Departamento de Ciencias Aplicadas

Ingeniería en Computación.

Programación avanzada.

Unidad de competencia III: “Técnicas de diseño de algoritmos”

Presenta:
M. en C. C. J. Jair Vázquez Palma.

Programación avanzada

Objetivos de la Unidad de la Unidad de Aprendizaje

- Analizar y diseñar sistemas de información.
- Utilizar eficazmente los lenguajes de programación.
- Responder eficazmente a nuevas situaciones informáticas.
- Analizar soluciones del entorno y problemas propios de ser tratados mediante sistemas computacionales.
- Aplicar los conocimientos en la práctica.
- Desarrollar la habilidad análisis y síntesis de información.

Unidad III: Técnicas de diseño de algoritmos.

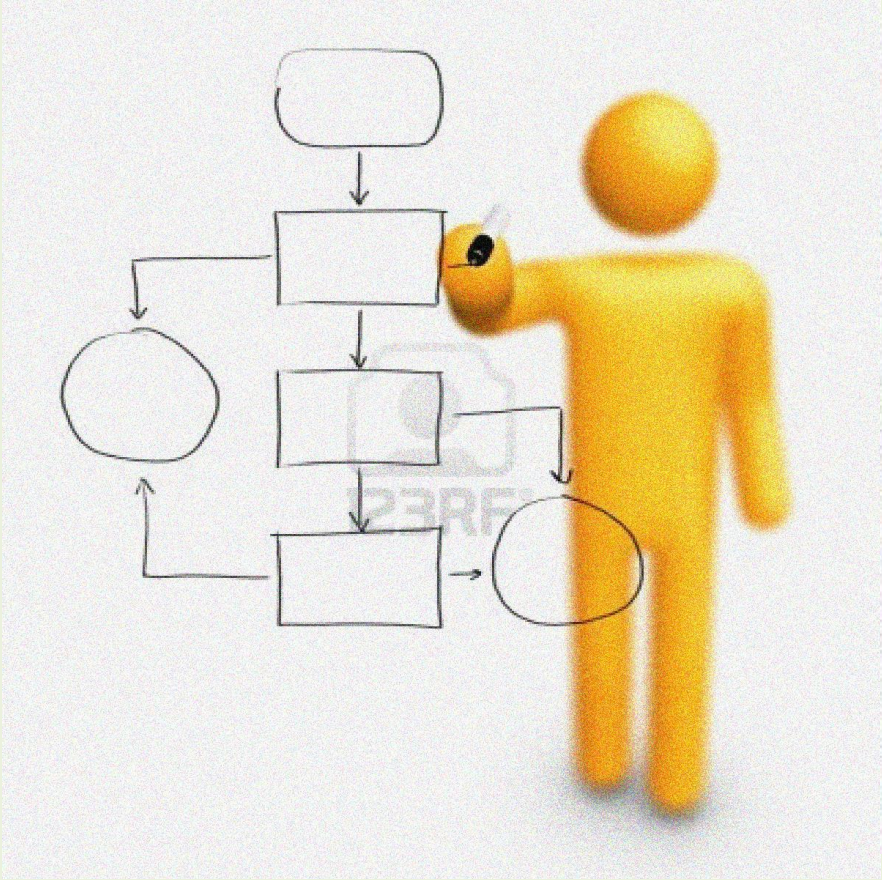
Contenido:

- Ecuaciones de recurrencia.
- Algoritmo voraces.
- Divide y vencerás.
- Programación dinámica.
- Vuelta atrás.
- Ramifica y poda.

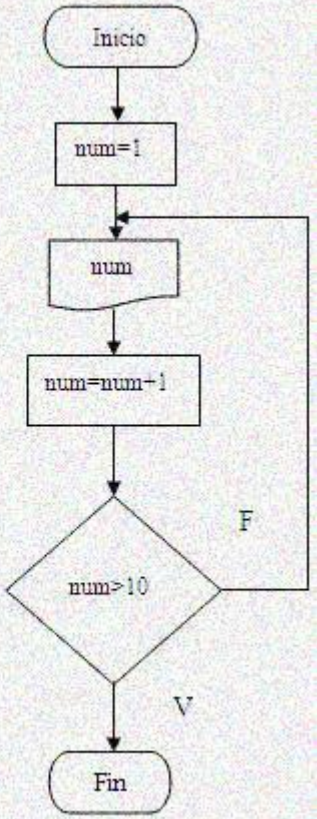
Objetivos de la Unidad de Competencia III

Diseñar algoritmos empleando diferentes técnicas de diseño dependiendo de la naturaleza del problema.

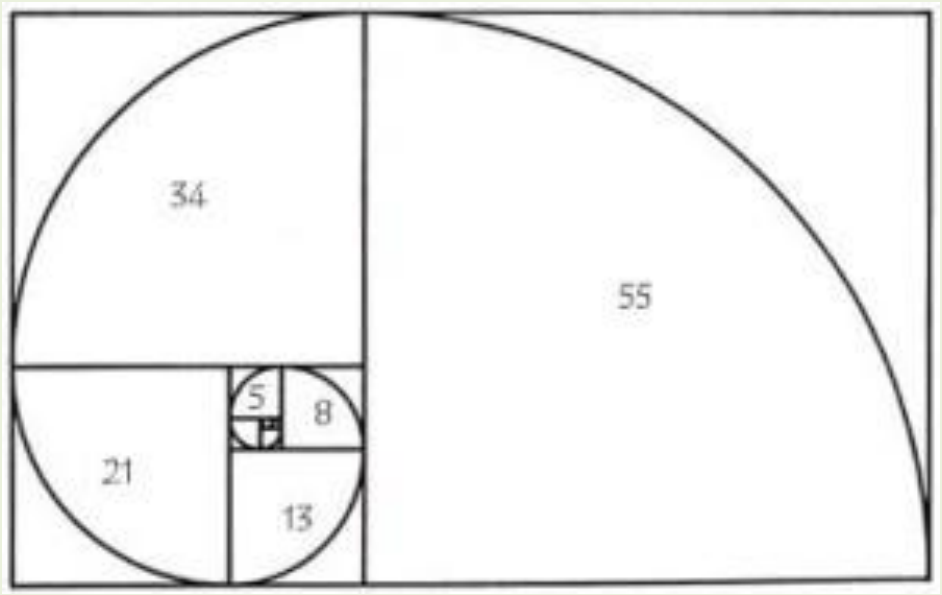
TÉCNICAS DE DISEÑO DE ALGOTIRMOS



```
Inicio
   num=1
   Repita
      Escribir num
      num=num+1
   Hasta  num>10
Fin algoritmo
```



Ecuaciones de recurrencia



Ecuaciones de recurrencia

Una ecuación de recurrencia define una función sobre los números naturales, digamos $T(n)$, en términos de su propio valor con uno o más enteros menores que n.

En otras palabras, $T(n)$ se define inductivamente, hay casos base que se definen aparte, y la ecuación de recurrencia sólo es válida para n mayor que los casos base.

Muchas funciones matemáticas enteroeresantes se pueden definir con ecuaciones de recurrencia, como los conocidos números de Fibonacci.

Ecuaciones de recurrencia

Las ecuaciones de recurrencia surgen de forma muy natural cuando se desea expresar los recursos empleados por procedimientos recursivos.

- Hoy en día, Fibonacci es más conocido por la famosa secuencia de números:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34…

- Formalmente hablando, los números de Fibonacci Fn son generados por una regla simple:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{Si } n > 1 \\ 1 & \text{Si } n = 1 \\ 0 & \text{Si } n = 0 \end{cases}$$

- Secuencia de números aplicada a campos del conocimiento: biología, demografía, arte, arquitectura, música, para nombrar algunos de los campos. Y junto con la potencia de dos, es en ciencias de la computación una de las secuencias favoritas.

Una idea es utilizar la definición recursiva de Fn . El pseudocódigo se muestra enseguida:

```
función fib(n){
      si n=0 retorna 0
      si n=1 retorna 1
      retorna fib(n-1) + fib(n-2)
}
```

Para números mayores existen invocaciones recursivas de fib( ), así tenemos:

T(0)=1, la revisión del primer condicional

T(1)=2, la revisión de dos condicionales

T(2)= La revisión del primer y segundo condicional y la invocación a función.

T(2)= 3 + T(1) + T(0)

T(2)=T(1)+T(0)+3=1+2+3=5

T(3)= La revisión de los dos condicionales mas la invocación recursiva de T(2)+T(1)

T(3)=T(2) + T(1)+3=5+2+3=10

T(4)=T(3)+T(2)+3=10+5+3=18

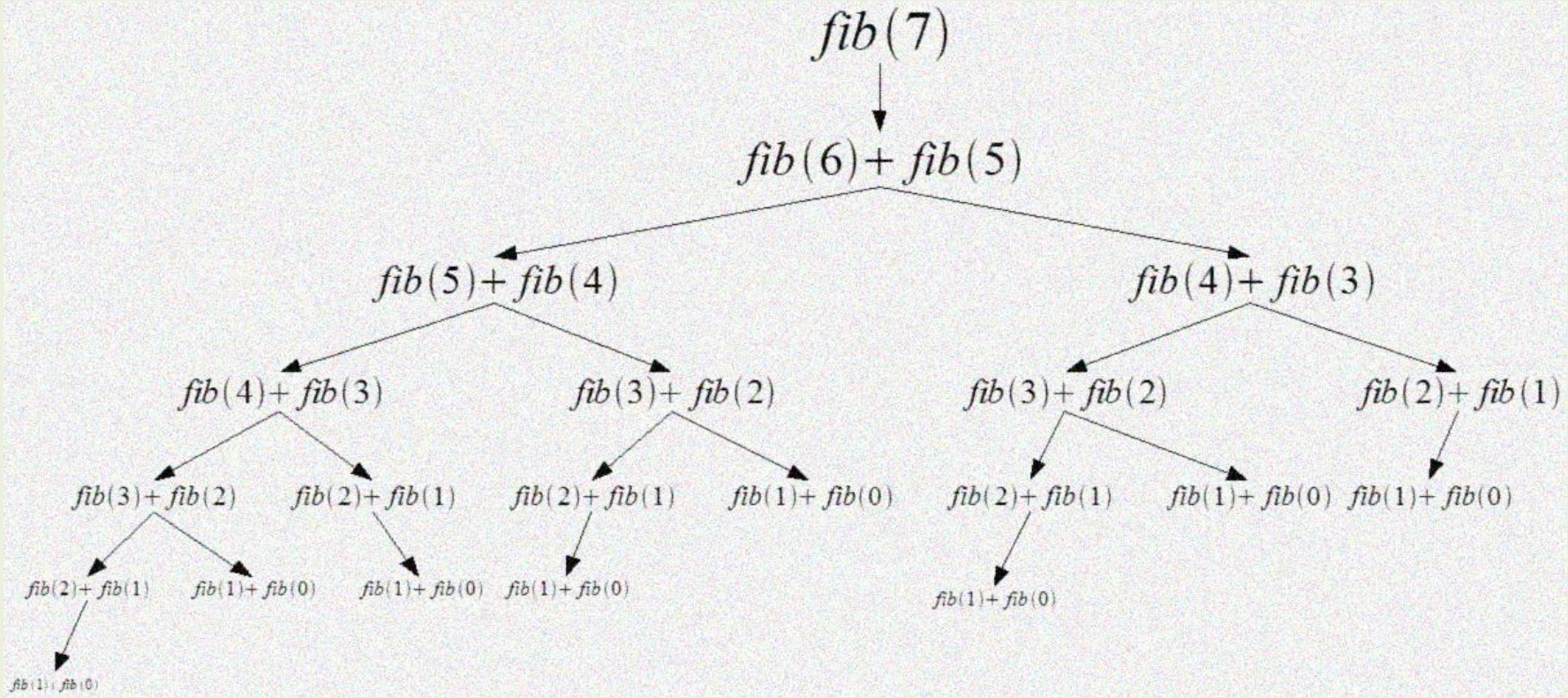
…

T(n) = T(n-1) + T(n-2) + 3 para n > 1.

- El tiempo de ejecución del algoritmo crece tan rápido como los números de Fibonacci: T(n) es exponencial en ***n***, el cual implica que el algoritmo es impráctico exceptuando para valores muy pequeños de ***n***.

- El algoritmo se vuelve ineficiente por la razón de que una llamada a *fib(n)* se tiene una cascada de llamadas recursivas en las cuales la mayoría de ellas son repetitivas.

Árbol recursivo de la función Fibonacci.



Se muestra en lenguaje C un algoritmo mucho más sencillo no recursivo:

```
#include <stdio.h>
main(){
        entero i, n=5, fibn_2,fibn_1, fibn;
        fibn_2=fibn_1=1;
        for ( i =2;i<=n; i++){
                fibn=fibn_2+fibn_1;
                prenterof ("fibo (%d) =%d\n", i , fibn);
                fibn_2=fibn_1;
                fibn_1=fibn;
        }
        getchar ();
}
```

Algoritmos voraces.

El esquema voraz se aplica normalmente a problemas de *decisión* y *optimización.*
Procede por pasos:

- En cada paso se toma una decisión de la que estamos seguros.
- Las decisiones tomadas nunca se reconsideran el algoritmo termina cuando no quedan decisiones por tomar.
- El algoritmo es correcto si podemos garantizar que la solución encontrada es siempre óptima.

MÉTODO CODICIOSO. (Greedy)

- La mayoría de estos problemas tiene ***n*** entradas y requiere tener un subconjunto que satisfaga ciertas restricciones. Cualquier subconjunto que satisfaga estas restricciones se conoce como solución factible.
- El problema requiere encontrar una solución factible que maximice o minimice una función objetivo dada. El método greedy sugiere que uno puede hacer un algoritmo que trabaje por pasos, considerando una entrada a la vez.
- En cada paso se realiza una decisión. Si la inclusión de la siguiente entrada da una solución no factible, la entrada no se adiciona a la solución parcial.

MÉTODO CODICIOSO. (Greedy), algoritmo.

Procedure Greedy.

```
//A(1:n) contiene n entradas.
solución ← φ
for i ← to n do
x ← select (A)
if feasible(solución, x) then
Solución ← UNION(Solución, x)
endif
repeat
return (solución)
end Greedy.
```

El Problema de la Mochila. (Knapsack Problem)

Se tienen ***n*** objetos y una mochila. El objeto ***i*** tiene un peso ***Wi*** y la mochila tiene una capacidad ***M***. Si una fracción ***Xi***, $0 \leq Xi \leq 1$ del objeto ***i*** se enteroroduce, se tendrá una ganancia *Pi Xi*. El objetivo es llenar la mochila de tal forma que maximice la ganancia. El problema es:

$$\text{Max} \sum_{1 \leq i \leq n} P_i X_i \qquad (1)$$

S. A.

$$\sum_{1 \leq i \leq n} W_i X_i \leq M \qquad (2)$$

$$0 \leq X_i \leq 1 \quad 1 \leq i \leq n \qquad (3)$$

Una posible solución es cualquier conjunto (X1, X2 , . . .,Xn) que satisfaga (2) y (3). Una solución óptima es una solución factible en el cual maximice la ganancia (1).

Ejemplo: n=3, M=20, (P1, P2, P3) = (25, 24, 15)

(W1, W2, W3)= (18, 15, 10)

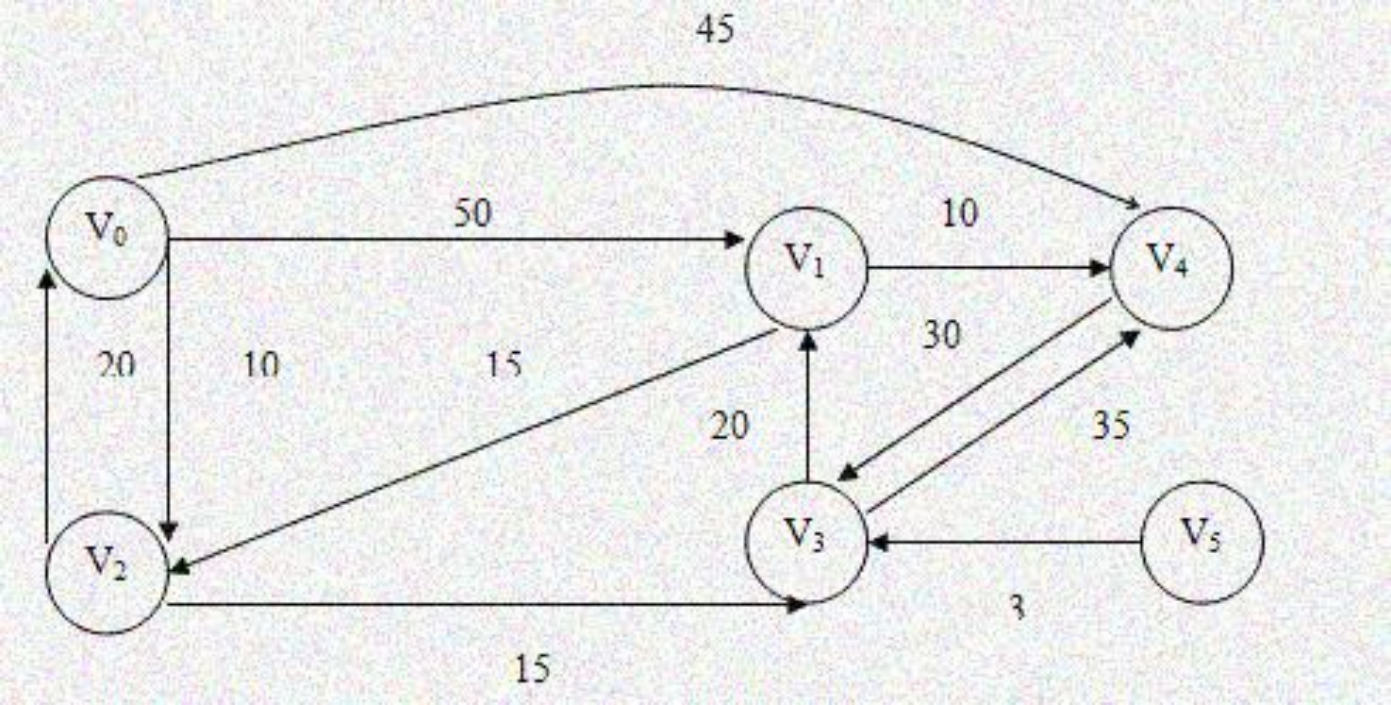
El siguiente algoritmo localiza el óptimo siempre y cuando P(i+1)/W(i+1) ≤ P(i)/W(i):

```
#include <stdio.h>
#define N 3
main(){
float Cu,M=20,
P[]={24,15,25},W[]={15,10,18},X
[]={0,0,0},ganancia=0;
float R[N];
entero i;
Cu=M;
for (i=0;i<N;i++){
if(W[i]>Cu)
break;
X[i]=1;
Cu=Cu-W[i];
}
if (i<N)
X[i]=Cu/W[i];
for (i=0;i<N;i++){
prenterof("X[%d]=%f..",i,X
[i]);
ganancia=ganancia+X[i]*P[i
];
}
prenterof("\nGanancia=%f\n
",ganancia);
getchar();
}
```

Single Source Shortest Paths (La ruta más corta a partir de un origen)

- Los grafos pueden ser utilizados para representar carreteras, estructuras de un estado o país con vértices representando ciudades y segmentos que unen los vértices como la carretera. Los segmentos pueden tener asignados pesos que marcan una distancia entre dos ciudades conectadas.
- La distancia de un trayecto es definido por la suma del peso de los segmentos. El vértice de inicio se definirá como el origen y el último vértice se definirá como el destino. El problema a considerar será en base a una gráfica dirigida G= (V, E), una función de peso c(e) para los segmentos de G y un vértice origen v0.
- El problema es determinar el trayecto más corto de v0 a todos los demás vértices de G. Se asume que todos los pesos son positivos.

Single Source Shortest Paths (La ruta más corta a partir de un origen).



Distancia mas corta entre V_0 a V_i = V_0 + V_2 + V_3 + V_1 = 45

Single Source Shortest Paths (La ruta más corta a partir de un origen)

Se observa que:

1. Si el siguiente trayecto más corto es al vértice u, entonces el trayecto inicia en V0, termina en u y va a través de los vértices localizados en S.
2. El destino del siguiente trayecto generado debe de ser aquel vértice u tal que la mínima distancia, DIST(u), sobre todos los vértices no en S.
3. Habiendo seleccionado un vértice u en 2 y generado el trayecto más corto de V_0 a u, el vértice u viene a ser miembro de S. En este punto la dimensión del trayecto más corto iniciando en V_0. irá en los vértices localizados en S y terminando en w no en S puede decrecer. Esto es, el valor de la distancia DIST(w) puede cambiar. Si cambia, entonces se debe a que existe un trayecto más corto iniciando en V_0 posteriormente va a u y entonces a w. Los vértices enteroermedios de V_0, a u y de w a w deben estar todos en S. Además, el trayecto V_0 a u debe ser el más corto, de otra forma DIST(w) no está definido en forma apropiada. También, el trayecto u a w puede no contener vértices enteroermedios.

En pseudocódigo el algoritmo es el siguiente:

```
Procedure SHORTEST-PATHS(v, COST, DIST, n)
//DIST(j) es el conjunto de longitudes del trayecto más corto del vértice
v al vértice j en la gráfica G con n vértices.
//G es representada por la matriz de costos COST(n, m)
Bolean S(1:n); real COST(1:n, 1:n), DIST(1:n)
enteroeger u, v, n, num, i, w
For i ← 1 to n do
S(i) ← 0; DIST(i) ← COST(v, i)
Repeat
S(v) ← 1 DIST(v) ← 0 // colocar el vértice v en S.
For num← 2 to n-1 do //determina n – 1 trayectos desde el vértice v.
Escoger u tal que DIST(u) = min{DIST(w)}
S(w) =0
S(u) ← 1 //Coloca el vértice u en S
For all w con S(w) =0 do
DIST(w) ← min(DIST(w), DIST(u) + COST(u, w))
Repeat
Repeat
End SHORTEST-PATH.
```

Divide y vencerás.

(Divide and conquer)

- Divide y Vencerás es una técnica de diseño de algoritmos que consiste en resolver un problema a partir de la solución de sub-problemas del mismo tipo, pero de menor tamaño.

- Si los sub-problemas son todavía relativamente grandes se aplicará de nuevo esta técnica hasta alcanzar sub-problemas lo suficientemente pequeños para ser solucionados directamente.

- Ello naturalmente sugiere el uso de la recursión en las implementaciones de estos algoritmos.

- Dada una función para computar ***n*** entradas, la estrategia divide y conquistaras sugiere dividir la entrada en ***k*** subconjuntos, 1< ***k*** ≤ ***n*** produciendo ***k*** sub-problemas. Estos sub-problemas deben ser resueltos y entonces un método debe ser encontrado para combinar las sub-soluciones dentro de una solución como un todo.

- Si el sub-problema es aún demasiado grande, entonces puede ser reaplicada la estrategia. Frecuentemente los sub-problemas resultantes de un diseño divide y conquistaras son del mismo tipo del problema original.

En general, un algoritmo divide-y-vencerás consta de tres pasos.

Paso 1. Si el tamaño del problema es pequeño, éste se resuelve aplicando algún método directo; en caso contrario, el problema original se separa en dos sub-problemas, de preferencia del mismo tamaño.

Paso 2. Los dos sub-problemas se resuelven de manera recurrente aplicando el algoritmo divide-y-vencerás a los sub-problemas.

Paso 3. Las soluciones de los sub-problemas se fusionan en una solución del problema original.

Divide → Conquista → Combina.

Una forma general de ver el modelo es:

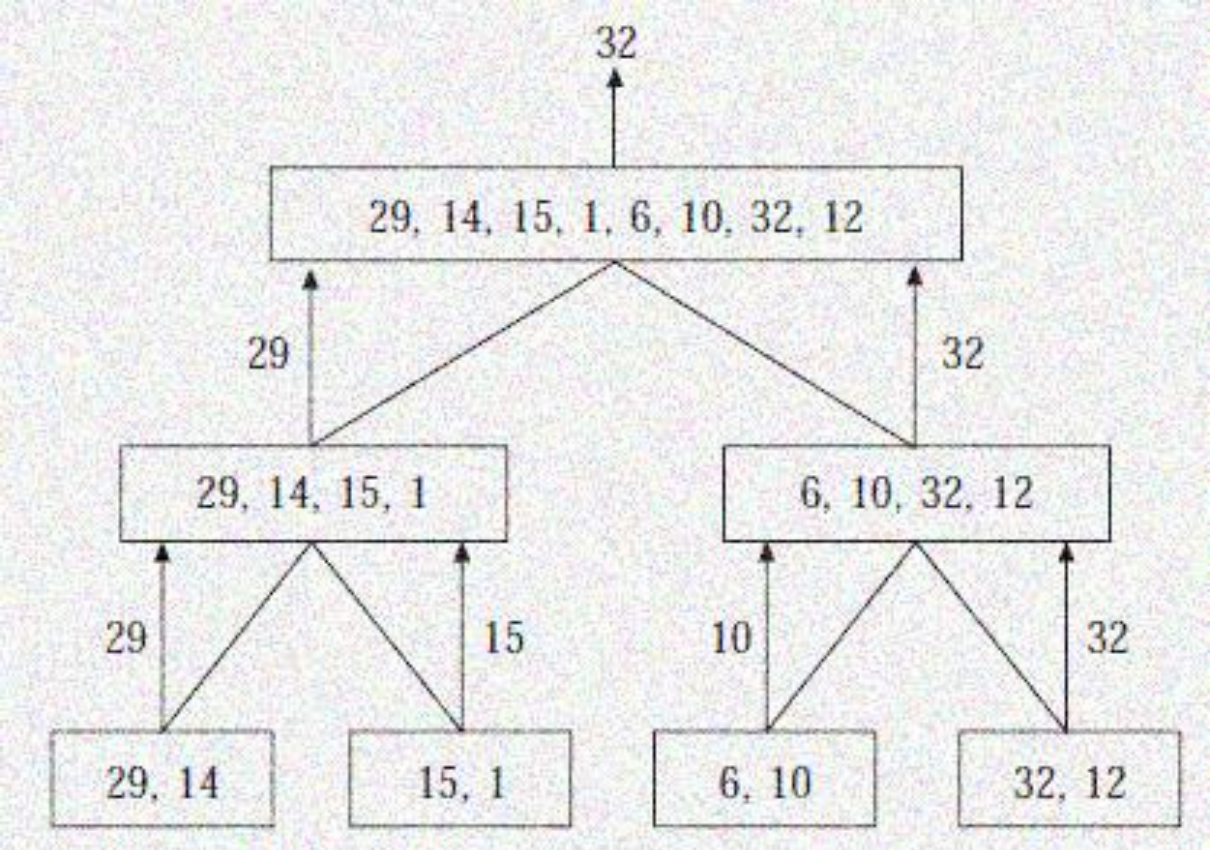
```
void DANDC(entero a[],p,q) {
//a[] es el arreglo a trabajar
//p y q son los subespacios a dividir
        entero m,p,q;
        if small(p,q)
                return(G((p,q)) //Resolución de manera directa
        else{
                m=divide(p,q);
        return(conbin acion(DANDC(a,p,m), DANDC(a,m+1,q))
        }
}
```

En este caso, la función ***small*** determina si se cumple una condición especifica para que el algoritmo se detenga, si no es así se crean dos sub-espacios en los cuales se puede subdivide el espacio en dos más pequeños.

- El principio divide y conquistaras se expresa en forma natural por un procedimiento recursivo. Por lo que se obtienen sub-problemas de menor dimensión de la misma clase, eventualmente se producen sub-problemas que son lo suficientemente pequeños que son resueltos sin la necesidad de dividirlos.

- Tres algoritmos vistos con anterioridad pertenecen a este paradigma:
    - Búsqueda binaria (Binary Search) .
    - Mezcla (Merge Sort) .
    - Ordenación rápida (Quick Sort) .

Ejemplo: Imagine que se tienen ocho números: 29, 14, 15, 1, 6, 10, 32 y 12. La estrategia divide-y-vencerás encuentra el máximo de estos ocho números, lo cual se muestra:



Otro ejemplo es el siguiente algoritmo: El problema es encontrar el máximo y el mínimo de un conjunto de ***n*** elementos en desorden. Un algoritmo directo sería:

```
#include <stdio.h>
main(){
        entero max,min,i,j,a[]={4,2,1,5,7,9,8,6,3,10};
        max=min=a[0];
        for(i=1;i<10;i++){
                if (a[i]>max)
                max=a[i];
                if (a[i]<min)
                        min=a[i];
                }
        prenterof("El maximo valor es %d y el minimo valor es
%d\n",max,min);
        getchar();
}
```

Al término de la recursión se obtienen el mínimo y máximo del conjunto dado. Se muestra el algoritmo en C:

```
#include <stdio.h>
void MaxMin(entero [],entero, entero, entero *, entero *);
entero max(entero, entero);
entero min(entero, entero);

main(){
        entero fmax,fmin,a[]={4,2,10,5,-7,9,80,6,3,1};
        MaxMin(a,0,9,&fmax,&fmin);
        prenterof("El maximo valor es %d y el minimo valor es
%d\n",fmax,fmin);
        getchar();
}
```

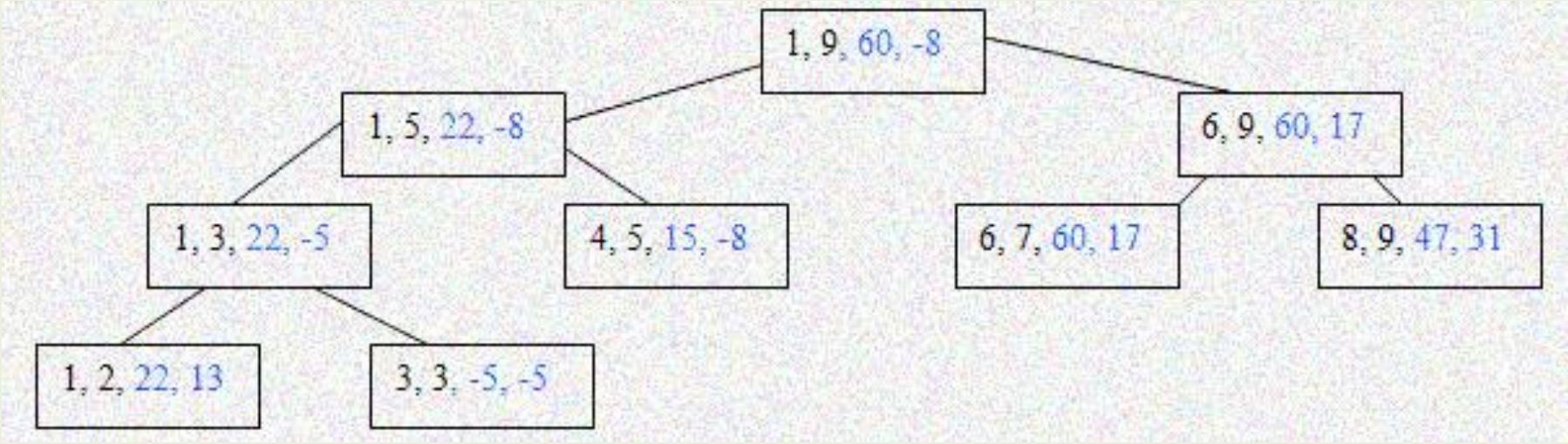
Mínimo y máximo de un conjunto dado recursivo.

```
void MaxMin(entero a[],entero i, entero j, entero * fmax, entero *fmin){
        entero gmax,gmin,hmax,hmin,mid;
        if (i==j)
                *fmax=*fmin=a[i];
        else if (i==j-1)
        if (a[i]<a[j]){
                *fmax=a[j];
                *fmin=a[i];
        }
        else{
                *fmax=a[i];
                *fmin=a[j];
        }
        else{
                mid=(i+j)/2;
                MaxMin(a,i,mid,&gmax,&gmin);
                MaxMin(a,mid+1,j,&hmax,&hmin);
                *fmax=max(gmax,hmax);
                *fmin=min(gmin,hmin);
        }
}
```

Mínimo y máximo de un conjunto dado recursivo.

```
entero max(entero g, entero h){
        if (g>h)
                return g;
        return h;
}

entero min(entero g,entero h){
        if (g<h)
                return g;
return h;
}
```

Árbol recursivo del algoritmo Max y Min.



- Programas a realizar bajo la técnica de diseño divide y vencerás utilizando memoria dinámica.

  - ***Búsqueda binaria (Binary Search) .***
  - ***Mezcla (Merge Sort).***
  - ***Ordenación rápida (Quick Sort).***
  - Multiplicación de enteros grandes(algoritmo Karatsuba).
  - Multiplicación de matrices cuadradas(algoritmo Strassen) .
  - Moda de un vector.
  - Exponenciación.

CÁLCULO DE DETERMINANTES EN FORMA RECURSIVA

El cálculo del determinante por menores de una matriz se puede definir en forma recursiva como:

$$DET(N, A_N) \begin{cases} \sum_{j=0}^{N-1} DET(N\text{-}1, A_{N-1}) * a[0][j] * (-1)^j & \text{Si } N > 2 \\ a[0][0] * a[1][1] - a[1][0] * a[0][1] & \text{Si } N = 2 \\ a[0][0] & \text{Si } N = 1 \end{cases}$$

ALGORITMO CÁLCULO DE DETERMINANTES EN FORMA RECURSIVA

```
//Función que regresa entero, recibe como parámetros un apuntador
doble(matriz) y entero(tamaño), realiza llamada a la función signo()
y subm()
entero determinante (entero **a, entero tam){
       si tam es igual 1
              regresa a[0][0]
       si tam es igual 2
              regresa a[0][0]*a[1][1]-a[1][0]*a[0][1]
       hacer entero m = 0
       hacer entero tam1=tam-1
       repetir con I desde 0 hasta I < tam
     hacer m = signo(i) * determinante(subm(i,a,tam1),tam1)*a[0][i]+m
regresa m
}
```

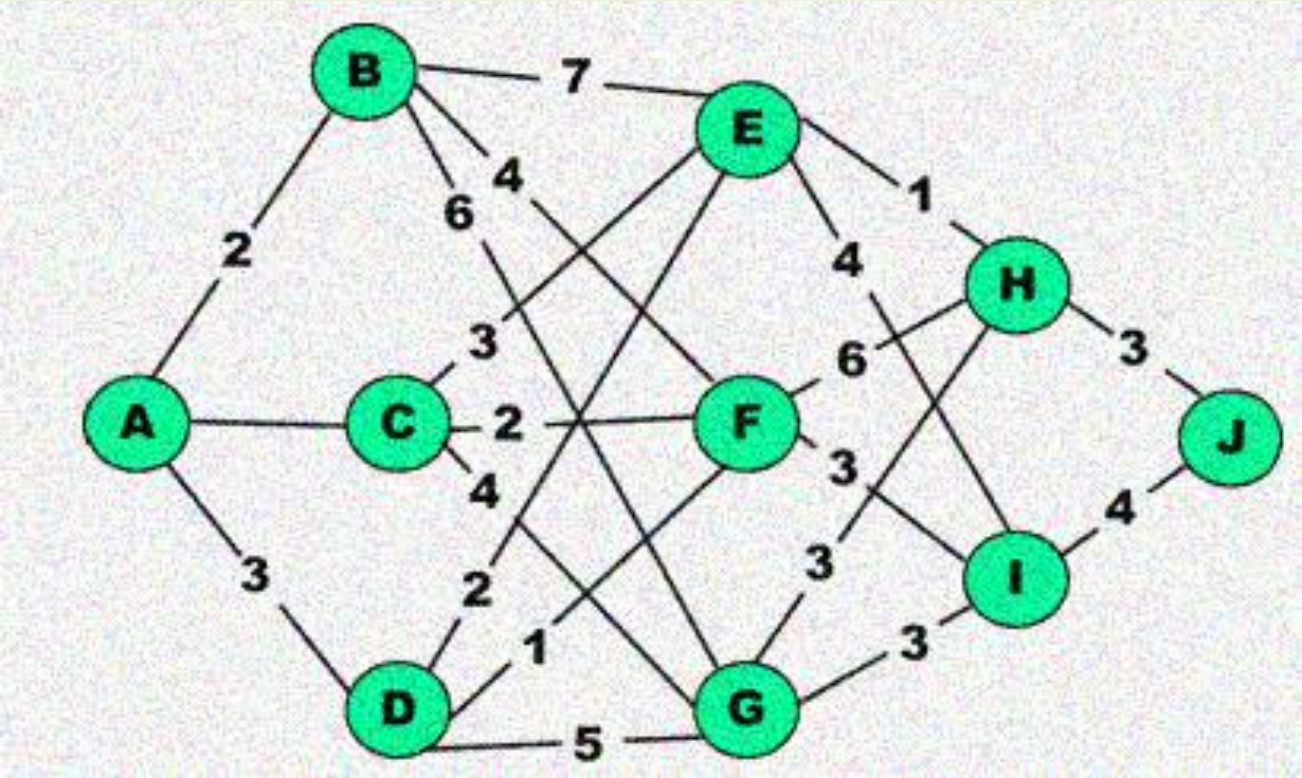
ALGORITMO CÁLCULO DE DETERMINANTES EN FORMA RECURSIVA

```
//Función que regresa un doble apuntador a entero, recibe como
parámetros un entero, doble apuntador a entero y entero.

entero** subm(entero k, entero **a, entero tam){
entero **b;
entero i,j
hacer b = malloc (tam)
repetir con j desde 0 hasta j < tam
       hacer b[j] = malloc (tam)
       repetir con I desde 0 hasta I < tam
              hacer m = 0
              repetir con j desde 0 hasta < tam+1
                     si j es diferente de k
                            hacer b[i][m] = a[i+1][j]
                            incrementar a m en 1
                     fin del si.
       fin del ciclo for
regresar b
}
```

ALGORITMO CÁLCULO DE DETERMINANTES EN FORMA RECURSIVA

```
//Función que regresa un entero, recibe como parámetro un entero.

entero signo(entero i){
        si i modulo 2 es igual 0
                regresa 1
        regresa -1
}
```

Programación dinámica.



Principio de optimización.

- El proceso de optimización puede ser visto como una secuencia de decisiones que nos proporcionan la solución correcta. Si, dada una subsecuencia de decisiones, siempre se conoce cuál es la decisión que debe tomarse a continuación para obtener la secuencia óptima, el problema es elemental y se resuelve trivialmente tomando una decisión detrás de otra, lo que se conoce como estrategia voraz.

- Contemplar un problema como una secuencia de decisiones equivale a dividirlo en problemas más pequeños y por lo tanto más fáciles de resolver como hacemos en Divide y Vencerás, técnica similar a la de programación dinámica.

Principio de optimización.

La programación dinámica se aplica cuando la subdivisión de un problema conduce a:

- Una enorme cantidad de problemas.
- Problemas cuyas soluciones parciales se solapan.
- Grupos de problemas de muy distenteroa complejidad.

Para que un problema pueda ser abordado por esta técnica ha de cumplir dos condiciones:

- La solución al problema ha de ser alcanzada a través de una secuencia de decisiones, una en cada etapa.
- Dicha secuencia de decisiones ha de cumplir el principio de óptimalidad.

Principio de optimización.

En general, el diseño de un algoritmo de Programación Dinámica consta de los siguientes pasos:

1. Planteamiento de la solución como una sucesión de decisiones y verificación de que ésta cumple el principio de optimalidad.
2. Definición recursiva de la solución.
3. Cálculo del valor de la solución óptima mediante una tabla en donde se almacenan soluciones a problemas parciales para reutilizar los cálculos
4. Construcción de la solución óptima haciendo uso de la información contenida en la anterior diapositiva.

El Problema del Agente Viajero (The Traveling Sales Person Problem, TSP)

Éste es un problema de permutaciones. Un problema de permutaciones usualmente será mucho más difícil de resolver que problemas en el que se escogen subconjuntos ya que existen ***n!*** permutaciones de ***n*** objetos mientras que sólo existen 2n diferentes subconjuntos de n objetos ($n! > 0(2^n)$).

n	2^n	n!
1	2	1
2	4	2
4	16	24
8	256	40320

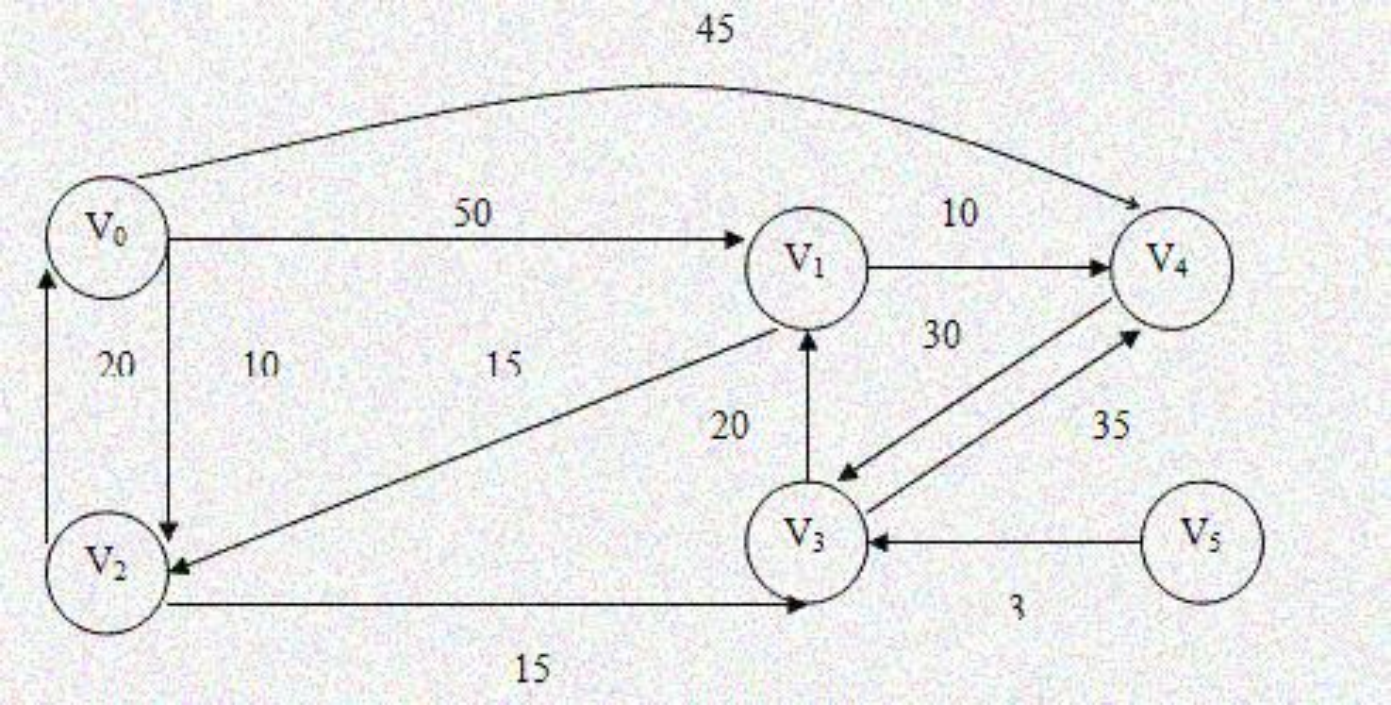
El Problema del Agente Viajero, problema.

- El problema del agente viajero es encontrar un camino que minimice los costos.
- Suponga que se tiene una ruta de una camioneta postal que recoge correo de cajas de correo localizados en ***n*** diferentes sitios. Una gráfica de ***n+1*** vértices puede ser usada para representar tal situación. Un vértice representa la oficina postal desde donde la camioneta postal inicia su recorrido y en el cual debe de retornar. La arista <i, j> tiene asignado un costo igual a la distancia desde el sitio i al sitio j. La ruta tomada por la camioneta postal es una gira y lo que se espera es minimizar el trayecto de la camioneta. El recorrido culmina regresando al vértice inicial, siendo el recorrido el mínimo costo.

Single Source Shortest Paths (La ruta más corta a partir de un origen)

- Los grafos pueden ser utilizados para representar carreteras, estructuras de un estado o país con vértices representando ciudades y segmentos que unen los vértices como la carretera. Los segmentos pueden tener asignados pesos que marcan una distancia entre dos ciudades conectadas.
- La distancia de un trayecto es definido por la suma del peso de los segmentos. El vértice de inicio se definirá como el origen y el último vértice se definirá como el destino. El problema a considerar será en base a una gráfica dirigida G= (V, E), una función de peso c(e) para los segmentos de G y un vértice origen v0.
- El problema es determinar el trayecto más corto de v0 a todos los demás vértices de G. Se asume que todos los pesos son positivos.

Single Source Shortest Paths (La ruta más corta a partir de un origen).



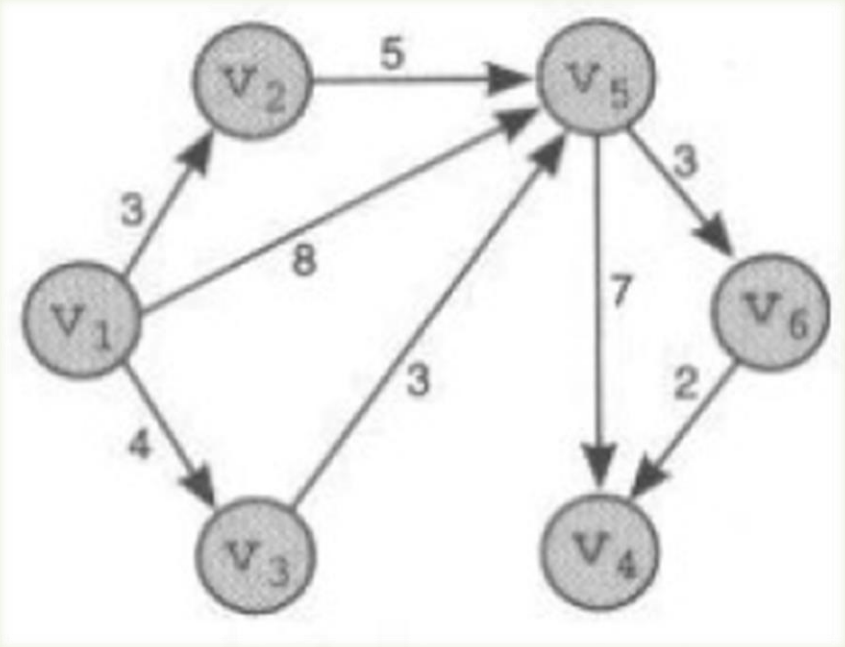
Distancia mas corta entre V_0 a V_i = V_0 + V_2 + V_3 + V_1 = 45

Single Source Shortest Paths (La ruta más corta a partir de un origen)

Se observa que:

1. Si el siguiente trayecto más corto es al vértice u, entonces el trayecto inicia en V0, termina en u y va a través de los vértices localizados en S.
2. El destino del siguiente trayecto generado debe de ser aquel vértice u tal que la mínima distancia, DIST(u), sobre todos los vértices no en S.
3. Habiendo seleccionado un vértice u en 2 y generado el trayecto más corto de V_0 a u, el vértice u viene a ser miembro de S. En este punto la dimensión del trayecto más corto iniciando en V_0. irá en los vértices localizados en S y terminando en w no en S puede decrecer. Esto es, el valor de la distancia DIST(w) puede cambiar. Si cambia, entonces se debe a que existe un trayecto más corto iniciando en V_0 posteriormente va a u y entonces a w. Los vértices enteroermedios de V_0, a u y de w a w deben estar todos en S. Además, el trayecto V_0 a u debe ser el más corto, de otra forma DIST(w) no está definido en forma apropiada. También, el trayecto u a w puede no contener vértices enteroermedios.

Ejemplo: en el grafo dirigido y con pesos no negativos, calcular el coste mínimo de los caminos desde el vértice 1 a los otros vértices.

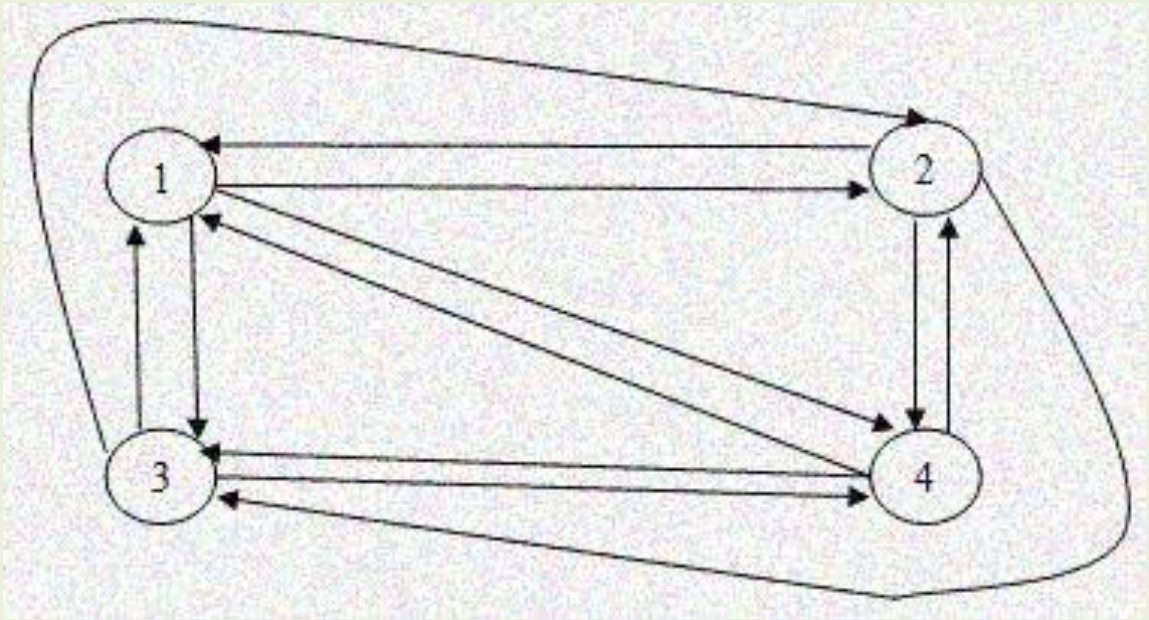


La matriz de adyacencia con los pesos de arcos:

	1	2	3	4	5	6
1	-	3	4	-	8	-
2	-	-	-	-	5	-
3	-	-	-	-	3	-
4	-	-	-	-	-	-
5	-	-	-	7	-	3
6	-	-	-	2	-	-

Ejemplo: Considere la siguiente gráfica donde el tamaño de las aristas se dan en la matriz.

La matriz de adyacencia con los pesos de arcos:



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Desde el principio de optimalidad se deduce que:

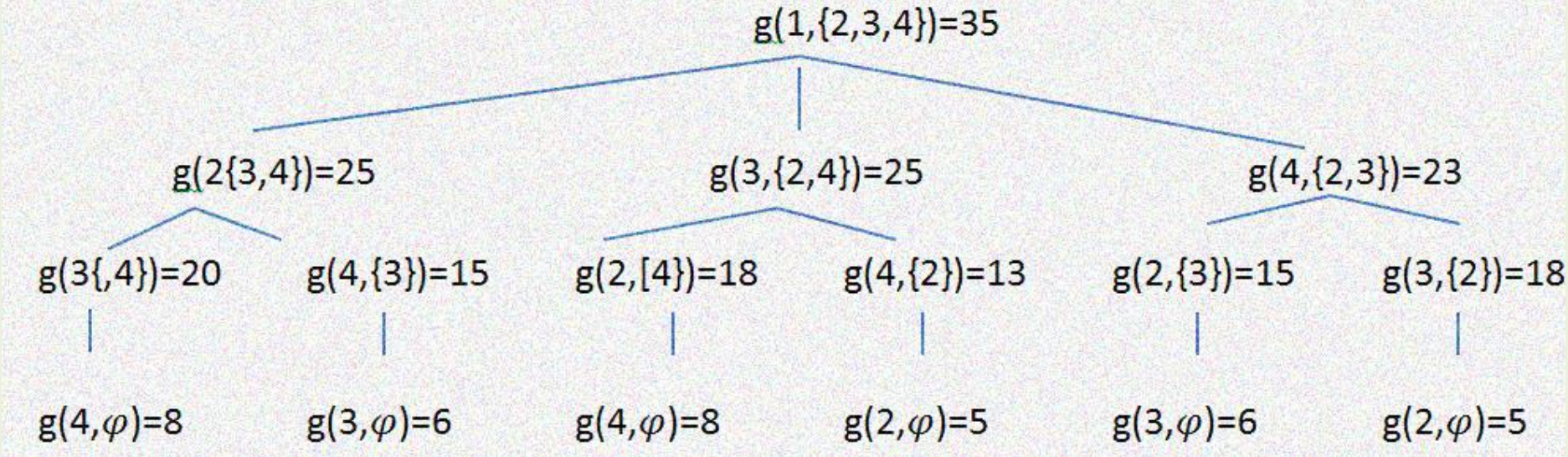
$$g(1, V-\{1\}) = \min_{2<=k<=n}\{C_{ik} + g(k, V-\{1,k\})\} \ldots (1)$$

Generalizando, se obtiene:

$$g(i, S) = \min_{j \epsilon S}\{C_{i\,k} + g(j, S - \{j\})\} \ldots (2)$$

Determinar el costo mínimo desde el punto inicial.

Árbol recursivo del agente viajero:



ALGORITMO PARA EL CALCULO TSP.

```
//Función principal de TSP, realiza llamadas a las funciones generar() y tsp().

funcion principal ()
Hacer enteros **cost, *m, r,v

//Solicitar el numero de vértices (matriz cuadrada).
**cost = malloc (v) // Crear matriz de filas y columnas de tamaño v.
* m = malloc (v)
Repetir con i desde 0 hasta v-1
          m[i]←0
Repetir con i desde 0 hasta v-1
        1. Repetir con j desde 0 hasta v-1
                si i ← j entonces
                        cost[i][j] ← 0
                si no
                        cost[i][j] ← 9999
        //Fin de paso 1
generar (cost,v)
//Solicitar vértice origen y asignar a r
```

ALGORITMO PARA EL CALCULO TSP.

```
//Continuación de función principal de TSP.

//Mostrando la matriz de adyacencia
2. Repetir con i desde 0 hasta v-1
        3. Repetir con j desde 0 hasta v-1
                Mostrar matriz cost[i][j]
        //Fin de paso 3
 Mostrar en pantalla
"Costo mínimo a partir de 'origen':", r,tsp(cost,m,v,r-1,v,r-1)
//Fin del paso 2.
```

ALGORITMO PARA EL CALCULO TSP.

```
//Función que recibe como parámetros un apuntador doble(matriz) y
entero(tamaño).
void generar (entero **cost, entero tam)
Hacer enteros i, nv ← 0, p, ad
//Con el numero 99, se termina la enteroroducción del vértice
1 Mientras nv < tam
        //Solicitar el numero de vértice
        hacer ad ←  numero de vértice
        2 si ad ← 99 entonces
                //Termina vértice nv+1
                incrementar nv en 1
        //Fin de la condición 2
        si no
        si, ad > tam entonces
                //Vértice no existente
        3 si no
                //Solicitar el peso del vértice
                p ← peso de vértice
                cost[nv][ad-1]← p
        //Fin del paso 3
//Fin del paso 1
```

ALGORITMO PARA EL CALCULO TSP.

```
//Función que regresa un entero y recibe como parámetros un apuntador
doble(matriz), apuntador a entero y 4 enteros

entero tsp (entero **cost, entero *t, entero d, o, v, r)
si d ← 1
        regresa cost [o][r]
hacer dist, dmin ← 999 enteros
hacer m[0]←1
repetir con i desde 0 hasta v-1
          1. Si m[i]←0 entonces
              Hacer dist ← cost[o][i]+ tsp(costo,m,d-1,i,v,r)
              Hacer m[i] ←0
              2. Si dist < dmin entonces
                dmin ← dist;

              //Fin de paso 2
          //Fin de paso 1
      Regresa dmin
```

Ejercicio:

1.- Dada la siguiente matriz de costos:

	1	2	3	4	5
1	0	10	20	10	5
2	5	0	10	15	6
3	7	6	0	10	7
4	8	6	5	0	6
5	3	4	2	3	0

Determinar el trayecto del agente viajero iniciando en el punto uno.

El Problema del Agente Viajero, algoritmo.

```
#include<stdio.h>
#include<stdlib.h>
void generar(entero **,entero);
entero tsp(entero **,entero *,entero,entero,entero,entero);
main(){
        entero **cost,*m,d,o,v,i,j,r;
        prenterof(" *************** TSP ***************\n");
        prenterof("\nenteroroduce el numero de vertices: ");
        scanf("%d",&v);
        cost=(entero **)malloc(sizeof(entero *)*v);
        for(i=0;i<v;i++)
                cost[i]=(entero *)malloc(sizeof(entero)*v);
        m=(entero *)malloc(sizeof(entero)*v);
        for(i=0;i<v;i++)
                m[i]=0;
        for(i=0;i<v;i++)
                for(j=0;j<v;j++){
                        if(i==j)
                        cost[i][j]=0;
                        else
                        cost[i][j]=9999;
                }
```

El Problema del Agente Viajero, algoritmo.

```
                generar(cost,v);
                prenterof("\nenteroroduce el origen:");
                scanf("%d",&r);
                prenterof("La matriz generada es: \n");
                for(i=0;i<v;i++){
                        for(j=0;j<v;j++){
                        prenterof("%6d",cost[i][j]); } prenterof("\n"); }
                prenterof("\n\nCosto minimo a partir de %d:
%d\n",r,tsp(cost,m,v,r-1,v,r-1));
        system("PAUSE");
}
```

El Problema del Agente Viajero, algoritmo.

```
void generar(entero **cost, entero
tam){
entero i,nv=0,p,ad;
prenterof("Para terminar de
enteroroducir un vertice enteroroduce
99\n");
while(nv<tam){
prenterof("Vertice %d a... \n",nv+1);
scanf("%d",&ad);
if(ad==99){
prenterof("Termino vertice
%d\n",nv+1);
nv++;}
else if(ad>tam)
prenterof("El vertice no existe \n");
else{
prenterof("enteroorduce el peso del
Vertice:");
scanf("%d",&p);
cost[nv][ad-1]=p;
}
}
}
```

```
entero tsp(entero **cost,entero
*m,entero d,entero o,entero
v,entero r){
if(d==1)
return cost[o][r];
entero dist,dmin=999;
m[o]=1;
for(entero i=0;i<v;i++)
if(m[i]==0){
dist=cost[o][i]+tsp(cost,m,d-
1,i,v,r);
m[i]=0;
if(dist<dmin){
dmin=dist;
}
}
return dmin;
}
```

Algoritmo Dijkstra.

{Se considera al vértice *1* como origen, *N* es el número de vértices, *S* y *D* son arreglos de *N* elementos y *M* es una matriz de *N* x *N* elementos.}

1 Agregar el vértice 1 a S.
2 Repetir con i desde 2 hasta *N*.
    Elegir un vértice *v* en (V-S) tal que D[v] sea el mínimo valor.
    Agregar *v* a S.
    2.1 Repetir para cada vértice *w* en (V-S)
        Hacer D[w] ← minimo (D[w],D[v]+M[v,w])
    2.2 Fin del ciclo paso 2.1
3 Fin del ciclo del paso 3

Fuente: “Estructura de datos”; Cairo-Guardati; 3ra Ed.

Algoritmo Dijkstra.

Datos de entrada.

1. Vértice inicial.
2. Numero de vértices.
3. Pesos entre vértices.

Datos de salida.

1. Matriz de adyacencia.
2. Mostrar el camino mas corto del vértice inicial a final, donde inicial y final son distenteroos vértices.

Técnicas de diseño.

1. Memoria dinámica.
2. Modularidad.

Vuelta a tras (backtracking).

- En la búsqueda de los principios fundamentales del diseño de algoritmos, ***backtracking*** representa una de las técnicas más generales. Varios problemas que tratan de la búsqueda de un conjunto de soluciones o buscan una solución óptima satisfaciendo algunas restricciones pueden ser resueltos usando backtracking.

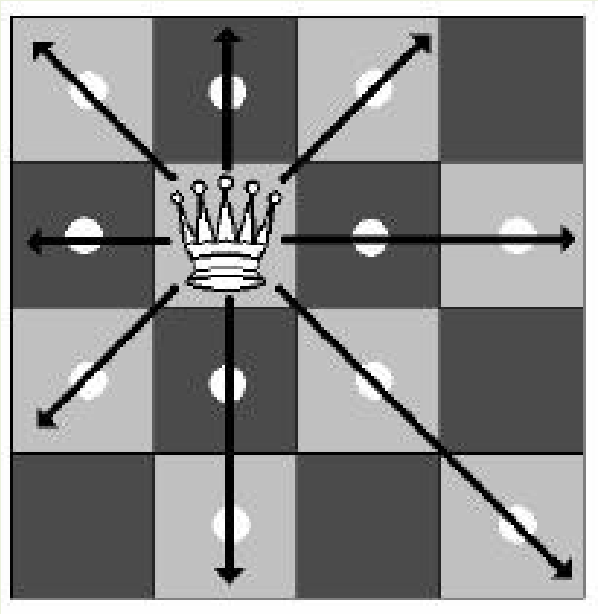
- Para utilizar el método backtracking, el problema debe de estar expresado en n-tuplas. (x_1, x_2, x_3, . . . , x_n) donde x_i se escoge desde un conjunto finito Si. A veces el problema a resolver es maximizar o minimizar una función P(x_1, x_2, x_3, . . . X_n). A veces se busca todos los vectores tales que satisfacen a P.

Vuelta a tras (backtracking).

- La virtud de backtracking es la habilidad para producir la misma respuesta con un menor número de pasos. Su idea básica es construir un vector y evaluarlo con la función $P(x_1, x_2, x_3, \ldots X_n)$ para probar si el vector recién formado tiene una oportunidad de ser el óptimo. La gran ventaja de éste método es: si se observa que el vector parcial (x_1, x_2, x_3, . . . X_n) no tiene la posibilidad de conducir hacia una posible solución óptima, entonces $m_{1+1}, \ldots m_n$ puede ser ignorado completamente.
- Varios de los problemas que se resuelven utilizando backtracking requieren que todas las soluciones satisfagan a un complejo conjunto de restricciones. Estas restricciones pueden ser divididas en dos categorías: explicitas e implícitas. Restricciones explicitas son reglas cuyas restricciones de cada x_i toman valores para un conjunto determinado.

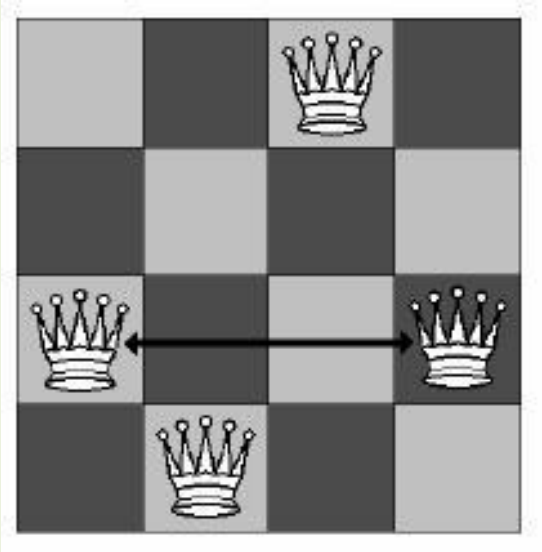
El problema de las ocho reinas (8-Queens).

- Un problema combinatorio clásico es colocar las ocho reinas en un tablero de ajedrez de tal forma que no se puedan atacar entre ellas, esto es que no existan dos reinas en la misma hilera, columna o diagonal. Enumerando las hileras y columnas del tablero del 1 al 8. Las reinas pueden enumerarse también del 1 al 8. Ya que cada reina debe estar sobre una hilera diferente, se puede asumir que la reina *i* se colocará en la hilera *i*.



8-Queens, problemática.

- Como cada reina puede amenazar a todas las reinas que estén en la misma fila, cada una ha de situarse en una fila diferente. Podemos representar las 8 reinas mediante un vector[1-8], teniendo en cuenta que cada índice del vector representa una fila y el valor una columna. Así cada reina estaría en la posición (i, v[i]) para i = 1-8.



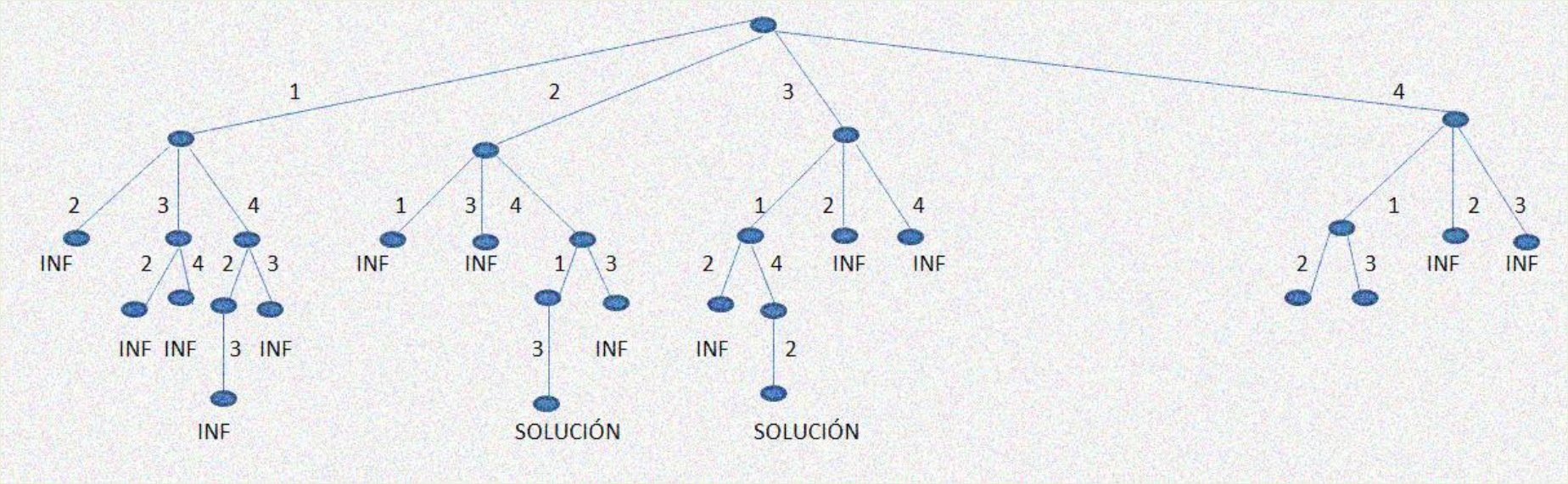
Descripción del Algoritmo:

El algoritmo que arroja la solución de nuestro problema, en el cual $S_{1 \ldots 8}$ es un vector global. Para imprimir todas las soluciones, la llamada inicial es *reinas (0,0,0,0).*

```
Procedimiento reinas(k, col, diag45, diag135)
//sol1 . . . k es el k prometedor
//col = {soli | 1 ≤ i ≤ k}
//diag45 ={soli - i + 1| 1 ≤ i ≤ k}
// diag135 ={soli + i - 1| 1 ≤ i ≤ k}
si k = 8 entonces //un vector 8-prometedor es una solución
        Escribir sol
Si no //explorar las extensiones (k+1) prometedoras de sol
        Para j←1 hasta 8 hacer
        Si j ≠ col y j − k ≠ diag45 y j + k ≠ diag135 entonces
        Solk + 1 ← j//sol1, . . .,k+1 es (k + 1) prometedor
        Reinas(k+1, col U {j}, diag45 U {j − k}, diag135 U {j + k})
```

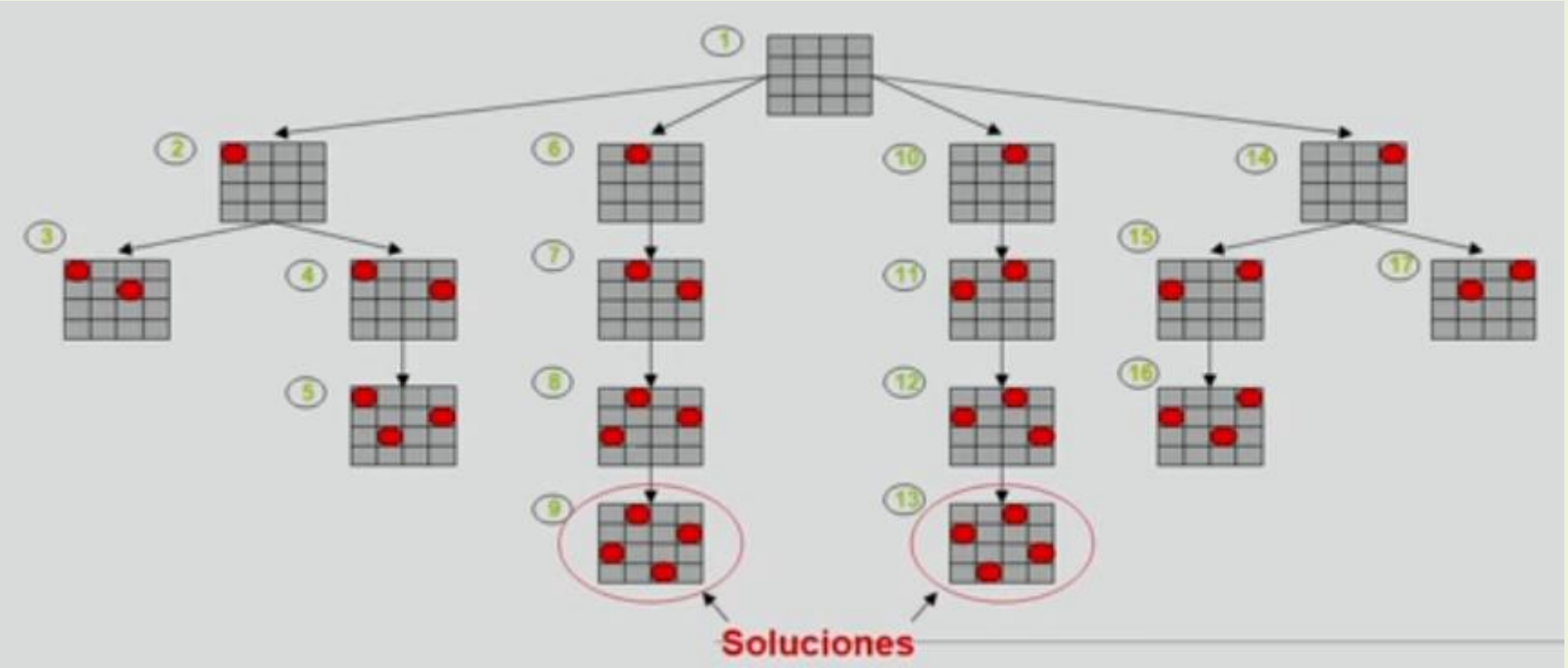
4 Queens.

Un ejemplo del árbol en profundidad con 4 reinas puede verse fácilmente como:



4 Queens.

Un ejemplo del árbol en profundidad con 4 reinas puede verse fácilmente como:



ALGORITMO N REINAS.

```
//Prototipos del algoritmo NReinas.

null marcar(entero **,entero,entero,entero);
null vaciar(entero **,entero);
null solucion(entero **,entero);
null dam(entero **,entero **,entero,entero,entero,entero);
null regresar(entero **,entero **,entero,entero,entero);
entero cont=0;  //Variable global


Funciones y variables globales del algoritmo.
```

ALGORITMO N REINAS.

```
//Función principal de NReinas, realiza llamadas a las funciones vaciar() y
dam().

funcion principal ()
Hacer enteros **matriz, **tablero, reinas, fila←0, columna ←0
reinas ← numero de reinas.
**matriz ← malloc (reinas)
**tablero ← malloc (reinas)
1.Repetir con i desde 0 hasta reinas-1
          matriz[i]← malloc (reinas)
          tablero[i] ← malloc (reinas)
          //Fin de paso 1
vaciar(matriz,reinas)
vaciar(tablero,reinas)
2. Repetir con i desde 0 hasta reinas-1
          dam(matriz,tablero,i,0,1,reinas)
          //Fin de paso 2
Si cont ← 0
        Imprimir: No hay soluciones para el problema entero, reinas
Fin de función principal
```

ALGORITMO N REINAS.

//Función *dam()* que no regresa parámetros, recibe como parámetros entero **matriz, entero **tablero, entero fila, entero columna, entero reinas, entero R; realiza la llamada a las funciones *marcar()*, *solución()* y *regresar()* con la llamada recursiva a *dam()*.

```
dam(){
matriz[fila][columna]←1
tablero[fila][columna]←1
marcar(matriz,R,fila,columna)
si reinas ← R
        solucion(tablero,reinas)
1. si no
        Repetir con j desde 0 hasta R-1
          si matriz [j][columna+1] ←0
                dam(matriz,tablero,j,columna+1,reinas+1,R)
      Fin de paso 1
regresar (matriz,tablero,fila,columna,R)
} //Fin de función dam().
```

ALGORITMO N REINAS.

//Función ***regresar()*** que no regresa parámetros, recibe como parámetros entero **matriz, entero **tablero, entero fila, entero columna, entero R; realiza la llamada a la función *marcar()*.

```
regresar(){
tablero[fila][columna]←0
1. repetir con i desde 0 hasta R-1
      2. repetir con j desde 0 hasta R-1
         matriz[i][j] ← 0
         Fin de paso 1
      Fin de paso 2
3. repetir con i desde 0 hasta R-1
      4. repetir con j desde 0 hasta R-1
               si tablero[i][j] ←1
               marcar(matriz,R,i,j)
      Fin de paso 4
Fin de paso 3

Fin de función.
```

ALGORITMO N REINAS.

```
//Función solucion() que no regresa parámetros, recibe como
parámetros entero **vect, entero reinas.

solucion(){
Mostrar variable ++cont
1. repetir con i desde 0 hasta reinas-1
      2. repetir con j desde 0 hasta reinas-1
         Mostrar vect[i][j]
         Fin de paso 2
         //Salto de linea
      Fin de paso 1
      //Salto de linea
Fin de función.
```

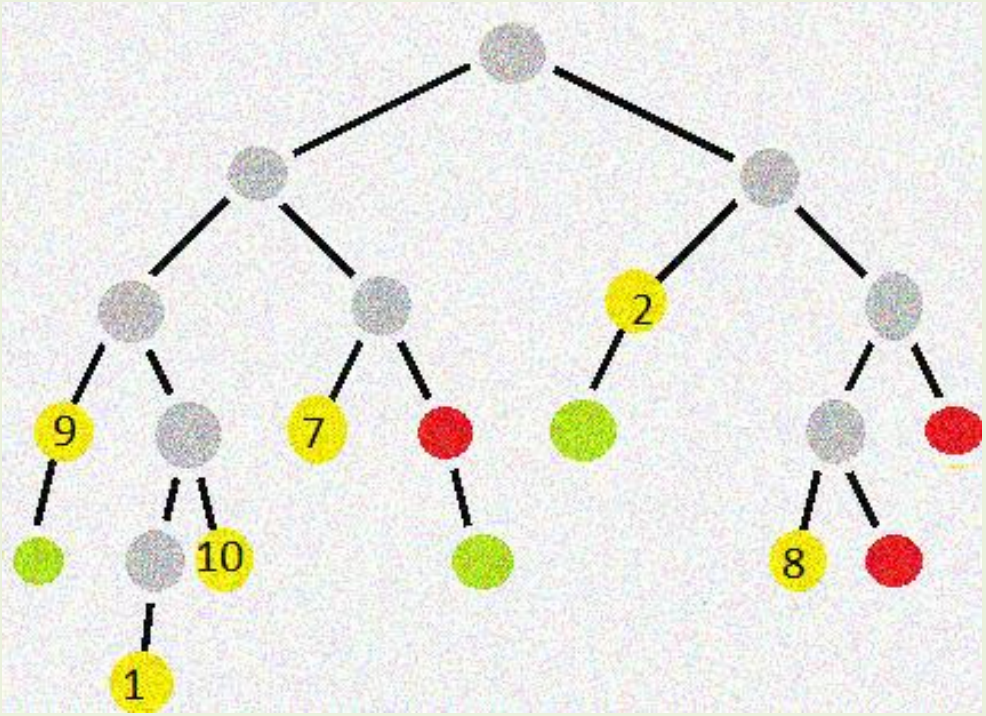
ALGORITMO N REINAS.

```
//Función vaciar() que no regresa parámetros, recibe como parámetros
entero **vect, entero reinas.

vaciar(){
repetir con i desde 0 hasta reinas-1
repetir con j desde 0 hasta reinas-1
        vect[i][j]←0
Fin de función.
```

ALGORITMO N REINAS.

```
//Función marcar() que no regresa parámetros, recibe como parámetros entero
**matriz, entero reinas, entero falfil, entero calfil.

marcar(){
repetir con fila desde 0 hasta reinas-1
repetir con columna desde 0 hasta reinas-1
si fila+columna ← falfil+calfil ó fila-columna←falfil-calfil
        matriz[fila][columna]←1
1. repetir con fila desde 0 hasta reinas-1
      matriz [falfil][fila]←1
      matriz [fila][calfil]←1
      //Fin del paso 1
Fin de función.
```

Ramificación y poda.



- El método de diseño de algoritmos **Ramificación y Acotamiento** (también llamado *Ramificación y Poda*) es una variante del Backtracking mejorado sustancialmente, se aplica mayoritariamente para resolver cuestiones o problemas de optimización.
- La técnica de **Ramificación y Acotamiento** se suele enteroerpretar como un árbol de soluciones, donde cada rama nos lleva a una posible solución posterior a la actual. La característica de esta técnica con respecto a otras anteriores es que el algoritmo se encarga de detectar en qué ramificación las soluciones dadas ya no están siendo óptimas, para «podar» esa rama del árbol y no continuar malgastando recursos y procesos en casos que se alejan de la solución óptima.

Ramificación.

- La idea clave del algoritmo de ramificación y poda es: si la menor rama para algún árbol nodo (conjunto de candidatos) A es mayor que la rama padre para otro nodo B, entonces A debe ser descartada con seguridad de la búsqueda. Este paso es llamado poda, y usualmente es implementado manteniendo una variable global *m* que graba el mínimo nodo padre visto entre todas las subregiones examinadas hasta entonces. Cualquier nodo cuyo nodo hijo es mayor que *m* puede ser descartado.
- La recursión se detiene cuando el conjunto candidato ***S*** es reducido a un solo elemento, o también cuando el nodo padre para el conjunto ***S*** coincide con el nodo hijo. De cualquier forma, cualquier elemento de ***S*** va a ser el mínimo de una función sin ***S***.

Ramificación.

El pseudocódigo del algoritmo de Ramificación y poda es el siguiente:

```
funcion RyP {
P = Hijos(x,k)
mientras ( no vacio(P) )
x(k) = extraer(P)
si esFactible(x,k) y G(x,k) < optimo
si esSolucion(x)
        Almacenar(x)
        sino
        RyP(x,k+1)
}
```

Donde:

G(x) es la función de estimación del algoritmo.

P es la pila de posibles soluciones.

esFactible es la función que considera si la propuesta es válida.

esSolución es la función que comprueba si se satisface el objetivo.

óptimo es el valor de la función a optimizar evaluado sobre la mejor solución encontrada hasta el momento.

NOTA: Usamos un menor que (<) para los problemas de **minimización** y un mayor que (>) para problemas de **maximización**.

Estrategias de Poda.

El objetivo principal será eliminar aquellos nodos que no lleven a soluciones buenas. Podemos utilizar dos estrategias básicas.

Estrategia 1

Si a partir de un nodo x_i se puede obtener una solución válida, entonces se podrá podar dicho nodo si la cota superior CS(x_i) es menor o igual que la cota inferior CI(x_j) para algún nodo j generado en el árbol.

Estrategia 2

Si se obtiene una posible solución válida para el problema con un beneficio B_j, entonces se podrán podar aquellos nodos x_i cuya cota superior CS(x_i) sea menor o igual que el beneficio que se puede obtener B_j (este proceso sería similar para la cota inferior).

Estrategias de Ramificación

La expansión de un árbol con distenteroas estrategias está condicionada por la búsqueda de la solución óptima. Debido a esto, todos los nodos de un nivel deben ser expandidos antes de alcanzar un nuevo nivel, cosa que es lógica ya que para poder elegir la rama del árbol que va a ser explorada, se deben conocer todas las ramas posibles.

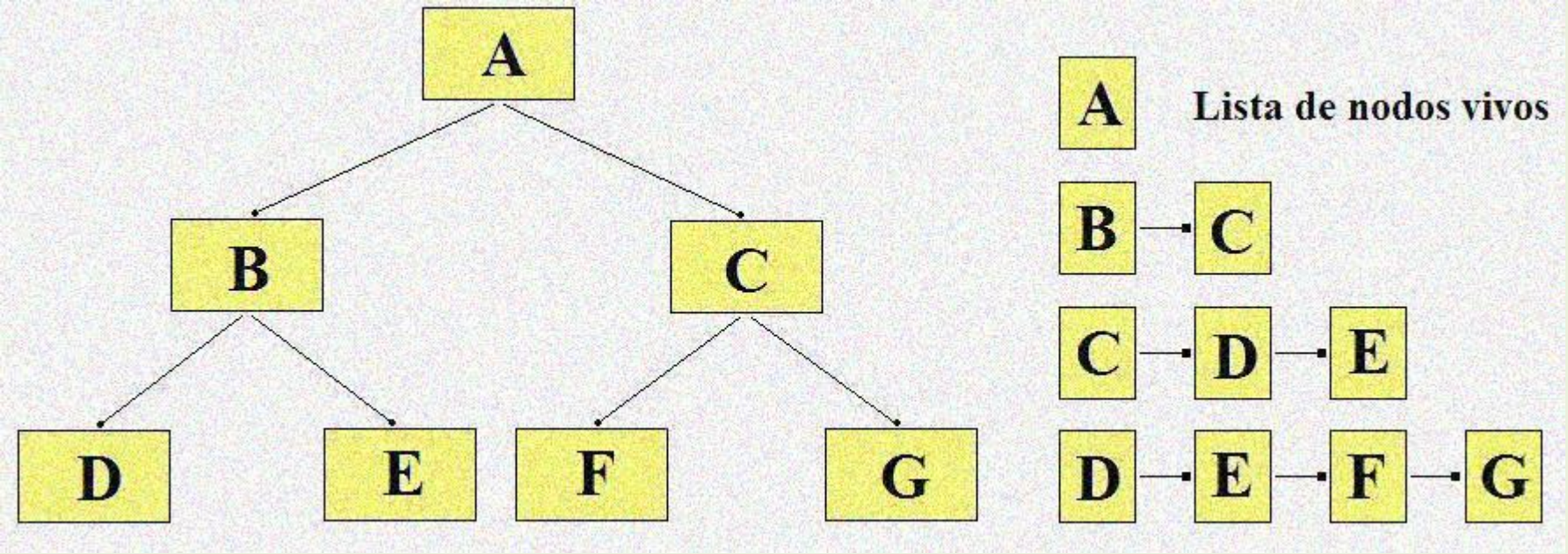
Todos estos nodos que se van generando y que no han sido explorados se almacenan en lo que se denomina Lista de Nodos Vivos (a partir de ahora LNV), nodos pendientes de expandir por el algoritmo.

Estrategias de Ramificación

La LNV contiene todos los nodos que han sido generados pero que no han sido explorados todavía. Según como estén almacenados los nodos en la lista, el recorrido del árbol será de uno u otro tipo, dando lugar a tres estrategias.

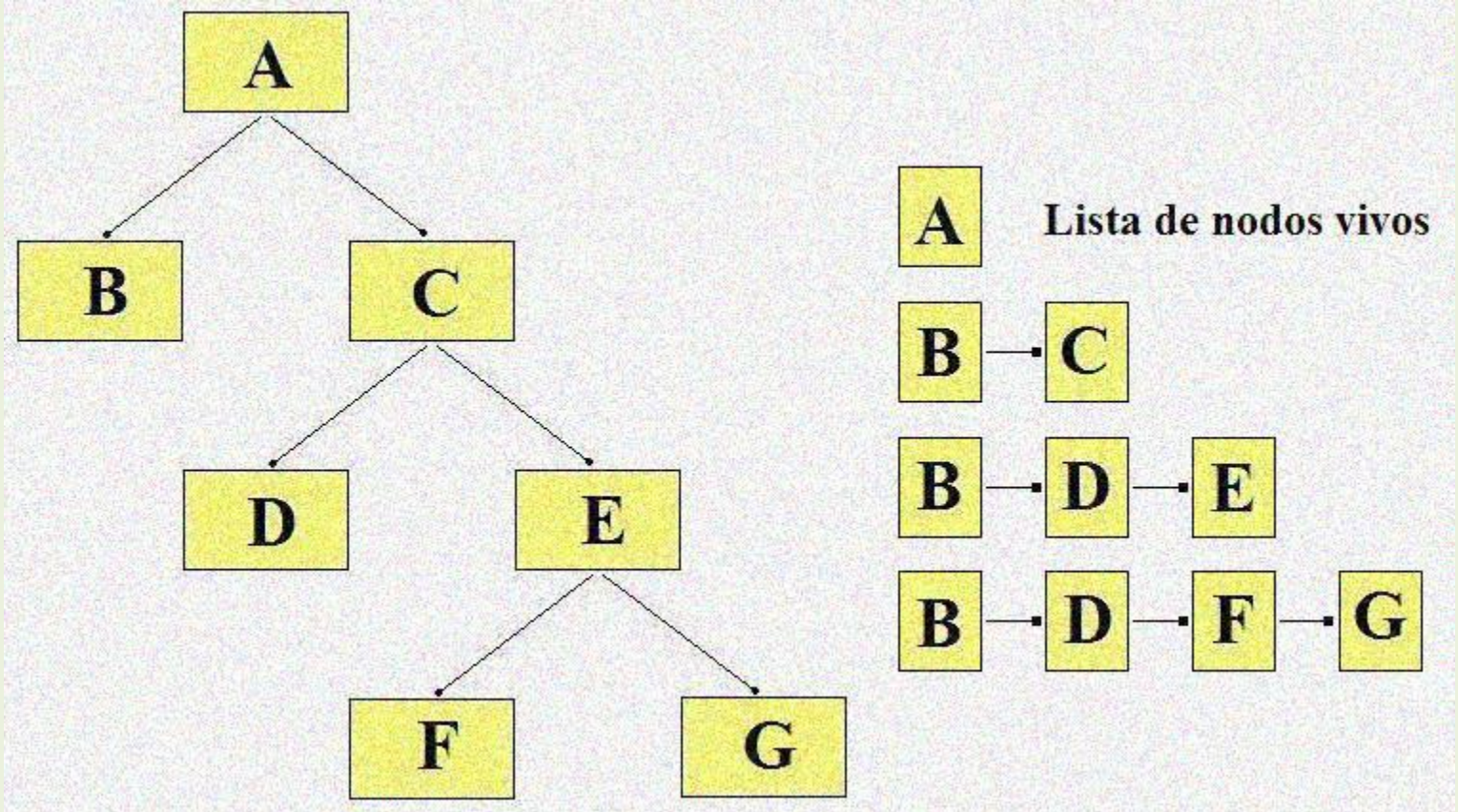
Estrategia FIFO

En la estrategia FIFO (First In First Out), la LNV será una cola, dando lugar a un recorrido en anchura del árbol.



Estrategia LIFO

En la estrategia LIFO (Last In First Out), la LNV será una pila, produciendo un recorrido en profundidad del árbol.



Estrategia de Menor Coste o LC

- Al utilizar las estrategias FIFO y LIFO se realiza lo que se denomina una búsqueda “a ciegas”, ya que expanden sin tener en cuenta los beneficios que se pueden alcanzar desde cada nodo. Si la expansión se realizase en función de los beneficios que cada nodo reporta (con una “visión de futuro”), se podría conseguir en la mayoría de los casos una mejora sustancial.

- Es así como nace la estrategia de Menor Coste o LC (Least cost), selecciona para expandir entre todos los nodos de la LNV aquel que tenga mayor beneficio (o menor coste). Por tanto, ya no estamos hablando de un avance “a ciegas”.

Ramificación y Poda "Relajado"

- Un variante del método de ramificación y poda más eficiente se puede obtener “relajando” el problema, es decir, eliminando algunas de las restricciones para hacerlo más permisivo.
- Cualquier solución válida del problema original será solución válida para el problema “relajado”, pero no tiene por qué ocurrir al contrario. Si conseguimos resolver esta versión del problema de forma óptima, entonces si la solución obtenida es válida para el problema original, esto querrá decir que es óptima también para dicho problema.
- La verdadera utilidad de este proceso reside en la utilización de un método eficiente que nos resuelva el problema relajado. Uno de los métodos más conocidos es el de Ramificación y Corte (Branch and Cut (versión inglesa)).

Ramificación y Corte

- Ramificación y corte es un método de optimización combinatoria para resolver problemas de enteros lineales, que son problemas de programación lineal donde algunas o todas las incógnitas están restringidas a valores enteros.

- Se trata de un híbrido de ramificación y poda con métodos de planos de corte.

- Resuelve problemas lineales con restricciones enteras usando algoritmos regulares simplificados.

Ramificación y Corte

- El algoritmo de planos de corte se usa para encontrar una restricción lineal más adelante que sea satisfecha por todos los puntos factibles enteros.

- Si se encuentra esa desigualdad, se añade al programa lineal, de tal forma que resolverla nos llevará a una solución diferente que esperamos que sea “menos fraccional”.

- Este proceso se repite hasta que ó bien, se encuentra una solución entera (que podemos demostrar que es óptima), ó bien no se encuentran más planos de corte.

BIBLIOGRAFÍA

- Cairó, Osvaldo y Guardati, Silvia. (2002). Estructuras de datos (2a. Edición). McGraw-Hill.
- Joyanes Aguilar L., Fundamentos de programación, algoritmos, estructuras de datos y objetos, 4ed. McGrawHill.
- Brassard G., Bratley P., Fundamento de algoritmia. Prentice Hall.
- Corona Nakumara M. A., Diseño de algoritmos y su codificación en el lenguaje C, McGrawHill.
- Dasgupta, Papadimitriou, Algorithms, Julio 2006.
- Baase Van G., Algoritmos computacionales, enteroroducción al análisis y diseño, 3ra Ed., Addison Wesley.