Tabla de contenido

Unidad 6
Recursividad

Temario: Concepto. Algoritmos recursivos. Seguimiento de la recursión. Tipos de recursividad. Algunos métodos recursivos de búsqueda y ordenación: M-Sort y Q-Sort. Comparación de eficiencia en métodos iterativos vs recursivos peor y mejor caso

Bibliografía:
Apuntes de Diseño de Algoritmos, Autor: J. L. Leiva O. Capítulo 3: Recursividad.
Curso de Programación en C/C++, Autor: Fco. Javier Ceballos Sierra. Capítulo 12: Algoritmos.
Curso de C (2003), autor: Salvador Pozo. Capítulo 24: Funciones V (Recursivas).
Introducción a la programación con C (2008), Autor: Andrés y Marzal Isabel Gracia. Capítulo 3: Funciones; 3.6. Recursivas.
DE GIUSTI, Armando, Algoritmos, datos y programas. Primera edición; Buenos Aires, Argentina; Pearson Education; 2001. Capítulo 8 "*Análisis de algoritmos: concepto de eficiencia*".

1. Concepto. Algoritmos recursivos

La recursividad es una técnica de programación muy potente que puede ser utilizada en lugar de la iteración. Permite diseñar algoritmos que dan soluciones elegantes y simples, y que generalmente son estructuradas y modulares.

¿En qué consiste realmente la recursividad? Es una técnica por la cual un algoritmo se invoca a sí mismo para resolver una "versión más pequeña" del problema original que le fue encomendado. Cuando se obtienen soluciones a problemas en los que una función o procedimiento se llama a sí mismo para resolver el problema, se tienen subprogramas recursivos.

Entonces, para diseñar una solución recursiva de un problema P, se procederá a dividir el problema de instancias más pequeñas como P1, P2,…, Pn; entendiendo que el problema Pi generado es de la misma naturaleza que el problema original P, pero en algún sentido es también más simple que P en su resolución.

La Técnica de divides y vencerás plantea la resolución de problemas en términos de una entrada de tamaño *n,* dividir la entrada en *k* subproblemas, $1<k<n$. Estos subproblemas (que son de la misma clase que el problema original), se resuelven independientemente y después se combinan sus soluciones parciales para obtener la solución del problema original.

Los procesos recursivos se componen de:

a) Una invocación a sí mismo (llamada recursiva); en donde el valor de los parámetros cambiará en cada llamada, volviéndolo un caso más sencillo que el anterior y aproximándose en cada llamado al caso base.

b) Una condición de terminación o salida, también conocida como caso base o degenerado que es la condición que no produce auto invocación y evita los ciclos infinitos.

Veamos un ejemplo, supongamos que deseamos calcular el Factorial de n. En matemáticas es frecuente definir un concepto en función del proceso usado para generarlo. Por ejemplo, una descripción matemática de n! es:

$$n! \begin{cases} \text{si } n = 0\text{; entonces } 1 \\ \text{si } n > 0\text{; entonces } n\,(n-1)! \end{cases}$$

Esta definición es recursiva, puesto que se expresa la función factorial en términos de sí mismo. A partir de esta definición, es fácil obtener un proceso recursivo para calcular el factorial de un número natural n en C.

Ejemplo 1:

```
1.  #include <stdio.h>
2.  int Fac_R (int);
3.  int main ()
4.  {
5.     int n, a;
6.     printf("Ingrese el valor de n: ");
7.     scanf("%d", &n);
8.     a = Fac_R(n);
9.     printf("El rdo de n! es:  %d", a);
10.    return 0;
11. }
12. int Fac_R(int n)
13. {
14.    if (n==0)
15.       return  1;
16.    else
17.       return  n*Fac_R(n-1);
18. }
```

De este programa se puede observar claramente que la función:

1. `Fac_R` se invoca a sí misma (esto es lo que la convierte en recursiva).
2. Cada llamada recursiva se realiza con un parámetro de menor valor `(n-1)` que el de la anterior llamada. Así, cada vez se está invocando a otro problema idéntico pero de menor tamaño, el cual cada vez se aproxima al caso base, cuando `n` vale `0`.
3. Existe un caso degenerado, base o de salida, `Fac_R(0) = 1`, en el que se actúa de forma diferente, esto es, ya no se utiliza la recursividad.
4. Se observa que el tamaño del problema disminuye asegurando que se llegará a este caso degenerado o caso base. Esto es necesario, porque de lo contrario el programa estaría ejecutándose indefinidamente.

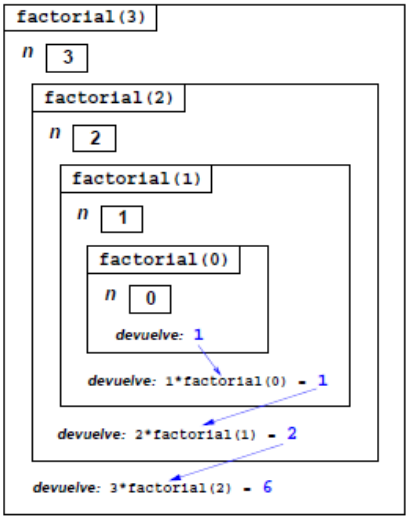
2. Seguimiento de la recursión: Traza

En general, en la pila se almacena el entorno asociado a las distintas funciones que se van activando. En particular, en un módulo recursivo, cada llamada recursiva genera una nueva zona de memoria en la pila, independiente del resto de llamadas.

Por ejemplo: Ejecución del factorial con `n = 3`.

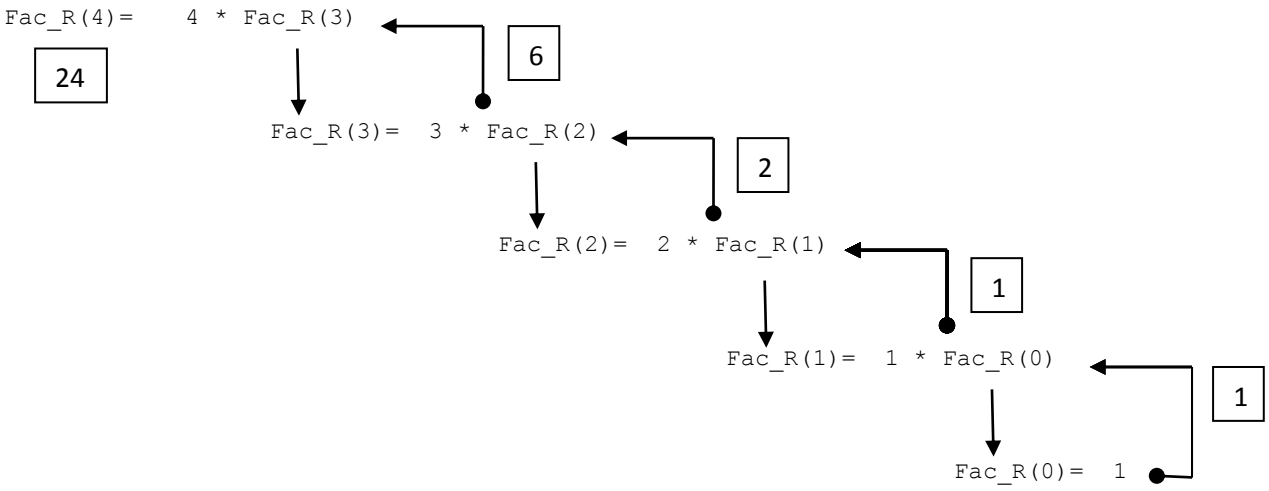
1. Dentro de factorial, cada llamada `return n*factorial(n-1);` genera una nueva zona de memoria en la pila, siendo `n-1` el correspondiente parámetro actual para esta zona de memoria. En cada llamada también queda pendiente la evaluación de la expresión y la ejecución del `return`.

2. El proceso anterior se repite hasta que la condición del caso base se hace cierta. Se ejecuta la sentencia `return1;` empieza la vuelta atrás de la recursión. Se evalúan las expresiones y se ejecutan los `return` que estaban pendientes.



También se suele representar la ejecución como una cascada, donde se expresa cada una de las llamadas al módulo recursivo, así como sus respectivas zonas de memoria y los valores que devuelven. Este proceso se conoce como traza de un algoritmo recursivo.

¿Qué ocurre si invocamos a Fac_R con n=4?



Para determinar si un programa recursivo está bien diseñado, se puede utilizar el método de las tres preguntas:

1. ¿Existe una salida no recursiva o caso base del algoritmo, y éste funciona correctamente para ella?

2. ¿Cada llamada recursiva al proceso se refiere a un caso más pequeño del problema original?

3. Suponiendo que las llamadas recursivas funcionan correctamente, así como el llamado al caso base, ¿funciona correctamente en todo el proceso planteado?

Si por ejemplo, aplicamos este método a la función Fac_R, podemos comprobar cómo las respuestas a las 3 preguntas son afirmativas, con lo que deducimos que el proceso recursivo está bien diseñado.

3. Tipos de recursividad

3.1. Indirecta (mutua)

Es el caso donde una función A llama a otra función B, que a su vez vuelve a llamar a la función A.

Recordemos que en C se debe conocer la definición o la declaración de una función antes de que sea llamada, ya que el compilador requiere conocer el tipo de retorno de la función y el número y tipo de sus parámetros. Para cumplir con este requerimiento basta declarar la función escribiendo el prototipo antes de la definición de `main`.

Volviendo a la recursión indirecta, supongamos que una función A necesita llamar a una función B y que B, a su vez, necesita llamar a A (recursión indirecta), la pregunta será ¿Cuál definimos primero? La solución es sencilla, da igual, siempre que se declare anticipadamente ambas funciones mediante sus respectivos prototipos.

Analicemos el siguiente ejemplo dado un número entero positivo `a` determinar si es par o impar:

Ejemplo 2:

```
1.  #include<stdio.h>
2.
3.  /*declaración de funciones mediante sus prototipos*/
4.  int par(int);
5.  int impar(int);
6.
7.  /*definición de main*/
8.  int main ()
9.  {
10.    int a, rdo;
11.    printf("Ingrese el valor de a: ");
12.    scanf("%d", &a);
13.    rdo = par(a);
14.    if (rdo==0)
15.       printf("El Nro es impar");
16.    else
17.       printf("El Nro es par");
18.    return 0;
19. }
20.
21. /*definición de funciones recursivas*/
22. int par(int a)
23. {
24.    if (a==0)
25.       return 1;
26.    else
27.       return (impar(a-1));
28. }
29.
30. int impar(int a)
31. {
32.    if (a==0)
33.       return 0;
34.    else
```

```
35.       return (par(a-1));
36. }
```

En la primera parte del programa se presenta la declaración de las funciones par e impar por medio de la definición de sus cabeceras (prototipo). Con esta declaración anticipada se adelanta la información acerca de qué tipos de valores aceptará y devolverán las funciones.

3.2. Directa

Es el caso donde una función A se llama a sí misma una o más veces directamente. Como ejemplo se plantea el caso de factorial de n, desarrollado anteriormente.

3.2.1. Directa Múltiple

Es el caso donde la función A se invoca a sí misma, más de una vez dentro de una misma instancia. Analicemos el ejemplo de la Multiplicación de Conejos.

Supongamos que partimos de una pareja de conejos recién nacidos, y queremos calcular cuantas parejas de conejos forman la familia al cabo de *n* meses si:

1. Los conejos nunca mueren.
2. Un conejo puede reproducirse al comienzo del tercer mes de vida.
3. Los conejos siempre nacen en parejas macho-hembra. Al comienzo de cada mes, cada pareja macho-hembra, sexualmente madura, se reproduce en exactamente un par de conejos macho-hembra.

Para un *n* pequeño, por ejemplo 5, la solución se puede obtener fácilmente a mano:

Mes 1: 1 pareja, la inicial.

Mes 2: 1 pareja, ya que todavía no es sexualmente madura.

Mes 3: 2 parejas; la original y una pareja de hijos suyos.

Mes 4: 3 parejas; la original, una pareja de hijos suyos nacidos ahora y la pareja de hijos suyos nacidos en el mes anterior.

Mes 5: 5 parejas; la original, una pareja de hijos suyos nacidos ahora, las dos parejas nacidas en los meses 3 y 4, y una pareja de hijos de la pareja nacida en el mes 3.

Si deseamos saber el número de parejas al cabo de *n* meses, para un *n* cualquiera, podemos construir un algoritmo recursivo fácilmente a partir de la siguiente relación:

$$\text{Parejas(n)} \begin{cases} \text{si n <= 2 entonces Parejas(n) = 1} \\ \text{si n > 2 entonces Parejas(n-1)+Parejas(n-2)} \end{cases}$$

En esta relación Parejas(n-1) son las parejas vivas en el mes *n-1*, y Parejas(n-2) son las parejas que nacen en el mes *n* a partir de las que había en el mes *n-2*.

La serie de números Parejas(1), Parejas(2), Parejas(3),... es conocida como la **Serie de Fibonacci**, la cual modela muchos fenómenos naturales.

A partir de esta definición, es fácil obtener un Programa recursivo para calcular la función Parejas para un número natural n en C.

Ejemplo 3:

```
1.  #include <stdio.h>
2.
3.  int Parejas_R (int);
4.  int main ()
5.  {
6.     int n, a;
7.     printf("Ingrese los meses: ");
8.     scanf("%d", &n);
9.     a = Parejas_R(n);
10.    printf("total de Parejas: %d", a);
11.    return 0;
12. }
13.
14. int Parejas_R(int n)
15. {
16.    if (n<=2)
17.       return  1;
18.    else
19.       return Parejas_R(n-1)+ Parejas_R(n-2) ;
20. }
```

3.2.2. Directa Lineal

Es el caso en donde la función A se invoca a sí misma como mucho una vez dentro de una misma instancia, sin embargo, esta invocación no es la última acción que realiza la función. Por ejemplo, veamos la implementación de la función factorial.

```
intFac_R(int n)
{
   intresu;
      if (n==0)
                  resu= 1;
              else
                  resu= n*Fac_R(n-1);
   return resu;
}
```

3.2.3. Directa Lineal Final

Es el caso en donde la función A se invoca a sí misma como mucho una vez dentro de una misma instancia, y esta llamada recursiva es la última acción que efectúa como parte del cuerpo de su definición.

Por ejemplo la función máximo común divisor `MCD_R`

```
int MCD_R (int m, int n)
{
   if (m % n == 0)
       return n;
   else
       return (MCD(n, m % n));
}
```

4. Algoritmos fundamentales recursivos

4.1. Búsqueda Binaria Recursiva

Se utiliza cuando el vector en el que se quiere determinar la existencia de un elemento, está previamente ordenado. Este algoritmo reduce el tiempo de búsqueda considerablemente, ya que disminuye exponencialmente el número de iteraciones necesarias.

Para implementar este algoritmo se compara el elemento a buscar con un elemento cualquiera del vector (normalmente el elemento central). Si el valor de comparación es mayor que el del elemento buscado se repite el procedimiento en la parte del vector que va desde el inicio de éste hasta el elemento tomado, en caso contrario se toma la parte del vector que va desde el elemento tomado hasta el final.

De esta manera se particional el vector de búsqueda en intervalos cada vez más pequeños, hasta que se obtenga un intervalo indivisible. Si el elemento no se encuentra dentro de este último entonces se deduce que el elemento buscado no se encuentra en todo el vector.

Un programa recursivo en C que utiliza este algoritmo se muestra a continuación:

Ejemplo 4:

```
1.  #include <stdio.h>
2.
3.  int BBR(int [],int, int, int);
4.
5.  int main()
6.  {
7.    int A[] = {2,3,4,5,7,8,9,10,11,13,17,20};
8.    int pri, ult, busca, rdo;
9.    ult=12;
10.   pri=0;
11.   busca=13;
12.   rdo=BBR(A,pri,ult,busca);
13.   printf("Posicion: %d", rdo);
14.   return 0;
15. }
16.
17. int BBR(intvec[], int ini, int fin, int Bus)
18. {
19.   int med;
20.   med=(ini+fin)/2;
21.   if (ini<=fin)
22.       if (Bus==vec[med]) return med;
23.       else
24.            if(Bus<vec[med])
25.                return BBR(vec, ini, (med-1), Bus);
26.            else
27.                return BBR(vec, (med+1),fin , Bus);
28.   else return -1;
29. }
```

4.2. Quick-Sort Recursivo

El algoritmo trabaja de la siguiente forma, se elige un elemento de la lista de elementos a ordenar, al que se le llama pivote.

Luego se reacomodan los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.

La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.

Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento.

Una vez terminado este proceso todos los elementos quedarán ordenados.

Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido. En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es $n \cdot \log(n)$ y en el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de n^2.

No es extraño, pues, que la mayoría de optimizaciones que se aplican al algoritmo se centren en la elección del pivote. Un programa recursivo en C que utiliza este algoritmo se muestra a continuación.

Ejemplo 5:

```
1.  #include <stdio.h>
2.
3.  void cargavec(int [], int);
4.  void quicksort(int [], int,  int);
5.  void mostrarvec( int [], int);
6.
7.  int main()
8.  {
9.     int n;
10.    int v[15];
11.    printf("Ingrese N: ");
12.    scanf("%d", &n);
13.    cargavec( v, n);
14.    quicksort(v, 1, n);
15.    mostrarvec(v, n);
16.    return 0;
17. }
18.
19. void cargavec(int a[], int tam)
20. {
21.    if(tam!=1)
22.       cargavec( a, tam-1);
23.    printf("Elemento %d: ", tam);
24.    scanf("%d", &a[tam]);
25. }
26.
27.    void q_sort(int l[], int ini, int fin)
28.    {
29.          int piv, izq, der, med;
30.
31.          if (ini<fin){
32.                piv=l[ini];
33.                izq=ini;
34.                der=fin;
35.                while (izq<der){
```

```
36.                         while (der>izq && l[der]>piv)
37.                               der--;
38.                         if (der>izq){
39.                               l[izq]=l[der];
40.                               izq++;
41.                         }
42.                         while (izq<der && l[izq]<piv)
43.                               izq++;
44.                         if (izq<der){
45.                               l[der]=l[izq];
46.                               der--;
47.                         }
48.                   }                   /*fin de while (izq<der)*/
49.                   l[der]=piv;         /*izq=der*/
50.                   med=der;
51.                   q_sort(l,ini,med-1);
52.                   q_sort(l,med+1,fin);
53.             }
54.       }
55. void mostrarvec( int a[], int tam){
56.       if (tam!=1)
57.          mostrarvec( a, tam-1);
58.       printf("%d ", a[tam]);
59. }
```

4.3. M-Sort Recursivo

`MergeSort` es un algoritmo de ordenamiento muy eficiente que utiliza la técnica "divide y vencerás". Su orden asintótico es n*log(n).

Para ordenar un vector lo divide por la mitad en dos segmentos y ordena cada segmento de forma recursiva es decir nuevamente divide cada segmento y ordena los subsegmentos de forma recursiva. Hasta que en algún momento los segmentos serán de longitud 1; es cuando se llama a la función `merge` (mezclar) que intercala los elementos del último segmento dividido colocando primero los elementos menores. El programa que utiliza este algoritmo en lenguaje C se muestra a continuación.

Ejemplo 6:

```
1.  #include <stdio.h>
2.
3.  void cargavec(int [], int);
4.  void mostrarvec( int [], int);
5.  void mergesort(int[], int, int);
6.
7.  int main()
8.  {
9.     int v[15], n;
10.    printf("Ingrese N: ");
11.    scanf("%d", &n);
12.    cargavec( v, n);
13.    mergesort(v, 1, n);
14.    mostrarvec( v, n);
15.    return 0;
16. }
17.
18. void cargavec(int a[], int tam)
19. {
20.    if(tam!=1)
```

```
21.       cargavec( a, tam-1);
22.    printf("Elemento %d: ", tam);
23.    scanf("%d", &a[tam]);
24. }
25.
26. void mostrarvec( int a[], int tam)
27. {
28.    if (tam!=1)
29.       mostrarvec( a, tam-1);
30.    printf("%d ", a[tam]);
31. }
32.
33. void merge(int l[],int ini, int m, int fin)
34. {
35.  int aux[20]; int i,j,k,t;
36.  k=0;
37.  i=ini;
38.  j=m+1;
39.  while (i<=m && j<=fin){
40.    k++;
41.    if (l[i]<l[j]){
42.           aux[k]=l[i];
43.           i++;
44.    }
45.    else{
46.           aux[k]=l[j];
47.           j++;
48.    }
49.  }
50.  for(t=i;t<=m;t++){ /*se ejecuta este ciclo o el que sigue, pero no los dos*/
51.    k++;
52.    aux[k]=l[t];
53.  }
54.  for(t=j;t<=fin;t++){
55.    k++;
56.    aux[k]=l[t];
57.  }
58.  for(t=1;t<=k;t++)     /*transfiere el vector mezclado al parámetro de salida*/
59.    l[ini+t-1]=aux[t];
60. }
61.
62. void mergesort(int l[], int ini, int fin)
63. {
64.    int m;
65.    if(ini<fin)
66.    {
67.       m= (ini+fin)/2;
68.       mergesort(l, ini, m);
69.       mergesort(l, m+1, fin);
70.       merge(l, ini, m, fin);
71.    }
72. }
```

5. El concepto de eficiencia

Hasta ahora, hemos desarrollado algoritmos para expresar la solución a problemas computacionales pero sin analizar que tan “buena” es la solución propuesta. Como para la mayoría de los problemas computacionales

se pueden desarrollar diferentes algoritmos, ¿cómo elegir la "mejor" alternativa?, ¿qué se requiere analizar?.

Pensar en la optimización de un algoritmo en algún sentido requiere analizar previamente su eficiencia.

Un algoritmo es eficiente si realiza una administración correcta de los recursos del sistema en el cual se ejecuta.

5.1.Análisis de eficiencia de algoritmos

Se puede analizar desde dos aspectos la eficiencia de un algoritmo:

- *Tiempo de ejecución*: se considerarán más eficientes aquellos algoritmos que cumplen con la especificación del problema en el menor tiempo de ejecución. A esta clase pertenecen aquellas aplicaciones con tiempo de respuesta finito.
- *Administración o uso de la memoria*: serán eficientes aquellos algoritmos que utilicen las estructuras de datos adecuadas de manera de minimizar la memoria ocupada. El énfasis está puesto en el volumen de la información a manejar en memoria.

En este apunte se analizará la eficiencia de los algoritmos según el tiempo de ejecución.

5.1.1. Estrategia del análisis de algoritmos según su tiempo de ejecución

Para poder medir la eficiencia de un algoritmo, según su tiempo de ejecución, es fundamental contar con una medida del trabajo que realiza.

En los algoritmos se toman, básicamente, dos operaciones elementales: las *comparaciones* de valores y las *asignaciones*, y se les asigna una unidad de medida a cada una.

Hay dos formas de estimar el tiempo de ejecución de un algoritmo:

a) Realizar un *análisis teórico*, calculando el número de asignaciones y de comparaciones que realiza.

b) Realizar un *análisis empírico*, aplicando distintos juegos de datos a la implementación del algoritmo, midiendo su tiempo de respuesta. Este análisis es fácil de implementar pero es afectado por algunos factores como:

   - La velocidad de la máquina.
   - Los datos empleados en la ejecución.

Es importante poder realizar un análisis teórico respecto al tiempo de ejecución para poder realizar una comparación relativa entre las diferentes soluciones algorítmicas, sin depender de los datos de experimentación.

Con este fin se introduce la siguiente definición: el tiempo de ejecución T(n) de un algoritmo se dice "*de orden f(n)*" cuando existe una función matemática f(n) que acota a T(n):

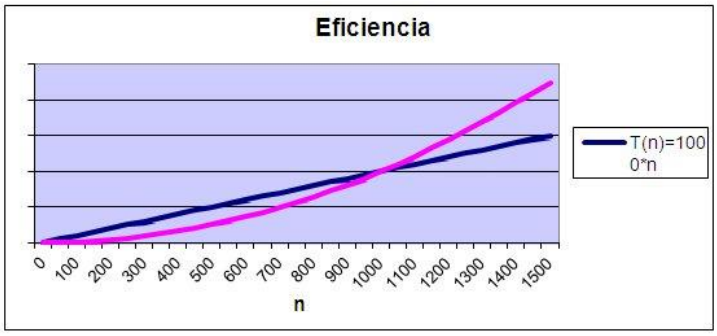
T(n)= O(f(n)) si existen constantes c y n_0 tales que $T(n) \leq c\ f(n)$ cuando $n \geq n_0$.

La definición anterior es una forma de establecer un orden relativo entre las funciones del tiempo de ejecución de los algoritmos T1(n) y T2(n). Según las funciones matemáticas que las acoten (f1(n) y f2(n)) se puede obtener una relación entre T1 y T2.

Por ejemplo, si T1(n)=O(2*n) y T2(n)=O(1/2*n*n), resulta claro que T1 es más eficiente que T2, para n>5.

Otro aspecto a analizar es la velocidad de crecimiento. Por ejemplo, si se compara la función T(n) = 1000*n, $f(n)=n^2$, para c=1; n>1000, $T(n)=O(n^2)$.

Ejemplo:



Al decir que T(n)=O(f(n)), se está garantizando que la función T(n) no crece más rápido que f(n), es decir que f(n) es un límite superior para T(n).

Reglas Generales

Para calcular el tiempo de ejecución de un algoritmo existen ciertas reglas a tener en cuenta para facilitar esta tarea.

Regla 1: Para lazos incondicionales.

El tiempo de ejecución de un lazo incondicional es, a lo sumo, el tiempo de ejecución de las sentencias que están dentro del lazo, multiplicadas por la cantidad de iteraciones que se realizan.

Es decir: cuando el número de iteraciones a realizar depende del tamaño de datos a procesar, la complejidad computacional del ciclo se incrementará con el tamaño de los datos de entrada, por ejemplo:

```
...
for (int i= 0; i < n; i++) {
    ...O(1) /*orden 1*/
}
```

La complejidad será: n* 1= n. Es de orden n.

Regla 2: Para lazos incondicionados anidados.

Se realiza el análisis desde adentro hacia fuera. El tiempo de ejecución de un bloque, dentro de lazos anidados, es el tiempo de ejecución del bloque multiplicado por el producto de los tamaños de todos los lazos incondicionados, por ejemplo:

```
...
for (int i= 0; i < n; i++) {
    for (int j= 0; j < n; j++) {
         O(1) /*orden 1*/
    }
}
```

La complejidad será: n*n*1= n^2. Es de orden n^2.

Regla 3: Para bloques de la forma `If condicion s1 else s2.`

El tiempo de ejecución no es mayor al max(t1,t2) donde t1 es el tiempo de ejecución de s1 y t2 el tiempo de ejecución de s2, por ejemplo:

```
...
if (dato<b[i])
   limite=m-1;
 else inicio=m+1;
```

t1=1, t2=1, max(t1,t2) costo constante. El orden de este fragmento es 1.

Regla 4: Para sentencias consecutivas.

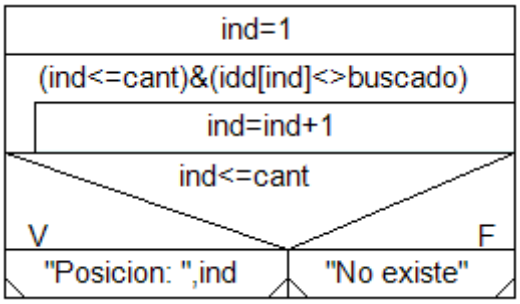
Un fragmento de código formado por dos bloques consecutivos, uno con tiempo T1(n) y otro con tiempo T2(n), el tiempo total es del orden del máximo entre el orden de T1 y T2, por ejemplo:

```
…
printf("Ingrese el valor de n: ");
scanf("%d",&n);
for (i= 1; i <=n; i++)
       if ((i%2) != 0)
              printf("%d  ",i);
…
```

El ciclo se realiza n veces, el cuerpo del bucle es de orden 1, al igual que las sentencias restantes. Es decir T1=1, T2=n, T3=1. El último tiempo (T3) corresponde en la que i evalúa a falso y ya no se entra al bucle. El fragmento del código es de orden n.

5.2.Análisis de los Algoritmos de búsqueda

5.2.1. Algoritmo de búsqueda secuencial



- Mejor caso (cuando es el primero): Se realiza una sola inspección.
- Peor caso (cuando no está): Se realizan n inspecciones.

Dado el siguiente código:

```
1.  ...
2.  ind=1;
3.  while ((ind<=n)&&(idd[ind] !=buscado))
4.     ind= ind++;
5.  if (ind<=n)
6.     printf("Posición: %d",ind);
```

Estimar su tiempo de ejecución, tener en cuenta las reglas generales, enunciadas anteriormente, para calcular el tiempo de ejecución de un algoritmo.

Para el mejor caso

El elemento buscado se encuentra en la primera posición del vector `idd`, vemos que se realiza:

Línea 2: una asignación.

Línea 3: una evaluación de la condición de iteración, dos comparaciones.

Línea 4: En este caso el cuerpo del ciclo no se ejecutará.

Línea 5: una comparación.

Costo=1+2+1=4, costo constante. Se dice que es de orden 1.

Para el peor caso

El elemento buscado no se encuentra en el vector idd, vemos que se realiza:

Línea 2: una asignación.

Línea 3: (n) evaluaciones de la condición de iteración (2 comparaciones) +1 comparación (cuando ind>n).

Línea 4: un incremento y una asignación, que se repiten n veces.

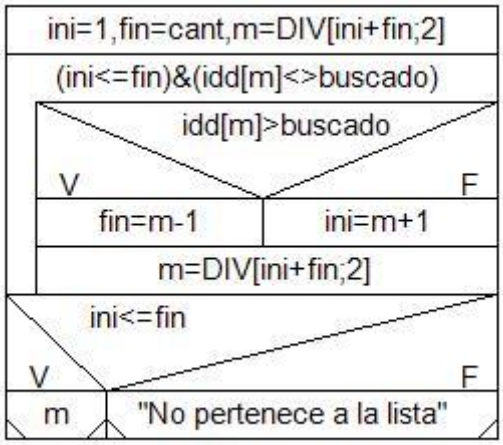
Línea 5: una comparación.

Costo=1+(2n+1)+2n+1=3+4n. Se dice que es del orden n.

La búsqueda secuencial requiere un tiempo proporcional a n.

El algoritmo es de orden n.

5.2.2. Algoritmos de búsqueda binaria



- Mejor caso: se realiza una sola inspección.
- Peor de los casos: la cantidad de iteraciones es del orden del $\log_2 (n)$.

5.3.Análisis de los métodos de ordenamiento

A continuación se realizará el análisis, para el mejor caso y para el peor caso, de los siguientes métodos de ordenamiento:

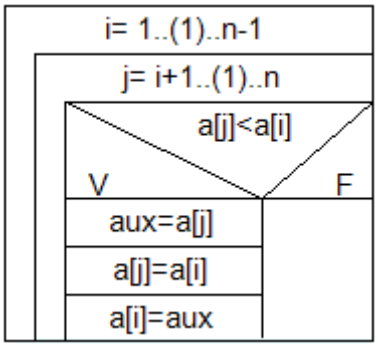
- Selección Directa
- Selección
- Intercambio
- Inserción

Para realizar este análisis se tomará como base la cantidad accesos y de comparaciones en el vector.

El mejor caso será cuando el vector ya se encuentre ordenado.

El peor caso será cuando el vector se encuentre ordenado a la inversa.

5.3.1. Método de Selección Directa

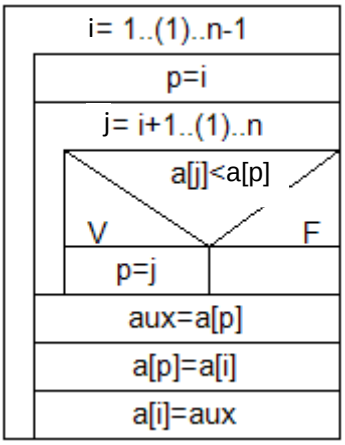


- Peor caso:
  - Cantidad de comparaciones: C=n-1 + n-2 + n-3 + … + 1 = n (n-1)/2
  - Cantidad de intercambios: I= 3C = 3/2 n (n-1)

  Costo = C + I = n (n-1)/2+3/2 n (n-1)=2 n (n-1)=2 n^2 - 2n

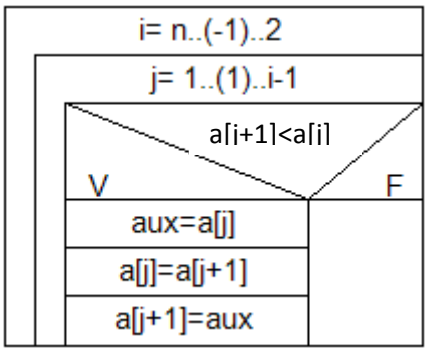
  Decimos que es de orden n^2
- Mejor caso: es igual al peor de los casos.

5.3.2. Método de Selección



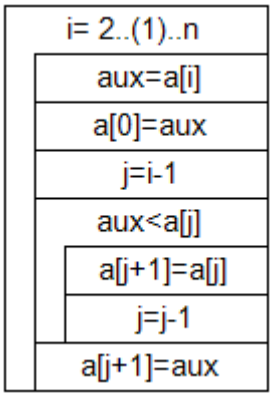
- Peor caso:
  - Cantidad de comparaciones: C=n-1 + n-2 + n-3 + … + 1 = n (n-1)/2
  - Cantidad de intercambios: I= 3 (n-1)
- Mejor caso: es igual al peor de los casos.

5.3.3. Método de intercambio (Burbuja de plomo)



- Peor caso:
  - Cantidad de comparaciones: C = (n-1)+(n-2)+…1=n(n-1)/2
  - Cantidad de intercambios: I = 3/2 n(n-1)
- Mejor caso: La cantidad de comparaciones es igual al del peor caso, sin embargo, no se realizan intercambios.

5.3.4. Método de Inserción (Baraja)



- Peor caso:
  - Cantidad de comparaciones: C = 1+2+3+…+(n-1)=n(n-1)/2
  - Cantidad de accesos al vector: en cada iteración de i se realizan (2 + j) intercambios.

    I = 2(n-1)+n(n-1)/2 = (n-1)(n+4)/2
- Mejor caso:
  - Cantidad de comparaciones: Se realiza una comparación por cada valor de i.

    C = n - 1
  - Cantidad de accesos al vector: no se realizan intercambios pero si se realizan dos accesos al vector por cada valor de i.

    I = 2(n - 1)

5.3.5. Comparación entre métodos de ordenamiento

Método	Selección	Intercambio	Inserción
Mejor caso	C = n(n-1)/2	C = n(n-1)/2	C=(n-1)

	I=3(n-1)	I=0	I=2(n-1)
Peor caso	C = n(n-1)/2 I=3(n-1)	C = n(n-1)/2 I=3n(n-1)/2	C = n(n-1)/2 I=(n-1)(n+4)/2

5.4.Comparación de eficiencia en métodos Iterativos vs recursivos

Hemos visto que la recursión es una técnica potente de programación para resolver, mediante soluciones simples y claras, problemas de incluso gran complejidad. Sin embargo, la recursión también tiene algunas desventajas desde el punto de vista de la eficiencia. Muchas veces un algoritmo iterativo es más eficiente que su correspondiente recursivo. Existen dos factores que contribuyen a ello:

5.5.La sobrecarga asociada con las llamadas a sub-algoritmos

- Este factor es propio de los sub-algoritmos en general, aunque es mayor con la recursividad, ya que una simple llamada inicial a un sub-algoritmo puede generar un gran número de llamadas recursivas.
- Aunque el uso de la recursión, como herramienta de modularidad, puede clarificar programas complejos que compensaría la sobrecarga adicional, no debemos usar la recursividad por el gusto de usarla.

Por ejemplo, la función recursiva `Factorial_R` presentada, no debería usarse en la práctica, podríamos fácilmente escribir una función iterativa `Factorial_I` tan clara como la recursiva y sin la sobrecarga de ésta.

5.6.La ineficiencia inherente de algunos algoritmos recursivos

Esta ineficiencia es debida al método empleado para resolver el problema concreto que se esté abordando. Por ejemplo, en nuestro algoritmo presentado para resolver el problema de la "parejas de conejos", apreciamos un gran inconveniente, los mismos valores son calculados varias veces. Por ejemplo, para calcular `Parejas_R(7)`, tenemos que calcular `Parejas_R(3)` cinco veces. Mientras más grande sea `n`, mayor ineficiencia.

Podemos construir una solución iterativa basada en la misma relación, que actúa hacia delante en lugar de hacia atrás, y calcula cada valor sólo una vez.

El hecho de que la recursividad tenga estas dos desventajas, no quiere decir que sea una mala técnica y que no se deba utilizar, sino que hay que usarla cuando realmente sea necesaria. Podemos encontrar algunas diferencias interesantes entre `Factorial_R` y `Factorial_I`, que se presentarán por lo general entre algoritmos recursivos e iterativos.

- El algoritmo iterativo utiliza una construcción cíclica (PARA, MIENTRAS,...) para controlar la ejecución, mientras que la solución recursiva utiliza una estructura de selección simple.
- La versión iterativa necesita un par de variables locales, mientras que la versión recursiva sólo una.