1.- Demuestre las siguientes afirmaciones. ¿Qué quieren decir?

(a) Si f(n) = O(log a), a > 0, entonces f(n) = O(log n).
(b) Si f(n) = o(g(n)) entonces f(n) + g(n) = O(g(n)).
$n \to +\infty$
(c) Si O(f(n)) < O(g(n)) entonces O(f(n) + g(n)) = O(g(n)). (Aplicar el apartado b)
(d) Si f(n) es la función logarítmica, polinómica o suma y/o producto de ellas, y para n potencia de 2 tenemos que otra determinada función g(n) es O(f(n)). Entonces, si g(n) es creciente, es O(f(n)) para todo n.

2.– Para resolver un determinado problema P_i, se dispone de dos algoritmos: A y B, con complejidades respectivas $f_A(n)$ y $f_B(n)$. Determine, según el siguiente cuadro, para qué valores de n conviene usar A ó B.

	$f_A(n)$	$f_B(n)$
P_1	n^2	$10.n$
P_2	2^n	$2.n^3$

	$f_A(n)$	$f_B(n)$
P_3	$n^2/\log n$	$n.(\log n)^2$
P_4	$n^3/2$	$n^{2,81}$

3.– Calcule el orden de complejidad de los algoritmos de ordenación por selección, inserción e intercambio directos (en el mejor y peor de los casos). Hágase también con las versiones mejoradas inserción binaria y sacudida, comparando los resultados. (Elija como función de complejidad en tiempo la suma del número de comparaciones más el de asignaciones).

4.– Calcule el orden de complejidad de cada uno de los siguientes fragmentos de programa, eligiendo previamente el tamaño y función de complejidad adecuados, para los casos en que Alg(m) sea

1º) O(1) y 2º) $O(m^2)$

```
(a) j = n; k = 1;
    while (j >= 1) {
       k++; j = n/k;
       a = Alg(j);
    }
```

```
(b) j = n; k = 1;
    while (j >= 1) {
       k++; j = n/k;
       for (i = 1; i <= j; i++)
          a = Alg(i);
    }
```

```
(c) j = n;
    while (j >= 1) {
       j = j/5;
       a = Alg(j);
    }
```

```
(d) for (i = 1; i <= n; i++)
       for (j = 1; j <= i; j++) {
          t = n+1;
          while (t > i) {
             a = Alg(t);
             t--;
          }
       }
```

5.– De una función de complejidad en tiempo T(n), en forma recurrente, para el siguiente algoritmo, asumiendo que n es potencia de 2. Deduzca de ella un orden de complejidad (en tiempo) que sea peor o igual al del algoritmo, y otro que sea mejor o igual. El procedimiento E es de orden O(1). ¿Cambiarían los resultados si quitamos la hipótesis de que n sea potencia de 2?.

```
int Camino( int s, int t, int n ) {
   int Cam = true;
   if (n = 1) {
      if (E(s,t) == true) return true ;
      else return false;
   }
   else {
      for ( i = 1 ; i <= n ; i++ )
         Cam = Cam && Camino(s,i,n/2) && Camino(i,t,n/2);
      return Cam;
   }
}
```

6.– El siguiente algoritmo (de Prim) busca un árbol generador mínimo para un grafo conexo valorado. V es un vector que guarda los vértices, X la matriz de costes y F un vector de arcos donde guardamos la solución. Se pide calcular el orden de complejidad del algoritmo (en tiempo) para los casos en que el tamaño del problema sea

a) n = número de arcos del grafo, y
b) m = número de nodos del grafo.

1. i ← 1; x ← V(1);
2. Hacer hasta que i = n
   2.1. Elegir X(a,b) = mín { X(V(x), V(y))/ 1≤x≤i, i<y≤n}
   2.2. F(i) ← (a,b);
   2.3. i ← i + 1;
   2.4. Intercambiar V(i) y V(b).

7.– Calcule el orden de complejidad del siguiente fragmento de programa, en donde Proc(n) es i) O(1) y ii) O(n)

```
for ( i = 1 ; i != n ; i++ ) {
   a = i ;
   do {
      Proc(a);
      a = a/2;
   } while (a != 0);
   j = n/2;
   for ( k = j ; k >= 1 ; k-- )
      Proc( k );
}
```

¿Cambia el orden de complejidad si quitamos el bucle `for` interior?
Repita los cálculos sustituyendo '`a = i`' por '`a = n`'.

8.– Calcule el orden de complejidad de las siguientes funciones o procedimientos

A)

```
void LF( int n , int *s ) {
   int i, x, y;
   if ( n < 1 )  s = 1;
   else {
      x = 1;
      for ( i = 2 ; i != n ; i++ ) x = x * i;
      y = 0;
      do {
         y++;
         x = x/4;
      } while ( x != 0 );
      s = y;
   }
}
```

B)

```
void MoverDisco( int Origen , int Destino ) {
   printf( " Movimiento Desde-Hacia " );
   printf( " Origen " );
   printf( " Destino " );
}

void Transferir( int n, int origen, int destino, int otro ) {
   if ( n > 0 ){
      Transferir( n-1, Origen, Otro, Destino);
      MoverDisco( Origen, Destino);
      Transferir( n-1, Otro, Destino, Origen);
   }
}
```

C) Asumiendo que se cumple la propiedad `x > 1`.

```
int PeroQueHaceEsto( int x, int y ) {
   if ( y > x ) {
      int c = PeroQueHaceEsto(2*x,y);
      while ( y > x ) {
         int i = 0;
         do {
            c = c + n*x*y ;
            i++;
         } while ( i*x <= y );
         x = 2 * x;
      }
   }
   return c;
}
```

D)

```
int Jun91( int n ) {
   int x = 1, y = 1;
   if ( n > 2 ) {
      y = Jun91(n/2);
      while ( x*x < n ) {
        x++;
        y = x + y;
      }
      y = 2 * Jun91(n/2) + x;
   }
   Jun91 = x + y;
}
```

E)

```
int Jun91( int n ) {
   int x , y;
   while ( n > 1 ) {
      x = 1;
      y = n;
      while ( n > x*x ) {
         x++;
         y = y – x;
      }
      n = n/2 ;
   }
   return (x + y)
}
```

F) Calcule la complejidad de las dos funciones siguientes:

```
int Estudio (int n) {
   int i, contador;

   contador = 0;
   for (i = 1; i <= n; i++) contador++;
   return contador;
}

int Examen (int n) {
   int j, k, contador;

   if (n < 2) return 1;
   else {
      contador = Estudio (n);
      j = 1; k = 1;
      while (j < n) {
         j = 3*j; k++;
         contador += Estudio(n);
      }
      return (contador + Examen(n/3) + k);
   }
} // Examen
```

G) Calcule la complejidad de las dos funciones siguientes:

```
int Adios (int n) {
   if (n <= 1) return 1;
   else return (1 + Adios(n-1));
} // Adios


int Despedida (int n) {
   int i, j, acumulador;

   if (n < 1) return 1;
   else {
      acumulador = Despedida(n/2) + Adios(n*n);
      j = 1;
      while (j < n) {
         acumulador += Adios(n);
         for (i = 1; i <= j; i++) acumulador++;
         j++;
      }
      return (acumulador);
   }
} // Despedida
```

H) Calcule la complejidad de las dos funciones siguientes:

```
int Una (int n) {
   if (n < 2) return 1;
   else return (2 * Una(n-1));
}

int Examen (int n) {
   int i, m, suma;

   if (n < 3) {
      suma = 0;
      for (i = 1; i <= n; i++) suma += Una(i);
      return suma;
   }
   else {
      m = n; suma = 0;
      while (m > 1) {
         m = m/3;
         for (i = 1; i <= 10; i++) suma += Una(n);
      }
      return (suma + Examen(n/3));
   }
} // Examen
```

I) Calcule la complejidad de las dos funciones siguientes:

```
int LaOtra (int n) {
   if (n < 2) return 1;
   else return (n * LaOtra(n/3));
}

int Examen (int n) {
   int i, suma;

   suma = 0;
   if (n < 2) {
      for (i = 1; i <= n; i++) suma += LaOtra(i);
      return suma;
   }
   else {
      for (i = 1; i <= n/2; i++) suma += LaOtra(n);
      return (suma + Examen (n/3));
   }
 } // Examen
```

J) Calcule la complejidad de las dos funciones siguientes:

```
int Uno (int n) {
   int K, Suma;

   K = 0; Suma = 0;
   for (i = 1; i <= n; i++) {
       Suma = Suma + n; K++;
   }
   while (Suma > 0) {
      Suma = Suma/2; K++;
   }
   return K;
   }
} // Uno


int Dos (int n) {
   int K, Suma;
   if (n < 2)
      return Uno(n*n);
   else {
      K = n; Suma = Dos (n/3);
      while (K > 0) {
         Suma = Suma + Factorial(n);
         K = K / 2;
      }
      return Suma;
   }
} // Dos
```

K) Calcule la complejidad de las dos funciones siguientes, indicando cuál tiene mayor orden de complejidad (debe implementar Factorial(n) y calcular su complejidad).

```
int Uno (int n) {
   int Suma, x;
   Suma = 0; x = 0;
   while (x*x < n) {
      Suma = Suma + Factorial(x); x++;
   }
   return Suma;
} // Uno


int Examen (int n) {
   int K, Suma;
   if (n < 3)
      return Uno(n*n);
   else {
      K = n; Suma = Examen (n/2);
      while (K > 0) {
         Suma = Suma + Factorial(n);
         K = K / 3;
      }
      return Suma;
   }
} // Examen
```

L) Calcule la complejidad de las dos funciones siguientes, indicando cuál tiene mayor orden de complejidad, donde `Dudua(n)` es una función de complejidad lineal.

```
int Uno (int n) {
   int Suma, x;
   Suma = 0; x = 0;
   while (x*x < n) {
      Suma = Suma + Dudua(x); x++;
   }
   return Suma;
} // Uno

int Examen (int n) {
   int K, Suma;
   if (n < 2) return Uno(2*n);
   else {
      K = n; Suma = Examen (n/2);
      while (K > 0) {
         Suma = Suma + Dudua(n);
         K = K / 4;
      }
      return Suma;
   }
} // Examen
```

M) Calcule la complejidad de las dos funciones siguientes, indicando cuál tiene mayor orden de complejidad.

```
int Uno (int n) {
   int Suma, x;
   Suma = 0; x = 0;
   while (x*x*x < n) {
      Suma = Suma + x; x++;
   }
   return Suma;
} // Uno

int Examen (int n) {
   int K, Suma;
   if (n < 2) return Uno(n*n*n);
   else {
      K = n; Suma = Examen (n/3);
      while (K > 0) {
         Suma = Suma + n;
         K = K / 4;
      }
      return Suma;
   }
} // Examen
```