ALGORITMOS DE BÚSQUEDA Y ORDENACIÓN

BÚSQUEDAS

1. Búsqueda Secuencial:

Consiste en recorrer la lista, desde el primer elemento hasta el último, comparando con el valor buscado
Devolverá:
Si la lista contiene el elemento = a su posición, si no lo contiene un valor de fallo (-1).

Opción 1: O(n)
Opción 2: Caso mejor (primera posición) **O(1)**
Caso medio y caso peor Orden **(n)**

Implementado en Pseudocódigo	Implementado en C
Opción 1 **Función** secuencial (T[1..n], valor) **inicio** posición←0 **para** i←1 hasta n hacer **inicio** **si** T[i]= valor **entonces** posición ← i **fin** **return** posición; **fin**	 **int** Secuencial(**int** *lista, **int** elementos,**int** valor) { **int** i, posicion; posicion=-1; **for**(i=0;i<elementos;i++) { **if**(lista[i]==valor) posicion=i; } **return** posicion; }
Opción 2 **función** secuencial (T[1..n],valor) **inicio** **para** i ¬ 1 **hasta** n **hacer** **inicio** **si** T[i] = valor **entonces** **devolver** i **fin** **devolver** 0 **fin**	 **int** Secuencial(**int** *lista, **int** elementos,**int** valor) { **int** i; **for**(i=0;i<elementos;i++) { **if**(lista[i]==valor) **return** i; } **return** -1; }

2. Búsqueda Binaria o dicotómica:

Algoritmo basado en la técnica "Divide y Venderás", la solución de todo caso suficientemente grande se reduce a un caso más pequeño, en este caso de tamaño la mitad.

1º Pasada se toma el elemento central de la lista. Tres posibilidades:

- Que sea el valor buscado, se terminó
- Que sea un valor mayor que el elemento buscado, se repite el proceso con la sublista izquierda
- Que sea un valor menor que el elemento buscado, se repite el proceso con la sublista derecha

Condición: la lista o vector debe estar ordenado en orden creciente

Opción Iterativa: Orden O(log n), superior e inferior.

Implementado en Pseudocódigo

Opción iterativa

```
función búsqueda_binaria (T[1..n],valor)
inicio
     i ← 1
     j ← n
  mientras i < j hacer
  inicio
    k ← (i+j)/2
    caso_de valor < T[k]: j ← k-1
            valor = T[k]: i,j ← k
            valor > T[k]: i ← k+1
  fin
  devolver i
fin
```

Implementado en C

```
int Binaria(int *lista,int elementos,int
valor)
{
  int i,j,k;
  i=0;
  j=elementos;
  while(i<j)
   {
     k=(i+j)/2;
     if(valor  < lista[k]) j=k-1;
     if(valor == lista[k]) i=j=k;
     if(valor  > lista[k]) i=k+1;
   }
  if(valor ==lista[i]) return i;
  else                 return -1;

}
```

Opción Resursiva

```
función b_recursiva (T[i..j],valor)
 inicio
   si i > j entonces
    devolver 0

   si no
   inicio
     k ← (i+j)/2
     caso_de valor < T[k]: j ← k-1
             valor = T[k]: devolver k
             valor > T[k]: i ← k+1
     devolver b_recursiva(T[i..j],valor)
   fin

 fin
```

```
int BRecursiva(int *lista,int i,
                     int j,int valor)
{
  int k;
  if (i> j)
    return -1;
  else
   {
     k=(i+j)/2;
     if(valor  < lista[k]) j=k-1;
     if(valor == lista[k]) return k;
     if(valor  > lista[k]) i=k+1;
    return BRecursiva(lista,i,j,valor);
   }
}
```

ORDENACIÓN

3. Burbuja

Va comparando elementos adyacentes y los intercambia si están desordenados, de esta manera los valores más pequeños, van burbujeando hacia la parte superior de la lista y los más grandes se hunden hacia la parte inferior.

Necesitamos dos bules, uno recorre la lista indicando la posición a la que debe burbujear, el elemento más pequeño de la lista desordenada, y otro interno que por cada iteración del externo, recorre la lista desordenada y se encarga del burbujeo.

El algoritmo está en **el orden (n^2) para todos los casos**

Implementado en Pseudocódigo	Implementado en C
`procedimiento ordenación_burbuja (T[1..n])` `inicio` `  para i ← 1 hasta n - 1 hacer` `  inicio` `  para j ← n - 1 hasta i (inc = -1) hacer` `  inicio` `  si T[j] > T[j + 1] entonces` `  intercambiar(T[j],T[j + 1])` `  fin` `  fin` `fin` `procedimiento intercambiar (ref T[x],` `  ref T[y])` `inicio` `  temp ←T[x]` `  T[x] ←T[y]` `  T[y] ←Temp` `fin`	`void Burbuja( int *lista, int elementos)` `{` `  int i,j;` `  for(i=0;i<elementos-1;i++)` `  {` `  for(j=elementos-2;j>=i;j--)` `  {` `  if(lista[j]>lista[j+1])` `  Intercambiar(lista,j,j+1);` `  }` `  }` `}` `void Intercambiar(int *lista,` `  int p, int i)` `{` `  int aux;` `  aux=lista[p];` `  lista[p]=lista[i];` `  lista[i]=aux;` `}`

4. Selección

Busca el elemento más pequeño(orden ascendente) en la lista desordenada que queda pendiente y lo coloca al final de la lista ordenada.
Al principio la lista ordenada está vacía, mientras que la lista desordenada contiene los elementos que se irán examinando para formar la ordenada.

Necesitamos dos bucles, uno externo que indica la posición a insertar de la lista ordenada, y otro interno, que con cada iteración del externo, recorre la lista desordenada y se encarga de buscar el elemento más pequeño a insertar.

El algoritmo está en **el orden (n^2) para todos los casos**

Implementado en Pseudocódigo	Implementado en C
`procedimiento selección(T[1..n])` `inicio` `  para i ←1 hasta n -1 hacer` `  inicio` `  minj ← i` `  minx ← T[i]` `  para j ← i+1 hasta n hacer` `  inicio` `  si T[j] < minx entonces` `  inicio` `  minj ← j` `  minx ← T[j]` `  fin` `  fin` `  T[minj] ← T[i]` `  T[i] ← minx` `  fin` `fin`	`void Seleccion( int *lista, int elementos)` `{` `  int i,j,menor,posicion;` `  for(i=0;i<elementos-1;i++)` `//el ultimo elemento queda ordenado` `  {` `  posicion=i;` `  menor=lista[i];` `  for(j=i+1;j<elementos;j++)` `  {` `  if(lista[j]<menor)` `  { posicion=j;` `  menor=lista[j];` `  }` `  }` `  lista[posicion]=lista[i];` `  lista[i]=menor;` `  }` `}`

5. Inserción

Insertar cada elemento de una lista desordenada en la posición que le corresponde en una lista ordenada.
Al principio la lista ordenada está formada por un único elemento; el primer elemento de la lista a ordenar.

Necesitamos dos bucle, uno externo que recorre la lista indicando el elemento de la lista desordenada a insertar, y uno interno que recorrerá la lista ordenada (comenzando por el final) buscando la posición que le corresponde al elemento a insertar (y realizando los desplazamientos precisos, para poder insertar directamente el elemento).

El algoritmo está en **el orden (n^2) para el caso medio y el peor**
el orden (n) para el caso mejor

Implementado en Pseudocódigo	**Implementado en C**
procedimiento inserción(T[1..n]) **inicio** **para** i ←2 **hasta** n **hacer** **inicio** x ←T[i] j ←i-1 **mientras** j>0 **y** T[j]>x **hacer** **inicio** T[j+1] ←T[j] j ←j-1 **fin** T[j+1] ←x **fin** **fin**	**void** Insercion(**int** *lista, **int** elementos) { **int** i,j,valor; **for**(i=1;i<elementos;i++) { j=i-1; valor=lista[i]; **while**(j > -1 && lista[j]>valor) { lista[j+1]= lista[j]; j--; } lista[j+1]=valor; } }

6.Inserción Binaria

Se trata de insertar cada elemento de una lista desordenada en la posición que le corresponde de una lista ordenada.

Al principio la lista ordenada está formada por un único elemento: el primer elemento de la lista a ordenar.

Se precisan tres bucles:

1. El primer bucle, nos indicará el elemento de la lista a ordenar.
2. El segundo bucle, nos dará la posición en la lista ordenada del elemento a ordenar (la búsqueda de la posición se hará de forma binaria).
3. El tercer bucle, desplazará los elementos necesarios de la lista, para poder insertar el nuevo elemento en la posición correspondiente.

El algoritmo está en **el orden ($n^2 \log n$)**

Implementado en Pseudocódigo	**Implementado en C**
Procedimiento Inserción_Binaria(T [1...n]) **inicio** **para** z ←2 hasta n hacer **inicio** valor ←T[z] i←1 j←z-1 **si** lista[z] > lista[z-1] entonces j=-1 k=z **mientras** i <= j **inicio** k←(i+j) / 2 **caso de** valor > T[k] i ← k+1 valor < T[k] j ← k-1 **fin** **para** i ←z-1 hasta k (decrementando) **inicio** T[i+1] ← T[i] **fin** T [k]← valor **fin** **fin**	**void** InsercionBinaria(**int** *lista, **int** elementos) { **int** z,j,i,k,valor; **for**(z=1;z<elementos;z++) { valor=lista[z]; i=0; j=z-1; **if**(lista[z]>lista[z-1])j=-1; k=z; // ya está en la posición correcta **while**(i<=j) { k=((i+j)/2); **if**(valor > lista[k]) i=k+1; **if**(valor < lista[k]) j=k-1; } **for(**i=z-1;i>=k;i--) lista[i+1]=lista[i]; lista[k]=valor; } }

7. Algoritmo Shellsort (Donal Shell)

Se propone como mejora al algoritmo de inserción.

Basado en suponer que los elementos desordenados son pocos y están en posiciones muy alejadas de la que les corresponde en la lista ordenada.
El algoritmo de inserción tiene un comportamiento natural

En vez de realizar comparaciones con los elementos de posiciones consecutivas, se realizan con elementos en posiciones separadas una distancia mayor que uno.

Se realizan varias pasadas del algoritmo de inserción rebajando la distancia de comparación en cada una, de manera que los elementos se acercan a la posición que les corresponde en la lista ordenada

En la última pasada del algoritmo de inserción la distancia de comparaciones es exactamente uno con lo que se garantiza que la lista queda ordenada (algoritmo de inserción visto)

Importante: Inc[1..n_inc]) es una lista con los incrementos crecientes. Obsérvese que para i=1, estamos en el algoritmo de inserción

La eficiencia de este algoritmo se basa en el hecho de que la ordenación con un determinado incremento no deshace el trabajo realizado anteriormente.

Implementado en Pseudocódigo

```
procedimiento Shell(T[1..n],Inc[1..n_inc])
inicio
    para i ← n_inc hasta 1 (inc=-1) hacer
    inicio
      incremento ← Inc[i]
      para j ← incremento + 1 hasta n hacer
      inicio
         x ←  T[j]
         k ←  j-incremento
         mientras k>0 y T[k]>x hacer
         inicio
           T[k + incremento] ←  T[k]
           k ←  k - incremento
         fin
         T[k + incremento] ←  x
      fin
    fin
fin
```

8. Ordenación Rápida o Quick Sort

Basado en la técnica "divide y vencerás", se trata de dividir el problema inicial en dos problemas más sencillos de resolver.

Una lista a ordenar → dos listas a ordenar de tamaño más pequeño.

La técnica es:

1. Escoger un elemento de la lista, denominado pivote.
2. Recolocar todos los elementos de la lista inicial, incluyendo el pivote, de manera que ésta quede dividida en dos sublistas, cumpliéndose que:

---

izquierda: menores que el pivote ← **pivote** → **derecha:** mayores que el pivote

De esta forma el pivote queda ordenado en la posición correspondiente de la lista, de la misma forma se ordenarán las dos sublistas generadas aplicando recursividad, hasta obtener sublistas de tamaño unidad.

En el peor caso (elegir como pivote el más pequeño de la lista, lista ordenada en el mismo orden a ordenar) **el orden es** ($\mathbf{n^2}$)
En el caso medio (si se cumplen una serie de suposiciones) **el orden es (n log n**)

Implementado en Pseudocódigo	Implementado en C
procedimiento OrdenaciónRápida(T[base..tope]) **inicio** **si** base < tope **entonces** **inicio** pivote ← colocar(T[base..tope]) ordenación_rápida(T[base,pivote-1]) ordenación_rápida(T[pivote+1,tope]) **fin** **fin** **función** colocar(T[base..tope]) **inicio** pivote ←base valor_pivote ←T[pivote] **para** índice ←base + 1 **hasta** tope **hacer** **inicio** **si** T[índice] < valor_pivote **entonces** **inicio** pivote ←pivote + 1 intercambiar(T[pivote],T[índice]) **fin** **fin** intercambiar(T[base],T[pivote]) **devolver** pivote **fin**	**void** OrdenacionRapida(**int** *lista, **int** base, **int** tope) { **int** pivote; **if**(base < tope) { pivote=Colocar(lista,base,tope); OrdenacionRapida(lista,base,pivote); OrdenacionRapida(lista,pivote+1,tope); } } **int** Colocar(**int** *lista,**int** base, **int** tope) { **int** pivote,i,valor_pivote; pivote=base; valor_pivote=lista[pivote]; **for**(i=base+1;i<tope;i++) { **if**(lista[i] < valor_pivote) { pivote++; Intercambiar(lista,pivote,i); } } Intercambiar(lista,base,pivote); **return** pivote; }

Podemos realizar unas mejoras al algoritmo para que el peor caso no se llegue a dar. Ya que el problema principal es la elección del pivote

1. Valor aleatorio de la lista.(pude darnos problemas)
2. Elegir lista((base+tope)/2), no tendríamos los problemas anteriores.
3. Elegir la mediana entre los valores, base, tope, (base+tope)/2

Versión de elección de un pivote diferente (pivote central)

Implementado en Pseudocódigo

```
procedimiento OrdenaciónRápida
                            (T[base..tope])
inicio
 i ←base
 j ←tope
 pivote ← (base + tope)/2
 x ← T[pivote]

 repetir
   mientras T[i] < x hacer i ←i + 1
   mientras T[j] > x hacer j ←j - 1
      si i <= j entonces
      inicio
        intercambiar(T[i],T[j])
        i ←i + 1, j ←j - 1
      fin
 hasta i > j

  si base < j entonces
     ordenación_rápida(T[base,j])
  si tope > i entonces
     ordenación_rápida(T[i,tope])
fin
```

Implementado en C

```
void OR_PivoteCentral (int *lista,
                    int base, int tope)
{
  int i,j, pivote, valor_pivote;
  i=base; j=tope;
  pivote=(base+tope)/2;
  valor_pivote=lista[pivote];

 do{
     while(lista[i] < valor_pivote) i++;
     while(lista[j] > valor_pivote) j--;
     if(i<=j)
      { Intercambiar(lista,i,j);
        i++;  j--;
      }
  }while(i<j);


 if(base < j)
     OR_PivoteCentral(lista,base,j);
 if(tope > i)
     OR_PivoteCentral(lista,i,tope);
}
```

Pseudocódigo de una versión en la que se utiliza otro algoritmo de ordenación en el momento que las listas alcanzan un tamaño demasiado reducido
El valor de mínimo depende de la implementación

Implementado en Pseudocódigo

```
procedimiento ordenación_rápida(T[base..tope])
inicio
   si tope - base >= mínimo entonces
   inicio
     pivote ←colocar(T[base,tope])
     ordenación_rápida(T[base,pivote-1])
     ordenación_rápida(T[pivote+1,tope])
   fin
   si no otro_algoritmo(T[tope,base])
Fin
```

Pseudocódigo de una versión en la que se utiliza un TAD pila para simular el comportamiento recursivo del algoritmo, eliminando las llamadas recursivas

Implementado en Pseudocódigo

```
procedimiento ordenación_rápida(T[1..n])
inicio
  push(mi_pila,1)
  push(mi_pila,n)
  mientras mi_pila no vacía hacer
  inicio
    pop(mi_pila,tope)
    pop(mi_pila,base)
    mientras base < tope hacer
    inicio
      pivote ← colocar(T[base,tope])
      push(mi_pila,pivote+1)
      push(mi_pila,tope)
      tope ←pivote - 1
    fin
  fin
fin
```

9. Ordenación por Fusión

Se basa en la técnica de "divide y vencerás":

Se divide la lista a ordenar en dos sublistas aproximadamente de la misma longitud, para pasar a fusionar de forma ordenada ambas sublistas, una vez han sido ordenadas separadamente
El algoritmo tiene una naturaleza recursiva: para ordenar las dos sublistas que se producen se puede aplicar la misma técnica. El final de la recursividad se tiene cuando tenemos sublistas de longitud unidad

La técnica que se sigue es:

1. Dividir la lista a ordenar en dos sublistas aproximadamente del mismo tamaño
2. Se ordenan las dos sublistas producidas (se puede utilizar cualquier algoritmo, incluyendo este mismo: recursividad)
3. Se fusionan las sublistas ordenadas de tal forma que se mantiene el orden.

El tiempo de ejecución, incluso en el peor caso, **en el orden (nlgn)**

Implementado en Pseudocódigo	Implementado en C

```
procedimiento Ordena_fusión(T[base..tope])

inicio
  si base < tope entonces
  inicio
    ordena_fusión(T[base,(base+tope)/2])
    ordena_fusión(T[(base+tope)/2+1,tope])
    fusionar(base, (base+tope)/2,
             (base+tope)/2+1, tope)
  fin
fin

procedimiento fusionar (base_A, tope_A,
                        base_B, tope_B)
inicio
  matriz A[1..n],B[1..m]
  ind_A ← 1; ind_B ← 1;
  ind_T ← base_A
  n ← tope_A-base_A+1;
  m ← tope_B-base_B+1

  A[1..n] ← T[base_A..tope_A]
  B[1..m] ← T[base_B..tope_B]

  mientras ind_A < n y ind_B < m hacer
  inicio
    si A[ind_A] < B[ind_B] entonces
    inicio
      T[ind_T] ← A[ind_A]
      ind_A ← ind_A + 1
    fin

    si no
    inicio
      T[ind_T] ← B[ind_B]
      ind_B ← ind_B + 1
    fin
    ind_T ← ind_T + 1
  fin

  si ind_A > n entonces
      T[ind_T..tope_B] ← B[ind_B..m]
  si no
      T[ind_T..tope_B] ← A[ind_A..n]

fin
```

```
void OrdenaFusión(int *lista, int base,
                                 int tope)
{
  if (base < tope)
   {
    OrdenaFusion(lista,base,(base+tope)/2);
    OrdenaFusion(lista,(base+tope)/2+1, tope);
    Fusionar(lista,base, (base+tope)/2,
             (base+tope)/2+1, tope);
   }
}

void Fusionar(int *lista, int baseA,
              int topeA,  int baseB, int topeB)
{
 int *listaA, *listaB;
 int A=0,B=0,T, n,m,i;
 T=baseA;
 n=topeA-baseA+1;
 m=topeB-baseB+1;

 listaA=(int *)malloc(sizeof(int)*n);
 listaB=(int *)malloc(sizeof(int)*m);

 if(!listaA || !listaB)
  {
   printf("No hay memoria suficiente");
   exit(1);
  }
 GenerarLista(lista, baseA, topeA, listaA);
 GenerarLista(lista, baseB, topeB, listaB);

 while(A <n && B < m)
  {
    if (listaA[A] < listaB[B])
      lista[T]=listaA[A++];
    else
      lista[T]=listaB[B++];
    T++;
  }
 if(A>n)
  {
   for (i=B;i<m;i++)
    lista[T++]=listaB[i];
  }
 else
  {
   for (i=A;i<n;i++)
   lista[T++]=listaA[i];
  }
  free(listaA);
  free(listaB);
}

void GenerarLista(int *Origen,int base,
                  int tope, int *Destino)
{
  int i,j=0;
  for (i=base;i<=tope;i++,j++)
   {
    Destino[j]=Origen[i];
   }
}
```

10. Radix Sort

Algoritmo de ordenación por distribución.
Pasos a seguir:

1. Distribución de los elementos en grupos, pilas o lotes, atendiendo a una determinada característica de dichos elementos.
2. Ordenación de los grupos individuales, por algún algoritmo conocido.
3. Combinación de las listas ordenadas.(concatenación)

La diferencia fundamental con los estudiados hasta ahora es que se debe conocer algo sobre la estructura o el rango de los elementos.

En el **caso mejor** el tiempo de ejecución estará en el **orden (n)** (ordenación lineal)
En el **caso peor :**

La distribución coloca todos los elementos en un único grupo.
Si a priori se conoce la distribución de los elemento esto se podrá ajustar, de manera que todos los grupos reciban aproximadamente el mismo número de elementos.

También es importante la estructura que se utilice al implementar los grupos. (matrices, listas)

Caso particular:
Algoritmo radix u ordenación por base (ordenación de números enteros con el mismo número de cifras).

La primera versión sería seguir paso por paso las tres fases, decidiendo el número de grupos, los rangos de cada grupo y el algoritmo de ordenación a utilizar en la segunda fase.
Una versión recursiva sería:

1. Tomar la cifra más significativa
2. Distribuir los elementos de la lista entre diez grupos o pilas
3. Aplicar el mismo mecanismo a cada uno de los grupos

Esta versión recursiva (intercambio de base) implica:
No se puede realizar la última fase hasta que no se terminen de ordenar todos los grupos, existe un gran problema de manejo de punteros en las sucesivas llamadas recursivas

Variación de la anterior versión:

1. Tomar la cifra menos significativa
2. Distribuir los elementos de la lista entre diez grupos o pilas
3. Concatenar los grupos en un única lista.
4. Aplicar los mismos pasos con la siguiente cifra hacia la izquierda

Esta versión es mejor:

1. mantiene el orden al pasar de cifras menos a más significativas.
2. No hace falta ningún algoritmo adicional de ordenación
3. No hay problemas de índices ni punteros, ya que se parte de un única lista.

Recibe el nombre de ordenación por base o por intercambio de base porque trata los elementos como números en una determinada base:
Utiliza las cifras decimales (base 10), esto indica también el número de pilas o grupos.

Implementado en Pseudocódigo	Implementado en C
procedimiento RadixSort (**ref** ↑Lista:Nodo; n_grupos, n_cifras) /*Lista es el puntero raíz de una lista enlazada */ **inicio** Nodo ↑pt_grupos [1..n_grupos], ↑pt_últimos[1..n_grupos], ↑pt_aux **para** i ← 1 **hasta** n_cifras **hacer** **inicio** **para** j ← 1 **hasta** n_grupos **hacer** **inicio** pt_grupos[j] ← pt_NULO pt_últimos[j] ← pt_NULO **fin** pt_aux ← Lista **mientras** pt_aux ≠ pt_NULO **hacer** **inicio** extraer la i-ésima cifra de pt_aux↑.llave j ← valor cifra obtenida añadir pt_aux al final de la lista pt_grupos[j], utilizando pt_últimos[j] pt_aux ← pt_aux↑.siguiente **fin** **para** j ← 1 **hasta** n_grupos **hacer** **inicio** **si** pt_grupos[j] ≠ pt_NULO **entonces** añadir lista pt_grupos[j] al final de la lista en construcción Lista **fin** **fin** **fin**	**void** RadixSort(nodo **lista, **int** n_grupos, **int** n_cifras) { nodo *pt_grupos[n_grupos]; nodo *pt_ultimos[n_grupos]; nodo *aux; **int** i,pos,j,pot=10,div=1; **for**(i=0;i<n_cifras;i++) { *//inicializo las matrices de punteros a NULL* **for**(j=0;j<n_grupos;j++) { pt_grupos[j]=NULL; pt_ultimos[j]=NULL; } aux=(*lista); **while** (aux) { j=((aux->num)/div)%pot; *//j será la cifras por la que vamos ordenar* **if**(!pt_grupos[j]) *//no hay ningun nodo en la posición j de la matriz grupos* { pt_grupos[j]=aux; pt_ultimos[j]=aux; } **else** *//como ya hay algún nodo en grupos, solamente tengo que engancharle a final* { (pt_ultimos[j])->siguiente=aux; pt_ultimos[j]=aux; } aux=aux->siguiente; } div*=10; (*lista)=NULL; pos=-1; *// va a guardar el último grupo en el que he estado.* **for**(j=0;j<n_grupos;j++) { **if**(pt_grupos[j]) { **if**(!(*lista)) (*lista)=pt_grupos[j]; **if**(pos!=-1) { pt_ultimos[pos] ->siguiente=pt_grupos[j]; pt_ultimos[j]->siguiente=NULL; *// lo pongo pq no se si va a volver a entrar* } pos=j; } } }*//fin for n_cifras* }

TÉCNICAS RECURSIVAS

Un algoritmo recursivo es aquel que en parte está formado por si mismo o se define en función de si mismo.

La principal ventaja que tiene, es la posibilidad de definir un conjunto infinito de objetos mediante una proposición finita.

Situación Idónea, es cuando el problema a resolver, la función por calcular o la estructura de datos por procesar ya están definidos en términos recursivos.

Es importante tener en cuenta que cada vez que un procedimiento se activa de forma recursiva se crea un nuevo conjunto de objetos locales en la pila del programa, esto puede dar lugar al desbordamiento de la pila.

Por eso, si se puede sustituir un algoritmo recursivo por uno iterativo, hay que hacerlo.
En general siempre es cierto que la recursividad se puede reemplazar por iteración+pila.

Pasos para emplear recursividad:

- Examinar varios casos sencillos (buscar un caso base y un método de resolución que funcione de forma general).
- Encontrar una regla de detención (hay fin de recursividad).

La herramienta fundamental del análisis de algoritmos recursivos es el árbol de recursividad.

Ejemplos de Algoritmos en los que **no se debe emplear recursividad**:

1. Factorial de un número

Factorial de un número 4 = 4*3*2*1 es decir 4*(4-1)!

Implementado en Pseudocódigo	**Implementado en C**
función factorial (n) **inicio** **si** n <= 1 **entonces devolver** 1 **sino devolver** n * factorial (n-1) **fin**	**int** Rfactorial(**int** n) { **int** i; **if** (n<=1) **return** 1; **else return** n*Rfactorial(n-1); }
función factorial_no (n) **inicio** resultado ← 1 **para** i ← 2 **hasta** n **hacer** resultado ← resultado * i **devolver** resultado **fin**	**int** factorial(**int** n) { **int** i, resultado=1; **for**(i=2;i<=n;i++) resultado*=i; **return** resultado; }

Cuando el árbol se reduce a una cadena, la transformación de la recursividad en iteración resulta fácil(generalmente) y ahorrará tiempo y espacio.

2. Fibonacci

Cálculo de los números Fibonacci:
Es una relación de números que se definen por la relación de recurrencia
$F_0 = 0$
$F_1 = 1$
$F_2 = F_{n-1} + F_{n-2}$ para n >= 2

Implementado en Pseudocódigo	**Implementado en C**
función fibonacci (n) **inicio** **si** n <= 0 **entonces devolver** 0 **si** n = 1 **entonces devolver** 1 **sino** **devolver** fibonacci(n-1) + fibonacci(n-2) **fin**	**int** RFibonacci(**int** n) { **if** (n<=0) **return** 0; **if** (n==1) **return** 1; **else** **return** RFibonacci(n-1) + RFibonacci(n-2); }
función fibonacci_no (n) **inicio** **si** n <= 0 **entonces devolver** 0 **si** n = 1 **entonces devolver** 1 **sino** **inicio** a ← 0, b ← 1 **para** i ← 2 **hasta** n **hacer** c ← a + b, a ← b, b ← c **devolver** c **fin** **fin**	**int** Fibonacci (**int** n) { **int** a=0,b=1,c,i; **if**(n<=0) **return** 0; **if**(n==1) **return** 1; **else** { **for**(i=2;i<=n;i++) { c=a+b; a=b; b=c; } **return** c; } }

Ejemplos de Algoritmos en los que **se debe emplear recursividad**:

3. Las Torres de Hanoi

En el momento de la creación del mundo, los sacerdotes recibieron una plataforma de bronce sobre la cual había tres agujas de diamante. En la primera aguja estaban apilados 64 discos de oro, cada uno ligeramente menor que el que está debajo de él. A los sacerdotes se les encomendó la tarea de pasarlos todos de la primera aguja a la tercera, con dos condiciones:

Sólo puede moverse un disco a la vez

Ningún disco podrá ponerse encima de otro más pequeño

Se dijo a los sacerdotes que, cuando hubieran terminado de mover los 64 discos, llegaría el fin del mundo.

Es un problema típico que se resuelve con la técnica divide y vencerás.

El paso clave: concentrarse en mover el último disco, no el primero.

Los pasos del algoritmo serían:

- Mover 63 discos de 1 a 2, utilizando la aguja 3 temporalmente.
- Mover disco 64 de 1 a 3.
- Mover los 63 discos de 2 a 3, utilizando la aguja 1 temporalmente.

Partiendo del árbol de recursividad se necesitan 2^n -1 movimientos para n discos.

Implementado en Pseudocódigo	**Implementado en C**
llamada inicial mueve(n,1,3,2) n=discos, a=origen, b=destino, c=aux **procedimiento** mueve(n, a, b, c) **inicio** **si** n > 0 **entonces** **inicio** mueve (n-1, a,c,b) escribe ("Mueve un disco de ",a,"a",b) mueve(n-1,c,b,a) **fin** **fin**	llamada inicial hanoi(n, 1, 3, 2); **void** hanoi(**int** n, **int** a, **int** b, **int** c) { **if**(n>0) { hanoi(n-1, a, c, b); printf("Mueve un disco de %d a %d",a,b); hanoi(n-1, c, b, a); } }

Algoritmos de Rastreo inverso o Backtracking. (ensayo error)

La tarea consiste en diseñar los algoritmos para encontrar las soluciones de problemas específicos sin seguir una regla fija de cálculo, sino por ensayo y error (tanteo).

El patrón común consiste en:

- Descomponer el proceso de tanteo en tareas parciales
- Generalmente estas se expresan espontáneamente en términos recursivos y consisten en explorar un número finito de subtareas

El proceso entero como un proceso de ensayo y búsqueda que poco a poco construye y rastrea (poda) un árbol de subtareas.

```
procedimiento rastrea
inicio
   soluciones parciales candidatas

    repetir
    inicio

      seleccionar siguiente candidata solución
      si solución aceptable entonces
      inicio
        registra solución parcial
        si solución incompleta entonces
        inicio
          rastrea
          si sin éxito solución final entonces
            elimina solución parcial
        fin
        sino
          marcar final búsqueda
     fin

    hasta final búsqueda o no más candidatas
    fin
fin
```

4. El problema de las ocho reinas

Hay que colocar ocho reinas en un tablero de ajedrez, de forma que ninguna reina se pueda comer a otra.

En el caso concreto que nos ocupa:
La tarea: colocar las ocho reinas
Se divide en ocho subtareas: colocar una reina en una columna, siempre que no se coma a las ya colocadas

Consiste en completar la búsqueda de la solución de un problema, construyendo soluciones parciales (tanteo), asegurándose siempre que sean coherentes con las exigencias del problema.
Si una solución parcial no es coherente a las exigencias del problema, se rastrea de forma inversa, suprimiendo la última solución parcial probada y probando otra.

Implementado en Pseudocódigo	Implementado en C

```
procedimiento reinas(i, ref exito)
inicio
  inicializar selección de posiciones
  para la i-ésima reina
  poner éxito a falso
  repetir
    selección siguiente posición
    si posición segura entonces
    inicio
      coloca reina
      si i < 8 entonces
      inicio
        reinas(i+1,exito)
        si no éxito entonces
          elimina reina
      fin
    sino
      poner éxito a verdadero
    fin
  hasta éxito o no más candidatas
fin
----------------------------------------
procedimiento rastreo (i,ref q)
inicio
  j ← 0
  repetir
   j ← j+1
   q ← FALSO
   si a[j] Y b[i + j] Y c[i - j] entonces
    inicio
      x[i] ← j
      a[j] ← b[i + j] ← c[i - j] ← FALSO
      si i < 8 entonces
      inicio
        rastreo (i + 1, q)
        si (no q) entonces
          a[j]←b[i+j]←c[i-j]←VERDADERO
       fin
   sino q ← VERDADERO
   fin
  hasta (q) O (j = 8)
fin
```

```
i = reina que voy a colocar(1 por columna)
j = n°de fila
*q = bandera, si llego a solución = 1
*tab = tablero, guarda el n°fila(j) de la i
*diaS = diagonal suma
*diaR = diagonal resta

con tab, diaS y diaR:
simulo el tablero ya que la diagonal
principal en la diferencia de coordenadas
(i-j) tiene un valor constante entre(-7..7)
y en la diagonal secundaria su suma(i+j)
también tiene un valor constante entre
(2..16), como nosotros no podemos definir
matrices con esos rangos, la diaS queda
igual pero a la diaR le tengo que sumar7.

ReinasUnaSol(0,0,tab,filas,diaS,diaR);
Las matrices tab,diaS y disR las
inicializo a uno.

void ReinasUnaSol(int i, int *q, int *tab,
          int *filas, int *diaS, int *diaR)
{

 int j=-1;
 do{
   j++;
   (*q)=0;
   if( filas[j]==1 && diaS[i+j]==1 &&
                      diaR[i-j+7]==1)
   {
     tab[i]=j;
     filas[j]=diaS[i+j]=diaR[i-j+7]=0;

     if(i<7)
      {
        ReinasUnaSol(i+1, q, tab, filas,
                           diaS, diaR);
        if((*q)==0)
        filas[j]=diaS[i+j]=diaR[i-j+7]=1;
      }
     else (*q)=1;
   }

 }while((*q)!=1 && j<7);
}
```

Para conseguir todas las soluciones posibles

Implementado en Pseudocódigo

```
procedimiento rastreo (i)
inicio
  para j ← 1 hasta 8 hacer
  inicio
    seleccionar siguiente posición
    si posición segura entonces
    inicio
      colocar reina
      si i < 8 entonces rastreo (i + 1)
      sino imprimir solución
      eliminar reina
    fin
  fin
fin
----------------------------------------
procedimiento rastreo (i)
inicio
   para j ← 1 hasta 8 hacer
   inicio

     si a[j] Y b[i+j] Y c[i-j] entonces
     inicio
       x[i] ← j
       a[j] ← b[i+j] ← c[i-j] ← FALSO

       si i < 8 entonces rastreo (i + 1)
       sino imprimir_solución

      a[j]←b[i+j]←c[i-j] ← VERDADERO
    fin

   fin
fin
```

Implementado en C

```
void ReinasTodasSol(int i, int *tab,
          int *filas, int *diaS, int *diaR)
{
  int j;

  for(j=0;j<8;j++)
   {
    if(filas[j]==1 && diaS[i+j]==1 &&
                      diaR[i-j+7]==1)
    {
      tab[i]=j;
      filas[j]= diaS[i+j]= diaR[i-j+7]=0;
      if(i<7)
         ReinasTodasSol(i+1,tab,
                        filas,diaS,diaR);
      else MuestraSolucion(tab,8);

      filas[j]=diaS[i+j]=diaR[i-j+7]=1;
    }
   }
}
void MuestraSolucion(int *matriz, int dim)
{
   int i,j,pos;
   for(j=0;j<dim;j++)
    { printf("\n");
      for(i=0;i<dim;i++)
      {
        if(matriz[i]==j) printf(" R ");
        else             printf(" O ");
      }
    }
}
```

5. El recorrido del caballo

Se tiene un tablero n*n con n^2 campos. Un caballo se pone en el campo de coordenadas iniciales x_0, y_0. Encontrar una cobertura de todo el tablero (si existe) moviéndolo conforme a las reglas del ajedrez. Es decir, calcular un recorrido de n^2-1 movimientos tales que cada campo del tablero sea visitado exactamente una vez.

Lo primero de todo vamos a suponer si nos situamos en la casilla central (simulando un eje de coordenadas (x,y) todos los movimientos que podríamos realizar, con nuestro caballo, y este resultado lo almacenamos en dos matrices.

```
const int ejex[]={2,1,-1,-2,-2,-1, 1, 2};
const int ejey[]={1,2, 2, 1,-1,-2,-2,-1};
```

Hay que tener en cuenta a la hora de movernos por el teclado, el no sobrepasar los límites del mismo, para eso utilizo las variables u, v que me guardarán la posición anterior del caballo y a las que voy a sumar los movimientos (x,y), para comprobar si es una posición segura.
Al ser rastreo inverso (Backtracking):
La tarea: que el caballo recorra las n*n casillas del tablero
*Se divide en (n*n) subtareas*: mover el caballo a una nueva casilla, en la que no haya estado ya.

Implementado en Pseudocódigo	Implementado en C
procedimiento rastreo (i,ref q) **inicio** k← 1 q← FALSO **repetir** u ←pos_x + eje[k] v ← pos_y + ejey[k] **si** u>=0 **Y** u<N **Y** v>=0 **Y** v<N **entonces** **inicio** **si** tab [u][v]=0 **entonces** **inicio** tab [u][v]=i **si** i < N*N **entonces** **inicio** rastreo(i+1,q) **si** (no q) **entonces** tab[u][v]=0 **fin** **sino** q ← VERDADERO **fin** **fin** k ← k+1 **mientras** (!q) Y (j < 8) **fin**	**void** Caballo(**int** tab[][N],**int** i, **int** pos_x, **int** pos_y, **int** *q) { **int** k,u,v; k=0; *q=0; **do**{ u=pos_x +ejex[k]; v=pos_y + ejey[k]; **if**(u>=0 && u<N && v>=0 && v< N) { **if** (tab[u][v]==0) { tab[u][v]=i; **if**(i< N*N) { Caballo(tab,i+1,u,v,q); **if**(!(*q)) tab[u][v]=0; } **else** *q=1; } } k++; }**while**(!(*q) && k<8); }

Todas las soluciones:

Implementado en Pseudocódigo	Implementado en C
procedimiento rastreo (i) **inicio** k← 1 **repetir** u ←pos_x + eje[k] v ← pos_y + ejey[k] **si** u>=0 **Y** u<N **Y** v>=0 **Y** v<N **entonces** **inicio** **si** tab [u][v]=0 **entonces** **inicio** tab [u][v]=i **si** i < N*N **entonces** rastreo(i+1) **sino** imprimeSolución **fin** **fin** k ← k+1 **mientras** (!q) Y (j < 8) **fin**	**void** Caballo(**int** tab[][N],**int** i, **int** pos_x, **int** pos_y) { **int** k,u,v; k=0; **do**{ u=pos_x + ejex[k]; v=pos_y + ejey[k]; **if**(u>=0 && u<N && v>=0 && v< N) { **if** (tab[u][v]==0) { tab[u][v]=i; **if**(i< N*N) Caballo (tab, i+1 ,u ,v); **else** ImprimeSol(tab); tab[u][v]=0; } } k++; }**while**(k<8); }

6. El problema de la mochila

Un esforzado correo desea llevar en su cartera exactamente v kilogramos, se tiene para elegir un conjunto de objetos de pesos conocidos. Se desea cargar la cartera con un peso que sea igual al objetivo. Es decir, dado un conjunto de pesos p1, p2, ..., pn (enteros positivos), estudiar si existe una selección de pesos que totalice exactamente el valor dado como objetivo, v.

Al ser rastreo inverso (Backtracking):
La tarea: cargar la mochila con un peso de valor v.
Se divide en pn subtareas: cargar la mochila con un peso que no sobrepase el valor n.

Implementado en Pseudocódigo

```
Datos a tener en cuenta
typedef struct t_objeto{
 int peso;    /* Peso del objeto */
 int cargado;/*Si esta en la mochila o
no*/
}objeto;

declaramos en el main:
int mochila[num_objetos]
objeto objetos[num_objetos]
/* Creamos aleatoriamente los objetos*/

ind_m = indice de la mochila.
ind_o = último elemento cargado en la mochila.
peso_m = peso que llevo guardado en la
mochila.
fin   = bandera, 1 si he encontrado solución.
--------------------------------------
procedimiento mochila
inicio
  soluciones parciales candidatas
  repetir
   inicio
     siguiente candidata a solución
     si solución aceptable entonces
     inicio
       registra solución parcial
       si solución incompleta entonces
       inicio
        mochila
        si no llego a solución final
        entonces elimina solución parcial
      fin
      sino marcar final búsqueda
   fin
  hasta final búsqueda o no más
      candidatas
fin
```

Implementado en C

```
void mochila (int *mochila, int ind_m,
                  int ind_o, int peso_m,
                  objeto *objetos, int *fin)
{
 int i=ind_o-1;

do{
   i++;
   (*fin)=0;

   if(!objetos[i].cargado &&
      (objetos[i].peso+peso_m)<=PESO)
   {

      objetos[i].cargado = 1;
      mochila[ind_m] = objetos[i].peso;
      peso_m += objetos[i].peso;

      if(peso_m < PESO)
      {
       mochila(mochila,ind_m+1,i,peso_m,
                          objetos, fin);
        if(!(*fin))
         {
          objetos[i].cargado=0;
          mochila[ind_m] = 0;
          peso_m -= objetos[i].peso;
         }
      }
      else (*fin) = 1;
   }
 } while(!(*fin) && i < num_objetos-1);
}
```

Una variante del problema anterior sería buscar la solución óptima, conocido también como el problema de la mochila: un viajante tiene que hacer las maletas seleccionando entre n artículos, aquellos cuya suma de valores (valor o precio total) sea un máximo (solución óptima) y la suma de sus pesos no exceda de un peso límite dado. Sigue tratándose de una estrategia de rastreo inverso para generar todas las soluciones posibles, pero en este caso, cada vez que se alcance una solución se guarda si es mejor que las anteriores según la restricción definida.

Implementado en Pseudocódigo	Implementado en C

Datos a tener en cuenta

```
typedef struct t_objeto{
 int peso;    /* Peso del objeto */
 int valor; // valor del objeto
 int cargado;/*Si esta ya cargado
}objeto;

declaramos en el main:
int mochila1[num_objetos]
int mochila2[num_objetos]
objeto objetos[num_objetos]
/* Creamos aleatoriamente los objetos*/

ind_m = indice de la mochila.
ind_o = último elemento cargado en la
mochila.
peso_m = peso que llevo guardado en la
mochila.
fin   = bandera, 1 si he encontrado
solución.
---------------------------------------
procedimiento mochila
inicio
  soluciones parciales candidatas
  repetir
   inicio
     siguiente candidata a solución
     si solución aceptable entonces
     inicio
       registra solución parcial
       si solución incompleta entonces
       inicio
        mochila
        si no llego a solución final
        entonces elimina solución
                                parcial
      fin
      sino marcar final búsqueda
   fin
  hasta final búsqueda o no más
      candidatas
fin
```

```
void cargar_mochila(int *mochila1,
                    int *mochila2,int valor1,
                    int *valor2, int ind_m,
                    int ind_o, int peso_m,
                    objeto *objetos)
{
 int valor=valor1;
 int i,j;

 for(i=ind_o;i<10;i++)
  {
    if(!objetos[i].cargado &&
       (objetos[i].peso+peso_m)<=PESO )
     {

      objetos[i].cargado = 1;
      mochila1[ind_m] = i;
      peso_m += objetos[i].peso;
      valor+= objetos[i].valor;

      if(peso_m < PESO)
        cargar_mochila(mochila1, mochila2,
        valor, valor2, ind_m+1,i+1, peso_m,
                                    objetos);
      else
      {
       ImprimeSolucion(mochila1, objetos);
       if(valor > (*valor2))
        {
          for(j=0;j<N_OBJETOS;j++)
             mochila2[j]=mochila1[j];
         (*valor2)=valor;
        }
      }
      objetos[i].cargado=0;
      mochila1[ind_m] = -1;
      peso_m -= objetos[i].peso;
      valor -= objetos[i].valor;
     }
  }
}
```

Voy comprando las soluciones y cojo la de más valor.

7. Números

Obtener todos los números de m (m<=9) cifras, todas ellas distintas de cero y distintas entre sí, de tal manera que el número formado por las n primeras cifras, cualquiera que sea n (n <= m), sea múltiplo de n. Por ejemplo, para m=4 son números válidos, entre otros, los siguientes:

1236 ya que 1 es múltiplo de 1, 12 de 2, 123 de 3 y 1236 de 4

9872 pues 9 es múltiplo de 1, 98 de 2, 987 de 3 y 9872 de 4

Es un problema de Backtracking

Implementado en Pseudocódigo	Implementado en C
procedimiento numeros **inicio** soluciones parciales candidatas **para** recorro todas las posibles soluciones **hacer** **inicio** siguiente candidata a solución **si** solución aceptable **entonces** **inicio** registra solución parcial **si** solución incompleta **entonces** numeros **sino** imprimo resultado elimino solución parcial **fin** **fin** **fin**	**for**(i=0;i<DIM;i++) *//inicializo num* num[i]=-1; Numeros(4,1,num,0); *//llamada inicial* **void** Numeros (**int** m, **int** cifra, **int** *num, **int** resultado) { **int** i,repite,j; **for**(i=1;i<DIM;i++) { repite = 0; **for**(j = 0;j < cifra; j++) **if**(num[j]== i) repite = 1; **if**(!((resultado+i)%cifra) && !repite) { resultado += i; num[cifra-1]=i; **if**(cifra < m) Numeros(m,cifra+1,num,(resultado*10)); **else** printf("\nresultado: %d", resultado); resultado/=10; resultado*=10; num[cifra-1]=-1; } } }

TIPOS ABSTRACTOS DE DATOS

– **Tipo de datos**
  - Define el conjunto de valores que puede tomar una variable,
  - Son dependientes del lenguaje de programación

– **Tipo abstracto de datos (TAD)**
  - Es un modelo matemático, junto con varias operaciones definidas sobre ese modelo
  - Generalmente, los algoritmos se implementan en función de los TAD, sin embargo, el TAD se debe representar en función de los tipos y operaciones básicas del lenguaje de programación

– **Estructuras de datos**
  - Son conjuntos de variables, quizá de tipos distintos, conectados entre sí de diversas formas
  - **Celda**, unidad básica de una estructura de datos. Caja que puede almacenar un valor de un tipo básico o compuesto
  - Las estructuras se crean:
    – Dando nombres a agregados de celdas
    – Interpretando los valores de algunas celdas como representantes de conexiones
    – entre celdas (opcional)
  - Mecanismos de agregación:
    – Matriz, vector, array o arreglo (unidimensional)
    – Estructura de registro
    – Archivo
    – Otros mecanismos de representar relaciones entre celdas:
      - Punteros o apuntadores
      - Cursores

Vamos a construir un TAD con sus operaciones necesarias para poder utilizar el procedimiento purga que se encarga de eliminar los elementos repetidos de una lista.

```
procedimiento purga(ref L : LISTA)
inicio
  act, sig : tipo_posición;
  act ← Primero(L)
  mientras act ≠ FIN(L) hacer
  inicio
    sig ← Siguiente(act,L)
    mientras sig ≠ FIN(L) hacer
    inicio
      si mismo(Recupera(act,L),Recupera(sig,L))entonces
          Suprime(sig,L)
      sino
          sig ← Siguiente(sig,L)
    fin
  act ← Siguiente(act,L)
  fin
fin
```

Listas (nivel de implementación)

1. Mediante matrices

- Los elementos se almacenan en celdas contiguas de una matriz
- Permite recorrer la lista con facilidad y agregar nuevos elementos al final (coste unitario)
- Insertar un nuevo elemento en mitad de la lista obliga a realizar un desplazamiento de todos los elementos a partir de la posición de inserción
- Análogo problema para suprimir un elemento, excepto el último

Declaraciones básicas:

Constante long_máx = 100
tipo
LISTA = **registro**
elementos: matriz[1..long_máx] de tipo_elemento
últ : entero (es el último ocupado)
fin
posición = entero

```
procedimiento Inserta(x:tipo_elemento;
                      p:posición; ref L:LISTA)
inicio
  q : posición
  si L.últ ≥ long_máx entonces
      error ("lista llena")
  sino si p > L.últ + 1 o p < 1 entonces
      error ("la posición no existe")
  sino
  inicio
    para q ← L.últ hasta p(inc=-1) hacer
        L.elementos[q+1] ← L.elementos[q]
    L.últ ← L.últ + 1
    L.elementos[p] ←x
  fin
fin
```

```
función Localiza(x:tipo_elemento;
                 ref L:LISTA):posición
inicio
  q : posición
  para q ← 1 hasta L.últ hacer
     si L.elementos[q] = x entonces
       devolver q

  devolver L.últ + 1
fin
```

```
función FIN(ref L: LISTA):posición
inicio
     devolver(L.últ + 1)
fin
```

```
procedimiento Suprime(p:posición;
                      ref L:LISTA)
inicio
  q : posición
  si p > L.últ o p < 1 entonces
     error ("la posición no existe")
  sino
  inicio
    L.últ ← L.últ - 1
    para q ← p hasta L.últ hacer
      L.elementos[q] ← L.elementos[q + 1]
  fin
fin
```

```
procedimiento Recupera(p:posición;
                       ref L:LISTA):elementos
inicio
  si p > L.últ o p < 1 entonces
     error ("la posición no existe")
  sino devolver L.elementos[p]
fin
```

```
función Imprime_lista(ref L:Lista)
inicio
  q :posición
  para q ← 1 hasta L.últ hacer
   imprimir L.elementos[q]
fin
```

```
función Primero(ref L: LISTA):posición
inicio
  si L.últ= 1 entonces devolver(L.últ + 1)
  sino devolver 1
fin
```

```
función Anula(ref L: LISTA): posición
inicio
  devuelve L.últ = 1
fin
```

```
función Siguiente(p:posición;
                  ref L:LISTA):posición
inicio
   si p = L.últ entonces
     devuelve L.últ +1
   si p = L.últ +1 entonces
     error ("la posición no existe")
   sino si p > L.últ O p < 1 entonces
     error ("la posición no existe")
   sino devuelve p+1
fin
```

```
función Anterior (p:posición;
                  ref L:LISTA):posición
inicio
   si p = 1 entonces
     error ("la posición no existe")
   sino si p > L.últ O p < 1 entonces
     error ("la posición no existe")
   sino devuelve p-1
fin
```

2. Mediante punteros (listas enlazadas)

- Los elementos se almacenan en nodos o celdas enlazadas sencillas, utilizando punteros para enlazar elementos consecutivos
- Las celdas o nodos correspondientes a elementos consecutivos en la lista no tienen que estar en posiciones consecutivas de memoria
- Se evitan los desplazamientos de elementos de la lista en las operaciones de inserción y supresión
- El precio: la memoria adicional ocupada por los punteros
- En esta representación, cada celda contiene:
  - un elemento de la lista
  - un apuntador a la celda con el siguiente elemento de la lista
- Si la lista es a1, a2,...,an, la celda que contienen ai tiene un apuntador a la celda que contiene el elemento ai+1, para i=1,2,...,n-1
- La celda contiene al elemento an posee una apuntador nulo, que no apuntará a ninguna celda.
- Como encabezamiento de la lista:
  - Un puntero
  - Una celda de encabezamiento o nodo ficticio.

Declaraciones básicas:

tipo

tipo_celda = **registro**
elemento : tipo_elemento
sig : ↑tipo_celda

fin

posición = ↑tipo_celda
LISTA = ↑tipo_celda

Operaciones para el TAD:

Sin nodo ficticio	Con nodo ficticio
Procedimiento Inserta, en este caso una posición antes de la indicada por p (suponiendo es legal) o, lo que es lo mismo, en la posición p, **mediante el atajo** **procedimiento** Inserta(x:tipo_elemento; p:posición) **inicio** **temp** : posición temp ← nueva_celda() **si** temp =NULO **entonces** error ("no hay memoria") **sino** **inicio** temp↑.elemento ← p↑.elemento p↑.elemento ← x temp↑.sig ← p↑.sig p↑.sig ← temp **fin** **fin**	Vale para ambos, para el primer y el último elemento **procedimiento** Inserta(x:tipo_elemento; p:posición) **inicio** **temp** : posición temp ← nueva_celda() **si** temp =NULO **entonces** error ("no hay memoria") **sino** **inicio** temp↑.elemento ← p↑.elemento p↑.elemento ← x temp↑.sig ← p↑.sig p↑.sig ← temp **fin** **fin**
función Localiza(x:tipo_elemento; L: LISTA):posición **inicio** q : posición q ← L **mientras** q ≠ NULO **hacer** **inicio** **si** q↑.elemento = x **entonces** **devolver** q **sino** q ← q↑.sig **fin** **devolver** q **fin**	Me salto el primero comparo directamente con el siguiente **función** Localiza(x:tipo_elemento; L: LISTA):posición **inicio** q : posición q ← L // posición ficticio **mientras** q↑.sig ≠ NULO **hacer** **inicio** **si** q↑.sig.elemento = x **entonces** **devolver** q↑.sig **sino** q ← q↑.sig **fin** **devolver** q↑.sig **fin**
función Recupera (p:posición ; L: LISTA): tipo elemento **inicio** **si** p = FIN(L) **entonces** error ("la posición no existe") **sino devolver** p↑.elemento **fin**	**función** Recupera (p:posición ; L: LISTA): tipo elemento **inicio** **si** p = FIN(L) **entonces** error ("la posición no existe") **sino devolver** p↑.elemento **fin**