribimos un algoritmo recursivo para resolverlo, probablemente tenga la forma de un algoritmo divide y conquista, tendríamos algo de $\mathcal{O}(2^n)$. Es algo muy ineficiente, y claramente el problema está en que se calcula muchas veces un mismo valor. Por ejemplo:

$$F(5) = \underbrace{F(4) + F(3)}_{F(5)} = \underbrace{\underbrace{F(3) + F(2)}_{F(4)} + \underbrace{F(2) + F(1)}_{F(3)}}_{F(5)} =$$

$$= \underbrace{\underbrace{\underbrace{F(2)}_{F(3)} + F(1)}_{F(4)} + \underbrace{\underbrace{F(1) + F(0)}_{F(2)}}_{F(4)} + \underbrace{\underbrace{F(1) + F(0)}_{F(2)}}_{F(3)} + F(1)}_{F(5)}$$

$$= \underbrace{\underbrace{\underbrace{F(1) + F(0)}_{F(2)} + F(1)}_{F(3)} + \underbrace{\underbrace{F(1) + F(0)}_{F(2)}}_{F(4)} + \underbrace{\underbrace{F(1) + F(0)}_{F(2)} + F(1)}_{F(3)}}_{F(5)}$$

Como podemos ver, $F(3)$ se está calculando dos veces y $F(2)$ se está calculando tres veces. Si en lugar de calcularlo varias veces, vamos almacenando el valor calculado, podríamos evitar muchos cálculos. Entonces almacenamos en una tabla los valores:

$F(0)$	$F(1)$	$F(2)$	$\cdots$	$F(n)$

Podemos plantear un algoritmo como sigue:

ALGORITMO FIBBONACI
Entrada: n: entero
Salida: r: entero
  si $n < 2$
    devolver 1
  sino
    Vector<entero> $tabla \leftarrow inicializarVector(n)$
    $tabla[0] \leftarrow 1$
    $tabla[1] \leftarrow 1$
    para $i \leftarrow 2$ **hasta** n
      $tabla[i] \leftarrow tabla[i-1] + tabla[i-2]$
    fin para
    devolver $tabla[n]$
  fin si

Si analizamos el costo vemos que es $\mathcal{O}(n)$, pero también tenemos que analizar el espacio que ocupa, que también está en $\mathcal{O}(n)$. Entonces otro factor importante a tener en cuenta cuando se usa esta técnica es la evaluación del espacio necesario para almacenar los resultados parciales.

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

```
    }else{
        VectorTDA<Integer> aux = new Vector<Integer>();
        aux.inicializarVector(n+1);
        aux.agregarValor(0,1);
        aux.agregarValor(1,1);
        for(int i = 2; i <=n;i++){
            aux.agregarValor(i,aux.recuperarValor(i-2)+aux.
                recuparar Valor(i-1));
        }
    }
    return aux.recuparar Valor(n);
}
```

Si comparamos esta técnica con Divide y Conquista, vemos que una resuelve el problema en forma recursiva, de forma top down, mientras que la otra la resuelve en forma iterativa en forma bottom up. Una requiere que los subproblemas sean independientes y la otra que se solapen. Si comparamos la técnica con los algoritmos greedy, vemos que uno se basa en iteraciones donde se hace una selección voraz, con los datos disponibles hasta el momento y nunca revisa decisiones tomadas mientras que la otra siempre selecciona lo mejor de lo calculado anteriormente para armar la solución.
Los pasos a seguir para plantear la solución son:

- Plantear la solución como una decisión entre soluciones menores
- Plantear la solución en forma recursiva
- Llenar la tabla de soluciones parciales de forma bottom up
- Reconstruir el resultado a partir de la tabla

4.2. Ejemplos

Vamos a analizar varios problemas que pueden ser resueltos con programación dinámica.

4.2.1. Cambio

El problema del cambio, que ya analizamos con un algoritmo greedy, vimos que no puede ser resuelto sin reconsiderar las alternativas elegidas. Vamos a plantear una solución basada en programación dinámica.
Tenemos un conjunto de n denominaciones de monedas y debemos alcanzar un valor v a devolver. Las soluciones parciales serán encontrar la cantidad mínima de monedas necesarias para devolver valores menores con la misma cantidad de

denominaciones, y la forma de devolver valores menores con menos denominaciones. Entonces para encontrar la solución de la cantidad de monedas para devolver un valor $j (j \leq v)$ con $i (i \leq n)$ denominaciones será la menor cantidad entre **a)** la mejor forma de devolver el mismo valor j con $i - 1$ denominaciones, es decir, no considero la última denominación que estoy analizando y **b)** la mejor forma de devolver j menos el valor de la denominación j más una moneda de la denominación j. Es decir si agrego una moneda de la denominación j, tengo que ver cuántas monedas necesitaba para devolver $j - valor(d_i)$ y sumarle uno. Si en cambio no agrego una moneda de la denominación que estoy analizando, tomo la cantidad de monedas que necesitaba para devolver el mismo valor j sin considerar la denominación i.
Para memorizar todos estos valores, se deberá construir una tabla con n filas, una para cada denominación y $v + 1$ columnas, una para cada valor posible entre 0 y v. En cada posición (i, j) de la tabla estará la menor cantidad de monedas necesarias para devolver un valor j considerando sólo i denominaciones, desde la 1 hasta la i. La solución al problema será el valor que está almacenado en la posición (n, v) de la tabla.
Los casos excepcionales que caen fuera del caso genérico son cuando alguno de los dos valores a comparar o los dos caen fuera de la tabla. Si el valor a devolver es 0, entonces claramente la cantidad de monedas es 0. Si no podemos verificar ni **a)** ni **b)**, es decir si estamos analizando la primera denominación, y el valor de ésta, excede el valor a devolver que estoy considerando, no tenemos solución y podríamos indicarlo con un $+\infty$, si no podemos verificar **a)** tomamos directamente el valor de **b)** más 1, si en cambio no podemos verificar **b)** tomamos directamente el valor de **a)**.
El planteo recursivo es entonces:

$$C[i,j] \in \begin{cases} 0 & \text{si } j = 0 \\ +\infty & \text{si } i = 1 \text{ y } d_i > j \\ 1 + C[i, j - d_i] & \text{si } i = 1 \text{ y} d_i \leq j \\ c[i-1, j] & \text{si } i > 1 \text{ y } d_i > j \\ \text{mín}(1 + C[i, j - d_i], c[i-1, j]) & \text{sino} \end{cases}$$

Como para resolver una posición de la tabla es necesario conocer los valores de la misma fila hacia la izquierda y de la misma columna hacia arriba, el orden razonable para llenar la tabla es haciendo un recorrido por filas y luego por columnas. Para el ejemplo de devolver 8 centavos con monedas de 1, 4 y 6, la tabla quedaría como la de la figura 4.1. Se puede ver con diferentes colores las diferentes opciones por las que se llenó cada posición y el resultado se puede ver en la intersección de la fila de la moneda 6 con el valor 8, que da como resultado 2 monedas.
El algoritmo para llenar la tabla sería el siguiente:

ALGORITMO CAMBIO
Entrada: d: Vector<entero>, v: entero
Salida: c: entero

```
M ← inicializarMatriz(máx(longitud(d), v + 1))
para i ← 0 hasta longitud(d) − 1
    M[i, 0] ← 0
    para j ← 1 hasta v
        si i = 0
            si d[j] > j
                M[i, j] ← ∞
            sino
                M[i, j] ← M[i, j − d[j]]
            fin si
        sino
            si d[j] > j
                M[i, j] ← M[i − 1, j]
            sino
                M[i, j] ← mín(M[i − 1, j], M[i, j − d[j]])
            fin si
        fin si
    fin para
fin para
devolver M[longitud(i) − 1, v]
```

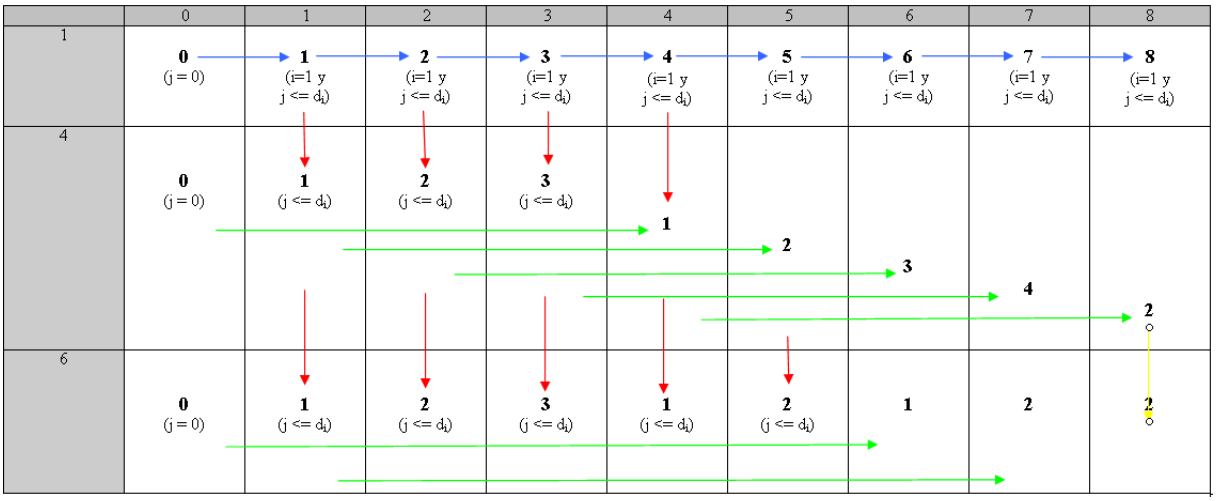


Figura 4.1: Tabla para algoritmo de devolución de cambio

El código Java quedaría así:

```
public static int cambio(VectorTDA<Integer> monedas, int
    valor) throws Exception{

   int i, j;
   int cantidadMonedas = monedas.capacidadVector();

   int n = Math.max(cantidadMonedas, valor+1);
   MatrizTDA<Integer> m = new Matriz<Integer>();
   m.inicializarMatriz(n);

   for(i = 0; i < cantidadMonedas;i++){
      for(j = 0; j <= valor; j++){
         if(i == 0 && j < monedas.recuperarElemento(0)){
            m.setearValor(i, j, 0);
         }else{
            if(i == 0){
               m.setearValor(i, j, 1+m.obtenerValor(i,j-
                  monedas.recuperarElemento(i)));
            }else{
               if(j<monedas.recuperarElemento(i)){
                  m.setearValor(i, j, m.obtenerValor(i
                     -1, j));
               }else{
                  if(m.obtenerValor(i-1, j)>0){
                        m.setearValor(i, j, Math.min(m.
                           obtenerValor(i-1, j), 1+m.
                           obtenerValor(i, j-monedas.
                           recuperarElemento(i))));
                  }else{
                     m.setearValor(i, j, 1+m.
                        obtenerValor(i, j-monedas.
                        recuperarElemento(i)));
                  }
               }
            }
         }
      }
   }
   return m.obtenerValor(cantidadMonedas-1, valor);
}
```

El costo de este algoritmo estará dado por el tamaño de la matriz que será $\mathcal{O}(cantidadMonedas \cdot valor)$. Se debe destacar que también ésta es la complejidad espacial del algoritmo, a diferencia de las otras técnicas que no tienen un alto costo en espacio.
Si quisiéramos saber qué monedas son las que componen la solución, basta con

recorrer la tabla en orden inverso partiendo de la última posición y analizando desde dónde se llegó hasta allí. Por ejemplo partiendo de la posición $M[3,8]$ se pasa a la $M[2,8]$, $M[2,4]$, $M[2,0]$. Movimiento a la izquierda significa que se toma una moneda de esa denominación, movimiento hacia arriba significa que no.

4.2.2. Mochila

Recordemos el problema de la mochila: Se tienen n objetos y una mochila. Para $i = 1, 2, ..n$, el objeto i tiene un peso positivo p_i y un valor positivo v_i. La mochila puede llevar un peso que no sobrepase P. Los objetos no pueden ser fraccionados. El objetivo es llenar la mochila de manera tal que se maximice el valor de los objetos transportados, respetando la limitación de capacidad impuesta. El algoritmo debe determinar el valor máximo que se podrá cargar. La diferencia con la versión anterior, es que los objetos no se pueden fraccionar. Eliminamos esta restricción que habíamos puesto en la versión anterior para poder resolverlo con la técnica *Greedy*.
Las instancias intermedias sobre las que basaremos nuestra solución serán considerando menos objetos y/o menos peso máximo. Cuando hablemos de $V(o,p)$ hablaremos del máximo valor que se puede obtener considerando hasta el objeto o, con $1 \leq o \leq n$ y un peso máximo de p con $0 \leq p \leq P$. Considerar un nuevo objeto para una solución parcial, puede llevarse a cabo de dos formas, considerar que dicho objeto realmente se agrega a la mochila o que no se agrega, en cuyo caso el valor alcanzado no se agrega. Ahora si se decide agregarlo, el valor alcanzado será el dado por la suma del valor del objeto o a agregar más el valor calculado previamente sin considerar el objeto seleccionado (éste valor previo es el obtenido para un peso p' dado por el peso p menos la resta del peso del objeto o seleccionado). Si una solución parcial es óptima, la nueva solución, calculada como el mejor de las dos opciones (agregar o no agregar el nuevo objeto al a mochila) también lo será. Entonces, en la matriz tendremos una fila por cada objeto y una columna por cada peso posible entre cero y el máximo peso de la mochila.
La función recursiva queda entonces como sigue:

$$V[o,p] \in \begin{cases} 0 & \text{si } o = 1 \text{ y } peso[o] > p \\ valor[o] & \text{si } o = 1 \text{ y } peso[o] \leq p \\ V[o-1,p] & \text{si } o > 1 \text{ y } peso[o] > p \\ \text{máx}(V[o-1,p], V[o-1,p-peso[o]] + valor[o]) & \text{sino} \end{cases}$$

La solución será $V[O,P]$.
Por ejemplo, supongamos que tenemos cuatro objetos con pesos 1, 4, 5 y 7 respectivamente y valores 10, 3, 5 y 8 respectivamente y una mochila con capacidad máxima de 8. La matriz será la siguiente:

i	v_i	p_i	0	1	2	3	4	5	6	7	8
1	10	1	0	10	10	10	10	10	10	10	10
2	3	4	0	10	10	10	10	13	13	13	13
3	5	5	0	10	10	10	10	13	15	15	15
4	8	7	0	10	10	10	10	13	15	15	18

y la solución es la que está en la posición M[4,8] = 18.

ALGORITMO MOCHILA
Entrada: O: Vector<Objeto>, P: entero
Salida: v: entero
  $M \leftarrow inicializarMatrix(longitud(O), P+1)$
  para $i = 0$ **hasta** $longitud(O) - 1$
    para $j = 0$ **hasta** P
      si $i = 0$ **Y** $peso[i] < j$
        $M[i, j] \leftarrow 0$
      sino
        si $i = 0$
          $M[i, j] \leftarrow valor[i]$
        sino
          si $peso[i] < j$
            $M[i, j] \leftarrow M[i-1, j]$
          sino
            $M[i, j] \leftarrow \max(M[i-1, j], M[i-1, j - peso[i]] + valor[i])$
          fin si
        fin si
      fin si
    fin para
  fin para
  devolver $M[longitud(O) - 1, P]$

El costo de este algoritmo estará dado por el tamaño de la matriz que será $\mathcal{O}(nP)$. Se debe destacar que también ésta es la complejidad espacial del algoritmo, a diferencia de las otras técnicas que no tienen un alto costo en espacio.
El código Java es este:

```
public static int mochila(VectorTDA<Integer> pesos,
   VectorTDA<Integer> valores, int p) throws Exception{

   int i, j;
   int n = pesos.capacidadVector();
   MatrizTDA<Integer> m = new Matriz<Integer>();
```

```
        m.inicializarMatriz(Math.max(n, p+1));

        for(i = 0; i < n;i++){
            for(j = 0; j <= p; j++){
                if(i == 0 && j < pesos.recuperarElemento(0)){
                    m.setearValor(i, j, 0);
                }else{
                    if(i == 0){
                        m.setearValor(i, j, valores.
                            recuperarElemento(i));
                    }else{
                        if(j<pesos.recuperarElemento(i)){
                            m.setearValor(i, j, m.obtenerValor(i
                                -1, j));
                        }else{
                            m.setearValor(i, j, Math.max(m.
                                obtenerValor(i-1, j), valores.
                                recuperarElemento(i)+m.obtenerValor
                                (i-1, j-pesos.recuperarElemento(i))
                                ));
                        }
                    }
                }
            }
        }

        return m.obtenerValor(n-1,p);
    }
```

4.2.3. Subsecuencia común más larga

Dada una secuencia $X = \{x_1, x_2, \cdots, x_m\}$, decimos que $Z = \{z_1, z_2, \cdots, z_k\}$ es una subsecuencia de X (siendo $k \leq m$) si existe una secuencia creciente $\{i_1, i_2, \cdots, i_k\}$ de índices de X tales que para todo $j = 1, 2, \cdots, k$ tenemos $x_{i_j} = z_j$.
Por ejemplo, $Z = \{BCDB\}$ es una subsecuencia de $X = \{ABCBDAB\}$ con la correspondiente secuencia de índices $\{2, 3, 5, 7\}$. Dadas dos secuencias X e Y, decimos que Z es una subsecuencia común de X e Y si es subsecuencia de X y subsecuencia de Y. Se desea encontrar la longitud de la subsecuencia de longitud máxima común a dos secuencias dadas.
Por ejemplo, si las subsecuencias fueran $X = abda$ e $Y = abacdeb$ la subsecuencia común más larga es abd y su longitud 3.
La solución a este problema estará dada por considerar soluciones parciales en las que se consideran los primeros i caracteres de la secuencia X y los primeros j

caracteres de la secuencia Y. Es decir que llamaremos $L(i, j)$ a la longitud de la subsecuencia común máxima entre X e Y siendo i un valor entre 0 y la longitud de X y j un valor entre 0 y la longitud de Y.
La optimalidad estará dada ya que al considerar un nuevo caracter en cada secuencia pueden suceder dos cosas, que ambos sean iguales o no lo sean, si lo son, sumamos 1 a la longitud que teníamos sin considerar a ninguno de los dos nuevos caracteres, y si no lo son, consideramos lo mejor que teníamos sin considerar a uno de los caracteres y sin considerar al otro.
La recurrencia tendrá la siguiente forma:

$$L[i,j] \in \begin{cases} 0 & \text{si } i = 0 \text{ o } j = 0 \\ L[i-1, j-1] + 1 & \text{si } i > 0 \text{ y } j > 0 \text{ y } X_i = Y_j \\ \text{máx}(L[i-1, j], L[i, j-1]) & \text{sino} \end{cases}$$

La solución será $L[X_n, Y_m]$.
La matriz para nuestro ejemplo de $X = abda$ e $Y = abacdeb$ quedaría entonces:

		a	**b**	**a**	**c**	**d**	**e**	**b**
	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1
b	0	1	2	2	2	2	2	2
d	0	1	2	2	2	3	3	3
a	0	1	2	3	3	3	3	3

y la solución será la esquina inferior derecha de la matriz con un valor de 3.
El algoritmo será

ALGORITMO SUBSECUENCIAMAXIMA
Entrada: X, Y: Vector<Caracter>
Salida: v: entero
  $M \leftarrow inicializarMatrix(longitud(X) + 1, longitud(Y) + 1)$
  para $i = 0$ **hasta** $longitud(X)$
    para $j = 0$ **hasta** $longitud(Y)$
      si $i = 0$ **O** $j = 0$
        $M[i, j] \leftarrow 0$
      sino
        si $X[i] = Y[j]$
          $M[i, j] \leftarrow M[i-1, j-1] + 1$
        sino
          $M[i, j] \leftarrow \text{máx}(M[i-1, j], M[i, j-1])$
        fin si
      fin si
    fin para

fin para
devolver $M[longitud(X), longitud(Y)]$

El costo de este algoritmo estará dado por el tamaño de la matriz que será $\mathcal{O}(nm)$ donde n y m son las longitudes de las secuencias X e Y respectivamente. Se debe destacar que también ésta es la complejidad espacial del algoritmo, a diferencia de las otras técnicas que no tienen un alto costo en espacio.
El código Java es:

```
public static int subsecuenciaMasLarga(VectorTDA<
   Character> s1, VectorTDA<Character> s2) throws
   Exception{

   int l_s1,l_s2,aux,aux2;
   MatrizTDA<Integer> m;

   l_s1 = s1.capacidadVector();
   l_s2 = s2.capacidadVector();

   m = new Matriz<Integer>();
   m.inicializarMatriz(Math.max(l_s1, l_s2));

   for(int i = 0; i<l_s1;i++){
      for(int j = 0; j <l_s2;j++){
         if(s1.recuperarElemento(i) == s2.
            recuperarElemento(j)){
            if(i == 0 || j == 0){
               aux = 0;
            }else{
               aux = m.obtenerValor(i-1, j-1);
            }
            m.setearValor(i, j, aux +1);
         }else{
            if(i == 0){
               aux = 0;
            }else{
               aux = m.obtenerValor(i-1, j);
            }
            if(j== 0){
               aux2 = 0;
            }else{
               aux2 = m.obtenerValor(i, j-1);
            }
            m.setearValor(i, j, Math.max(aux,aux2));
```

```
            }
        }
    }

    return m.obtenerValor(l_s1-1,l_s2-1);
}
```

4.2.4. Multiplicación encadenada de matrices

Sea $M = M_1 \times M_2 \times \cdots \times M_n$ una multiplicación encadenada de matrices de dimensiones conocidas. Como la multiplicación es asociativa, hay varias maneras de resolver secuencias de productos matriciales. Se desea conocer la menor cantidad posible de multiplicaciones necesarias para resolver el problema. Dadas dos matrices M_1 y M_2 de dimensiones $m_1 \times m_2$ y $m_2 \times m_3$ respectivamente, la cantidad necesaria de multiplicaciones es $m_1 \cdot m_2 \cdot m_3$.
Por ejemplo, si tengo tres matrices M_1, M_2 y M_3 de 5×1, 1×4 y 4×3 respectivamente. El producto $M_1 \cdot M_2 \cdot M_3$ se puede resolver como $(M_1 \cdot M_2) \cdot M_3$ o como $M_1 \cdot (M_2 \cdot M_3)$. En el primer caso tendremos primero una multiplicación de 5×1 por 1×4 y luego 5×4 por 4×3 que da un total de $20 + 60 = 80$. En el segundo caso tendremos una multiplicación de 1×4 por 4×3 y luego 5×1 por 1×3 que da un total de $12 + 15 = 27$. Claramente conviene la segunda opción.
Para representar las dimensiones de las matrices utilizaremos un vector desde la posición 0 hasta la n y la dimensión de la matriz i estará dada por las posiciones del vector $i - 1$ e i. Para multiplicar desde la matriz i hasta la matriz j se deberá buscar el mejor k entre i y j tal que se minimice la cantidad de multiplicaciones para multiplicar desde i hasta k más la cantidad de multiplicaciones para multiplicar desde $k + 1$ hasta j más la cantidad de multiplicaciones para multiplicar ambos resultados. Por ejemplo, si tenemos que encontrar $M_1M_2M_3M_4$ con los diferentes k resultaría $(M_1)(M_2M_3M_4)$ o $(M_1M_2)(M_3M_4)$ o $(M_1M_2M_3)(M_4)$. Se deberá tomar la mejor de estas opciones. En cada opción, la cantidad de multiplicaciones estará dada por lo necesario para resolver lo que está dentro de cada par de paréntesis más la multiplicación de ambos grupos de paréntesis.
Este caso, a diferencia de los anteriores requerirá un recorrido de la matriz en un orden diferente. Para encontrar un valor $M[i, j]$ necesitaremos $M[i, k]$ y $M[k+1, j]$ con $i \leq k < j$, y esos valores se encuentran en la misma fila de $M[i, j]$ a su izquierda y en la misma columna hacia abajo respectivamente. Entonces, deberemos recorrer la matriz por diagonales para tener disponibles todos los valores necesarios para calcular cada nuevo valor o por filas, de abajo hacia arriba.
La recurrencia quedará entonces planteada como sigue:

$$M[i,j] \in \begin{cases} 0 & \text{si } i = j \\ \min\limits_{i \leq k < j} \{M[i,k] + M[k+1,j] + d_{i-1}d_k dj\} & \text{si } i < j \end{cases}$$

La solución será $M[1, n]$.
Por ejemplo, para cuatro matrices con un vector de distancias

4	1	5	3	6

	1	**2**	**3**	**4**
1	0	20	27	57
2		0	15	33
3			0	90
4				0

Y el resultado será $M[1, 4] = 57$.
El algoritmo tendrá la siguiente forma:

ALGORITMO PRODUCTOMATRICES
Entrada: d Vector<entero>
Salida: v: entero
entero $n \leftarrow longitud(d) - 1$
$M \leftarrow inicializarMatriz(n, n)$
para $i = n - 1$ **hasta** 0
para $j = i$ **hasta** $n - 1$
$M[i, j] \leftarrow 0$
para $k = i$ **hasta** $j - 1$
si $M[i, j] > 0$ **O** $M[i, k] + M[k + 1, j] + d[i - 1]d[k]d[j] < M[i, j]$
$M[i, j] \leftarrow M[i, k] + M[k + 1, j] + d[i - 1]d[k]d[j]$
fin si
fin para
fin para
fin para
devolver $M[0, n - 1]$

El costo está dado por el tiempo de llenado de la matriz que será de $\mathcal{O}(n^3)$ que responde a los tres bucles for del algoritmo. La complejidad espacial está en $\mathcal{O}(n^2)$.
El código Java es:

```
public static int productoMatrices(VectorTDA<Integer> v)
   throws Exception{
   int n = v.capacidadVector();
   MatrizTDA<Integer> m = new Matriz<Integer>();
   m.inicializarMatriz(n);
```

```
    for(int i = n-1; i>0;i--){
       for(int j = i;j<n;j++){
          m.setearValor(i, j, 0);
          for(int k = i; k < j; k++){
             if(m.obtenerValor(i,j)>0){
                if(m.obtenerValor(i, k)+m.obtenerValor(k
                   +1, j)+v.recuperarElemento(i-1)*v.
                   recuperarElemento(k)*v.
                   recuperarElemento(j) < m.obtenerValor(
                   i,j)){
                   m.setearValor(i, j, m.obtenerValor(i,
                      k)+m.obtenerValor(k+1, j)+v.
                      recuperarElemento(i-1)*v.
                      recuperarElemento(k)*v.
                      recuperarElemento(j));
                }
             }else{
                   m.setearValor(i, j, m.obtenerValor(i,
                      k)+m.obtenerValor(k+1, j)+v.
                      recuperarElemento(i-1)*v.
                      recuperarElemento(k)*v.
                      recuperarElemento(j));
             }
          }
       }
    }

    return m.obtenerValor(1,n-1);
 }
```

4.3. Grafos

4.3.1. Caminos mínimos: Floyd

El segundo tipo de búsqueda de caminos mínimos que veremos(el anterior fue Dijkstra en el capítulo de algoritmos Greedy) es el que busca el costo del camino mínimo entre cualquier par de nodos del grafo, siempre que este exista. A diferencia del anterior, este algoritmo funciona sólo con un grafo y no necesita un vértice particular. Al igual que el algoritmo anterior, el resultado es un grafo con los mismos nodos que el grafo de entrada, pero tiene una arista entre cada par de nodos y su peso es el costo de un camino mínimo.
Para poder ejecutar el algoritmo es necesario contar con un grafo dirigido, con todas sus aristas con peso positivo. Esto no significa que el grafo deba ser conexo,

es decir, puede haber pares de nodos que no estén unidos por una arista, pero si ésta existe, debe tener un peso positivo.

Si ponemos todos los posibles caminos entre cualquier par de vértices, tendríamos una complejidad exponencial, que no es lo que estamos buscando. Si en cambio planteamos una solución en la que buscamos el costo entre cada par de nodos contemplando pasar por todos los demás y lo planteamos como lo mejor de no contemplar uno de los nodos y luego probar agregarlo, podríamos llegar a una solución óptima basado en soluciones menores óptimas también.
Es decir si tengo la mejor forma de llegar de cada nodo a cualquier otro, pasando por los un subconjunto de los vértices como nodos intermedios y luego contemplamos pasar además por un vértice k, podríamos refinar cada resultado parcial y ver si utilizando el nodo k se puede mejorar. Para esto partiremos de una matriz M_0 en donde tenemos la mejor forma de llegar de cada vértice a todos los demás sin pasar por ningún vértice intermedio e iremos encontrando las matrices intermedias M_1, M_2 hasta llegar a M_n donde n es la cantidad de vértices del grafo. M_0 estará inicializado con 0 en su diagonal, el valor de la arista en el grafo original cuando exista arista o ∞ si no existe arista. Luego M_k se calculará como sigue $M_k[i,j] = \text{mín}(M_{k-1}[i,j], M_{k-1}[i,k] + M_{k-1}[k,j])$
Para la implementación, reemplazaremos la matriz por un grafo. El algoritmo en pseudocódigo es el siguiente:

ALGORITMO FLOYD
Entrada: G grafo de entrada
Salida: R: grafo con una arista con la distancia mínima entre cada par de vértices

```
inicializarGrafo(R)
copiarGrafo(G, R)
Vertices_K = Vertices(G)
para cada k ∈ Vertices_K
  Vertices_I = Vertices(G)
  para cada i ∈ Vertices_I
    Vertices_J = Vertices(G)
    para cada j ∈ Vertices_J
      si i ≠ j Y existeArista(R, i, k) Y existeArista(R, k, j)
        si existeArista(R, i, j)
          si pesoArista(R, i, k) + pesoArista(R, k, j) <
          pesoArista(R, i, j)
            agregarArista(R, i, j, pesoArista(R, i, k) +
            pesoArista(R, k, j))
          fin si
        sino
```

$agregarArista(R, i, j, pesoArista(R, i, k) + pesoArista(R, k, j))$
fin si
fin si
fin para
fin para
fin para
devolver R

El costo de este algoritmo está dado por el costo de los tres ciclos, que está en $\mathcal{O}(n^3)$.
El grafo resultado de aplicar el algoritmo sobre el grafo de la figura 4.2 es el que se ve en la figura 4.3.

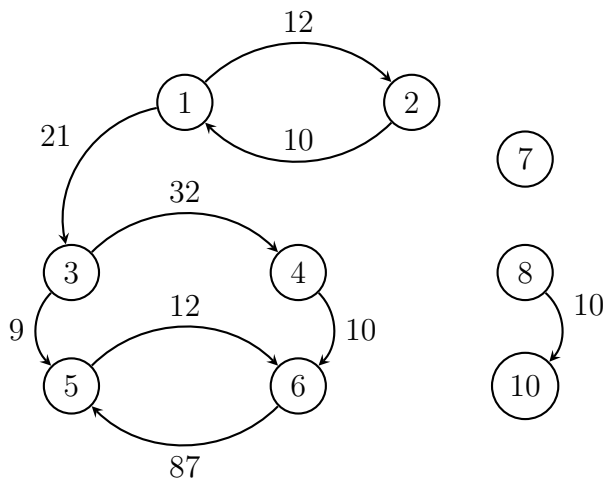


Figura 4.2: Ejemplo de grafo

El código Java es el siguiente:

```
public static GrafoDirTDA<Integer> floyd(GrafoDirTDA<
    Integer> g){

    ConjuntoTDA<Integer> conjuntoI,conjuntoJ,conjuntoK;
    int i,j,k;
    GrafoDirTDA<Integer> r = new GrafoDir<Integer>();
    r.InicializarGrafo();

    //Copio el grafo original
    conjuntoK = g.Vertices();
    while(!conjuntoK.conjuntoVacio()){
        k = conjuntoK.elegir();
```

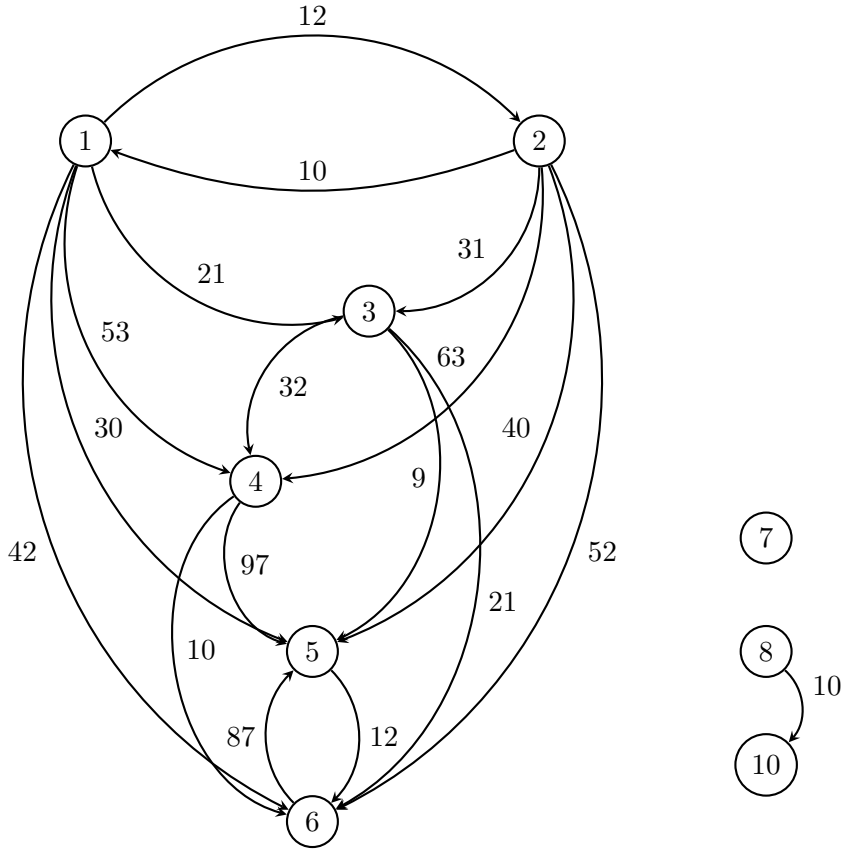


Figura 4.3: Resultado del algoritmo de Floyd

```
        conjuntoK.sacar(k);
        r.AgregarVertice(k);
    }

    conjuntoK = g.Vertices();
    while(!conjuntoK.conjuntoVacio()){
        k = conjuntoK.elegir();
        conjuntoK.sacar(k);
        conjuntoI = g.Adyacentes(k);
        while(!conjuntoI.conjuntoVacio()){
            i = conjuntoI.elegir();
            conjuntoI.sacar(i);
            r.AgregarArista(k, i, g.PesoArista(k, i));
        }
    }

    conjuntoK = g.Vertices();
    while(!conjuntoK.conjuntoVacio()){
        k = conjuntoK.elegir();
        conjuntoK.sacar(k);
        conjuntoI = g.Vertices();
        conjuntoI.sacar(k);
        while(!conjuntoI.conjuntoVacio()){
            i = conjuntoI.elegir();
            conjuntoI.sacar(i);
            if(r.ExisteArista(i, k)){
                conjuntoJ = r.Adyacentes(k);
                conjuntoJ.sacar(i);
                while(!conjuntoJ.conjuntoVacio()){
                    j = conjuntoJ.elegir();
                    conjuntoJ.sacar(j);
                    if(r.ExisteArista(i, j)){
                        if(r.PesoArista(i, k)+r.PesoArista(k,
                            j)< r.PesoArista(i, j)){
                            r.AgregarArista(i, j, r.PesoArista(
                                i, k)+r.PesoArista(k, j));
                        }
                    }else{
                        r.AgregarArista(i, j, r.PesoArista(i,
                            k)+r.PesoArista(k, j));
                    }
                }
            }
        }
    }
```

```
        return r;
    }
```

4.3.2. Problema del viajante

Un viajante de comercio debe visitar n ciudades. Cada ciudad está conectada con las restantes mediante carreteras de longitud conocida. El problema consiste en hallar la longitud de la ruta que deberá tomar para visitar todas las ciudades retornando a la ciudad de partida, pasando una única vez por cada ciudad y de modo tal que la longitud del camino recorrido sea mínima.
El problema consiste en buscar un camino desde el origen y hasta el origen, pasando por todos los demás nodos del grafo. Para ello se buscará la mejor opción de elegir un vértice distinto al origen y sumar el costo de llegar en el grafo del origen a dicho vértice más la mejor forma de llegar de dicho vértice al destino pasando por todos los demás. Es decir, tenemos que buscar el camino más corto que partiendo del origen, pase por todos los vértices y vuelva al origen, para ello iremos buscando los caminos más cortos que pasen por todos los vértices menos uno, menos dos, y así hasta los caminos más cortos que partiendo de un vértice, no pasen por ningún otro vértice y lleguen al destino. Este último valor sale directamente del grafo original tomando el valor de la arista.
Llamaremos D a la función que dado un vértice v del grafo y un conjunto de vértices C del grafo que no contiene al primero, devuelve la distancia más corta de un camino que partiendo de v y pasando por todos de C llegue al destino. El caso más sencillo es cuando C está vacío, que la distancia estará dada por el valor de la arista en el grafo original, los demás casos serán planteados a partir de las mejores soluciones de caminos con menos vértices.
El planteo recursivo entonces será el siguiente:

$$D(v, C) \in \begin{cases} pesoArista(v, destino) & \text{si } C = \emptyset \\ \min\limits_{v \notin C, w \in C} \{pesoArista(v, w) + D(w, C \setminus \{w\})\} & \text{sino} \end{cases}$$

Si analizamos esta matriz tendrá tantas filas como vértices tenga el grafo, y la cantidad de columnas estará dada por la cantidad de subconjuntos posibles de armar de el conjunto total de vértices, este es el conjunto de partes de un conjunto y su cardinalidad estará dada por 2^n. Es decir tendremos una matriz de $n2^n$ valores con lo cual el tiempo y espacio para resolver esta matriz será extremadamente alto. Dado este costo, no se justifica resolver este problema con la técnica que estamos analizando y dejaremos la solución para analizar en el próximo capítulo. Este es uno de los casos en donde aún no se han encontrado soluciones en tiempos razonables y en el último capítulo definiremos como problemas $\mathcal{NPC}$ o $\mathcal{NP}$-completos.

Capítulo 5

Backtracking

5.1. Características generales de la técnica

En los capítulos anteriores hemos analizado varias técnicas que nos permitían encontrar soluciones para diversos problemas. Las características de éstas soluciones es que eran eficientes, es decir que si bien no pudimos demostrar que los resolvimos de la mejor manera posible, se resolvían en un tiempo razonable. Ahora trataremos una serie de problemas para los que no nos fue posible encontrar soluciones eficientes basándonos en alguna propiedad de la solución. Para estos casos no habrá otra opción que realizar un estudio exhaustivo entre las posibles soluciones.
Para este estudio exhaustivo utilizaremos un grafo en el que iremos explorando los diferentes caminos para llegar a la solución. Esto implica dos cosas importantes, la primera es determinar qué forma tendrá la solución que queremos alcanzar, y la segunda es entender el significado de cada vértice del grafo como una etapa más próxima a una posible solución. Para facilitar el estudio y teniendo en cuenta las propiedades del grafo del que estamos hablando, pasaremos a hablar de un árbol (ya que descartaremos los ciclos que nunca nos llevarán a una solución) y llamaremos a este árbol el árbol de expansión. Dicho árbol nunca se construirá en forma efectiva sino que se irá recorriendo en forma abstracta. En cada nodo del árbol se tomará una decisión que es la que se cree mejor para alcanzar la solución. Si dicha decisión no fue la correcta, se volverá a dicho nodo y se tomará una decisión diferente. De aquí sale el nombre de la técnica como *Backtracking* o vuelta atrás.
La forma de recorrer este árbol es parecida a lo que llamamos un recorrido en preorden o profundidad para el caso de grafos. Avanza tanto como puede hasta que llega un punto en que no puede seguir, en dicho punto vuelve hasta el último paso en donde se tomó una decisión.
Por ejemplo, dado un conjunto de números C y un valor v, mostrar todos los subconjuntos de C en los cuales la suma de sus elementos sea v. En una posible solución al problema, cada elemento de C puede estar o no estar, con lo cual

podríamos tener una vector de 0s y 1s o un vector booleano con dimensión igual a la cardinalidad de C. Como cada posición del vector puede tomar dos posibles valores, la cantidad total de posibles soluciones es 2^n.
Si armamos el árbol completo del ejemplo para $C = \{13, 20, 7\}$ y $v = 20$ el árbol sería el siguiente

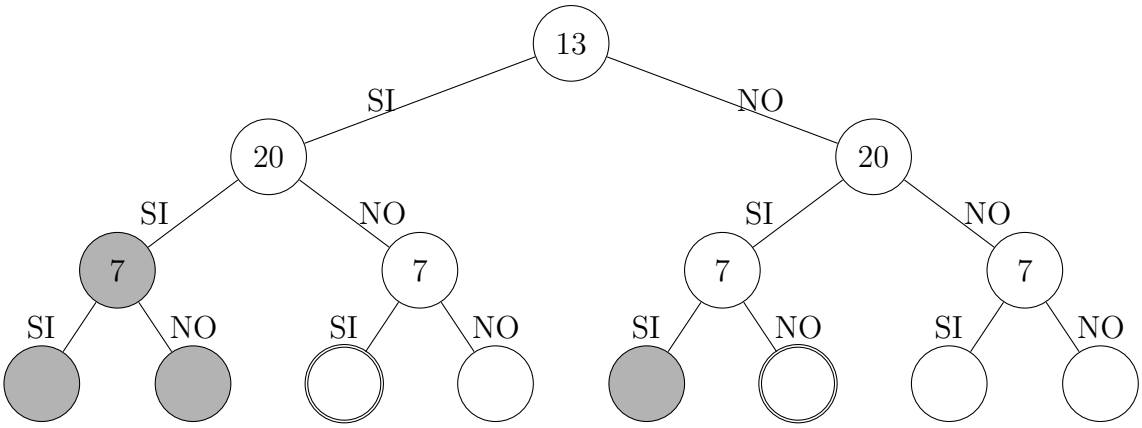


Figura 5.1: Arbol de expansión

donde están marcados los nodos que son solución con doble círculo. Además los nodos que están en gris son nodos que no habría que explorar nunca ya que la suma de elementos desde la raíz hasta el nodo anterior ya es superior a v.
La forma de proceder de backtracking es darle a cada etapa de la solución (en este caso, cada posición del vector) todos los posibles valores y para cada uno de esos valores, se generan todos los posibles valores para las etapas siguientes. Cuando no haya más etapas se verifica si el vector contiene o no una solución. Según sea el caso del problema que se esté resolviendo, el algoritmo podrá parar cuando encuentra la primera solución o podrá seguir buscando otras soluciones o mejores soluciones.
Para el ejemplo anterior, el algoritmo queda como sigue:

```
ALGORITMO SUMAM
Entrada: v: Vector<entero>, m: entero, ActSolucion: Vector<entero>,
  ActSuma: entero, etapa: entero
  para i = 0 hasta 1
    ActSolucion[etapa] ← i
    ActSuma ← ActSuma + v[etapa] * i
    si etapa = tamanio(v)
      si ActSuma = m
        mostrar(ActSolucion)
      fin si
```

sino
si $ActSuma < m$
$sumaM(v, m, ActSolucion, ActSuma, etapa + 1)$
fin si
fin si
fin para

y el código java correspondiente es

```
private static void sumaM(VectorTDA<Integer> v, int m,
    VectorTDA<Integer> ActSolucion, int ActSuma, int
    etapa) throws Exception{

     for(int i=0; i <= 1; i++){
         ActSolucion.agregarElemento(etapa, i);
         ActSuma += v.recuperarElemento(etapa)*i;
         if(etapa == v.capacidadVector()-1){
             if(ActSuma == m){
                 for(int j = 0; j < v.capacidadVector();
                    j++){
                     if(ActSolucion.recuperarElemento(j)
                         == 1){
                         System.out.print(v.
                            recuperarElemento(j)+",");
                     }
                 }
                 System.out.println("");
             }
         }else{
             if(ActSuma <= m){
                 sumaM(v,m,ActSolucion,ActSuma,etapa+1);
             }
         }
     }
}
```

En una solución con backtracking se deben identificar los siguientes elementos: formato de la solución, etapas, valores posibles, función solución.
Un algoritmo de Backtracking tendrá la siguiente forma:

ALGORITMO BACKTRACKING
Entrada: S: Solución parcial, E: etapa

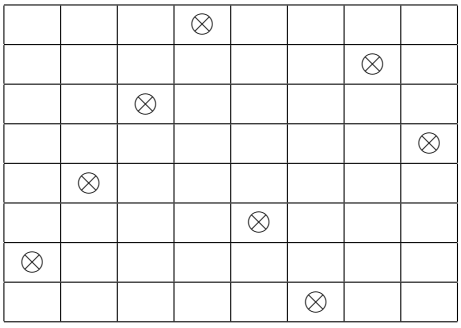
```
Valores ← generarValoresPosibles(S, E)
para cada v ∈ Valores
  agregar(S, E, v)
  si factible(S)
    si esSolucion(S)
      mostrar(S)
    sino
      backtracking(S, E + 1)
    fin si
  fin si
  eliminar(S, E)
fin para
```

Si quisiéramos analizar el costo de un algoritmo planteado con este esquema, podemos ver las llamadas recursivas que serán v para cada etapa donde v es la cantidad de valores posibles para cada etapa, esto deriva en soluciones del tipo $\mathcal{O}(v^n)$. Para el peor caso, donde $v = n$ podemos llegar a $\mathcal{O}(n^n)$ o para el mejor caso podemos llegar a $\mathcal{O}(2^n)$. Como un caso intermedio, si del n^n no contemplamos los casos de repeticiones, podemos tener un $\mathcal{O}(n!)$.

5.2. Ejemplos

5.2.1. N reinas

El enunciado del problema es el siguiente: *Dado un tablero de $N \times N$ ubicar N reinas de modo tal que no se coman entre sí. Una reina come a otra si está en la misma fila, columna o diagonal.* Encontrar una solución para el problema.
Por ejemplo, para $N = 8$ una posible solución es:



La solución obvia sería una matriz booleana con valores verdaderos en las posiciones donde están las reinas. Como esta matriz sería rala, es decir, tendría N valores en uso de los N^2 disponibles, y teniendo en cuenta que nunca habrá más de una reina en cada fila, se podría considerar un vector de N posiciones que

indica para cada fila, en qué columna se ubicará la reina. Es decir que el vector deberá contener los números del 1 al N. Con esta solución no podríamos tener más de una reina en la misma fila. Además si evitamos que aparezcan números repetidos, estaremos asegurando que no haya más de una reina en una misma columna. Faltaría chequear la restricción de las diagonales. Dada una fila particular i y una columna particular j donde estamos probando colocar una reina, las diagonales de esta ubicación estarán k filas más arriba ($1 \leq k < i$) y desplazadas k columnas a derecha o a izquierda. Con toda esta información, podemos armar el algoritmo como sigue:

```
ALGORITMO REINAS
Entrada: S: Vector<entero>, E: entero, n:entero
Salida: verdadero-falso
  booleano solucion ←falso
  S[E] ← 0
  mientras S[E] < n − 1 Y NO solucion
    S[E] ← S[E] + 1
    si ReinaValida(S, E)
      si E = n − 1
        solucion ←verdadero
      sino
        solucion ← Reinas(S, E + 1, n)
      fin si
    fin si
  fin mientras
  devolver solucion
```

```
ALGORITMO REINAVALIDA
Entrada: S: Vector<entero>, E: entero
Salida: verdadero-falso
  para i = 0 hasta E − 1
    si S[i] = S[E] O |S[i] − S[E]| = |i − E|
      devolver falso
    fin si
  fin para
  devolver verdadero
```

Este algoritmo está en $\mathcal{O}(n^n)$ ya que prueba todas las posibles formas de llenar el vector de tamaño n con los números del 1 al n. Corta cuando encuentra la primera solución o cuando prueba todas las opciones y no hay solución posible. El código Java es el siguiente:

```
public static void reinas(int n)throws Exception{

    VectorTDA<Integer> v = new Vector<Integer>();
    v.inicializarVector(n);

    if(reinas(n,0,v)){
        Mostrar(v);
    }else{
        System.out.println("No hay solución");
    }
}

static boolean reinas(int n, int k, VectorTDA<Integer>
   v)throws Exception{
    boolean solucion = false;
    v.agregarElemento(k, 0);

    while(v.recuperarElemento(k) < n  && !solucion){
        v.agregarElemento(k, v.recuperarElemento(k)+1);
        if(reinasValido(k,v)){
            if(k +1 == n){
                solucion = true;
            }else{
                solucion = reinas(n,k+1,v);
            }
        }
    }
    return solucion;
}

static boolean reinasValido(int k, VectorTDA<Integer> v
   )throws Exception{
    for(int i = 0; i < k;i++){
        if(v.recuperarElemento(i) == v.
           recuperarElemento(k) || Math.abs(v.
           recuperarElemento(i)-v.recuperarElemento(k))
            == Math.abs(i-k))
            return false;
    }

    return true;
}
```

5.2.2. Partición en mitades

Dado un conjunto de n enteros se desea encontrar, si existe, una partición en dos subconjuntos disjuntos, tal que la suma de sus elementos sea la misma.
Aquí la estrategia consistirá en colocar cada elemento en el conjunto 1 o en el conjunto 2 y al tener cada elemento en alguno de los conjuntos ver si la suma de los elementos de cada conjunto es igual. En una estrategia un poco más elaborada se podría llevar el acumulado de cada conjunto y así no hace falta hacer la suma al final.
El algoritmo sería:

ALGORITMO PARTICIONMITAD
Entrada: S: Vector<entero>, E: entero, v: Vector<entero>
para $i = 1$ **hasta** 2
$S[E] \leftarrow i$
si $E = longitud(v) - 1$
$suma1 \leftarrow 0; suma2 \leftarrow 0$
para $j = 0$ **hasta** $longitud(v) - 1$
si $S[j] = 1$
$suma1 \leftarrow suma1 + v[j]$
sino
$suma2 \leftarrow suma2 + v[j]$
fin si
fin para
si $suma1 = suma2$
$Mostrar(S)$
fin si
sino
$particionMitad(v, S, E + 1)$
fin si
fin para

Este algoritmo está en $\mathcal{O}(2^n)$ y encuentra todas las posibles soluciones al problema. El código java es el siguiente:

```
public static void particionMitad(VectorTDA<Integer> v)
   throws Exception{
     VectorTDA<Integer> ActSolucion = new Vector<Integer
        >();
     ActSolucion.inicializarVector(v.capacidadVector());
     particionMitad(v,ActSolucion,0);
 }
```

```
    private static void particionMitad(VectorTDA<Integer> v
       , VectorTDA<Integer> ActSolucion, int etapa)throws
       Exception{

        for(int i=0; i <= 1; i++){
            ActSolucion.agregarElemento(etapa, i);
            if(etapa == v.capacidadVector()-1){
                int suma1=0, suma2=0;

                for(int j = 0; j < v.capacidadVector();j++)
                   {
                    if(ActSolucion.recuperarElemento(j) ==
                       1){
                        suma1 += v.recuperarElemento(j);
                    }else{
                        suma2 += v.recuperarElemento(j);
                    }
                }

                if(suma1 == suma2){
          Mostrar(ActSolucion);
                }
            }else{
                particionMitad(v,ActSolucion,etapa+1);
            }
        }
    }
```

5.2.3. Cambio de monedas

El problema del cambio de monedas es ya conocido y lo hemos resuelto en forma correcta con algoritmo de programación dinámica y en forma incorrecta con un algoritmo Greedy. En este problema, a diferencia de los anteriores tendremos que buscar la mejor solución, para ello necesitaremos contar con parámetros adicionales, además de la solución parcial y la etapa, que nos indiquen cuál es la mejor solución hasta el momento. Esta información servirá para dos propósitos, por un lado servirá para comparar la nueva solución encontrada y por otro lado servirá para no seguir explorando caminos que ya sabemos que no nos conducirán a una solución mejor, por ejemplo si sabemos que ya tenemos una solución con tres monedas y la solución actual ya está en cuatro monedas, no seguiremos explorando ese camino.
La estrategia será ir armando la solución mientras no se alcance el importe deseado o no se pase del mismo, y las opciones en cada etapa serán todas las posibles monedas.

El algoritmo será entonces:

ALGORITMO CAMBIO
Entrada: S: Vector<entero>, S_v: entero, M: Vector<entero>, M_c: entero
E: etapa, v: Vector<entero>, C: entero
para $i = 0$ **hasta** $longitud(v) - 1$
$S[E] \leftarrow v[i]$
$S_v \leftarrow S_v + v[i]$
si $S_v = C$
si NO M_c **O** $E < M_c$
$M_c \leftarrow E + 1$
$copiarVector(S, M)$
fin si
sino
si $S_v < C$
$M_c = Cambio(S, S_v, M, M_v, E + 1, v, C)$
fin si
fin si
$S_v \leftarrow S_v - v[i]$
fin para
devolver M_c

El costo de este algoritmo estará dado por $\mathcal{O}(V^C)$ donde V es la cantidad de denominaciones de monedas disponibles y C es el cambio a devolver. El código java es:

```
public static void monedas(VectorTDA<Integer> monedas,
   int v)throws Exception{
  VectorTDA<Integer> solucionActual = new Vector<
     Integer>();
  solucionActual.inicializarVector(v);
  VectorTDA<Integer> solucionMejor = new Vector<Integer
     >();
  solucionMejor.inicializarVector(v);
  int cantidad = monedas(monedas, v, solucionActual, 0,
      solucionMejor, -1,0);
MostrarSolucion(MejorSolucion);
 }

 public static int monedas(VectorTDA<Integer> monedas,
    int v, VectorTDA<Integer> solucionActual, int
    valorActual, VectorTDA<Integer> solucionMejor, int
    valorMejor, int etapa) throws Exception{
```

```
    for(int i = 0; i < monedas.capacidadVector(); i++){
        solucionActual.agregarElemento(etapa, monedas.
            recuperarElemento(i));
        valorActual += monedas.recuperarElemento(i);
        if(valorActual == v){
            if(etapa < valorMejor || valorMejor == -1){
                valorMejor = etapa+1;
                copiarVector(solucionActual,solucionMejor,
                    etapa+1);
            }
        }else if(valorActual < v){
            valorMejor = monedas(monedas,v,solucionActual,
                valorActual,solucionMejor,valorMejor,etapa
                +1);
        }
        valorActual -= monedas.recuperarElemento(i);
    }
    return valorMejor;
}
```

5.2.4. Camino con menos escalas

El enunciado del problema es el siguiente: *Dado un sistema de rutas aéreas modelado mediante un grafo, determine la ruta que contenga la mínima cantidad de escalas entre un par de ciudades dadas.*
En este caso, tenemos un problema de optimización, en particular de minimización. En este caso entonces como parámetros adicionales al grafo de entrada, el origen y destino, tendremos el camino que se está buscando actualmente, la longitud del camino actual, el mejor camino encontrado, la longitud del mejor camino encontrado y la etapa.
La solución será un vector con el camino, en cada etapa se puede seleccionar cualquier vértice, de los no visitados.

ALGORITMO CAMINOMENOSESCALAS
Entrada: G: grafo, o: origen, d: destino, $mejorCamino$, $longitudMejor$, $actualCamino$, $etapa$
Salida: $escalas$: cantidad de escalas de la mejor solución encontrada
$actualCamino[etapa] \leftarrow o$
$etapa \leftarrow etapa + 1$
si $o = d$
  si $longitudMejor = -1$ **O** $etapa < longitudMejor$

```
      mejorCamino ← actualCamino
      devolver etapa
    sino
      devolver longitudMejor
    fin si
  sino
    adyacentes ← adyacentes(G, o)
    para cada o ∈ adyacentes
      si NO pertenece(actualCamino, o)
        longitudMejor ←
            CaminoMenosEscalas(G, o, d, mejorCamino, longitudMejor,
                actualCamino, etapa)
      fin si
    fin para
    devolver longitudMejor
  fin si
```

En este caso, donde se evalúan todas las posibles combinaciones para encontrar la mejor, está en $\mathcal{O}(n!)$.
El código en Java es el siguiente:

```
public static int rutasAereas(GrafoDirTDA<Integer> g,
   int c1, int c2)throws Exception{
     VectorTDA<Integer> mejorCamino = new Vector<Integer
        >();
     VectorTDA<Integer> actualCamino = new Vector<
        Integer>();

     ConjuntoTDA<Integer> vertices = g.Vertices();
     int i = 0;
     while(!vertices.conjuntoVacio()){
         i++;
         vertices.sacar(vertices.elegir());
     }

     mejorCamino.inicializarVector(i);
     actualCamino.inicializarVector(i);

     return rutasAereas(g,c1,c2,mejorCamino,-1,
        actualCamino,0);
 }

private static int rutasAereas(GrafoDirTDA<Integer> g,
   int c1, int c2, VectorTDA<Integer> mejorCamino, int
```

```
      mejorCosto, VectorTDA<Integer> actualCamino, int
      actualCosto) throws Exception{

        ConjuntoTDA<Integer> adyacentes;
        actualCamino.agregarElemento(actualCosto, c1);
        actualCosto++;

        if(c1 == c2){
            if(mejorCosto == -1 || actualCosto < mejorCosto
               ){
                copiarVector(actualCamino,mejorCamino,
                   actualCosto);
                return actualCosto;
            }else{
                return mejorCosto;
            }
        }else{
            adyacentes = g.Adyacentes(c1);
            while(!adyacentes.conjuntoVacio()){
                c1 = adyacentes.elegir();
                adyacentes.sacar(c1);
                if(!pertenece(actualCamino,actualCosto,c1))
                   {
                  mejorCosto = rutasAereas(g,c1,c2,
                     mejorCamino,mejorCosto,actualCamino,
                     actualCosto);
               }
            }
            return mejorCosto;
  }
   }
```

5.2.5. Mochila

El problema de la mochila que ya hemos resuelto con otras técnicas, es muy simple de plantear con la técnica de Backtracking. Básicamente se basa en entender que cada objeto puede estar o no estar en la solución, que la suma de pesos no sobrepase la capacidad de la mochila y que la suma de valores sea máxima. La solución será un vector booleano o un vector con 0's y 1's. Como entrada, además de los pesos, los valores y la capacidad se necesita la etapa, la solución actual con su peso y la mejor solución con su peso.
El algoritmo para resolver el problema es el siguiente:

ALGORITMO MOCHILA
Entrada: *Objetos*: Vector<Objeto>, *capacidad*: entero, *mejorSolucion*: Vector<entero>, *valorMejorSolucion*: entero, *solucionActual*: Vector<entero>, *valorSolucionActual*,*pesoSolucionActual*,*etapa*:entero
Salida: entero
para $i = 0$ **hasta** 1
$solucionActual \leftarrow i$
$pesoSolucionActual \leftarrow pesoSolucionActual + i * Objetos[etapa].peso$
$valorSolucionActual \leftarrow valorSolucionActual + i * Objetos[etapa].valor$
si $pesoSolucionActual \leq capacidad$
si $etapa = longitud(Objetos) - 1$
si $valorMejorSolucion = -1$ **O** $valorMejorSolucion < valorSolucionActual$
$mejorSolucion \leftarrow solucionActual$
$valorMejorSolucion \leftarrow valorSolucionActual$
fin si
sino
si $etapa < longitud(Objetos) - 1$
$valorMejorSolucion \leftarrow$
$Mochila(Objetos, capacidad, mejorSolucion, valorMejorSolucion, solucionActual, valorSolucionActual, etapa + 1)$
fin si
fin si
fin si
fin para
devolver *valorMejorSolucion*

El costo de este algoritmo está dado por los posibles valores de la solución, que es el vector de 0's y 1's, es decir que está en $\mathcal{O}(2^n)$. El código Java es el siguiente:

```
public static VectorTDA<Integer> mochila(VectorTDA<
   Integer> pesos, VectorTDA<Integer> valores, int
   capacidad) throws Exception{

   VectorTDA<Integer> mejorSolucion = new Vector<Integer
      >();
   mejorSolucion.inicializarVector(pesos.capacidadVector
      ());

   VectorTDA<Integer> actualSolucion = new Vector<
      Integer>();
```

```
        actualSolucion.inicializarVector(pesos.
            capacidadVector());

        int etapa = 0;

        int mejorValor=-1;
        int actualValor = 0;
        int actualPeso = 0;

        mejorValor = mochila(pesos,valores,capacidad,
            mejorSolucion,mejorValor,actualSolucion,
            actualValor,actualPeso,etapa);

        return mejorSolucion;

    }

    public static int mochila(VectorTDA<Integer> pesos,
        VectorTDA<Integer> valores, int capacidad, VectorTDA
        <Integer> mejorSolucion, int mejorValor, VectorTDA<
        Integer> actualSolucion, int actualValor,int
        actualPeso,int etapa) throws Exception{

        for(int i = 0; i < 2; i++){
            actualSolucion.agregarElemento(etapa, i);
            actualPeso += i*pesos.recuperarElemento(etapa);
            actualValor += i*valores.recuperarElemento(etapa);
            if(actualPeso <= capacidad){
                if(etapa +1 == valores.capacidadVector()){
                    if(mejorValor == -1 || mejorValor <
                        actualValor){
                        copiarVector(actualSolucion,mejorSolucion
                            ,pesos.capacidadVector());
                        mejorValor = actualValor;
                    }
                }else{
                    if(etapa +1 < valores.capacidadVector()){
                        mejorValor = mochila(pesos,valores,
                            capacidad,mejorSolucion,mejorValor,
                            actualSolucion,actualValor,actualPeso,
                            etapa+1);
                    }
                }
            }
        }
```

```
        return mejorValor;
    }
```

5.2.6. Asignación de tareas

El enunciado de este problema dice *Se tienen* ***m*** *procesadores idénticos y* ***n*** *tareas con un tiempo de ejecución dado. Se requiere encontrar una asignación de tareas a procesadores de manera de minimizar el tiempo de ejecución del total de tareas.* La estrategia en este caso será ir asignando a cada una de las **n** tareas uno de los **m** procesadores posibles. Como es un problema de optimización, además de la etapa y la solución parcial habrá que guardar la mejor solución encontrada hasta el momento. Esto significa que la etapa estará dada por cada una de las tareas, las posibilidades para cada etapa será cada uno de los procesadores. La solución parcial y la mejor solución serán vectores de tamaño **n** donde el contenido será un valor entre 1 y **m**. Para comparar la mejor solución con la solución actual, se cuenta con un método auxiliar que calcula para cada procesador la suma de las tareas que tiene asignadas y así, con el máximo de todos esos tiempos se sabe el tiempo de ejecución total.

ALGORITMO TAREAS
Entrada: m, $tareas$, $solucionParcial$, $mejorSolucion$, $valorMejorSolucion$, $etapa$
Salida: $valorMejorSolucion$
  para $i = 1$ **hasta** m
    $solucionParcial[etapa] \leftarrow i$
    si $etapa + 1 = tamanio(tareas)$
      $valorSolucionParcial \leftarrow sumaTareas(m, tareas, solucionParcial)$
      si $valorMejorSolucion = -1$ **O** $valorSolucionParcial < valorMejorSolucion$
        $mejorSolucion \leftarrow solucionParcial$
        $valorMejorSolucion \leftarrow valorSolucionParcial$
      fin si
    sino
      $valorMejorSolucion \leftarrow Tareas(m, tareas, solucionParcial, mejorSolucion, valorMejorSolucion, etapa + 1)$
    fin si
  fin para
  devolver $valorMejorSolucion$

ALGORITMO SUMATAREAS
Entrada: m, $tareas$, $solucionParcial$
Salida: $maximoTiempo$
  $procesadores \leftarrow inicializarVector(m)$
  para $i = 1$ **hasta** $tamanio(tareas)$
    $procesadores[solucionParcial[i]] \leftarrow procesadores[solucionParcial[i]] + tareas[i]$
  fin para
  $maximoTiempo \leftarrow -1$
  para $i = 1$ **hasta** m
    si $procesadores[i] > maximoTiempo$
      $maximoTiempo \leftarrow procesadores[i]$
    fin si
  fin para
  devolver $maximoTiempo$

En este caso, como se analizan todas las posibles asignaciones de procesadores a tareas, sabemos que eso está en $\mathcal{O}(n^m)$.
El código Java es el siguiente:

```
public static VectorTDA<Integer> asignarTareas(int m,
    VectorTDA<Integer> tareas) throws Exception{
  VectorTDA<Integer> asignacionActual = new Vector<
     Integer>();
  asignacionActual.inicializarVector(tareas.
     capacidadVector());
  VectorTDA<Integer> asignacionMejor = new Vector<
     Integer>();
  asignacionMejor.inicializarVector(tareas.
     capacidadVector());
  asignarTareas(m,tareas,asignacionActual,
     asignacionMejor,-1,0);
  return asignacionMejor;

}

private static int asignarTareas(int m, VectorTDA<
   Integer> tareas, VectorTDA<Integer> asignacionActual
   , VectorTDA<Integer> asignacionMejor, int
   valorAsignacionMejor, int tarea) throws Exception{
   for(int i = 0; i < m; i++){
     asignacionActual.agregarElemento(tarea, i);
     if(tarea + 1 == tareas.capacidadVector()){
```

```
            int valorAsignacionActual = asignarTareasSuma(m
               ,tareas,asignacionActual);
            if(valorAsignacionActual < valorAsignacionMejor
                || valorAsignacionMejor == -1){
               copiarVector(asignacionActual,
                  asignacionMejor,tareas.capacidadVector())
                  ;
               valorAsignacionMejor = valorAsignacionActual
                  ;
            }
         }else{
            valorAsignacionMejor = asignarTareas(m,tareas,
               asignacionActual,asignacionMejor,
               valorAsignacionMejor,tarea+1);
         }
      }
      return valorAsignacionMejor;
   }

   private static int asignarTareasSuma(int m, VectorTDA<
      Integer> tareas, VectorTDA<Integer> asignacionActual
      ) throws Exception{
     VectorTDA<Integer> procesadores = new Vector<Integer
        >();
     procesadores.inicializarVector(m);

     for(int i = 0; i < m;i++){
        procesadores.agregarElemento(i,0);
     }

     for(int i = 0; i < asignacionActual.capacidadVector()
        ;i++){
        procesadores.agregarElemento(asignacionActual.
           recuperarElemento(i),procesadores.
           recuperarElemento(asignacionActual.
           recuperarElemento(i))+tareas.recuperarElemento(
           i));
     }

     int mayor = -1;
     for(int i = 0; i < m; i++){
        if(procesadores.recuperarElemento(i)>mayor){
           mayor = procesadores.recuperarElemento(i);
        }
     }
     return mayor;
```

```
}
```

5.2.7. Caminos Hamiltonianos

El problema de los caminos Hamiltonianos consiste en encontrar un camino dentro de un grafo que comience y termine en el mismo vértices, que pase por todos los vértices del grafo y que no repita ninguno. El problema del viajante visto en el capítulo anterior, es una variante de este problema que de todos los caminos Hamiltonianos, se queda con el de menor longitud.
Para resolver este problema habrá que determinar cuáles son las etapas, cuáles las posibilidades para cada etapa, qué forma tendrá la solución parcial.
Una estrategia simple de resolución es generar las $n!$ permutaciones de los n nodos y para cada una, ver si existe una arista para cada par de nodos contiguos en cada permutación.
La otra alternativa es ir generando solamente las permutaciones válidas analizando los vértices adyacentes.
La solución parcial se irá almacenando en un vector del tamaño de la cantidad de nodos. En cada etapa i, se irá definiendo qué nodo va en ese punto del camino. El primer nodo se fija al azar. Debe existir una arista entre cada par de nodos que ocupan posiciones contiguas en la solución. Además deberá existir una arista entre el último nodo de la solución y el primero, para cerrar el ciclo.
Suponiendo que el primer nodo se elige al azar y ya está en la primera posición de la solución parcial y que etapa comienza en la posición 2, el algoritmo para encontrar todos los caminos hamiltonianos sería el siguiente:

ALGORITMO HAMILTONIANO
Entrada: G, $solucionParcial$, $etapa$
  para cada $v \in adyacentes(G, solucionParcial[etapa-1])$
    si NO $pertenece(v, solucionParcial)$
      $solucionParcial[etapa] \leftarrow v$
      si $etapa < tamanio(solucionParcial)$
        $Hamiltoniano(G, solucionParcial, etapa+1)$
      sino
        si $existeArista(G, solucionParcial[etapa], solucionParcial[1])$
          $mostrar(SolucionParcial)$
        fin si
      fin si
    fin si
  fin para

Claramente, para el peor de los casos, que seria un grafo completo, donde todos los vértices están unidos por medio de una arista con todos los demás, el orden de este algoritmo es $\mathcal{O}(n!)$.
Si quisiéramos resolver el problema del viajante en base a este problema, deberíamos quedarnos con la mejor de las soluciones. Para ello se deberán agregar dos parámetros más al algoritmo, que sería el mejor ciclo encontrado hasta el momento, y el costo de ese ciclo para no volver a calcularlo en cada comparación con un nuevo ciclo encontrado. Además en el lugar donde se muestra el ciclo encontrado, se deberá comparar su costo contra el costo del mejor ciclo encontrado hasta el momento y si lo mejora, se deberá reemplazar el mejor ciclo encontrado hasta el momento con el nuevo ciclo encontrado.

5.3. Ramificación y Poda

La técnica de ramificación y poda, conocida por su nombre en inglés como *Branch & Bound* es una variante de la técnica de Backtracking. Se basa en la exploración de un árbol y tiene dos principales diferencias con la técnica pura. Ambas diferencias se basan en la forma de recorrer el árbol.
En backtracking el árbol se recorre en profundidad mientras que aquí el árbol se recorrerá según convenga, siguiendo por el nodo más prometedor, el que lleve a una solución más rápidamente o el que ayude a descartar mayor cantidad de opciones del árbol. Esa será la estrategia de ramificación.
Además, si se detecta que un nodo nunca nos llevará a una buena solución, directamente se eliminará y no se recorrerá ninguna solución que pase por dicho nodo. Esa será la estrategia de poda.
Según esta explicación queda claro que la técnica sirve para resolver problemas de optimización.
A diferencia de backtracking, en donde el siguiente nodo a explorar del árbol se genera o avanzando una etapa o por vuelta atrás, en esta técnica, en cada momento existen uno o varios nodos candidatos a ser el siguiente nodo a explorar. A cada uno de estos nodos candidatos a ser explorados los llamaremos nodos vivos. Tendremos un conjunto de nodos vivos, que inicializaremos con el primer nodo. Luego, en forma iterativa, y hasta finalizar la búsqueda, iremos seleccionando del conjunto de nodos vivos el más prometedor, es decir, aquel que suponemos que nos llevará a una buena solución y lo más rápido posible. A partir de este nodo seleccionado, explotamos todos los posibles nodos siguientes, y los agregamos al conjunto de nodos vivos. Esto en cuanto a la estrategia de ramificación. El conjunto de nodos vivos será aquella estructura que mejor nos sirva, pudiendo ser en muchos casos una cola con prioridad.
A lo anterior, faltaría completarlo con dos ideas. La principal, es que cuando el nodo elegido del conjunto de nodos vivos es una solución, debemos hacer algo, según el problema. Podría ser frenar la ejecución si buscamos la primera solución o

compararlo contra la solución anterior encontrada si buscamos la mejor solución. La segunda idea para complementar lo anterior, es la idea de poda, es decir que tanto al momento de explotar los siguientes nodos y antes de agregarlos al conjunto de nodos vivos, como al momento de seleccionar uno para continuar la búsqueda, podríamos definir que no nos sirve. Esto seria la poda. Para podar, se trabaja con la idea de una cota. En cada nodo vivo se calcula una cota, que es el valor de la mejor solución que se podría alcanzar desde ese nodo. Si el mejor valor que se podría encontrar para un nodo es peor que el valor de la mejor solución encontrada hasta el momento, ese nodo se podrá podar. También se podría calcular una segunda cota, para el valor de la peor solución que se podría alcanzar desde un nodo. Si el peor valor que se puede alcanzar desde un nodo es mejor que el mejor valor que se puede alcanzar desde otro, ese otro nodo se puede podar, ya que nunca alcanzará una solución mejor que la del nodo actual. A cada una de estas cotas las llamaremos cota inferior y cota superior respectivamente y según se trate de un problema de minimización o de maximización será el rol que cada una tenga.
Veremos estos dos casos en un ejemplo. Supongamos el árbol de expansión de la figura 5.2 para resolver un problema de maximización.

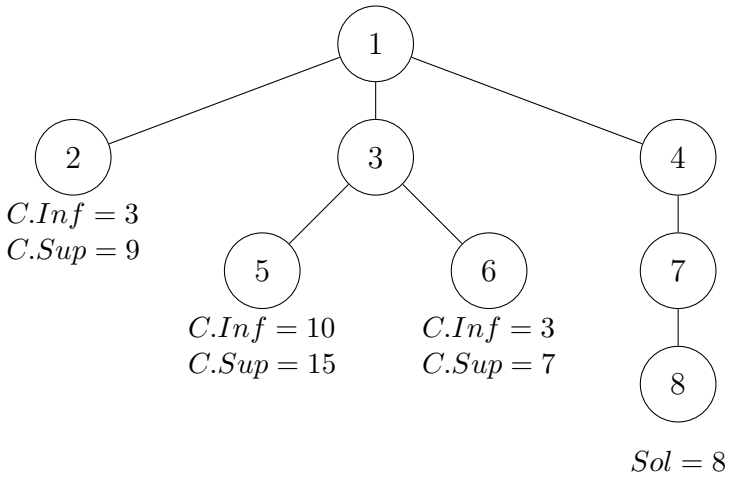


Figura 5.2: Árbol de poda

Hasta ahora en el nodo 8 se encontró una solución con valor 8. En el conjunto de nodos vivos hay tres nodos, el 2, el 5 y el 6. La primera pregunta será cuál de ellos seleccionar? Esa será la estrategia de ramificación. Podría ser aquel que tenga una cota inferior más alta, es decir, aquel que me prometa, para el pero de los casos, la mejor de soluciones. Otra estrategia sería elegir aquel que tenga la mayor de las cotas superiores, es decir, aquel que me prometa para el mejor de los casos, lo mejor. Ambas serían válidas. Ahora veamos qué pasa con la estrategia de poda. Por un lado, todo nodo que tenga su cota superior menor que la mejor solución

encontrada se podrá descartar, ya que no alcanzaremos nada mejor. En este caso, el nodo 6 se podrá podar. Por otro lado, todo nodo que tenga una cota superior menor que la cota inferior de otro se podrá podar, ya que tampoco alcanzaremos una solución buena. En este caso el nodo 2 se podrá podar, porque lo mejor que se podría alcanzar es 9, que es peor que el peor valor que se podría alcanzar por el nodo 5. Es así que en este caso solo convendrá seguir buscando por el nodo 5.
Si en cambio, estuviéramos trabajando con un problema de minimización, las reglas se inviertes. Es decir, se podrá podar un nodo cuya cota inferior sea mayor que la mejor solución encontrada y se podrá podar podar un nodo cuya cota inferior sea mayor que la cota superior de otro nodo.
Para simplificar la codificación, se mantendrá en todo momento un valor `Cota` actualizado con el máximo entre la mejor solución encontrada y el máximo de todas las cotas inferiores de los nodos explotados. Y se podrá podar todo nodo cuya cota superior sea menor o igual a `Cota`.
Análogamente, para un ejercicio de minimización se mantendrá en todo momento un valor `Cota` actualizado con el mínimo entre la mejor solución encontrada y el mínimo de todas las cotas superiores de los nodos explotados. Y se podrá podar todo nodo cuya cota inferior sea mayor o igual a `Cota`.
Al igual que en backtracking, en este caso la solución podrá tener una de tres características: **a)** finalizar al encontrar la primera solución **b)** encontrar todas las soluciones **c)** encontrar la mejor solución. Presentaremos el modelo de solución para el caso de encontrar la mejor solución, ya que las dos primeras son una modificación del mismo algoritmo.
La estructura de un nodo tendrá como mínimo la información necesaria para poder expandirlo, podarlo si corresponde y determinar si es una solución. La información mínima será la *Solución Parcial*, la *Etapa*, la *Cota Inferior* y la *Cota Superior*. Además se podrán agregar todos los valores necesarios según el problema. La Estructura de Nodos Vivos (ENV) como dijimos, será aquella que nos permita ramificar con el mejor criterio.

ALGORITMO R&P

$ENV \leftarrow inicializarEstructura()$
$nodoRaiz \leftarrow CrearNodoRaiz()$
$agregar(ENV, nodoRaiz)$
$Cota \leftarrow ActualizarCota(Cota, nodoRaiz)$
$mejorSolucion \leftarrow NULO$
mientras $ENV \neq \emptyset$
  $nodo \leftarrow primero(ENV)$
  si NO $Podar(nodo, Cota)$
    $hijos \leftarrow GenerarHijos(nodo)$
    para cada $hijo \in hijos$
      si NO $Podar(hijo, Cota)$

```
            si esSolucion(hijo)
              si esMejor(mejorSolucion, hijo)
                mejorSolucion ← hijo
                Cota ← ActualizarCota(Cota, hijo)
              fin si
            sino
              agregar(ENV, hijo)
              Cota ← ActualizarCota(Cota, hijo)
            fin si
          fin si
        fin para
      fin si
    fin mientras
```

El costo de una solución aplicando esta estrategia no necesariamente mejora al backtracking tradicional. El costo dependerá de la cantidad de nodos explorados. En el peor de los casos, si se exploran todos los nodos, el costo es igual al de backtracking mientras que si se aplica una buena estrategia de ramificación y de poda, se podrán eliminar muchos nodos y así explorar menos. Esto significa que se podrá mejorar mucho el tiempo para el caso promedio, pero no el costo para el peor caso.

5.3.1. Mochila

Resolveremos el ya conocido problema de la mochila aplicando la técnica de ramificación y poda. La solución parcial se podrá almacenar en un vector de tamaño de la cantidad de objetos y el contenido será un 0 o un 1 según si el objeto pertenece o no a esa solución. Supondremos que los objetos están ordenados de mayor a menor según su valor absoluto, es decir estarán ordenados de mayor a menor según la relación *valor/peso*. Las opciones de ramificación en cada nodo serán siempre dos, según se incluya o no se incluya al nodo evaluado en la solución. La estrategia de ramificación será seleccionar aquel nodo que tenga una mayor cota superior, o sea el más prometedor, aunque pueda no ser el mejor. La cota inferior para cada nodo será el valor de la solución parcial hasta ese nodo y la cota superior será el valor de la solución parcial hasta ese nodo más el máximo peso que se podría obtener desde esa etapa hasta el final. Una opción sería suponer que la capacidad restante de la mochila se completa con la cantidad necesaria del próximo nodo. Si bien esto es imposible porque no se puede llevar mas de un objeto de cada uno, sí es verdad que ese valor será una cota superior por la forma en que están ordenados los objetos.
En cada nodo almacenaremos la solución parcial, la etapa, el peso actual, el valor actual y las dos cotas.

SolucionParcial
etapa
PesoParcial
ValorParcial
CotaInferior
CotaSuperior

Y el algoritmo principal será

```
ALGORITMO R&P_MOCHILA
Entrada: Objetos, Capacidad
Salida: mejorSolucion
  Cota ← 0
  ENV ← inicializarColaPrioridad()
  nodoRaiz ← CrearNodoRaiz(Objetos, Capacidad)
  agregar(ENV, nodoRaiz, nodoRaiz.CotaSuperior)
  Cota ← ActualizarCota(nodoRaiz, Cota)
  mejorSolucion ← NULO
  mientras ENV ≠ ∅
    nodo ← Primero(ENV)
    EliminarPrimero(ENV)
    si NOPodar(nodo, Cota)
      hijos ← GenerarHijos(nodo, Objetos, Capacidad)
      para cada hijo ∈ hijos
        si NOPodar(hijo, Cota)
          si esSolucion(hijo)
            si esMejorSolucion(mejorSolucion, hijo)
              mejorSolucion ← hijo
              Cota ← ActualizarCota(hijo, Cota)
            fin si
          sino
            Agregar(ENV, hijo, hijo.CotaSuperior)
            Cota ← ActualizarCota(hijo, Cota)
          fin si
        fin si
      fin para
    fin si
  fin mientras
```

Y los algoritmos auxiliares son:

ALGORITMO CREARNODORAIZ
Entrada: $Objetos, Capacidad$
Salida: $nodo$
  $nodo \leftarrow inicializarNodo()$
  $nodo.SolucionParcial \leftarrow inicializarVector(tamanio(Objetos))$
  $nodo.etapa \leftarrow 0$
  $nodo.PesoParcial \leftarrow 0$
  $nodo.ValorParcial \leftarrow 0$
  $nodo.CotaInferior \leftarrow 0$
  $nodo.CotaSuperior \leftarrow ActualizarCotaSuperior(nodo, Objetos, Capacidad)$

ALGORITMO ACTUALIZARCOTASUPERIOR
Entrada: $nodo, Objetos, Capacidad$
Salida: $valor$
  si $nodo.etapa = tamanio(Objetos)$
    devolver $nodo.ValorParcial$
  sino
    devolver $nodo.ValorParcial + (Capacidad - nodo.PesoParcial) * Objetos[nodo.etapa + 1].valor/Objetos[nodo.etapa + 1].peso$
  fin si

ALGORITMO ACTUALIZARCOTA
Entrada: $nodo, Cota$
Salida: $Cota$
  si $nodo.CotaInferior > Cota$
    devolver $nodo.CotaInferior$
  sino
    devolver $Cota$
  fin si

ALGORITMO PODAR
Entrada: $nodo, Cota$
Salida: V/F
  si $nodo.CotaSuperior < Cota$
    devolver V
  sino

```
    devolver F
  fin si
```

ALGORITMO GENERARHIJOS
Entrada: $nodo, Objetos, Capacidad$
Salida: $Hijos$

```
Hijos ← inicializarConjunto()
si nodo.etapa < tamanio(Objetos)
  hijo1 ← inicializarNodo()
  hijo1.SolucionParcial ← copiar(nodo.SolucionParcial)
  hijo1.SolucionParcial[nodo.etapa + 1] ← 0
  hijo1.etapa ← nodo.etapa + 1
  hijo1.PesoParcial ← nodo.PesoParcial
  hijo1.ValorParcial ← nodo.ValorParcial
  hijo1.CotaInferior ← hijo1.ValorParcial
  hijo1.CotaSuperior ← ActualizarCotaSuperior(hijo1, Objetos, Capacidad)
  agregar(Hijos, hijo1)
  si nodo.PesoParcial + Objetos[nodo.etapa + 1].peso ≤ capacidad
    hijo1 ← inicializarNodo()
    hijo1.SolucionParcial ← copiar(nodo.SolucionParcial)
    hijo2.SolucionParcial[nodo.etapa + 1] ← 1
    hijo2.etapa ← nodo.etapa + 1
    hijo2.PesoParcial ← nodo.PesoParcial + Objetos[nodo.etapa +
    1].peso
    hijo2.ValorParcial ← nodo.ValorParcial + Objetos[nodo.etapa +
    1].valor
    hijo2.CotaInferior ← hijo2.ValorParcial
    hijo2.CotaSuperior ← ActualizarCotaSuperior(hijo2, Objetos, Capacidad)
    agregar(Hijos, hijo2)
  fin si
fin si
```

ALGORITMO ESSOLUCION
Entrada: $nodo$
Salida: V/F

```
si nodo.etapa = tamanio(nodo.SolucionParcial)
  devolver V
sino
  devolver F
```

fin si

ALGORITMO esMejorSolucion
Entrada: $mejorSolucion, nodo$
Salida: V/F
si $nodo.ValorParcial > mejorSolucion.ValorParcial$
devolver V
sino
devolver F
fin si

Una posible codificación del problema sería:

```
public static class mochilaNodo{
  VectorTDA<Integer> actualSolucion;
  int etapa;
  int actualPeso;
  int actualValor;
  int cInf;
  int cSup;
}

public static void mochilaNodoRaiz(mochilaNodo raiz,
   VectorTDA<Integer> valores, VectorTDA<Integer> pesos
   , int capacidad) throws Exception{
  raiz.actualSolucion = new Vector<Integer>();
  raiz.actualSolucion.inicializarVector(valores.
     capacidadVector());
  for(int i = 0; i < valores.capacidadVector();i++){
     raiz.actualSolucion.agregarElemento(i, 0);
  }
  raiz.etapa = -1;
  raiz.actualPeso = 0;
  raiz.actualValor = 0;
  raiz.cInf = 0;
  raiz.cSup = mochilaCotaSuperior(valores,pesos,raiz,
     capacidad);
}

public static int mochilaCotaSuperior(VectorTDA<Integer
   > valores, VectorTDA<Integer> pesos, mochilaNodo n,
   int capacidad) throws Exception{
```

```
      if(n.etapa +1 == valores.capacidadVector()){
         return n.actualValor;
      }else{
         return n.actualValor + (capacidad - n.actualPeso)*
            valores.recuperarElemento(n.etapa+1)/pesos.
            recuperarElemento(n.etapa+1);
      }
   }

   public static int mochilaGenerarHijos(mochilaNodo n,
      VectorTDA<Integer> valores, VectorTDA<Integer> pesos
      , int capacidad, VectorTDA<mochilaNodo> hijos)
      throws Exception{

      int cantHijos = 0;

      if(n.etapa +1 < valores.capacidadVector()){
         cantHijos = 1;
         mochilaNodo h1 = new mochilaNodo();
         h1.actualSolucion = new Vector<Integer>();
         h1.actualSolucion.inicializarVector(valores.
            capacidadVector());
         copiarVector(n.actualSolucion,h1.actualSolucion,n.
            actualSolucion.capacidadVector());
         h1.actualSolucion.agregarElemento(n.etapa+1, 0);
         h1.actualPeso = n.actualPeso;
         h1.actualValor = n.actualValor;
         h1.etapa = n.etapa+1;
         h1.cInf = n.actualValor;
         h1.cSup = mochilaCotaSuperior(valores,pesos,h1,
            capacidad);

         hijos.agregarElemento(0, h1);

         if(n.actualPeso + pesos.recuperarElemento(n.etapa
            +1) <= capacidad){
            cantHijos = 2;
            mochilaNodo h2 = new mochilaNodo();
            h2.actualSolucion = new Vector<Integer>();
            h2.actualSolucion.inicializarVector(valores.
               capacidadVector());
            copiarVector(n.actualSolucion,h2.actualSolucion
               ,n.actualSolucion.capacidadVector());
            h2.actualSolucion.agregarElemento(n.etapa+1, 1)
               ;
            h2.actualPeso = n.actualPeso+pesos.
```

```
            recuperarElemento(n.etapa+1);
        h2.actualValor = n.actualValor+valores.
            recuperarElemento(n.etapa+1);
        h2.etapa = n.etapa+1;
        h2.cInf = h2.actualValor;
        h2.cSup = mochilaCotaSuperior(valores,pesos,h2,
            capacidad);
        hijos.agregarElemento(1, h2);
      }
    }

    return cantHijos;
  }

  public static int mochilaValorPoda(mochilaNodo n, int
      cota){
    return Math.max(n.cInf, cota);
  }

  public static boolean mochilaEsSolucion(mochilaNodo n){
    return (n.etapa+1 == n.actualSolucion.capacidadVector
        ());
  }


  public static VectorTDA<Integer> rypMochila(VectorTDA<
      Integer> valores, VectorTDA<Integer> pesos, int
      capacidad) throws Exception{

    int cota = 0;

    mochilaNodo nAux;

    mochilaNodo raiz = new mochilaNodo();
    mochilaNodoRaiz(raiz,valores,pesos,capacidad);

    mochilaNodo mejorSolucion = raiz;

    ColaPrioridadTDA<mochilaNodo> LNV = new ColaPrioridad
        <mochilaNodo>();
    LNV.InicializarCola();

    LNV.AgregarElemento(raiz, raiz.cSup);

    cota = mochilaValorPoda(raiz,cota);
```

```
    VectorTDA<mochilaNodo> hijos = new Vector<mochilaNodo
       >();
    hijos.inicializarVector(2);
    int cantHijos;

    while(!LNV.ColaVacia()){
       nAux = LNV.RecuperarMaxElemento();
       LNV.EliminarMaxPrioridad();
       if(nAux.cSup >= cota){

          cantHijos = mochilaGenerarHijos(nAux,valores,
             pesos,capacidad,hijos);
          for(int i = 0 ; i< cantHijos; i++){
             if (hijos.recuperarElemento(i).cSup >= cota)
                {
                if(mochilaEsSolucion(hijos.
                   recuperarElemento(i))){
                   if(hijos.recuperarElemento(i).
                      actualValor >= cota){
                      mejorSolucion = hijos.
                         recuperarElemento(i);
                      cota = hijos.recuperarElemento(i).
                         actualValor;
                   }
                }else{
                   cota = mochilaValorPoda(hijos.
                      recuperarElemento(i),cota);
                   LNV.AgregarElemento(hijos.
                      recuperarElemento(i), hijos.
                      recuperarElemento(i).cSup);
                }
             }
          }
       }
    }

    return mejorSolucion.actualSolucion;
  }
```

5.3.2. Asignación de tareas

El enunciado del problema dice *Se tienen* $\boldsymbol{n}$ *empleados y* $\boldsymbol{n}$ *tareas, se conoce el tiempo que tardará el empleado* $\boldsymbol{i}$ *para realizar la tarea* $\boldsymbol{j}$*. Se requiere encontrar una asignación de una tarea a cada empleado de manera de minimizar el tiempo*

de ejecución del total de tareas.
La solución parcial se podrá almacenar en un vector de tamaño de la cantidad de empleados y el contenido será un valor entre 1 y **n** que será la tarea asignada a cada empleado. Las opciones de ramificación en cada nodo serán las posibles tareas a asignar al empleado correspondiente, es decir aquellas que aún no hayan sido asignadas a un empleado en una etapa anterior. La estrategia de ramificación será seleccionar aquel nodo que tenga una mayor cota superior, o sea el más prometedor, aunque pueda no ser el mejor.
La cota inferior para cada nodo será el valor de la solución parcial hasta ese nodo. También podría ser ese valor más el resultado de asignar a cada empleado pendiente (o sea de una etapa posterior) la tarea de menor costo, de las aún no asignadas. De igual forma, la cota superior será el valor de la solución parcial hasta ese nodo más el máximo valor que se podría obtener asignando a cada empleado de una etapa superior, la tarea de mayor costo, de las aún no asignadas. El costo de calcular estas cotas será superior al que se utilizó en el problema anterior.
En cada nodo almacenaremos la solución parcial, la etapa, el valor actual y las dos cotas.
A diferencia del problema anterior, donde el objetivo era maximizar un valor, aquí estamos minimizando un tiempo.
Si bien el algoritmo principal no variará, salvo en los datos de entrada que en este caso será una matriz de $n \times n$ donde $M[i, j]$ será el tiempo que tarda el empleado i para realizar la tarea j.
Los algoritmos auxiliares serán entonces:

ALGORITMO CREARNODORAIZ
Entrada: M
Salida: *nodo*
  $nodo \leftarrow inicializarNodo()$
  $nodo.SolucionParcial \leftarrow inicializarVector(tamanio(M))$
  $nodo.etapa \leftarrow 0$
  $nodo.ValorParcial \leftarrow 0$
  $nodo.CotaInferior \leftarrow 0$
  $nodo.CotaSuperior \leftarrow ActualizarCotaSuperior(nodo, M)$

ALGORITMO ACTUALIZARCOTASUPERIOR
Entrada: $nodo, M$
Salida: *valor*
  si $nodo.etapa = tamanio(M)$
    devolver $nodo.ValorParcial$

```
  sino
    valor ← nodo.ValorParcial
    para i = nodo.etapa + 1 hasta tamanio(M)
      valor ← valor+elegirMaximaNoAsignada(nodo.SolucionParcial, M, i)
    fin para
    devolver valor
  fin si
```

Algoritmo ActualizarCota

Entrada: $nodo, Cota$
Salida: $Cota$

```
  si nodo.CotaSuperior < Cota
    devolver nodo.CotaSuperior
  sino
    devolver Cota
  fin si
```

Algoritmo Podar

Entrada: $nodo, Cota$
Salida: V/F

```
  si nodo.CotaInferior > Cota
    devolver V
  sino
    devolver F
  fin si
```

Algoritmo GenerarHijos

Entrada: $nodo, M$
Salida: $Hijos$

```
  Hijos ← inicializarConjunto()
  para j = 1 hasta tamanio(M)
    si NO TareaUsada(nodo, j)
      hijo ← inicializarNodo()
      hijo.SolucionParcial ← copiar(nodo.SolucionParcial)
      hijo.SolucionParcial[nodo.etapa + 1] ← j
      hijo.etapa ← nodo.etapa + 1
      hijo.ValorParcial ← nodo.ValorParcial + M[nodo.etapa + 1, j]
```

```
        hijo.CotaInferior ← hijo.ValorParcial
        hijo.CotaSuperior ← ActualizarCotaSuperior(hijo, M)
        agregar(Hijos, hijo)
      fin si
    fin para
```

```
Algoritmo esSolucion
Entrada: nodo
Salida: V/F
  si nodo.etapa = tamanio(nodo.SolucionParcial)
    devolver V
  sino
    devolver F
  fin si
```

```
Algoritmo esMejorSolucion
Entrada: mejorSolucion, nodo
Salida: V/F
  si nodo.ValorParcial < mejorSolucion.ValorParcial
    devolver V
  sino
    devolver F
  fin si
```

5.3.3. Problema del viajante

Si quisiéramos resolver el problema del viajante analizado previamente, utilizando estrategias de ramificación y poda, la estrategia será la misma. Como es un problema de minimización, ya que se trata de encontrar el ciclo hamiltoniano de menor costo, sólo es imprescindible calcular una cota inferior. La cota superior es opcional. Si no la calculamos, la cota general sólo se actualizará cuando se vayan encontrando las soluciones candidatas. Se podrá seleccionar como cota inferior el valor del camino de la solución parcial o se podrá mejor sumándole el costo de la arista saliente de menor valor de cada uno de los vértices aún no visitados del grafo en la solución parcial. Como de todos los vértices pendientes de visitar, se deberá "salir"por una arista de las existentes, la menor de todas las aristas será una cota inferior, aunque posiblemente no sea una solución real al problema. El algoritmo será similar a los ya vistos.

5.4. Resolución de juegos

Existen determinados juegos de antagonismo, como el TA-TE-TI, los fósforos, y las damas que pueden ser resueltos utilizando la técnica de Backtracking, debido a que podemos modelar los estados del juego mediante un árbol. Cada nivel del árbol será una etapa del juego, y las posibles jugadas para el jugador de turno. Cada jugador en su turno intentará maximizar su resultado. Como consecuencia de intentar maximizar el jugador A será minimizar al jugador B, la técnica se denominará *MINMAX*. Si en lugar de considerar todas las posibles situaciones del juego, porque el árbol sería muy grande, podemos predecir sus resultados haciendo uso de la idea de la técnica de Ramificación y Poda, utilizaremos la teoría de la *poda alfa beta*.
Para ejemplificar la teoría del Backtracking de juegos vamos a utilizar el juego de los fósforos. *Se disponen de 11 fósforos. Dos jugadores pueden retirar por turno 1, 2 o 3 fósforos. Pierde aquel jugador que retira el último fósforo. Construir un algoritmo que determine para un jugador si existe una secuencia de jugadas ganadora.*
Al principio del juego se poseen 11 fósforos. El primer jugador puede tener tres opciones, es decir: tomar uno, dos o tres fósforos; y el segundo jugador tiene también tres opciones. De esta manera, podemos modelar el juego completo mediante un árbol ternario como el que se muestra en la figura 5.3.

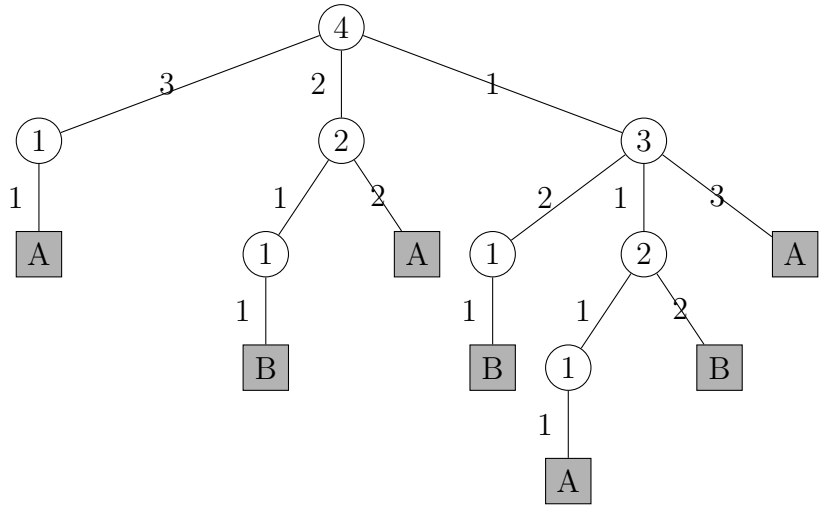


Figura 5.3: Arbol del juego de los fósforos

En la figura 5.3, escribimos la cantidad de fósforos disponibles para sacar por un jugador dentro del nodo, en este caso comenzamos con cuatro fósforos. Escribimos la cantidad de fósforos tomadas por el jugador junto a los arcos. La cantidad de fósforos de un nodo menos los fósforos tomados en el arco dará la cantidad de fósforos del nodo inferior. Obviamente, el juego terminará de forma inmediata si

la cantidad de fósforos llega a cero. En ese punto se define quién fue el ganador, suponiendo que comienza jugando el jugador A. En la figura se muestran algunas de las posibles alternativas del juego a modo de ejemplo.

5.4.1. MINMAX

Para evitar confusión, pensamos siempre en términos de primer jugador y consideramos el beneficio como el resultado que el primer jugador recibe al fin del juego. Si gana el segundo jugador, entonces el resultado es negativo. De esta forma, el primer jugador desea maximizar el resultado, en tanto que el segundo jugador desea minimizar el resultado. Los jugadores serán llamados MAX y MIN de ahora en adelante. Cuando es el turno del jugador MAX, el nodo es de color gris, en tanto que cuando es el turno del jugador MIN, el color del nodo es blanco. Re-dibujaremos ahora el juego como se muestra en la figura 5.4, donde el número dentro del nodo indica el resultado obtenido en el juego (se considera que el jugador MAX gana si obtiene el valor 1 y el jugador MIN gana si obtiene el valor -1).

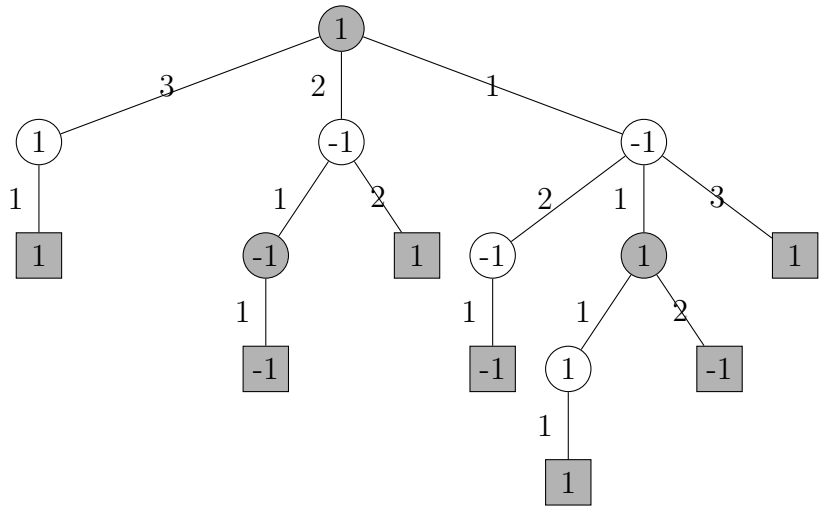


Figura 5.4: MIN-MAX

Al principio, sólo conocemos el valor dentro de los nodos terminales(cuadrados en fondo negro). 1 si ganó MAX y -1 si ganó MIN. La pregunta es *¿Quién es el ganador si ambos jugadores juegan de forma óptima?*
Es fácil responder a esa pregunta observando los nodos del nivel más bajo y ascendiendo hacia la raíz.
Consideremos el tercer nodo del segundo nivel donde quedan tres fósforos en la pila, y es el turno del jugador MIN. Obviamente, MIN no va a tomar ni tres fósforos ni uno, ya que perdería directamente o daría la posibilidad de ganar a MAX, MIN debería tomar dos fósforos y dejar con un solo fósforo a MAX y

así ganar el juego. De esta manera, si el juego alcanza el nivel de ese nodo, el resultado será -1 y podremos escribir -1 dentro del nodo. Es decir que el jugador MIN tomará de sus hijos el de menor valor. Como tiene dos hijos con valor 1 y uno con valor -1, ese es el que tomará.
De igual forma, en el nodo raíz, es el turno del jugador MAX y quedan cuatro fósforos, él tomará definitivamente 3 fósforos, dado que el máximo entre sus tres hijos, $max(1, -1, -1) = 1$. De esta manera podemos escribir 1 dentro del nodo raíz. Esto significa que con cuatro fósforos, el jugador MAX tiene asegurado el éxito.
Como complemento, si en un nodo MAX ya encontramos un hijo con valor 1, no será necesario explorar los demás hijos, ya que 1 es el mejor valor que se podría obtener y por cualquiera de las otras ramas no lo podríamos mejorar. De igual forma, si en un nodo MIN encontramos un hijo con valor -1, no será necesario seguir buscando por otras ramas. Este concepto se podrá perfeccionar con la estrategia de poda alfa beta.
La implementación de un backtracking para el juego de los fósforos sería la siguiente:

```
public int juegoFosforos(int cantFosforos, int
   nivelJuego, ConjuntoTDA<Jugada> jugadas){
   Jugada jugada = new Jugada();
   if (cantFosforos == 0){
      return -1*nivelJuego;
   }else{
      int hijos = 1;
      int proxNivelJuego;
      int valor = 0;
      boolean podar = false;
      if (nivelJuego == 1){
           proxNivelJuego = -1;
           valor = -1;
      }else{
         proxNivelJuego = 1;
         valor = 1;
      }
      while (hijos <= 3 && hijos <= cantFosforos && !
         podar){
         cantFosforos -= hijos;
         jugada.cantFosforos = hijos;
         jugada.jugada = nivelJuego;
         jugadas.agregar(jugada);
         if (nivelJuego == 1){
            valor = Math.max(valor, juegoFosforos(
               cantFosforos, proxNivelJuego, jugadas));
         }else{
```

```
                valor = Math.min(valor, juegoFosforos(
                    cantFosforos, proxNivelJuego, jugadas));
            }
            if ((nivelJuego == 1 && valor == 1) || (
                nivelJuego == -1 && valor == -1)){
                podar = true;
            }else{
                cantFosforos += hijos;
                jugadas.sacar(jugada);
            }
            hijos++;
        }
        return valor;
    }
}
```

En base a lo anterior podemos describir el esquema general para backtracking de juegos de la siguiente manera:

ALGORITMO BACKJUEGO
Entrada: e: estado, n: nivel
Salida: v: valor
  si $hoja(e, n)$
    devolver $directo(e, n)$; `//calcula la solución directa, retorna 1, -1 o 0`
  sino
    $nrohijo \leftarrow 1$
    si $n = max$
      $valor \leftarrow -\infty$
    sino
      $valor \leftarrow +\infty$
    fin si
    mientras $hijos(e, nrohijo, n, siguiente)$
      //hijos retorna 1 o 0 si tiene o no un hijo, siguiente retorna el hijo
      si NO $podado(siguiente, n)$
        si $n = max$
          $valor \leftarrow maximo(valor, backjuego(siguiente, min))$
        sino
          $valor \leftarrow minimo(valor, backjuego(siguiente, max))$
        fin si
      fin si
      $nrohijo++$
    fin mientras

```
    devolver valor
fin si
```

Capítulo 6

Complejidad computacional

Hasta ahora nos hemos dedicado al estudio de la algoritmia, que si repasamos el concepto, tiene que ver con el estudio de técnicas, eficiencia y eficacia. Existe otra ciencia, que complementa a la primera que se denomina *Complejidad*.
En el análisis de costo de nuestros algoritmos, nos encargamos de analizar la eficiencia de un algoritmo para resolver un problema, pero no teníamos forma de saber si la solución encontrada es la mejor o si se podría encontrar una solución para el mismo problema con un costo menor. De eso se ocupa la complejidad, de analizar si sería posible encontrar una solución mejor para un algoritmo.
Existen diferentes técnicas para el análisis de la complejidad computacional de un problema. Está la técnica del árbol de decisión, que arma un árbol con las diferentes soluciones que puede tener un problema como las hojas del árbol, y los nodos internos como puntos de decisión de la solución de un problema. En base a eso se puede calcular la mínima cantidad de pasos que serían necesarios para alcanzar todas las posibles soluciones. Otra técnica es la llamada técnica del adversario que consiste en ir tomando en cada etapa la "peor" decisión posible para el algoritmo, que lo conduzca a una decisión. La longitud de ese "peor" camino será la complejidad mínima del problema. En todas estas técnicas, además de ser muy difíciles de demostrar y probar, se llega a una cota inferior teórica pero no práctica. Esto significa que si demostramos que un problema A tiene una complejidad de $\mathcal{O}(n \log n)$ podría existir una solución en ese orden, pero no necesariamente la encontraremos, con lo cual podría perder interés. Lo interesante es que la misma idea se puede usar para demostrar que no se podría encontrar nada mejor y eso sí es importante, por ejemplo el ordenamiento tiene una complejidad $\mathcal{O}(n \log n)$, eso significa que por más que sigamos buscando nuevas técnicas, no encontraremos ninguna con un orden mejor al ya encontrado. Dicho de otra forma, la complejidad computacional es una cota inferior para todos los algoritmos que resuelven un problema.
Como dijimos que es muy difícil encontrar la complejidad exacta de un problema, se apela a la idea de equivalencia entre problemas. Para eso se define la idea de reducibilidad lineal. Un problema A se reduce a otro B si se puede aplicar una

transformación sobre una entrada X de A de tal forma de obtener una entrada Y para B y al aplicar $B(Y)$ se obtiene una solución para A. Si A es reducible a B y B es reducible a A, entonces los problemas son equivalentes. Los problemas equivalentes tienen una complejidad similar. Esto es muy ventajoso ya que si se encuentra un algoritmo mejor para un problema se pueden mejorar los algoritmos para los otros problemas.
Existen varios problemas para los que no se han encontrado soluciones eficientes, por ejemplo el problema del viajante, satisfacibilidad de fórmulas lógicas, etc. Un algoritmo se dice *eficiente* si tiene un costo polinómico que lo resuelve. Esto se basa en la idea de que un problema $\mathcal{O}(n^k)$ no tendrá casi nunca un k demasiado grande. Entonces, un problema se dice *tratable* si tiene un algoritmo eficiente que lo resuelve. Esta clase de algoritmos son los que denominaremos $\mathcal{P}$ aunque existen excepciones, por ejemplo algoritmos donde los valores de k son muy altos o las constantes multiplicativas ocultas son muy grandes. La forma de demostrar que un problema está en esta clase $\mathcal{P}$ es encontrando el algoritmo o reduciéndolo a otro conocido.
Veamos una simple tabla en la que suponemos que cada instrucción tarda un microsegundo en ejecutarse, para entender porqué definimos tratabilidad como equivalente de tiempo polinómico.

	1	10	50	100	500
n	1/1000000 seg	1/100000 seg	1/20000 seg	1/10000 seg	1/2000 seg
n^3	1/1000000 seg	1/1000 seg	0.125 seg	1 seg	2.08 min
n^5	1/1000000 seg	0.1 seg	5.min	2.77 horas	361 días
2^n	1/500000 seg	16/15625 seg	37.5 años	400 billones de siglos	–
$n!$	1/1000000 seg	3.6 seg	–	–	–
n^n	1/1000000 seg	2.7 horas	–	–	–

Existe un conjunto de problemas para los cuales aún no se encontraron soluciones en tiempo polinómico para resolverlos, pero dado el problema y una "posible solución" se puede encontrar un algoritmo que en tiempo polinómico determina si la "posible solución" es una solución real al problema. Todos estos problemas para los cuales se puede verificar en tiempo polinómico una posible solución estarán en una clase que denominaremos $\mathcal{NP}$ (del inglés, tiempo polinómico no determinístico). Otra definición para esta clase de problemas surge de su sigla en inglés, que es posible encontrar un algoritmo no determinístico que resuelva el problema en tiempo polinómico. En realidad esto es cierto a medias, porque el algoritmo no determinístico podría encontrar la solución o podría no encontrarla en ese tiempo. Por ejemplo, no se encontró ninguna solución en tiempo polinómico para el problema del ciclo hamiltoniano pero dado un grafo y un ciclo es fácil verificar en tiempo polinómico si dicho ciclo es hamiltoniano.
Es fácil demostrar que $\mathcal{P} \subseteq \mathcal{NP}$ ya que si existe un algoritmo que resuelve en

tiempo polinómico un algoritmo también existirá un algoritmo que en tiempo polinómico verifique que una entrada es solución para el problema.
Lo que aún no se pudo demostrar es si $\mathcal{NP} \subseteq \mathcal{P}$, o lo que es lo mismo $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$. Para demostrar que esto no es cierto bastaría con demostrar que existe algún problema que está en $\mathcal{NP}$ que no está en $\mathcal{P}$.
Existe una subclase de $\mathcal{NP}$ que son los problemas $\mathcal{NP}$ completos o $\mathcal{NPC}$. Son aquellos problemas para los que no se encontró ningún algoritmo que los resuelva en tiempo polinómico pero tampoco fue posible aún demostrar que es imposible encontrarlo. Más formalmente, un problema L es $\mathcal{NP}$ completo o $\mathcal{NPC}$ si está en $\mathcal{NP}$ y además cualquier otro problema $X \in \mathcal{NP}$ puede reducirse a él en tiempo polinómico. Ejemplos de problemas de esta clase son:

- **CLIQUE**: dado un grafo, encontrar el mayor subgrafo que sea fuertemente conexo.
- **Satisfacibilidad**: encontrar una combianación de valores lógicos para que una fórmula bien formada sea verdadera
- **Ciclo hamiltoniano**: encontrar un ciclo hamiltoniano.
- **Viajante**: encontrar el mínimo ciclo hamiltoniano.
- **Camino más largo**: encontrar el camino más largo entre un par de vértices de un grafo.
- **Cubrimiento de nodos**: sea $G = \langle N, A \rangle$, encontrar el mínimo $N' \subseteq N$ tal que para todo $a \in A$ alguno de los extremos de de a está en N'.

Esta clase de problemas hace sospechar que $\mathcal{P} \neq \mathcal{NP}$ pero aún no se pudo demostrar. Como todos estos problemas son equivalentes, podrían pasar dos cosas. La primera es que se encuentre una solución en tiempo polinómico para alguno de estos problemas, con lo cual existiría una solución para todos ellos y se demostraría que $\mathcal{P} = \mathcal{NP}$. Lo otro que puede pasar es que se demuestre que no es posible encontrar una solución en tiempo polinómico para alguno de estos problemas, con lo cual no existiría tal solución para ninguno y demostraríamos que $\mathcal{P} \neq \mathcal{NP}$.
Por último, existe otra clase de problemas, que son los problemas $\mathcal{NP}hard$ o $\mathcal{NP}dificil$ o $\mathcal{NP}complejo$, son aquellos problemas para los que no se pudo demostrar que están en $\mathcal{NP}$ pero sí se puede demostrar que que cualquier problema en $\mathcal{NP}$ se puede reducir a ellos. Es decir, de las dos condiciones de $\mathcal{NPC}$, cumple con la segunda y no la primera. Esto significa que $\mathcal{NPC} \subseteq \mathcal{NP}hard$. Esta definición implica decir que si un problema es $\mathcal{NP}hard$ es al menos tan difícil como cualquier problema en $\mathcal{NP}$.
En resumen, el diagrama de las clase vistas sería el de la figura 6.1 y si en cambio se lograra demostrar que $\mathcal{P} = \mathcal{NP}$, entonces el diagrama quedaría como en la figura 6.2.

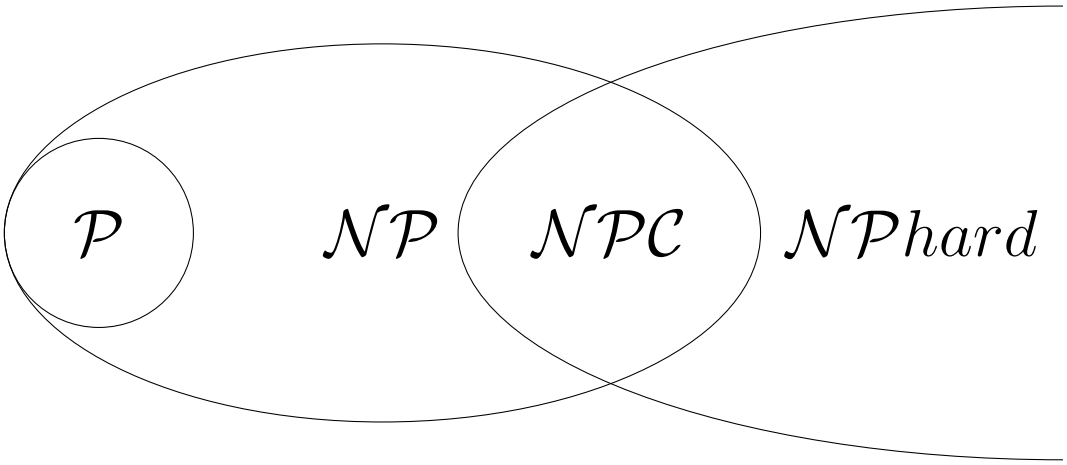


Figura 6.1: Diagrama de clases de complejidad con $\mathcal{P} \neq \mathcal{NP}$

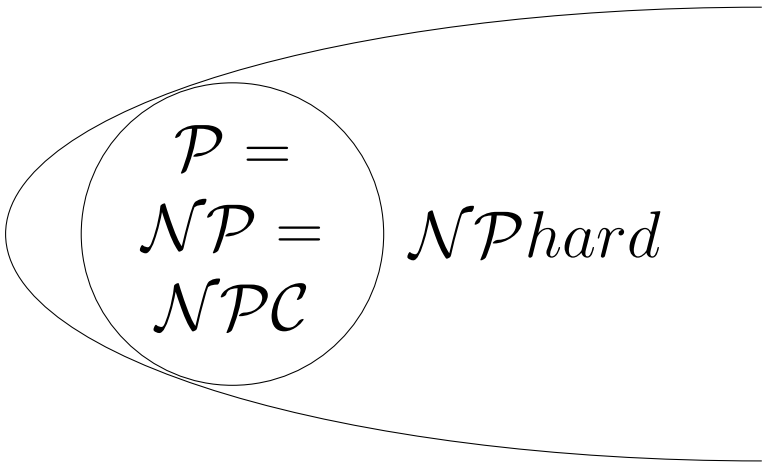


Figura 6.2: Diagrama de clases de complejidad con $\mathcal{P} = \mathcal{NP}$

Apéndice A

Java

Para el desarrollo de la materia, seguiremos utilizando el lenguaje de programación *Java*. Agregaremos algunos elementos del lenguaje que nos servirán para los propósitos de la materia. Se trata de un manejo básico de errores y los tipos genéricos.

A.1. Generics

El uso de tipos de datos genéricos nos permite abstraernos de los tipos de datos concretos para el desarrollo de un módulo en particular. El ejemplo más claro es para todos aquellas clases que administran conjuntos de datos, las también llamadas colecciones. Por ejemplo, hemos visto ya pilas, colas, conjuntos, etc. Todos ellos deberían funcionar de igual forma si almacenamos elementos enteros, caracteres o incluso otra estructura, por ejemplo una pila de pilas.
La forma de definir en Java una clase que utiliza un tipo de datos genérico, es la siguiente:

```
class Pila<T>{
  ...
}
```

Luego, en todo el desarrollo de la clase, se utilizará el tipo genérico `T` para cualquier declaración y/o utilización de una variable de ese tipo. Siguiendo con el ejemplo anterior, tendríamos:

```
class Pila<T>{

  T Tope(){
     ...
  }

  void Apilar(T elemento){
```

```
        ...
    }
}
```

En el momento en que decidamos utilizar una clase con un tipo genérico, tanto en la declaración como en la creación, deberemos indicar de qué tipo estamos hablando. Los tipos válidos son todos aquellos a los que se asocia un objeto y no un tipo simple. Para los tipos simples, se deberán utilizar las clases java que los convierten en objetos, por ejemplo `Integer`, `Double`, `Character`, etc. Entonces, para utilizar una pila de enteros, tendremos:

```
Pila<Integer> p = new Pila<Integer>();
```

El uso de interfaces, como venimos haciendo desde Programación II no influye más que en colocar el tipo genérico también en la interface:

```
interface PilaTDA<T>{
    ...
    void Apilar(T elem);
    T Tope();
    ...
}

class Pila<T> implements PilaTDA<T>{

    T Tope(){
        ...
    }

    void Apilar(T elemento){
        ...
    }
}
```

y para utilizarlo:

```
PilaTDA<Integer> p = new Pila<Integer>();
```

A.2. Excepciones

Java, al igual que otros lenguajes de programación, ofrece un mecanismo de tratamiento de errores basado en excepciones. Hasta ahora la única forma de comunicación entre un método y su llamador eran los parámetros en una dirección y el retorno en el otro sentido. Para indicar que dentro del método no se pudo llegar al resultado esperado por algún motivo, la única alternativa que teníamos era utilizar un valor de retorno fuera del dominio del problema y el llamador se debía encargar de interpretarlo. El ejemplo más claro es cuando se esperaba un resultado entero positivo para un método y el retorno era un -1 como señal de error.
Las excepciones vienen a suplir este inconveniente, incorporando una nueva forma de comunicación llamada excepciones. Una excepción no es otra cosa que un objeto que viaja desde el método llamado hacia el método llamador. Indica una salida del método en forma errónea. Para utilizarlo, se deben tener en cuenta los siguientes puntos:

- El método llamado debe indicar en su prototipo que puede disparar una excepción.
- Si se trata de la implementación de un método de una interface, el prototipo en la interface ya debe indicar que puede disparar una excepción.
- El método llamador se debe encargar de tratar la excepción.
- Una alternativa al tratamiento de la excepción es no tratarla y dejar que el llamador del llamador la trate. En este último caso, se deberá indicar también en el método llamador que puede disparar una excepción.

Las excepciones son objetos de alguna clase que puede ser predefinida por el lenguaje o definida por el usuario. Para nuestros ejemplo utilizaremos siempre el tipo predefinido `Exception`. La forma de indicar que un método puede devolver una excepción es la siguiente:

```
void Apilar(T elem) throws Excepcion{
   ...
}
```

La forma de disparar una excepción es la siguiente:

```
void Apilar(T elem) throws Excepcion{

   //controla que no haya más elementos que la capacidad
       del vector
   if(i>100){
```

```
            throw new Exception("No hay suficiente espacio en
                la pila.");
        }
        ...
    }
```

Este código hace que la ejecución del método Apilar se detenga y el hilo de controla sigue en el método llamador.
Quien llama al método apilar, debe atrapar la excepción disparada, de la siguiente forma:

```
    ...
    try{
        p.Apilar(3);
    }catch(Exception e){
        System.out.println(e->getMessage());
    }
    ...
```

En este último ejemplo apilamos el valor 3, y contemplamos la opción de que dicha invocación dispare una excepción. Si ese es el caso, la excepción será `e`. La forma de mostrar el error es invocando al método `getMessage`. En el momento de hacer el tratamiento de la excepción se podría optar por disparar otra excepción, en cuyo caso se sale también de la ejecución del método llamador o se puede mostrar el mensaje en pantalla y seguir la ejecución normal.
También existe la posibilidad de realizar la llamada fuera de un bloque `try` en cuyo caso habría que colocar en el prototipo del método llamador la declaración `throws Exception` igual que en el otro método.

A.3. TDA

En esta sección vamos a definir los diferentes TDA que vamos a utilizar durante la materia. La mayoría de ellos ya fueron utilizados en la materia anterior.

A.3.1. Vector

La clase `Vector<E>` representa un vector de tamaño fijo donde el contenido es de un tipo genérico. La interface es la siguiente:

```
public interface VectorTDA<E>{

  void agregarElemento(int posicion, E elemento);
    //Permite agregar un elemento en una posición dada.
```

```
  int capacidadVector();
    //Devuelve la capacidad del vector

  void EliminarElemento(int posicion);
    //Eliminar un elemento del vector dada una posición

  void inicializarVector(int n);
    //Inicializa la estructura del vector con capacidad n

  E recuperarElemento(int posicion);
    //Permite recuperar el elemento de una posición dada.
}
```

A.3.2. Matriz

La clase `Matriz<E>` representa una matriz cuadrada de tamaño fijo y su contenido es de un tipo genérico. La interface es la siguiente:

```
public interface MatrizTDA<E>{

  void inicializarMatriz(int n);
    //Inicializa una matriz de nxn

  int obtenerDimension();
    //Devuelve la dimensión que tiene la matriz, es decir,
       si es una matriz de nxn, el valor que retorna es n

  E obtenerValor(int i, int j);
    //Obtiene el valor de la celda fila i y columna j

  void setearValor(int i, int j, E elem);
    //Setea un valor en la posición fila i y columna j
}
```

A.3.3. Cola con prioridad

La clase `ColaPrioridad<E>` recibe elementos con una prioridad asociada, y permite recuperar el elementos con mayor prioridad o menor prioridad.

```
public interface ColaPrioridadTDA<E>{

  void AgregarElemento(E e, double p);
```

```
    //Agrega el elemento e con la prioridad asociada p

  boolean ColaVacia();
    //Devuelve true si la cola no tiene elementos y false
       en caso contratrio

  void EliminarMaxPrioridad();
    //Elimina el elemento con mayor prioridad

  void EliminarMinPrioridad();
    //Elimina el elemento con menor prioridad

  void InicializarCola();

  E RecuperarMaxElemento();
    //Recupera el valor del elemento con mayor prioridad

  double RecuperarMaxPrioridad();
    //Recupera la prioridad del elemento con mayor
       prioridad

  E RecuperarMinElemento();
    //Recupera el valor del elemento con menor prioridad
  double RecuperarMinPrioridad();
    //Recupera la prioridad del elemento con menor
       prioridad
}
```

A.3.4. Conjunto

La clase `Conjunto<E>` de elementos que no permite elementos repetidos.

```
public interface ConjuntoTDA<E>{

  void agregar(E elemento);
    //PRECONDICION: conjunto Inicializado Agrega el
       elemento al conjunto siempre que ya no exista en el
       mismo

  boolean conjuntoVacio();
    //PRECONDICION: conjunto Inicializado Devuelve true si
       el conjunto no tiene elementos y false en caso
       contrario

  E elegir();
```

```
    //PRECONDICION: conjunto No Vacio Devuelve un elemento
       del conjunto en forma aleatoria

  void inicializarConjunto();
    //Incializa la estructura para poder comenzar a
       utilizarla

  boolean pertenece(E elemento);
    //PRECONDICION: conjunto Inicializado Devuelve true si
       el elemento pertenece al conjunto y false en caso
       contrario

  void sacar(E elemento);
    //PRECONDICION: conjunto Inicializado Elimina del
       conjunto un elemento dado
}
```

A.3.5. Conjunto con repetidos

La clase `ConjuntoRep<E>` es un conjunto que permite almacenar elementos repetidos. Su interface es la siguiente:

```
public interface ConjuntoRepTDA<E>{

  void agregar(E elemento);
    //PRECONDICION: conjunto Inicializado Agrega el
       elemento al conjunto siempre que ya no exista en el
       mismo

  boolean conjuntoVacio();
    //PRECONDICION: conjunto Inicializado Devuelve true si
       el conjunto no tiene elementos y false en caso
       contrario

  E elegir();
    //PRECONDICION: conjunto No Vacio Devuelve un elemento
       del conjunto en forma aleatoria

  void inicializarConjunto();
    //Incializa la estructura para poder comenzar a
       utilizarla

  boolean pertenece(E elemento);
    //PRECONDICION: conjunto Inicializado Devuelve true si
       el elemento pertenece al conjunto y false en caso
```

```
        contrario

  void sacar(E elemento);
    //PRECONDICION: conjunto Inicializado Elimina del
        conjunto un elemento dado
}
```

A.3.6. Grafo no dirigido

La clase `Grafo` representa un grafo no dirigido y su interface es la siguiente:

```
public interface GrafoTDA<E>{

  ConjuntoTDA<E> Adyacentes(E vertice);
    //PRECONDICION: grafo Inicializado, [vertice] existente
        .

  void AgregarArista(E vertice1, E vertice2, int costo);
    //PRECONDICION: grafo Inicializado, [Vertice1] y [
        Vertice2] existentes.

  void AgregarVertice(E vertice);
    //PRECONDICION: grafo Inicializado, y el vertice no
        debe existir

  E Elegir();
    //PRECONDICION: grafo Inicializado, Vertices del grafo
        No Vacio

  void EliminarArista(E vertice1, E vertice2);
    //PRECONDICION: grafo Inicializado, [Vertice1] y [
        Vertice2] existentes.

  void EliminarVertice(E vertice);
    //PRECONDICION: grafo Inicializado, [vertice] existente
        .

  boolean ExisteArista(E vertice1, E vertice2);
    //PRECONDICION: grafo Inicializado, [Vertice1] y [
        Vertice2] existentes.

  void InicializarGrafo();
    //Permite inicializar la estructura para poder comenzar
        a utilizarla
```

```
  int PesoArista(E vertice1, E vertice2);
    //PRECONDICION: grafo Inicializado, [Vertice1] y [
       Vertice2] existentes.

  ConjuntoTDA<E> Vertices();
    //PRECONDICION: grafo Inicializado
}
```

A.3.7. Grafo dirigido

La clase `GrafoDir` representa un grafo dirigido y su interface es la siguiente:

```
public interface GrafoDirTDA<E>{

  ConjuntoTDA<E> Adyacentes(E vertice);
    //PRECONDICION: grafo Inicializado, [vertice] existente
       .

  void AgregarArista(E vertice1, E vertice2, int costo);
    //PRECONDICION: grafo Inicializado, [Vertice1] y [
       Vertice2] existentes.

  void AgregarVertice(E vertice);
    //PRECONDICION: grafo Inicializado, y el vertice no
       debe existir

  E Elegir();
    //PRECONDICION: grafo Inicializado, Vertices del grafo
       No Vacio

  void EliminarArista(E vertice1, E vertice2);
    //PRECONDICION: grafo Inicializado, [Vertice1] y [
       Vertice2] existentes.

  void EliminarVertice(E vertice);
    //PRECONDICION: grafo Inicializado, [vertice] existente
       .

  boolean ExisteArista(E vertice1, E vertice2);
    //PRECONDICION: grafo Inicializado, [Vertice1] y [
       Vertice2] existentes.

  void InicializarGrafo();
    //Permite inicializar la estructura para poder comenzar
        a utilizarla
```

```
  int PesoArista(E vertice1, E vertice2);
    //PRECONDICION: grafo Inicializado, [Vertice1] y [
       Vertice2] existentes.

  ConjuntoTDA<E> Vertices();
    //PRECONDICION: grafo Inicializado
}
```

Bibliografía

[AA98] J. Hopcroft Y J. Ullman A. Aho. *Estructuras de Datos y Algoritmos*. Addison Wesley, Ubicación en biblioteca UADE: 681.3.01/A292, 1998.

[Bra96] P. Brassard, G. y Bratley. *Fundamental of Algorithmics*. Prentice Hall, Ubicación en biblioteca UADE: Piso 3 - Estantería 1 / Piso 1, 1996.

[Wei04] Mark Allen Weiss. *Estructuras de datos en JAVA*. Addison Wesley, Ubicación en biblioteca UADE: 004.43 WEI est [1a] 2004, 2004.

[yJC06] J. Lewis y J. Chase. *Estructuras de Datos con Java. Diseño de Estructuras y Algoritmos*. Addison Wesley Iberoamericana, 2006.

[yRT02] M. Goodrich y R. Tamassia. *Estructura de Datos y Algoritmos en Java*. Editorial Continental, 2002.