1. Recurrencias: Fórmulas Maestras

General I:

- $T(n) = aT(n/b)+cn^k$,
  con $T(1)=c$, $b>1$, $k>=0$

- $T(n)=\Theta(n^k)$ si $a<b^k$
- $T(n)=\Theta(n^k \log n)$ si $a=b^k$
- $T(n)=\Theta(n^{\log_b a})$ si $a>b^k$

General II:

$$T(n)=\begin{cases} f(n) & si\ 0\leq n< c \\ aT(n-c)+bn^k & si\ c\leq n \end{cases}$$

$$Entonces$$

$$T(n)=O(n^k)\ si\ a<1$$

$$T(n)=O(n^{k+1})\ si\ a=1$$

$$T(n)=O(a^{n/c})\ si\ a>1$$

Ecuación Característica:

- $T(n) = aT(n/b)+cn^k$
- El polinomio característico es

  $(x-a)(x-b^k)$

$O(n^{\log_b a})$ si $a>b^k$

$O(n^k)$ si $a<b^k$

- Si $a = b^k$ entonces las soluciones son

  $O(n^k \log n)$

2. Divide y Vencerás

Pseudocódigo: (f=fin. Ej: fhacer=fin hacer)

```
método divideYVencerás(x) retorna y
  si x es suficientemente sencillo entonces
    // caso directo
    retorna algoritmoDirecto(x)
  fsi
  // caso recursivo
  descompone x en subproblemas x1, x2, .., xs
  desde i := 1 hasta s hacer
    // llamadas recursivas
    yi := divideYVencerás(xi)
  fhacer
  // combina las soluciones
  y := combinación de las soluciones parciales (yi)
  retorna y
fmétodo
```

Ejemplo DyV (Búsqueda Binaria):

```
método búsquedaBinaria(entero[1..n] t, entero i,
                       entero j, entero x) retorna entero
  // calcula el centro
  k := (i + j)/2
  // caso directo
  si i > j entonces
    retorna -1   // no encontrado
  fsi
  si t[k] = x entonces
    retorna k   // encontrado
  fsi
  // caso recursivo
  si x > t[k] entonces
      retorna búsquedaBinaria(t, k+1, j, x)
  sino
      retorna búsquedaBinaria(t, i, k-1, x)
  fsi
fmétodo
```

Algoritmo de ordenación Quicksort:

```
Procedimiento quicksort (T[i..j])
   // ordena un array T[i..j] en orden creciente
   Si j-i es pequeño entonces Insercion(T[i..j])
   en caso contrario
      pivote (T[i..j], l)
      // tras el pivoteo, si i<=k<l, T[k]<=T[l]
      // y si l<k<=j, T[k]>T[l]
      quicksort (T[i..l-1])
      quicksort (T[l+1..j])
```

```
Procedimiento pivote (T[i..j],l)

   p=T[i]
   k=i; l=j+1;
   repetir k=k+1 hasta T[k]>p o k>=j
   repetir  l=l-1 hasta T[l]<=p
   mientras k<l hacer
       intercambiar T[k] y T[l]
       repetir k=k+1 hasta T[k]>p
       repetir l=l-1 hasta T[l]<=p
   intercambiar T[i] y T[l]
```

3. Algoritmos Greedy (Voraces)

Pseudocódigo:

```
Voraz(C : conjunto de candidatos) : conjunto solución
    S = Ø
    mientras C ≠ Ø y no Solución(S) hacer
        x = Seleccion(C)
        C = C – {x}
        si factible(S U {x}) entonces
            S = S U {x}
        fin si
    fin mientras
    si Solución(S) entonces
        Devolver S
    en otro caso
        Devolver "No se encontró una solución"
    fin si
```

El enfoque Greedy suele proporcionar soluciones óptimas, pero no hay garantía de ello. Por tanto, siempre habrá que estudiar la **corrección del algoritmo** para verificar esas soluciones

Pasos para desarrollar un Greedy:

Conjunto de Candidatos (C) : representa al conjunto de posibles decisiones que se pueden tomar en cada momento.

Conjunto de Seleccionados (S): representa al conjunto de decisiones tomadas hasta este momento.

Función Solución: determina si se ha alcanzado una solución (no necesariamente óptima).

Función de Factibilidad: determina si es posible completar el conjunto de candidatos seleccionados para alcanzar una solución al problema (no necesariamente óptima).

Función Selección: determina el candidato más prometedor del conjunto a seleccionar.

Función Objetivo: da el valor de la solución alcanzada.

Ejemplos Greedy(Colorea Grafos):

```
// coloreado de grafos utilizando una heurística
// voraz
método coloreaGrafoVoraz(Grafo g)
  Lista nodosNoColoreados=g.listaDeNodos()

  mientras nodosNoColoreados no está vacía hacer
    n=nodosNoColoreados.extraePrimero()
    c=un color no usado aún
    n.colorea(c)

    para cada nodo nnc en nodosNoColoreados hacer
      si nnc no tiene ningún vecino de color c
      entonces
        nodosNoColoreados.extrae(nnc)
        nnc.colorea(c)
      fsi
    fhacer

  fhacer // mientras nodosNoColoreados no está vacía

fmétodo
```

Ejemplos Greedy(Mochila):

```
método mochila(real[1..n] p, real[1..n] v, real P)
                                      retorna real[1..n]
   peso := 0
   mientras peso < P y quedan objetos hacer
      i := seleccionaMejorObjeto
      si peso + p[i] <= P entonces
         x[i] := 1                    peso := peso + p[i]
      sino
         x[i] := (P-peso)/p[i]        peso := P
      fsi
   fhacer
   retorna x
fmétodo
```

Ejemplos Greedy(Viajante):

```
// problema del viajante usando heurística voraz
método viajanteVoraz(Grafo g)
  Lista ciudadesPorVisitar=g.listaDeNodos()
  c=ciudadesPorVisitar.extraePrimera()
  Lista itinerario={c} // comienza en c

  mientras ciudadesPorVisitar no está vacía hacer
    cprox=ciudad más próxima a c en ciudadesPorVisitar

    // elimina la más próxima de la lista a visitar
    // y la añade al itinerario
    ciudadesPorVisitar.extrae(cprox)
    itinerario.añadeAlFinal(cprox)

    c=cprox  // cprox pasa a ser la ciudad actual

  fhacer

  // vuelve al comienzo
  itinerario.añadeAlFinal(itinerario.primera())
fmétodo
```

3.1. Exploración de grafos: Algoritmos para AGM (Árbol generador Minimal)

Procedimiento Kruskal:

```
 Kruskal_I (Grafo G(V,A))
{ set<arcos> C(A);
  set<arcos> S;              // Solución inic. Vacía
  Ordenar(C);    // de menor a mayor costo
  while (!C.empty() && S.size()!=V.size()-1) { //No solución
     x = C.first(); //seleccionar el menor
     C.erase(x);
      if (!HayCiclo(S,x))  //Factible
        S.insert(x);
      }
   if (S.size()==V.size()-1) return S;     // Hay solución
   else return “No_hay_solucion”;
}
```

Procedimiento Prim:

FUNCION PRIM (G = (V, A)) conjunto de aristas.
(Inicialización)
    T = ∅ (Contendrá las aristas del AGM que buscamos).
    U = {un miembro arbitrario de V}
    MIENTRAS $|U|$!= n HACER
        BUSCAR e = (u,v) de longitud mínima tal que
            $u \in U$ y $v \in V - U$
        T = T + e
        U = U + v
DEVOLVER (T)

Implementación Prim:

```
Funcion Prim (L[1...n, 1...n]: conjunto de aristas
{al comienzo solo el nodo 1 se encuentra en U}
    T = ∅ (contendrá las aristas del AGM)
    Para i = 2 hasta n hacer
        MasProximo [i] = 1; DistMin [i] = L[i, 1];
    Repetir n – 1 veces
        min = ∞;
        Para j = 2 hasta n hacer
            Si 0 ≤ DistMin [j] < min entonces min = DistMin [j];
                                              k = j;
        T = T + (MasProximo [k], k);
        DistMin [k] = -1; (estamos añadiendo k a U)
        para j = 2 hasta n hacer
            si L[j, k] < DistMin [j] entonces
                DistMin [j] = L[j, k];
                MasProximo [j] = k;
    Devolver T.
```

3.2. Exploración de grafos: Algoritmos para caminos mínimos

Procedimiento Dijkstra:

```
C = {2, 3, ..., n } // el nodo 1 es el origen; implicitamente S={1}
PARA i = 2 HASTA n HACER d[i] = c[1, i]
                         p[i] = 1
REPETIR n – 2 VECES
      v = algún elemento de C que minimice d[v]
      C = C – {v} // implicitamente se añade v a S
      PARA CADA w ∈ C HACER
              SI d[w] > d[v] + c[v, w] ENTONCES
                 d[w] = d[v] + c[v ,w]
                 p[w] = v
DEVOLVER d
```

Implementación Dijkstra:

Para mejorar usamos una cola con prioridad de vértices con campos d y p

ALGORITMO DE DIJKSTRA

```
Para cada v en V
  d[v]= infinito
  p[v] =  null
d[s]=0
set<vertices> S; // Vértices seleccionados está vacio
priorityqueue Q;
Para cada  v en V
  Q.insert(v);
while (!Q.empty())
  v = Q.delete-min()
  S.insert(v)     // incluimos v en vértices seleccionados
  Para cada  w en Adj[v]
    if d[w] > d[v] + c(v,w)
      d[w] = d[v] + c(v,w)
      p[w] = v
```

4. Backtracking y Branch&Bound (BK y BB)

Pseudocódigo BK (recursivo):

```
void back_recursivo(Solucion & Sol, int k)
{
if ( k == Sol.size())
    Sol.ProcesaSolucion();
else {
    Sol.IniciaComp(k);
    Sol.SigValComp(k);
    while (!Sol.TodosGenerados(k) {
      if (Sol.Factible(k))
          back_recursivo(Sol, k+1);
      Sol.SigValComp(k);
     }
  }
}
```

Pseudocódigo BK (iterativo):

```
void back_iterativo (Solucion & sol)   //BK ITERATIVO
{  int k = 0;       // k representa la componente actual
  sol.IniciaComp(k);  //Se inicializa la primera componente a NULO
  while (k >= 0) {
        sol.SigValComp(k); // Probamos el sig. valor para X[k]
        if (sol.TodosGenerados(k))
                k--; //Generados todos, por tanto backtracking
        else {
        if (sol.Factible(k))   // X[k] satisface restric
           { if (k == sol.Size() -1 )     // solución completa
                sol.ProcesaSolucion();
             else  {
                k++; // En caso contrario, ir a siguiente componente
                sol.IniciaComp(k);
             }
           }  else {   ....  // Si el vector solución actual no es factible }
        }
}
```

Ejemplos BK (Problema de las 8 reinas):

```
Void n_reinas(Solucion & sol, int i, bool & solfound)
{
if ( i == Sol.size() )   {Sol.ImprimeSolucion(); solfound=true;}
else {
    Sol.IniciaComp(i);
    Sol.SigValComp(i);
    while (!Sol.TodosGenerados(i) && !solfound) {
      if (Sol.Factible(i)){
          n_reinas(Sol, i+1,solfound);
          Sol.LiberarPosiciones(i);  // tras vuelta atras,
      }                              //  libera posiciones ocup.
      Sol.SigValComp(i);
     }
  }
}
```

El algoritmo imprime todas soluciones,....

Cómo para al encontrar la 1ª?

Deteniendo las llamadas recursivas usando un valor booleano

Ejemplos BK (Suma subconjuntos):

Procedimiento SUMASUB (s,k,r)

{Los valores de X(j), $1 \le j < k$, ya han sido determinados. $s = \sum_{1..k-1} W(j)X(j)$ y $r = \sum_{k..n} W(j)$. **Los W(j) están en orden creciente.**

Se supone que $W(1) \le M$ y que $\sum_{1..n} W(i) \ge M$}

Begin

{Generación del hijo izquierdo. Nótese que $s+W(k)+r \ge M$ ya que Fact(k-1) =true}

X(k) = 1

If s + W(k) = M Then For i = 1 to k print X(j)

Else   If $(s + W(k) + W(k+1)) \le M$

Then SUMASUB (s + W(k), k+1, r-W(k))

{Generación del hijo derecho y evaluación de Fact(k) }

If $s + r - W(k) \ge M$ and $s + W(k+1) \le M$

Then X(k) = 0

SUMASUB(s, k+1, r-W(k))

end

Pseudocódigo BB:

Esquema B&B

```
Algoritmo_B&B( )
{  contenedor<solucion> C;   solucion sol;
C.inserta(sol); // Raiz del arbol de estados
do {   sol=C.Selecciona_Siguiente_Nodo();
         if (sol.Factible()) {
            k = sol.Comp(); // Componente del e nodo;
            k++; // Siguiente componente
            for (sol.PrimerValorComp(k); sol.HayMasValores(k); sol.SigValComp(k))
                if (sol.Factible())  // Incluye las cotas
                     if (sol.NumComponentes()==sol.size())  //Es solucion?
                          sol.ActualizaSolucion();
                     else C.insert(sol);
         }
} while (!C.empty() )
}
```

¿Qué contenedor tenemos que utilizar?

Queue = Criterio FIFO

Stack = Criterio LIFO

Ejemplos BB (Mochila):

```
Solucion Branch_and_Bound(int n_objetos )
{
  priority_queue<Solucion> Q;
  Solucion n_e(n_objetos), mejor_solucion(n_objetos) ;  //nodoe en expansion
  int k;
  float CG=0;   // Cota Global
  float ganancia_actual;

 Q.push(n_e);
 while ( !Q.empty() && (Q.top().CotaLocal() > CG) ){
    n_e = Q.top();
   Q.pop();
    k = n_e.CompActual();
    for ( n_e.PrimerValorComp(k+1); n_e.HayMasValores(k+1);n_e.SigValComp(k+1)) {
           if ( n_e.EsSolucion() ){
             ganancia_actual = n_e.Evalua();
            if (ganancia_actual > CG) { CG = ganancia_actual; mejor_solucion = n_e; }
         }  else if ( n_e.Factible( ) && n_e.CotaLocal()>CG )
                Q.push( n_e );
     }
  }
 return mejor_solucion;
}
```

5. Programación Dinámica(PD)

Algoritmo General de Programación Dinámica (APD):

Algoritmo 1: Algoritmo de Programación Dinámica (APD)

Entrada: Funciones de costo $c_0, \ldots, c_N$, restricciones $A(x)$ y dinámica del sistema f.

Salida: Funciones de costo mínimo $J_0, \ldots, J_N$ y política óptima $\hat{\pi}$.

$J_N(x) \leftarrow c_N(x)$

para cada $k = N-1, N-2, \ldots, 0$ **hacer**

  para cada $x \in X$ **hacer**

    $J_k(x) \leftarrow \min_{a \in A(x)} \{c_k(x,a) + J_{k+1}(f(x,a))\}$

    $h_k(x) \leftarrow \arg\min_{a \in A(x)} \{c_k(x,a) + J_{k+1}(f(x,a))\}$

  fin

fin

Algoritmo Iterativo de Programación Dinámica (APD):

Algoritmo 2: Iteración de Valores

Entrada: Función de costo c, factor de descuento β, dinámica del sistema f, restricciones $A(x)$, función continua y acotada w_0 inicial y criterio de paro $\epsilon > 0$.

Salida: Función de valor v y política estacionaria óptima h aproximadas.

repetir

  para cada $x \in X$ **hacer**

    $w_{k+1}(x) \leftarrow \min_{a \in A(x)} \{c(x,a) + \beta w_k(f(x,a))\}$

    $v(x) \leftarrow w_{k+1}(x)$

    $h(x) \leftarrow \arg\min_{a \in A(x)} \{c(x,a) + \beta v(f(x,a))\}$

  fin

hasta que $\|w_{k+1} - w_k\| < \epsilon$

Pasos para diseñar un algoritmo de PD:

Uso de la programación dinámica:

1. Caracterizar la estructura de una solución óptima.
2. Definir de forma recursiva la solución óptima.
3. Calcular la solución óptima de forma ascendente.
4. Construir la solución óptima a partir de los datos almacenados al obtener soluciones parciales.

Ejemplos PD (Devolver Cambio):

```
For i=1 to n
  m[i,0]=0;
For i=1 to n
  For j=1 to M
    If (i==1 && j<c[i]) m[i,j]=10e30;
    Else if (i==1) m[i,j]=1+m[i,j-c[1]];
         Else if (j<c[i]) m[i,j]=m[i-1,j];
              Else m[i,j]=min(m[i-1,j],1+m[i,j-c[i]]);
Return m[n,M];
```

La eficiencia del algoritmo es O(nM)

Ejemplos PD (Mochila):

Algoritmo *01Knapsack*(*S, W*):

Input: Conjunto ***S de n*** objetos, beneficio ***bi*** y peso ***wi***; capacidad máxima ***W***

```
for w ← 0 to W do
   B[0,w] = 0
for k ← 1 to n do
   B[k,0] = 0
   for w ← 1 to wk -1 do
         B[k,w] = B[k-1,w]
   for w ← wk to W do
         if B[k-1,w −wk] + bk > B[k-1,w] then
               B[k,w] = B[k-1,w −wk] + bk
         else B[k,w] = B[k-1,w]
return B[n,W]
```

Ejemplos PD (LCS):(Less Common Subsequence)

Algoritmo LCS

```
LCS-Length(X, Y)
m = length(X)  // get the # of symbols in X
n  = length(Y) // get the # of symbols in Y
for i = 1 to m  c[i,0] = 0   // special case: Y_0
for j = 1 to n   c[0,j] = 0   // special case: X_0
for i = 1 to m                   // for all X_i
      for j = 1 to n                    // for all Y_j
            if ( x[i] == y[j] )
                  c[i,j] = c[i-1,j-1] + 1
            else c[i,j] = max( c[i-1,j], c[i,j-1] )
return c                          Eficiencia O(nm)
```

5.2. Exploración de grafos: Algoritmos para caminos mínimos

```
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        D[i][j] = coste(i,j);

for (k=1; k<=n; k++)
    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
            if (D[i][k] + D[k][j] < D[i][j] )
               D[i][j] = D[i][k] + D[k][j];
```

Eficiencia $O(n^3)$

6. Bonus:

Algoritmos greedy:

Se construye la solución incrementalmente, utilizando un criterio de optimización local.

Programación dinámica:

Se descompone el problema en subproblemas solapados y se va construyendo la solución con las soluciones de esos subproblemas.

Divide y vencerás:

Se descompone el problema en subproblemas independientes y se combinan las soluciones de esos subproblemas.