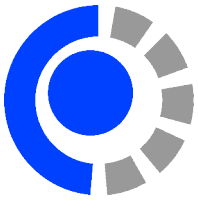
Universidad Nacional del Comahue
Facultad de Informática
Departamento de Programación



Diseño de Algoritmos

Trabajo Práctico 1

Ignacio Alejandro Navarro Oliva

2024

Índice

Introducción

En este informe se detallan los ejercicios y su resolución de trabajos prácticos de la materia Diseño de Algoritmos, de la Licenciatura en Ciencias de la Computación, carrera dictada en la Universidad del Comahue, Neuquén. Aquellos ejercicios que requieran de código, si bien algunas porciones de este están presentadas en este informe, también se encuentran en un repositorio de GitHub.

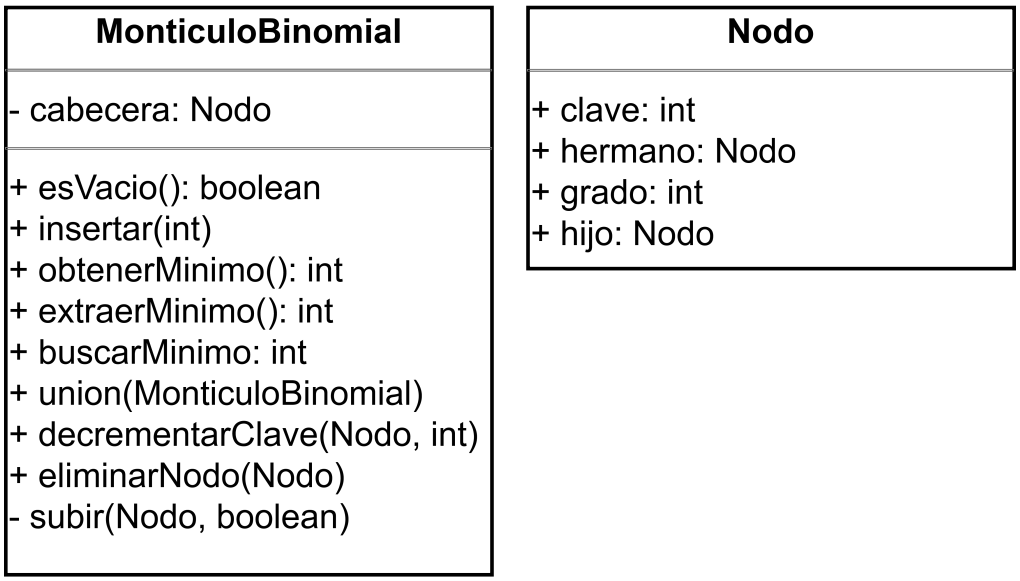


Figura 1: Diagramas UML para las clases *MonticuloBinomial* y *Nodo*

1. Estructuras de Datos Avanzadas y Análisis Amortizado

1.1. Ejercicio 1: montículos binomiales

En este ejercicio se observan las operaciones de un montículo binomial en pseudocódigo y su eficiencia. Dichas operaciones se especifican en el diagrama UML de la figura 1.

1.1.1. *esVacio()*

esVacio()

```
booleano esVacio() {
    retornar cabecera == null;
}
```

Es claro que la operación tiene una complejidad temporal constante $O(1)$.

1.1.2. *insertar(int key)*

insertar(int, key)

```
vacio insertar(entero clave) {
    cabecera = union(nuevo MonticuloBinomial(nuevo Nodo(clave)));
}
```

Insertar un elemento tiene una eficiencia temporal de $O(log\ n)$, ya que la operación *unión* tiene dicho costo.

1.1.3. *obtenerMinimo()*

obtenerMínimo()

```
entero obtenerMinimo() {
    si (cabecera == null) {
        retornar null;
    } sino {
        nuevo min = cabecera;
        nuevo siguiente = cabecera.hermano;

        mientras (siguiente <> null) {
            si (siguiente < min) {
                min = siguiente;
            }
            siguiente = siguiente.hermano;
        }
    }

    retornar min.clave;
}
```

El algoritmo de búsqueda para el mínimo elemento tiene una eficiencia de $O(log\ n)$. Si se quiere, es posible implementar un puntero que siempre apunte al mínimo elemento del montículo.

1.1.4. *extraerMínimo()*

extraerMínimo()

```
entero extraerMinimo() {
    si (cabecera == null)
        retornar null;

    nuevo min = cabecera; nuevo siguiente = min.hermano;
    nuevo minAnt = null;  nuevo siguienteAnt = min;
    mientras (siguiente <> null) {
        si (siguiente < min) {
            min = siguiente;
            minAnt = siguienteAnt;
        }
        siguienteAnt = siguiente;
        siguiente = siguiente.hermano;
    }
    quitarRaiz(min, minAnt);

    retornar min.clave;
}
```

El algoritmo para extraer el mínimo elemento tiene una eficiencia de $O(log\ n)$. La operación privada *quitarRaíz* está definida de la siguiente manera:

quitarRaíz()

```
vacio quitarRaiz(Nodo raiz, Nodo anterior) {
    si (raiz == cabecera)
        cabecera = raiz.hermano;
    sino
        anterior.hermano = raiz.hermano;

    nuevo nuevaCabecera = null;
    nuevo hijo = raiz.hijo;

    mientras (hijo <> null) {
        nuevo siguiente = hijo.hermano;
        hijo.hermano = nuevaCabecera;
        hijo.padre = null;
        nuevaCabecera = hijo;
        hijo = siguiente;
    }

    nuevo nuevoMonticulo = MonticuloBinomial(nuevaCabecera);
    cabecera = union(nuevaCabecera);
}
```

1.1.5. *unión()*

unión()

```
MonticuloBinomial union(MonticuloBinomial otro) {
    cabecera = mezclar(otro);

    si (cabecera == null) {
        return null;
    }

    nuevo anterior = null;
    nuevo x = cabecera;
    nuevo siguiente = x.hermano;

    mientras (siguiente <> null) {
        si ((x.grado <> siguiente.grado) o
                ((siguiente.hermano <> nil) y
                (x.grado == siguiente.hermano.grado)) {
            anterior = x;
            x = siguiente;
        } sino {
            si (x.clave <= siguiente.clave) {
                x.hermano = siguiente.hermano;
                vincular(siguiente, x);
            } sino {
                si (anterior == null) {
                    cabecera = siguiente;
                } sino {
                    anterior.hermano = siguiente;
                }
                vincular(x, siguiente);
                x = siguiente;
            }
        }
        siguiente = x.hermano;
    }

}
```

El algoritmo de *unión* es utilizado por varias operaciones mostradas anteriormente. Esta operación une 2 montículos binomiales $O(log\ n)$. Primero se conectan las cabeceras de ambos montículos con la operación privada *mezclar*. Luego la operación *unión* adapta la estructura para que cumpla las condiciones de un montículo binomial. La operación *vincular* simplemente reordena dos montículos binomiales de mismo orden, combinándolos. Dicha operación también es la que asigna los órdenes o grados de los nodos que se van insertando al montículo. La operación *mezclar* está definida de la siguiente manera:

mezclar()

```
Nodo mezclar(MonticuloBinomial otro) {
    nuevo cabecera1 = cabecera;
    nuevo cabecera2 = otro.cabecera;
    nuevo nuevaCabecera = null;
    nuevo actual = null;

    mientras (cabecera1 <> null y cabecera2 <> null) {
        si (cabecera1.grado <= cabecera2.grado) {
            sino (nuevaCabecera == null) {
                nuevaCabecera = cabecera1;
            } sino {
                actual.hermano = cabecera1;
                cabecera1 = cabecera1.hermano;
            }
        } sino {
            si (nuevaCabecera == null) {
                nuevaCabecera = cabecera2;
            } sino {
                actual.hermano = cabecera2;
                cabecera2 = cabecera2.hermano;
            }
        }
        actual = nuevaCabecera;
    }

    si (cabecera2 <> null) {
        nuevaCabecera.agregarHermano(sinHermanos(cabecera2));
    }
}
```

Por último, se incluye la definición de la operación privada *vincular*:

vincular(nodo,nodo)

```
vacio vincular(Nodo menor, Nodo mayor) {
    mayor.hermano = menor.hijo;
    menor.hijo = mayor;
    menor.grado++;
}
```

Finalmente, las operaciones restantes *decrementarClave* y *eliminarNodo* (junto a su operación auxiliar *subir*, también llamada *bubbleUp* en inglés), tienen una complejidad temporal de $O(log\ n)$.

1.2. Ejercicio 2: conjuntos disjuntos

El ejercicio 2 propone construir algoritmos para las operaciones de *buscar()* y *fusionar()* entre conjuntos disjuntos con una representación en arreglos, y calcular su eficiencia.

Para facilitar la implementación, en lugar de realizarlo en lenguaje de pseudocódigo, este ejercicio se presenta en lenguaje Java.

Una estructura de datos de conjuntos disjuntos tiene un número de conjuntos con elementos únicos que no se comparten entre ninguno de ellos (ni con ellos mismos). Generalmente, se toman elementos de un rango determinado (por ej., de 1 a 100) y se les asigna un conjunto. Esto facilita su representación con arreglos, ya que las posiciones del arreglo simbolizan el elemento o clave, mientras que el elemento guardado en el arreglo representa el conjunto en el que está. Se puede almacenar un *desfase* si los elementos no necesariamente empiezan desde 0 (Java).

En cuanto a las operaciones que tiene esta estructura, contamos con dos métodos: *buscar* y *fusionar*. En este caso, la operación *buscar* es bastante trivial: del elemento buscado x, se retorna el valor $conjunto[x]$. Al ser simplemente un acceso a un arreglo, la operación tiene una eficiencia de $O(1)$.

En el caso de la operación *fusionar*, se define de la siguiente manera:

fusionar(int a, int b)

```java
public void fusionar(int a, int b) {
    // conjuntos no válidos
    if (a >= cantConjuntos || b >= cantConjuntos) {
        return;
    }

    int i = Math.min(a, b);
    int j = Math.max(a, b);

    for (int k = 0; k < conjunto.length; k++) {
        if (conjunto[k] == j) {
            conjunto[k] = i;
        }
    }
}
```

Al iterar sobre todo el arreglo para verificar que todo elemento de un conjunto se coloque en otro, la eficiencia de este algoritmo es de $O(n)$.

1.3. Ejercicio 3: árbol Trie

En este ejercicio se requiere implementar y analizar en Java un algoritmo que almacene, utilizando un árbol Trie, un diccionario de sinónimos. El algoritmo debe permitir:

- Almacenar palabras en el diccionario

- Agregar sinónimos a una palabra
- Mostrar todos los sinónimos de una palabra dada
- Listar todas las palabras del árbol (sin sus sinónimos)

El código del árbol Trie en sí no está expuesto en este informe, pero sí las operaciones que pide el ejercicio. La implementación completa se puede obtener desde el repositorio de GitHub.

Un árbol Trie es un árbol en el que sus nodos solo almacenan un elemento atómico, en este caso letras. De este modo, se hace menos redundante la búsqueda de una palabra ya que simplemente se busca letra por letra hasta encontrar un nodo hoja (o no poder seguir el camino).

El almacenamiento de un árbol Trie se lleva a cabo utilizando un *HashMap*, el cual establece la relación entre una letra y su nodo. La inserción de una palabra se realiza descomponiendola en letras, y recorriendo el árbol desde la raíz para agregar las letras que falten. Finalmente, al último nodo se le asigna el atributo *esHoja*, es cual establece que siguiendo ese camino, dicho nodo completa una palabra.

Para la segunda operación a implementar, se agregará una lista como atributo a cada nodo para que almacene sus sinónimos. Se utiliza el algoritmo del método *buscar* para encontrar un nodo y verificar que complete una palabra (a través del atributo *esHoja*), para luego añadir una palabra a su lista de sinónimos.

buscarNodo, utilizado en *buscar(palabra)* y *añadirSinónimo(palabra, sinonimo)*

```
private NodoTrie buscarNodo(String palabra) {
    HashMap<Character, NodoTrie> hijos = raiz.getHijos();
    int longitud = palabra.length();
    boolean sigue = true;

    NodoTrie nodo = null;
    int i = 0;

    while(i < longitud && sigue) {
        char c = palabra.charAt(i);
        if (hijos.containsKey(c)) {
            nodo = hijos.get(c);
            hijos = nodo.getHijos();
            i++;
        } else {
            nodo = null;
            sigue = false;
        }
    }

    return nodo;
}
```

Mostrar todos los sinónimos de una palabra sigue la misma lógica que la operación anterior, solo que al encontrar la palabra, se retorna la lista de sinónimos.

Las tres operaciones mostradas anteriormente poseen una eficiencia de $O(n)$, siendo n la longitud de la palabra.

Finalmente, para listar todas las palabras del árbol Trie, necesitamos recorrer por todas las claves del *HashMap* (es decir, las letras), para ir acumulando las palabras en una lista auxiliar. Dicho recorrido se realiza de forma recursiva. Para completar la palabra letra por letra, se emplea una variable tipo *String*, llamada prefijo, la cual irá acumulando cada letra hasta verificar que su nodo asociado es una hoja. Luego se retorna la lista auxiliar con todas las palabras encontradas.

listarPalabras(), junto a su operación auxiliar

```
public List listarPalabras() {
    List palabras = new ArrayList<>();
    listarPAux(raiz, "", palabras);
    return palabras;
}

private void listarPAux(NodoTrie nodo, String prefijo, List palabras) {
    if (nodo.esHoja()) {
        palabras.add(prefijo);
    }
    for (char c : nodo.getHijos().keySet()) {
        listarPalabrasAux(nodo.getHijos().get(c), prefijo + c, palabras);
    }
}
```

Como esta operación recorre la completitud del árbol, tiene una eficiencia de $O(n)$, siendo n todos los nodos del árbol.

1.4. Ejercicio 4

El ejercicio pide lo siguiente:

1. Demostrar que si se incluye una operación de decrementar una unidad en el contador binario presentado en clase, en las diapositivas de análisis amortizado, n operaciones costarán como mucho $\Theta(nk)$ en tiempo, donde k es la cantidad de dígitos del número binario.

2. Se realiza una secuencia de n operaciones sobre una estructura de datos dada. La operación i-ésima cuesta i si i es una potencia exacta de 2, y 1 en otro caso. Calcular mediante el método agregado el costo amortizado de cada operación.

El contador binario que menciona el primer inciso se visualiza en la figura 2. Similarmente con las sucesivas operaciones *incrementar* con costo $O(k)$, realizar n veces la operación *decrementar* tiene un costo total de $O(nk)$, siendo k la cantidad de dígitos del número en binario.

En el caso de la operación i-ésima con el costo mencionado, el análisis amortizado de una secuencia

Valor Cont	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Costo Total
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figura 2: Contador binario presentado en clase

de n operaciones sería $O(i/2n)$. Se divide por 2 ya que los casos a considerar son aquellos con costo de potencia de 2, los demás son ignorados al ser un costo constante.

2. Balance espacio-tiempo

2.1. Ejercicio 1

Para adaptar el algoritmo de ordenamiento por conteo con elementos repetidos, se cuenta con un arreglo con 26 posiciones en las que se contará cada letra, sin importar si es mayúscula o minúsucula. Cada letra se cuenta primero convirtiéndola a minúscula y luego restándole la letra 'a' para obtener el índice. Luego, en el arreglo para contar se suma una unidad al índice obtenido. Todo la lógica del algoritmo sigue siendo la misma que el *countingSort* original. En el repositorio de GitHub se encuentran el algoritmo original y el modificado para funcionar con letras. Cabe recalcar que ambos admiten elementos repetidos.

2.2. Ejercicio 2

Dado un patrón de longitud m en un texto con longitud n, $n \leqslant m$, se requiere dar ejemplos del peor y mejor caso cuando se utiliza el algoritmo de Horspool.

El mejor caso sería que prácticamente el patrón sea el texto completo. De ese modo, solamente ocurre una iteración del algoritmo. Para el peor caso es posible armar un texto compuesto solo por *aes*, en el cual el patrón a buscar sea una cadena que comience con cualquier otra letra distinta de *a*, pero que esté seguida de *aes*. Por ejemplo, tomemos el texto *aaaaaa* con el patrón *paa*: el algoritmo solo avanzará una posición en el texto después de comparar el patrón completo, lo cual daría un costo de $O(nm)$.

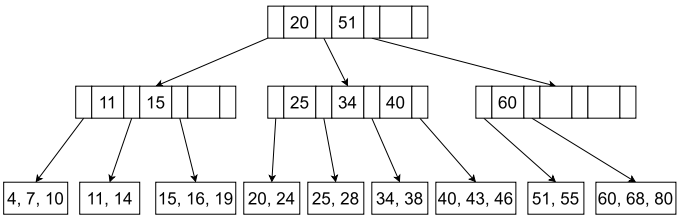


Figura 3: Árbol B

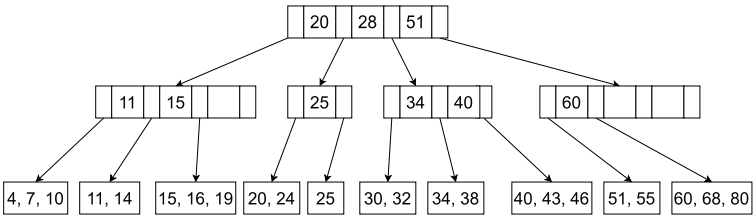


Figura 4: Árbol B con los elementos 30 y 32 insertados

2.3. Ejercicio 3

Una aproximación al problema planteado, que pide generar un listado de palabras diferentes y la cantidad de veces que aparecen en un texto utilizando hashing, es tomar la palabra encontrada como la clave de un HashMap, y que su valor asociado sea su cantidad de apariciones. Entonces, cuando se encuentre una palabra distinta, se inserta en el HashMap con 1 como cantidad inicial asociada (de modo que se pueda leer como un par $< Palabra, Ocurrencias >$). Luego, al encontrar una palabra ya registrada, solamente se incrementa el contador del par. El código en Java puede visualizarse en el siguiente link: hashing.

2.4. Ejercicio 4

En la figura 3 se visualiza el árbol B del ejercicio. Se pide insertar los elementos 30 y 32 y dibujar el árbol resultante, el cual se observa en la figura 4.

3. Algoritmos sobre grafos

3.1. Ejercicio 1

La implementación estática de un grafo se realiza diseñando una matriz de adyacencia de tamaño $N \times N$. Esta implementación se desempeña mejor en ciertas operaciones que la implementación dinámica, que utiliza listas de adyacencia. La implementación dinámica tiene la ventaja que puede almacenar información adicional de los nodos y arcos en los propios objetos de la lista, mientras que en una matriz de adyacencia esta información se agrega externamente. Se podrían reemplazar los números de una matriz de adyacencia por objetos para lograr guardar información adicional, pero esto requiere mucho más espacio. Una distinción clásica entre estructuras estáticas y dinámicas es su escalabilidad: las estáticas tienen un tamaño fijo, mientras que las dinámicas son modificables en tiempo de ejecución.

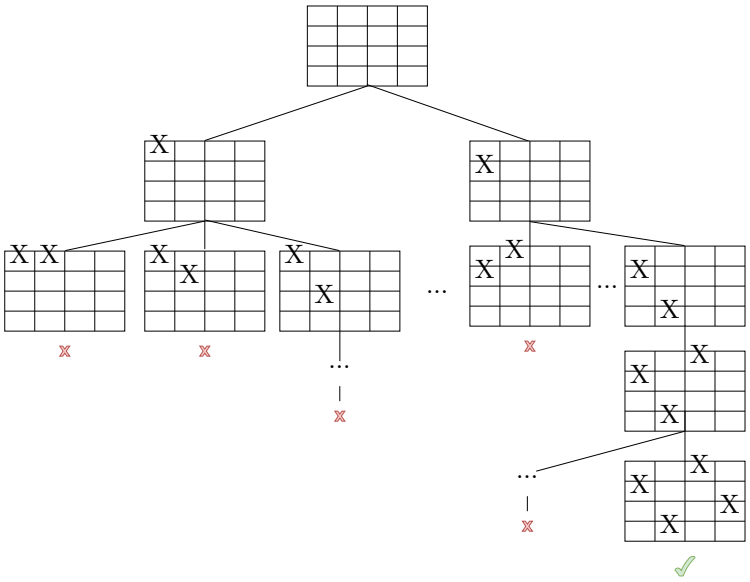


Figura 5: Resolución del problema de n reinas con $n = 4$ utilizando backtracking

	Lista de adyacencia	Matriz de adyacencia
Espacio	$O(\|V\| + \|E\|)$	$O(\|V\|^2)$
Añadir vértice	$O(1)$	$O(\|V\|^2)$
Añadir arco	$O(1)$	$O(1)$
Quitar vértice	$O(\|E\|)$	$O(\|V\|^2)$
Quitar arco	$O(\|V\|)$	$O(1)$

3.2. Ejercicio 2

Para resolver el problema de n reinas utilizando backtracking, podemos considerar recorrer primero las columnas del tablero y verificar si las reinas no se atacan entre sí. Si lo hacen, volvemos al estado anterior y buscamos otra posible solución. En la figura 5 se observa cómo se resuelve el problema para $n = 4$ usando backtracking.

3.3. Ejercicio 3

Con la siguiente tabla, aplicar la técnica de ramificación y poda al problema de asignación de tareas:

	t1	t2	t3
A	25	18	15
B	28	21	30
C	22	19	23

Primero buscamos una cota superior:

- Sumando los elementos de la diagonal principal obtenemos $25 + 21 + 23 = 69$
- Sumando los elementos de la diagonal secundaria obtenemos $15 + 21 + 22 = 58$

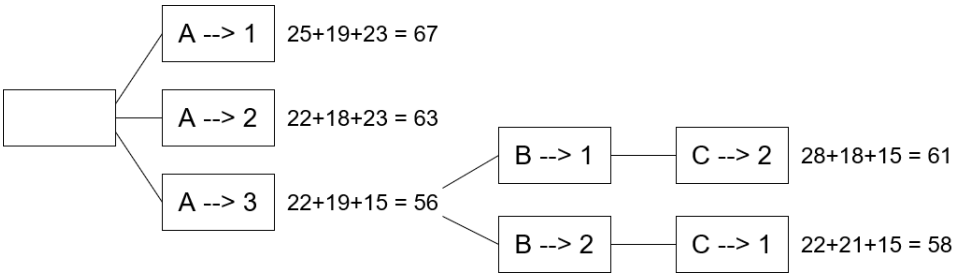


Figura 6: Ramificación y poda para el problema de asignación de tareas

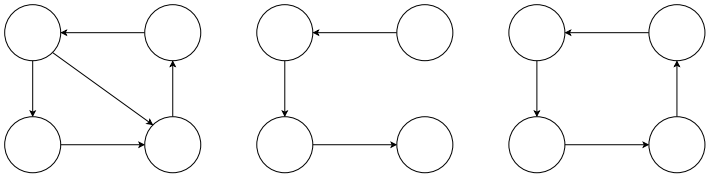


Figura 7: De izquierda a derecha: ejemplos 1, 2 y 3

- Sumando los menores elementos de cada columna obtenemos $22 + 18 + 15 = 55$

Se puede concluir que la solución optima yace en el intervalo $[55, 58]$. En la figura 6 se puede observar la técnica y la solución óptima.

3.4. Ejercicio 4

A continuación se darán ejemplos o por qué no existen tales ejemplos para los siguientes grafos:

1. Grafo con un circuito hamiltoniano pero sin un circuiot euleriano
2. Grafo con un circuito euleriano pero sin un circuito hamiltoniano
3. Grafo con un circuito hamiltoniano y un circuito euleriano
4. Grafo con un ciclo que incluye todos los vértices pero sin un circuito hamiltoniano ni un circuito euleriano

Los 3 primeros casos son posibles, y se visualizan en la figura 7. Sin embargo, el último caso es imposible. Que un grafo contenga un ciclo que incluye todos los vértices ya posibilita, al menos, un circuito hamiltoniano.