Ficha 04

Estrategia Divide y Vencerás
El Teorema Maestro del Análisis de Algoritmos

1.] Introducción.

La tarea de encontrar y plantear un algoritmo que resuelva un problema dado (cosa que debe hacer casi todo el tiempo un programador) requiere conocimientos en el campo de ese problema, pero también paciencia, creatividad, ingenio, disciplina para realizar pruebas, capacidad de autocrítica… y aún así en muchos casos encontrar una solución no es un proceso simple ni obvio. A veces un programador encuentra una solución que parece correcta pero luego descubre fallas o demuestra que la misma no es correcta para todos los casos. Otras veces se encuentra una solución, pero luego se comprueba que la misma es poco eficiente (ya sea porque su tiempo de ejecución es alto o porque ocupa demasiada memoria, por ejemplo). Y muchas veces la solución simplemente no aparece.

En este sentido, a lo largo de los años los investigadores y los especialistas en el área de los algoritmos han estudiado y propuesto diversas técnicas básicas que ayudan a plantear algoritmos para situaciones específicas. Estas técnicas no implican una garantía de éxito, sino simplemente un punto de partida para intentar encontrar una solución frente a un problema particular. Si el programador conoce lo suficiente acerca del problema, y domina los principios de aplicación de cada técnica, posiblemente podrá entonces plantear alguna de ellas para intentar dar con una solución al problema. Nada garantiza que lo logre, y si lo logra, nada garantiza que la solución encontrada sea eficiente, pero incluso así el programador habrá tenido elementos para comenzar a trabajar y al menos poder descartar algunos caminos.

Tradicionalmente, las técnicas o estrategias de planteo de algoritmos que se sugieren como básicas son las siguientes:

- ***Fuerza Bruta***: Consiste en explorar sistemáticamente, una por una, todas y cada una de las posibles combinaciones de solución para el problema dado. A modo de ejemplo, los algoritmos de *ordenamiento simples* (*Seleccción*, *Burbuja* e *Intercambio directo*) son esencialmente algoritmos de *fuerza bruta*. Por lo general, los algoritmos obtenidos por *fuerza bruta* resultan intuitivamente simples de comprender e implementar (de hecho, un algoritmo de fuerza bruta suele ser lo primero que se le ocurre a un programador) pero por otra parte suelen ser muy ineficientes (o bien son muy lentos o el tamaño de la entrada es grande, o bien ocupan demasiados recursos de memoria) ya que la esencia del planteo consiste justamente en no ahorrar pasos.

  En la medida de lo posible, si se tiene un algoritmo de *fuerza bruta* para un problema, el programador debería esforzarse en refinarlo, eliminar elementos de redundancia lógica o agregar elementos que permitan ahorrar trabajo y tratar de obtener una solución mejor. Sin embargo, es notable que para muchos problemas muy estudiados y comunes, sólo se conocen soluciones generales de fuerza bruta, aunque para casos especiales de esos problemas, se aplican estrategias de optimización que permiten mejorar el rendimiento de la solución. Un conocido ejemplo de esta situación, es el famoso *Problema del Viajante* (o del

Vendedor Ambulante): dadas *n* ciudades y las distancias entre ellas, establecer un recorrido que permita pasar una sóla vez por todas ellas pero recorriendo la menor distancia total. La solución trivial de fuerza bruta enumera todos los posibles recorridos y se queda con el más corto, pero eso lleva a un tiempo de ejecución *O(n!)*, que lo vuelve inaplicable si el número de ciudades se hace mayor a 20. Hasta hoy, una de las mejores soluciones se basa en técnicas de *Programación Dinámica* con tiempos de ejecución $O(n^2 2^n)$. No se conoce una solución general más eficiente para todos los casos de este problema, pero se pueden aplicar estrategias que mejoren la situación si se sabe que el número de ciudades es pequeño, o estrategias subóptimas que obtienen soluciones aproximadas aunque no necesariamente óptimas.

- ***Recursión***: La *recursión* o *recursividad* es la propiedad que permite que un proceso se invoque a si mismo una o más veces como parte de la solución. Si se establece que un problema puede ser entendido en base a versiones más pequeñas y manejables de si mismo, entonces un *planteo recursivo* puede ayudar a lograr un algoritmo que normalmente será muy claro para entender e implementar, al costo de utilizar cierta cantidad de memoria adicional en el segmento de stack del computador (y algo de tiempo extra para gestionar el apilamiento en ese stack). En muchos casos, este costo no es aceptable y es preferible plantear un *algoritmo no recursivo* que resuelva el mismo problema. Muchos ejemplos conocidos que se usan para explicar el funcionamiento de la *recursividad* (cálculo del factorial de un número, cálculo de un término de la secuencia de Fibonacci, etc) son justamente casos en los que la *recursión* ***no*** es aconsejable.

  A pesar de los costos extras en uso de memoria y tiempo de ejecución, para algunos problemas de lógica compleja la recursión constituye una herramienta que posibilita el planteo lógico en forma clara y concisa, frente a planteos no recursivos extensos, intrincados y sumamente difíciles de mantener frente a cambios en los requerimientos. Casos así, por ejemplo, se dan en algoritmos para generar gráficas fractales o en situaciones más conocidas como la implementación de algoritmos de inserción y borrado en árboles de búsqueda equilibrados.

- ***Vuelta Atrás (Backtracking)***: Se trata de una técnica que permite explorar en forma incremental un conjunto de potenciales soluciones (*soluciones parciales*) a un problema, de manera que si se detecta que una *solución parcial* no puede ser una solución válida, se la descarta junto a todas las candidatas que podrían haberse propuesto a partir de ella. Típicamente, se implementa mediante *recursión* generando un árbol de invocaciones recursivas en el que cada nodo constituye una solución parcial. Como cada nueva invocación recursiva agrega un nodo en el nivel siguiente del árbol por la inclusión de un nuevo paso simple en el proceso, entonces es fácil ver que si se llega a un punto en el cual se obtiene una solución no válida, puede anularse toda la rama que llevó a esa solución y continuar el análisis en ramas vecinas. Cuando es aplicable, el *backtracking* suele ser más eficiente que la enumeración por *fuerza bruta* de todas las soluciones, ya que con *backtracking* pueden eliminarse muchas soluciones sin tener que analizarlas.

  La estrategia de *backtracking* se usa en problemas que admiten la idea de solución parcial, siempre y cuando se pueda comprobar en forma aceptablemente rápida si una solución parcial es válida o no. Ejemplos muy conocidos de aplicación se dan con el problema de las *Ocho Reinas* (tratar de colocar ocho reinas en un tablero de ajedrez, sin que se ataquen entre ellas), o en el planteo de soluciones para juegos de *Palabras Cruzadas*, o en el planteo de tableros de *Sudoku* o en general, en problemas de *optimización combinatoria*.

- ***Algoritmos Ávidos o Devoradores (Greedy Algorithms)***: Un *algoritmo ávido* es aquel que aplica una regla intuitivamente válida (una *heurística*) en cada paso local del proceso, con la espezanza de obtener finalmente una solución global óptima. Para muchos problemas esta estrategia no produce una soución óptima, pero suele servir de todos modos para intentar plantear soluciones óptima locales, que luego puedan aproximar una solución óptima final en un tiempo aceptable.

  La ventaja de un *algoritmo ávido* (cuando es correcto) es que por lo general lleva a una solución simple de entender, muy directa de implementar y razonablemente eficiente. Pero la desventaja es que como no siempre lleva a una solución correcta, se debe realizar una *demostración de la validez* del algoritmo que podría no ser sencilla de hacer. En general, si se puede demostrar que una *estrategia ávida* es válida para un problema dado, entonces la misma suele producir resultados más eficientes que otras estrategias como la *programación dinámica*.

  Algunos ejemplos de aplicación conocidos en los cuales un *algoritmo greedy* se aplica con éxito son los algoritmos de *Prim* y de *Kruskal* para obtener el árbol de expansión mínimo de un grafo, el algoritmo de *Dijkstra* para obtener el camino más corto entre nodos de un grafo, y el algoritmo de *Huffman* para construir árboles que permitan obtener la codificación binaria óptima para una tabla de símbolos, de forma de reducir el espacio ocupado por un mensaje armado con esos símbolos.

- ***Divide y Vencerás***: Es la estrategia que analizaremos con detalle en esta ficha de estudio. Consiste en tratar de dividir el lote de datos en dos o más subconjuntos de tamaños aproximadamente iguales, procesar cada subconjunto por separado y finalmente unir los resultados para obtener la solución final. Normalmente, se aplica *recursión* para procesar a cada subconjunto y el proceso total podrá ser más o menos eficiente en cuanto a tiempo de ejecución debendiendo de tres factores: la cantidad de invocaciones recursivas que se hagan, el factor de achicamiento del lote de datos (por cuanto se divide al lote en cada pasada) y el tiempo que lleve procesar en forma separada a un subconjunto. Como veremos en esta misma ficha, el *Teorema Maestro del Análisis de Algoritmos* permite obtener una medida asintótica (en notación *Big O*) del tiempo de ejecución en base a esos tres factores. Ejemplos muy conocidos de aplicación exitosa de la técnica de *divide y vencerás*, son los algoritmos *Quicksort* y *Mergesort* para ordenamiento de arreglos.

- ***Programación Dinámica***: Esta técnica sugiere almacenar en una tabla las soluciones obtenidas previamente para los subproblemas que pudiera tener un problema mayor, de forma que cada subproblema se resuelva sólo una vez y luego simplemente se obtengan sus soluciones consultando la tabla si esos subproblemas volvieran a presentarse. Esto tiene mucho sentido: en muchos planteos originalmente basados (por ejemplo) en divide y vencerás, suele ocurrir que al dividir un problema en subproblemas se observe que varios de estos últimos se repiten más de una vez, con la consecuente pérdida de tiempo que implicaría el tener que volver a resolverlos.

  La idea de la programación dinámica es entonces, resolver primero los subproblemas más simples, y luego ir usando esas soluciones hacia arriba para combinarlas y resolver problemas con entradas mayores. Esto es especialmente útil en problemas en los que el número de subproblemas que se repiten crece en forma exponencial. Se aplica en problemas de optimización: encontrar la mejor solución posible entre varias alternativas presentes.

Algunos ejemplos en los que la programación dinámica se aplica con éxito son el algoritmo de *Bellman-Ford* y el algoritmo de *Floyd-Warshall* para encontrar el camino más corto entre dos vértices de un grafo, o los diversos algoritmos conocidos para *alineación de secuencias*, cuyo objetivo es encontrar la mínima cantidad de cambios que deben hacerse en una secuencia de entrada para convertirla en otra.

2.] La estrategia Divide y Vencerás.

La estrategia general de *Divide y Vencerás* propone que el problema estudiado se divida en subproblemas de la misma clase, luego se resuelvan cada uno de estos subproblemas (típicamente en forma recursiva) y finalmente se "unan las partes" resueltas para lograr el resultado final. Muchas veces, y dependiendo de ciertos factores que oportunamente veremos, esta estrategia favorece el diseño de algoritmos muy eficientes en tiempo de ejecución.

Como se dijo, en ciertos casos la estrategia *Divide y Vencerás* puede aplicarse en numerosos problemas muy conocidos para lograr soluciones con mejor o mucho mejor tiempo de ejecución en comparación a soluciones intuitivas de *fuerza bruta*, que suelen consistir en aplicar una y otra vez una idea directa y simple, pero de tal forma que la cantidad de operaciones a realizar aumenta de manera dramática a medida que aumenta la cantidad *n* de datos (o sea, a medida que aumenta el tamaño del problema).

Consideremos a modo de ejemplo introductivo, una situación clásica como es el *ordenamiento* de un arreglo de *n* componentes. Sabemos que existen numerosos algoritmos simples y directos (que terminan siendo de *fuerza bruta*…) para lograr el ordenamiento. Uno de ellos es el conocido algoritmo de *Selección Directa* (que citamos ahora a modo de elemento de comparación): la idea básica es hacer una pasada sobre el arreglo, encontrar el menor valor de los elementos que se recorren y colocar ese valor en la casilla 0. Luego repetir la idea, a partir de la casilla 1 (ya que la 0 está ya ordenada): se recorre el arreglo sin considerar a la casilla 0, se busca el siguiente menor, y se lo traslada a la casilla 1 con lo cual quedarían ordenadas las dos primeras casillas. Se prosigue sistemáticamente así, a lo largo de *n-1* pasadas, ordenando de a una casilla por vez, hasta terminar. El siguiente método implementa ese algoritmo, suponiendo que *v* es el vector a ordenar (conteniendo números enteros) y que está declarado como atributo de la misma clase que contiene al método:

```
public void seleccion()
{
    int n = v.length;
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (v[i] > v[j])
            {
                int aux = v[i];
                v[i] = v[j];
                v[j] = aux;
            }
        }
    }
}
```

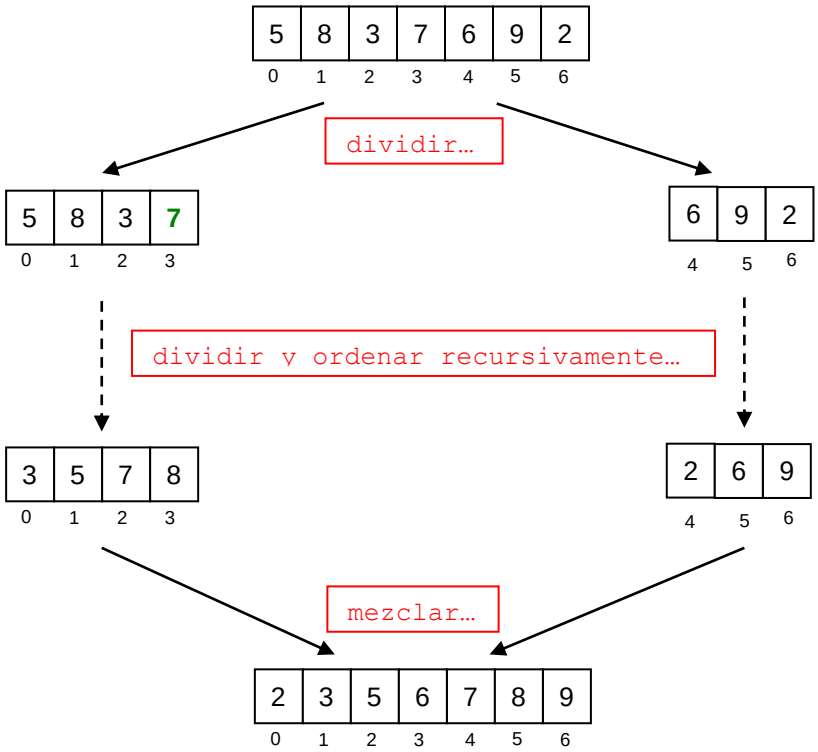
Un rápido análisis de ese algoritmo muestra dos ciclos for anidados que llevan a que la cantidad de comparaciones que se hacen hasta terminar el proceso es $O(n^2)$. La pregunta es: *¿podemos hacerlo mejor?*

El algoritmo de *Selección Directa* es un ejemplo típico de una idea simple, directa y obvia aplicada por *fuerza bruta*: ***todo*** *el trabajo posible se realiza*, lleve el tiempo que lleve, sin intentar un ahorro de tiempo y esfuerzo. Podemos ver que la idea funcionará, aunque será penosamente lenta cuando *n* sea muy grande: a la larga el arreglo efectivamente quedará ordenado pero considere el costo en tiempo de ejecución si *n* es 100000 (cien mil) o 1000000 (un millón)…

Existen muchos otros algoritmos de ordenamiento de tiempo de ejecución cuadrático para el peor caso. Pero como bien sabemos, también existen otros mucho más eficientes, de tiempo de ejecución *subcuadrático* para el peor caso. Hemos analizado a los más conocidos de esos algoritmos en una ficha anterior, y hemos visto que el algoritmo *Quicksort* aplica la idea de *divide y vencerás*. Pero ahora podemos presentar otro algoritmo muy conocido como un nuevo ejemplo de aplicación de la estrategia misma estrategia: el algoritmo de *ordenación por mezcla* o *Mergesort*, que fue planteado originalmente por *John von Neumann* en 1945.

3.] El algoritmo Mergesort para ordenamiento de arreglos.

La idea esencial del *Merge Sort* es dividir el arreglo en dos mitades, ordenar cada mitad por separado, y luego *fusionar* ambas mitades ordenadas de forma de producir que el arreglo final quede ordenado a su vez. El proceso final de *fusión* es el que se conoce en inglés como *merge*. Gráficamente:



La división del arreglo en dos para ordenar cada mitad, puede hacerse recursivamente: cada mitad puede ordenarse volviéndose a dividir y aplicando sobre ella el algoritmo completo de forma de continuar hasta que la partición tomada tenga un solo elemento. El siguiente conjunto de métodos aplica la idea, suponiendo que el arreglo *v* a ordenar está declarado en la misma clase que los métodos. También se supone declarado en esa clase el arreglo auxiliar *temp* usado para copiar al vector original hasta que se haga el proceso de fusión:

```
 // atributos de la clase:
// private int[] v, int[] temp;

public void ordenar()
{
   int n = v.length;
   temp = new int[n];
   sort(0, n - 1);
}

private void sort(int izq, int der)
{
   if(izq < der)
   {
      int centro = (izq + der) / 2;
      sort(izq, centro);
      sort(centro + 1, der);
      merge(izq, centro, der);
   }
}

private void merge(int izq, int centro, int der)
{
   for(int i = izq; i <= der; i++) temp[i] = v[i];

   int i = izq, j = centro + 1, k = izq;
   while(i <= centro && j <= der)
   {
     if(temp[i] <= temp[j])
     {
        v[k] = temp[i];
        i++;
     }
     else
     {
        v[k] = temp[j];
        j++;
     }
     k++;
   }

   while(i <= centro)
   {
      v[k] = temp[i];
      k++;
      i++;
   }
}
```

¿Qué tan bueno es este algoritmo en tiempo de ejecución? Podemos ver que el método *sort()* hace dos llamadas recursivas tomando cada una de ellas la mitad del arreglo. Al terminar esas dos llamadas, se aplica el método *merge()* que hace una *única pasada* sobre el total de los datos y los traslada ordenados al arreglo de salida. Por lo tanto, *merge()* es *O(n)*. Para analizar el algoritmo *sort()*, pongámoslo en un esquema de pseudocódigo:

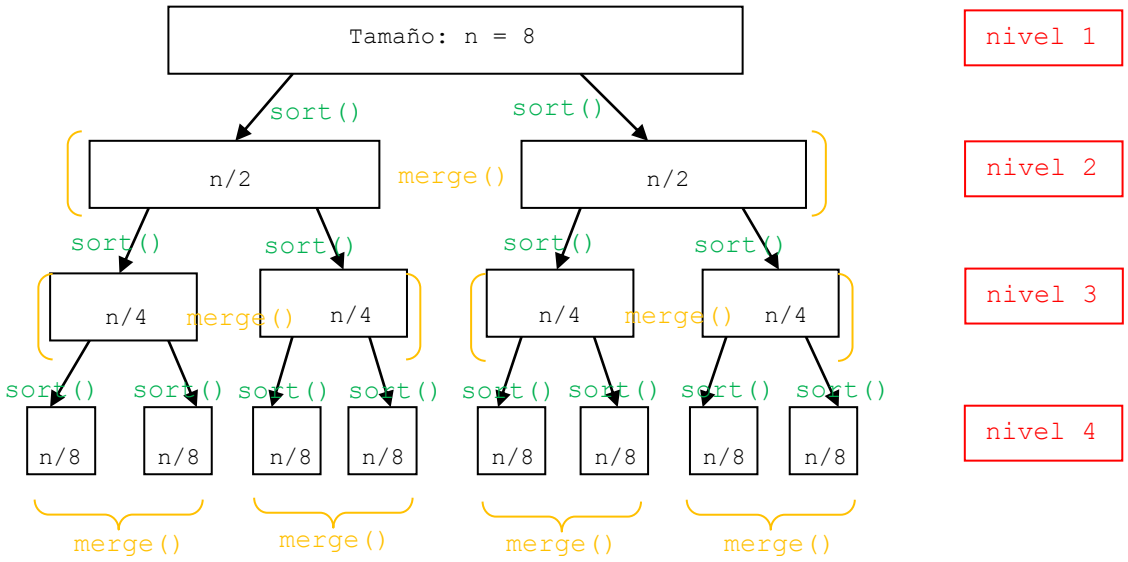
```
sort():
        centro = índice central del subarreglo actual
        sort(mitad izquierda)
        sort(mitad derecha)
        merge() [ ==> tiempo adicional: O(n)]
```

Podemos ver que el proceso consta de dos llamadas recursivas (cuyo tiempo de ejecución o costo ignoramos) más la ejecución de un proceso que en este caso es de *tiempo lineal* (*merge()*). Esto significa que el tiempo total empleado por el *algoritmo completo*, dependerá del tiempo que lleve la ejecución de las dos llamadas recursivas a *sort()*, por lo cual necesitamos averiguar cuántas invocaciones recursivas serán realizadas y cuánto tiempo empleará cada una.

Para ello ayudará un ejemplo. Sea $n = 8$ la cantidad de elementos del arreglo (para facilitar la explicación suponemos que n es una potencia de 2, pero el análisis no cambia para cualquier otro valor). Al ser invocado por primera vez, el método *sort()* hace dos llamadas recursivas (nivel 1 del gráfico siguiente) luego de las cuales *merge()* tiene que fusionar dos subarreglos de tamaño $n/2 = 4$, y el recorrido conjunto de ambos lleva a $4 + 4 = n/2 + n/2 = n = 8$ iteraciones ($O(n)$ en el nivel 1).

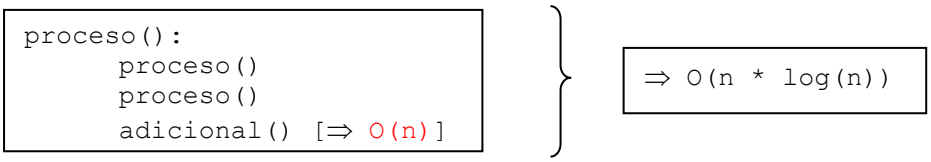
Luego en el nivel 2 se hacen dos llamadas recursivas a *sort()* para *cada subarreglo*, por lo cual finalmente *merge()* tendrá que fusionar primero dos subarreglos de tamaño $n/4$ y luego otros dos de tamaño también $n/4$… por lo cual el recorrido conjunto es $(n/4 + n/4) + (n/4 + n/4) = n$ (y tenemos $O(n)$ también en el nivel 2)

En cada nivel de llamadas recursivas, el tamaño del subarreglo se divide por 2, y aunque es cierto que *merge()* trabaja entonces cada vez menos, el hecho es que *merge()* se invoca varias veces por nivel, con un tiempo de ejecución combinado que siempre es $O(n)$. ¿Cuántos niveles de llamadas a *sort()* tiene el árbol? La respuesta es: $k = 4 = 1 + \log_2(n)$ [o sea: 1 por la primera invocación a *sort()* y luego tantas como se pueda dividir sucesivamente a n por 2 hasta llegar a un cociente de 1 (que es igual a $\log_2(n)$)]. En notación O mayúscula se puede prescindir de las constantes, por lo que la cantidad de niveles con llamadas a *sort()* es $O(\log(n))$.



Por lo tanto, si la cantidad total de niveles de invocación a *sort()* es $O(\log(n))$ y cada nivel tiene un tiempo de ejecución $O(n)$ entonces el tiempo total es $O(n*\log(n))$ lo cual es subcuadrático: puede verse que $n * \log(n) < n * n = n^2$ por lo que $O(n*\log(n)) < O(n^2)$.

¿Qué tan general es este resultado? Hemos aplicado la estrategia *divide y vencerás* para diseñar un algoritmo de ordenamiento, y hemos obtenido un tiempo de ejecución de *O(n * log(n))*… ¿puede inferirse que siempre será así? ¿Toda vez que se aplique la estrategia *divide y vencerás* se obtendrá una solución con tiempo de ejecución *O(n* log(n))*? Probaremos más adelante que la respuesta es *Sí* siempre y cuando el proceso analizado tenga la misma estructura que el proceso *sort()*: dos llamadas recursivas aplicadas sobre lotes de datos de tamaño aproximadamente igual a la mitad del lote actual, más un proceso adicional de tiempo de ejecución lineal (*O(n)*):



Finalmente, observemos que el algoritmo *Mergesort* es efectivamente veloz, pero tiene una contraparte muy desfavorable: utiliza un arreglo auxiliar del mismo tamaño que el que se quiere ordenar, a modo de buffer para ir reteniendo las particiones ordenadas o para ir generando el vector ordenado (según como se implemente). Ese espacio adicional de memoria puede hacer que su aplicación sea prohibitiva en casos de arreglos demasiado grandes, aún cuando existen variantes de implementación que minimizan ese espacio adicional. Sin embargo, el *Merge Sort* es un algoritmo muy usado en cursos de programación y/o diseño de algoritmos pues permite introducir con relativa sencillez al mundo del análisis de algoritmos, la estrategia *divide y vencerás* y la recursividad.

4.] Relaciones de recurrencia.

Hemos visto que la recursividad puede ser una herramienta muy potente para la resolución de ciertos problemas, sobre todo si se combina con alguna estrategia como *divide y vencerás*. Hemos analizado algunos problemas muy conocidos en los cuales la estrategia *divide y vencerás* basada en recursividad da lugar a algoritmos muy eficientes en tiempo de ejecución (particularmente, *Mergesort* y *Quicksort*, de orden subcuadrático).

Sin embargo, el análisis oportunamente efectuado para probar que esos dos algoritmos particulares son *O(n*log(n))* nos mostró dos hechos: el primero es que el análisis propiamente dicho terminó siendo más intuitivo que formal, y el segundo es que el mismo análisis (sin importar si era formal o informal) sirvió para ambos casos.

Una forma de proveer mayor formalismo al análisis (y enfrentar el primer hecho citado en el párrafo anterior) cuando aparece la recursividad en el diseño de un algoritmo, es incorporar las llamadas *relaciones de recurrencia*. Y contando con ellas, podremos mostrar que el segundo hecho antes citado no fue una casualidad, sino que ese resultado podría haberse previsto de acuerdo al *Método Maestro* o *Teorema Maestro* del análisis de algoritmos, que luego introduciremos.

Una *relación de recurrencia* es una ecuación matemática basada en una definición recursiva. La relación de recurrencia define una secuencia recursiva de términos, y cada término de la secuencia es una función de los términos anteriores. En pocas palabras, si podemos plantear una *relación de recurrencia* para la solución de un problema, entonces puede escribirse una solución recursiva para ese problema, y viceversa.

Consideremos (a modo de ejemplo) el siguiente método recursivo para el planteo de la búsqueda binaria (asuma que el arreglo *v* en el cual se quiere buscar, es atributo de la misma clase que el método mostrado, y que el arreglo *v* está convenientemente creado, cargado y ordenado de menor a mayor):

```
public int busquedaBinaria(int x)
{
   return search(0, v.length-1, x);
}

private int search(int izq, int der, int x)
{
   if(izq > der) return -1;

   int c = (der + izq) / 2;
   if(v[c] == x) return c; //O(1)

   if(x > v[c]) return search(c + 1, der, x);
   return search(izq, c - 1, x);
}
```

Intuitivamente, sabemos que el algoritmo de búsqueda binaria es *O(log(n))* para el peor caso, siendo *n* el tamaño del arreglo en el cual se busca: esto surge de ver que el arreglo de parte en dos mitades de igual tamaño, se desecha una y se prosigue con la otra. La cantidad de veces que puede dividirse a *n* por dos, tomar el cociente y seguir dividiendo por dos hasta llegar a un cociente de uno, es igual a *log(n)*.

Pero también podemos hacer un análisis más formal mediante *relaciones de recurrencia*: queremos saber cuántas comparaciones hará nuestro algoritmo en el peor caso. Llamemos *C(n)* a la cantidad de comparaciones que se requiere hacer en el peor caso, dado el arreglo completo de *n* componentes.

Es fácil ver que esa cantidad es igual a 1 (la comparación que debemos hacer contra el valor central) más la cantidad de comparaciones que deberemos hacer en una de las dos mitades (de tamaño n/2). Esto lleva a la siguiente ecuación [a]:

$$C(n) = C(n/2) + 1 \qquad [\text{con } n >= 2 \text{ y } C(1) = 1] \qquad [a]$$

La expresión [a] es una *relación de recurrencia*: el término *C(n)* se calcula como una función de *C(n/2)* más algún término adicional (el valor *1* en este caso). El algoritmo de búsqueda binaria queda formal y completamente descripto (en términos de análisis de cantidad de comparaciones) con esta relación. Y no solo eso: podemos ver que cualquier algoritmo que proceda a realizar un proceso *p* sobre uno solo de los *n* datos, luego parta en dos mitades de igual tamaño al lote inicial de *n* datos, deseche una mitad y repita la idea en la restante, será también caracterizado completamente por esta relación: la misma permite calcular cuántas veces se aplicará el proceso *p* a lo largo de todo el algoritmo en el peor caso (el proceso *p* puede ser cualquiera mientras sea de *orden constante* [*O(1)*]: una comparación [como en nuestro caso], una suma, una salida por dispositivo externo, una concatenación, etc.) En pseudocódigo, el algoritmo recursivo básico de la búsqueda binaria sería entonces:

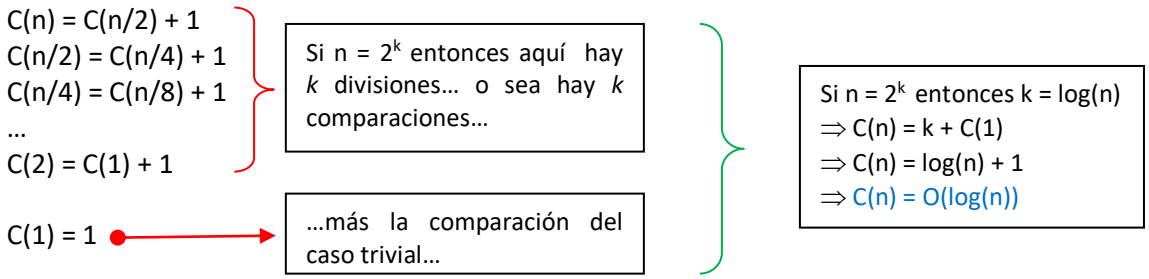
```
search(x):
    si (partición es tamaño 1) retornar -1
    c = indice central
    si (v[c] ==  x) retornar índice central

    si (v[c] > x) retornar search(parte derecha, x)
    else retornar search(parte izquierda, x)
```

Un proceso de tiempo constante: O(1)

Una llamada recursiva (solo una de las dos se activa)

Y bien: cuando el análisis se plantea en términos de una relación de recurrencia, la solución final se puede obtener de varias formas posibles. Una de ellas es la *aplicación en cascada* de la misma fórmula sobre sí misma, hasta llegar al caso trivial (que en la expresión [a] es *C(1) = 1*). Para simplificar, se puede suponer que *n* es potencia de 2 con lo que el proceso de dividir por 2 está siempre bien definido (o sea: $n = 2^k$ para algún k > 0, con lo que $k = log(n)$) y esta suposición no altera el resultado general. Entonces queda:



Este resultado es el mismo que ya habíamos obtenido intuitivamente al analizar en forma directa el algoritmo, pero las relaciones de recurrencia aportan más precisión: en rigor, la cantidad de comparaciones *C(n)* es igual exactamente a *log(n) + 1* en el peor caso.

El uso de recurrencias podria aplicarse también al *Mergesort*: si recordamos, la idea era partir en dos mitades de igual tamaño al arreglo, ordenarlas en forma recursiva, y luego mezclar (merge) las dos partes ordenadas. Eso llevaba a dos invocaciones recursivas y luego un proceso lineal (*O(n)*) para producir el merge:

```
sort(n):
        sort(mitad izquierda) // n/2
        sort(mitad derecha)  // n/2
        merge()   [⇒ O(n)]
```

Este esquema sugiere que se aplican dos llamadas recursivas, cada una sobre la mitad de los datos, y luego un proceso *p* de *tiempo lineal* (*O(n)*). Por lo tanto, si llamamos *T(n)* al tiempo esperado de ejecución para el total de *n* datos, entonces las dos llamadas recursivas insumirán un tiempo total igual a *2T(n/2)*. Usando recurrencias, llegamos a:

$$T(n) = 2T(n/2) + p \quad [n >= 2, T(1) = 1, p = O(n)]$$

Esta recurrencia será obtenida en cuanto algoritmo aplique la misma idea del pseudocódigo dado para *sort()*. Si se aplican recurrencias en cascada (y algún truco adicional…) puede probarse que el resultado será igual a $T(n) = n * log(n) + n$ lo cual implica que $T(n) = O(n*log(n))$.

5.] El Teorema Maestro del Análisis de Algoritmos.

En las secciones anteriores hemos visto diversos problemas que pueden resolverse mediante algoritmos recursivos, en el contexto de la estrategia *divide y vencerás* o en base a alguna variante de esta estrategia. También fue notable que diversos planteos pueden tener básicamente la misma estructura, y que en función de esa estructura pueden obtenerse relaciones de recurrencia de forma general, con la misma solución en cuanto a análisis de tiempo de ejecución o conteo de procesos críticos.

Puede parecer que hemos llegado a un punto en el ya no hay nada que agregar: el planteo recursivo de problemas mediante variantes de divide y vencerás producirá soluciones esquemáticamente similares, y su análisis puede hacerse en base a la expansión y resolución de la respectiva relación de recurrencia. Sin embargo, la situación práctica real es muy diferente…

Numerosos problemas de diversa complejidad pueden aparecer y ser planteados mediante recursividad y relaciones de recurrencia. Pero cada uno puede requerir que la cantidad de llamadas recursivas a realizar sea distinta (en la busqueda binaria es solo una, pero en Mergesort y en Quicksort es dos). El proceso adicional a realizar por fuera de esas llamadas recursivas podría ser de distintos órdenes para el peor caso (en la búsqueda binaria ese proceso era de orden constante [*O(1)*] pero en Mergesort y en Quicksort era de orden lineal [*O(n)*]). Finalmente, los datos podrían dividirse en subconjuntos que no sean necesariamente de tamaño igual a la mitad del lote actual (factor de achicamiento = 2) sino dividirse por tercios (factor de achicamiento = 3) o por cuartos, o por quintos…

Como puede observar el lector, el panorama es bastante más amplio de lo que podría haberse supuesto. Está claro que cada caso que surja de combinar estas tres variables, derivará en una relación de recurrencia diferente, y la misma tendrá una solución distinta que deberá calcularse. Esto supone que el programador / analista de algoritmos deberá eventualmente esforzarse una y otra vez en plantear soluciones matemáticas que no siempre serán obvias ni simples de obtener, con la consecuente pérdida de tiempo implicada.

Una solución sistemática para este problema surgió en 1990 con la publicación de la primera edición del libro *Introduction to Algorithms* de *Cormen*, *Leiserson*, *Rivest*, *Stein* y *Clifford* (aunque estos dos últimos se sumaron como co-autores en la segunda edición de 2001). En ese clásico libro (uno de los más citados de la historia de las ciencias de la computación) se introdujo (con su correspondiente demostración) por primera vez el llamado *Método Maestro* o *Teorema Maestro*, mediante el cual se pueden resolver *asintóticamente* y en forma sistemática, por simple y directa sustitución de variables, diversas relaciones de recurrencia cuando estas se ajustan a determinados patrones. En otras palabras, el *Método Maestro* permite obtener una relación de orden en notación *Big O* en forma directa si se cumplen ciertas premisas en la relación de recurrencia analizada[1].

La mecánica básica de presentación y aplicación del *Teorema Maestro* puede formalizarse así: Si se tiene una relación de recurrencia general ajustada al patrón:

$T(n) = aT(n/b) + O(n^d)$ [con a >= 1, b > 1 y tales que a, b, d NO DEPENDEN de n]

donde:

n = cantidad total de datos a procesar.
a = cantidad de llamadas recursivas (o "subproblemas recursivos")
b = factor de achicamiento del problema (y tal que se supone constante)
d = exponente de *n* en el proceso adicional (y define el orden de ese proceso)

entonces:

[1] La fuente esencial en la que está basada la explicación del *Teorema Maestro* que sigue en esta sección, es el material del curso "*Design and Analysis of Algorithms I*" – Stanford University (a cargo de *Tim Roughgarden*, Associate Professor): https://www.coursera.org/courses.

$$T(n) = \begin{cases} O(n^d * \log(n)) & \text{si } a = b^d \quad [\text{caso 1}] \\ O(n^d) & \text{si } a < b^d \quad [\text{caso 2}] \\ O(n^{\log_b(a)}) & \text{si } a > b^d \quad [\text{caso 3}] \end{cases}$$

Note que en el caso 1 la base del logaritmo no es importante (como se demostró oportunamente), pero en el caso 3 la base del logaritmo ES RELEVANTE, ya que ese logaritmo está en el exponente de *n* y forma parte del cálculo de la constante que justamente dará ese exponente en la expresión *Big O* final.

Así de simple. Así de directo. Y como prueba de la potencia de este método, veamos algunos ejemplos conocidos.

- ✓ *Búsqueda Binaria*: la versión recursiva de este algoritmo llevaba a la recurrencia siguiente:

$$T(n) = T(n/2) + O(1)$$

lo cual es trivialmente lo mismo que:

$$T(n) = 1 * T(n/2) + O(n^0)$$

Con lo que: $a = 1$ $b = 2$ $d = 0$
Entonces: $a = 1$, $b^d = 2^0 = 1$

$\Rightarrow$ $a = b^d$
$\Rightarrow$ caso 1: $T(n) = O(n^d * \log(n))$
$\Rightarrow$ $T(n) = O(n^0 * \log(n))$
$\Rightarrow$ $T(n) = O(\log(n))$ [cosa que coincide con lo que ya sabíamos… ☺]

- ✓ *Merge Sort*: la versión recursiva de este algoritmo llevaba a la recurrencia siguiente:

$$T(n) = 2T(n/2) + O(n)$$

lo cual es trivialmente lo mismo que:

$$T(n) = 2 * T(n/2) + O(n^1)$$

Con lo que: $a = 2$ $b = 2$ $d = 1$
Entonces: $a = 2$, $b^d = 2^1 = 2$

$\Rightarrow$ $a = b^d$
$\Rightarrow$ caso 1: $T(n) = O(n^d * \log(n))$
$\Rightarrow$ $T(n) = O(n^1 * \log(n))$
$\Rightarrow$ $T(n) = O(n * \log(n))$ [¡¡☺☺☺!!]

Note que en el caso promedio, esta misma recurrencia es la que describe al algoritmo *Quick Sort*.

✓ *Multiplicación de números enteros*: El conocido algoritmo de multiplicación de números enteros puede dar lugar a variantes que llevan a interesantes ejemplos[2]. Supongamos (para simplificar el análisis) que se quiere multiplicar dos números *x*, *y* de *n* dígitos cada uno. El algortimo básico de la escuela primaria (si *n = 4*) luce así:

Sean x = 2345 y = 1768

⇒

```
   2345
 *  1768
------
  18760
 140700
+1641500
2345000
------
4145960
```

Cada uno de los *n* dígitos de *y* se multiplica por cada uno de los n dígitos de *x*, lo cual es $O(n^2)$, y luego se hace una suma final que lleva algún tiempo extra no relevante frente a $O(n^2)$.

El proceso completo es $O(n^2)$. Un intento de mejorar el rendimiento de la solución se conoce como *Algoritmo de Karatsuba*, el cual inicialmente consiste en aplicar de alguna forma la estretegia *Divide y Vencerás*. Para ello, el primer paso básico es re-escribir los números *x* e *y* en la forma:

$$x = 10^{n/2}a + b$$
$$y = 10^{n/2}c + d$$

donde *a, b, c, d* son números de *n/2* dígitos cada uno: el número *a* contiene los primeros *n/2* dígitos de *x* y *b* contiene los restantes *n/2* dígitos. Se procede igual con *y* para obtener la siguiente representación:

x = 2345
⇒ n = 4 a = 23 b = 45
⇒ $x = 10^{n/2} * a + b$
⇒ $x = 10^2 * 23 + 45$

y = 1768
⇒ n = 4 c = 17 d = 68
⇒ $y = 10^{n/2} * c + d$
⇒ $y = 10^2 * 17 + 68$

Luego, se plantea la multiplicación de *x* por *y*, quedando:

$$x * y = (10^{n/2} * a + b) * (10^{n/2} * c + d)$$
$$x * y = 10n * ac + 10^{n/2} * (ad + bc) + bd \quad [*]$$

La expresión [*] puede tratarse resolviendo recursivamente los productos *ac*, *ad*, *bc*, *bd* y luego obteniendo en forma normal la suma final (que implica llenar con ceros y sumar dígito

[2] Ejemplo citado en el material del curso "*Design and Analysis of Algorithms I*" – Stanford University (a cargo de *Tim Roughgarden*, Associate Professor): https://www.coursera.org/courses. En estas notas, hemos ampliado el ejemplo y aclarado algunos elementos matemáticos básicos, así como la notación general.

a dígito, lo cual es *O(n)*). Tendríamos así 4 invocaciones recursivas. En cada una, el tamaño de cada número se divide por 2 (factor de achicamiento = 2). Y el proceso adicional por fuera de la recursión sería de orden lineal. Por lo tanto, una relación de recurrencia para esta primera aproximación a la solución de *Karatsuba* sería:

$$T(n) = 4 * T(n/2) + O(n^1)$$

Con lo que para el Método Maestro, quedarían:

$a = 4 \quad b = 2 \quad d = 1$

Entonces: $a = 4, \quad b^d = 2^1 = 2$

$\Rightarrow \quad a > b^d$

$\Rightarrow \quad$ caso 3: $T(n) = O(n^{\log_b(a)})$

$\Rightarrow \quad T(n) = O(n^{\log_2(4)})$

$\Rightarrow \quad$ **$T(n) = O(n^2)$** [¿☹?]

Aparentemente, el esfuerzo no sirvió de nada, ya que finalmente obtuvimos (otra vez…) un tiempo de orden cuadrático. Sin embargo, la versión final del *Algoritmo de Karatsuba* se basa en aplicar una pequeña mejora (a veces conocida como "*truco de Gauss*"): note que en la expresión [*] de la multiplicación:

$$x * y = 10n * ac + 10^{n/2} * (ad + bc) + bd$$

el factor *ac* y el factor *bd* se calculan en forma recursiva (lo que lleva 2 de las 4 invocaciones que se necesitan en total). El cálculo del factor *(ad + bc)* necesitaría las otras 2 (y luego una suma). Sin embargo, podemos ver que si hacemos:

$$(a + b) * (c + d) = ac + bd + ad + bc$$

entonces en la suma del miembro derecho ya tendríamos calculados los valores de *ac* y *bd*, y por lo tanto, para obtener *ad + bc* sólo deberíamos hacer:

$$(a + b) * (c + d) - ac - bd = ad + bc$$

Lo cual incluye 3 multiplicaciones y algunas sumas. El *algoritmo de Karatsuba* (en pseudocódigo) podría entonces verse así:

```
// suponemos que x e y tienen ambos n dígitos, para simplificar…
multiplicar(x, y):
        si (n == 1) retornar x * y
        sea x = 10^(n/2) * a + b
        sea y = 10^(n/2) * c + d
        r1 = multiplicar(a, c)
        r2 = multiplicar(b, d)
        r3 = multiplicar(a+b, c+d) – r1 – r2
        retornar 10^n * r1 + 10^(n/2) * r3 + r2
```

Para el Método Maestro, ahora tenemos 3 invocaciones recursivas (a = 3). El tamaño del problema se achica en un factor de 2 (b = 2). Y el proceso adicional por fuera de la recursión es de orden lineal (d = 1): la última línea del algoritmo calcula una suma de números de aproximadamente *n* dígitos, lo cual implica llenar con ceros y sumar dígito a dígito, en una sola pasada: *O(n)*. Nos hemos ahorrado solo *una* invocación recursiva… ¿qué tan bueno será esto en relación al orden cuadrático que obtuvimos sin aplicar el *truco de Gauss*? Veamos:

La relación de recurrencia final sería ahora de la forma:

$$T(n) = 3 * T(n/2) + O(n^1)$$

Con lo que para el Método Maestro, quedarían:

$a = 3 \quad b = 2 \quad d = 1$

Entonces: $a = 3, \quad b^d = 2^1 = 2$

$\Rightarrow \quad a > b^d$

$\Rightarrow$ caso 3: $T(n) = O(n^{\log_b(a)})$

$\Rightarrow \quad T(n) = O(n^{\log_2(3)})$

$\Rightarrow \quad \mathbf{T(n) = O(n^{1.589})}$ [!☺!] [¡subcuadrático!]

Conclusión: el *algoritmo de Karatsuba* simplemente ahorra una de cuatro posibles invocaciones recursivas, pero con eso logra que un algoritmo cuadrático se convierta en subcuadrático (y a estas alturas, debería el lector comprender que la ganancia en rendimiento es MUY notable…)

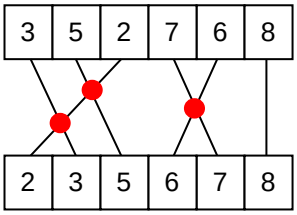
La demostración del *Teorema Maestro* puede analizarse en el libro "*Estructuras de Datos en Java*", de *Mark Allen Weiss* (página 195 y siguientes). Si bien el estudio de la demostración del *Teorema Maestro* no es obligatorio, citamos una fuente consulta para aquellos que deseen ampliar el tema.

6.] Aplicación y caso de análisis.

Para facilitar a los alumnos aplicar lo visto, presentamos un problema ya conocido (fue tratado en una asignatura previa y sirvió como base para una actividad práctica) que tiene alguna relación conceptual con el ordenamiento: el problema del *Conteo de Inversiones*, que se basa en una defición sencilla: dado un arreglo *v* cualquiera de *n* componentes, se designa como una *inversión* a todo par de componentes *(v[i] , v[j])* para el cual se cumpla que *i < j* pero *v[i] > v[j]*. En definitiva, una *inversión* es un par de componentes para el cual el elemento de la izquierda es mayor que el de la derecha. Ejemplo:

Sea v = {3, 5, 2, 7, 6, 8} ⇒ hay 3 inversiones: (3,2), (5,2) y (7,6)

El problema del *Conteo de Inversiones* es entonces, como puede adivinarse, el siguiente: dado un arreglo *v* de *n* componentes, determinar cuántas inversiones tiene *v*. Para el ejemplo anterior la respuesta obvia es 3. Una manera intuitiva de resolverlo consiste en escribir la lista de números tal como está en el arreglo, y debajo escribir la misma lista pero ordenada de menor a mayor. A continuación, trazar líneas que unan los números iguales. La cantidad de puntos de intersección de esas líneas (marcados con círculos rojos en el gráfico) es la cantidad de inversiones:



Resolver este problema tiene algunas motivaciones interesantes. Por lo pronto, ayuda a poder medir o testear la eficiencia de algún algoritmo de ordenamiento, ya que el número de inversiones total de alguna manera indica qué tanto trabajo tendrá que hacer el ordenamiento. Pero puede pensarse en otras aplicaciones menos técnicas: por ejemplo, supongamos que tenemos una lista de ítems ordenados por preferencia (por ejemplo, la lista de películas más vistas, la lista de libros best seller del último mes o la lista de automóviles preferidos por los usuarios) y queremos comprobar qué tan similar es esa lista con respecto a otra (por ejemplo, la lista de mis propias preferencias personales en cuanto a películas, libros o autos). Incluso podríamos querer saber qué tan similares a la "lista rankeada oficial" son varias otras listas. El conteo de inversiones con respecto a la lista rankeada nos dará una buena medida de la similaridad entre las listas comparadas.

Y bien… ¿cómo resolver algorítmicamente el problema del conteo de inversiones? Una primera solución (que es la que se sugirió en la asignatura anterior) trivial, intuitiva, obvia, directa y de fuerza bruta es modificar ligeramente el algoritmo de ordenamiento por *Selección Directa*, de forma que en lugar de ordenar el arreglo, simplemente cuente las inversiones:

```
public long contar()
{
    int n = v.length;
    long c = 0;
    for( int i = 0; i < n - 1; i++ )
    {
        for( int j = i + 1; j < n; j++ )
        {
            if( v[i] > v[j] )
            {
              c++;
            }
        }
    }
    return c;
}
```

Es claro que esta solución funciona: el valor ubicado en la casilla *i* es sucesivamente comparado con *todos y cada uno de los valores ubicados a la derecha de él*, por lo cual cada vez que la condición *(v[i] > v[j])* sea *true*, habremos encontrado una inversión. El problema ya lo sabemos: el tiempo de ejecución de este algoritmo es $O(n^2)$ y resulta prácticamente inaplicable si *n* es grande o muy grande…

¿Podemos hacerlo mejor? Para nosotros, eso equivale a preguntar: *¿Podemos plantear un algoritmo de tiempo subcuadrático?* La respuesta, nuevamente, es sí. Aunque rápidamente nos apuramos a aclarar dos hechos:

i. La solución que propondremos está basada en la estrategia de *divide y vencerás* aplicada sobre la idea del *Mergesort*… por lo cual tendremos un tiempo de ejecución de *O(n*log(n))*. Pero

enfrentaremos el mismo problema que hemos citado para el *Merge Sort*: usaremos al menos un arreglo auxiliar para permitir aplicar la rutina de fusión, con el consecuente espacio extra de memoria.

ii. Desde estas notas expondremos el algoritmo básico y daremos todas las ideas fundamentales, pero la implementación final quedará a cargo de los alumnos… ☺…

Comencemos: la estrategia *divide y vencerás* requiere partir el arreglo en dos subarreglos de la mitad del tamaño original, luego trabajar recursivamente sobre ellas y aplicar un proceso adicional con tiempo *O(n)* (si se quiere un tiempo de ejecución subcuadrático). Está claro que el proceso adicional que debemos desarrollar es el *propio conteo de inversiones*: en el subarreglo actual, debemos aplicar un recorrido que incluya el conteo de inversiones y garantizar para él un tiempo lineal.

Un análisis detallado del panorama que debemos prever, sugiere que al partir el arreglo tendremos entonces que una inversión $v_{i,j}$ puede ser de uno de los tres tipos de inversiones siguientes:

izquierda:	si *i* y *j* están en la primera mitad del arreglo *(i, j <= n/2)*
derecha:	si *i* y *j* están en la segunda mitad del arreglo *(i, j >= n/2)*
split:	si *i* está en la primera mitad y *j* en la segunda *(i <= n/2 <= j)*

Vemos un ejemplo gráfico simple para aproximarnos a la idea: Supongamos un arreglo *v* de *n = 8* componentes, tal como el que sigue:

v = {4, 5, 2, 6, 8, 3, 7, 9}

Partiendo en dos mitades de tamaño *n/2 = 4* a este arreglo, nos quedarían:

subarreglo 1: {4, 5, 2, 6} - subarreglo 2: {8, 3, 7, 9}

Según la clasificación anterior, tenemos:

Inversiones izquierdas:	(4,2) – (5, 2)	total: 2 inversiones
Inversiones derechas:	(8,3) – (8, 7)	total: 2 inversiones
Inversiones split:	(4,3) – (5,3) – (6,3)	total: 3 inversiones

Podemos deducir que las inversiones izquierdas y derechas pueden contarse directamente mediante el mismo proceso recursivo de partir el vector. Pero las inversiones split son un problema especial: si el vector se parte y cada subarreglo se procesa por separado, no tendremos forma de "ver" las inversiones split… Evidentemente, el proceso para contar las inversiones split debe pensarse especialmente. La primera idea del método de conteo (que llamaremos *count()*) a un nivel muy general, podría ser la siguiente:

```
count():
        si (tamaño del arreglo o subarreglo == 1) retornar 0
        sea x = count(mitad izquierda del arreglo o subarreglo)
        sea y = count(mitad derecha del arreglo o subarreglo)
        sea z = countSplitInversions(arreglo o subarreglo completo)
        retornar x + y + z
```

En este esquema de pseudocódigo, el método *count()* cumple el papel que cumplía el método *sort()* en el algoritmo *Mergesort* y presenta dos llamadas recursivas para procesar las mitades izquierda y

derecha del arreglo. La primera llamada debería partir el arreglo y contar las inversiones izquierdas (cosa que hace…) La segunda llamada parte de nuevo el vector, procesa la mitad derecha y cuenta la inversiones derechas. Es el método *countSplitInversions()* el que debe contar las inversiones split en *tiempo lineal*.

La idea para hacer eso es intuitivamente simple, y puede implementarse a partir del algoritmo *Mergesort*. Veamos el mismo ejemplo que pusimos más arriba:

v = {4, 5, 2, 6, 8, 3, 7, 9}

Supongamos que el proceso *countSplitInversions()* además de contar las inversiones, procede a ordenar el arreglo tal como lo hace el método *merge()* en el algoritmo *Merge Sort*. De esta forma, cuando llegue el momento de hacer la fusión final, quedarían dos subarreglos ordenados así:

s1: {2, 4, 5, 6} y s2: {3, 7, 8, 9}

Ahora bien: al hacer la fusión, se recorren ambos subarreglos, se toma el primer valor de cada uno, y se comparan esos valores. El menor de ellos se lleva al vector ordenado y se avanza al siguiente en el subarreglo original: en el ejemplo, se toma el 2 en *s1* y el 3 en *s2*, y como 2 es menor a 3, se lleva al vector de salida:

v = {2 }

Se toma el siguiente en s1, que es el 4, y se compara otra vez con el 3 de *s2*. En este caso, el menor es el 3 y debería llevarse al vector de salida:

v = {2, 3 }

Hasta aquí lo que mostramos es el algoritmo *merge()* tal cual como en *Mergesort*. Pero note que en este último caso, el valor menor (el 3) vino del subarreglo *s2*… lo cual automáticamente implica que hemos detectado una inversión con respecto al 4 de *s1*. Y más aún: como ambos subarreglos están ordenados, en particular está ordenado *s1* y eso también implica que el 3 que venía de *s2* será también menor que *todos los valores que siguen al 4 hasta el final de s1*… por lo que hemos descubierto también todas las inversiones con respecto al 3 sin tener siquiera que hacer las comparaciones. Esto finalmente, es lo que garantiza que no tendremos que hacer que *countSplitInversions()* despliegue un proceso cuadrático para buscar todas las inversiones split: puede hacerlo en una sola pasada sobre ambos subarreglos, obteniendo un tiempo lineal (¡que era lo que buscábamos!)

Conclusión: el algoritmo *countSplitInversions()* solo debe ordenar por fusión el arreglo de salida (tal como lo hace *merge()* en *Mergesort()*), pero además debe prestar atención al momento en que se detecta que un elemento de *s2* es menor que el correspondiente de *s1* y en ese momento sumar al contador general de inversiones el número de elementos que restan en *s1* desde el valor actual hasta el final (en nuestro caso, al detectar que el 3 viene de *s2* y es menor al 4 de *s1*, debemos sumar 3 al contador general de inversiones, ya que comenzando en el 4 hay 3 elementos a partir de allí en *s1*: el 4, el 5 y el 6).

Por lo que ya sabemos de *Mergesort* y la estrategia *Divide y Vencerás*, el tiempo ejecución de *count()* será *O(n * log(n))* si el proceso *countSplitInversions()* es *O(n)*, y hemos visto también que si se programa en base a la idea del párrafo anterior eso estará garantizado. Observemos nuevamente,

que el método propuesto será veloz en comparación al orden cuadrático, pero usará bastante memoria adicional ya que la base de *Mergesort* emplea un vector adicional para hacer la fusión.

Un detalle final: la idea es procesar el vector y *devolverlo tal como fue recibido*, sin modificar su contenido (al fin y al cabo, quien nos de un vector solo para saber cuantas inversiones tiene no esperará que le devolvamos ordenado ese arreglo… y hasta podríamos estar creando serios problemas al modificar la estructura relativa de los datos originales…) Por lo tanto considere seriamente estudiar la forma de salvar el arreglo original y operar sobre una copia del mismo, para luego recuperar el original, o alguna variante que permita volver sobre el original. Y entienda que esto eventualmente usará aún más memoria extra… Los detalles de implementación de este algoritmo, se dejan para el alumno.