UNIVERSITAT DE BARCELONA

Trabajo Final de Grado

Grado en Ingeniería Informática

Facultad de Matemáticas e Informática

Universidad de Barcelona

Herramienta didáctica basada en web de visualización de algoritmos secuenciales, paralelos y basados en árboles

---

Miguel Dominguez Lopez

Tutor: Ricardo Jorge Rodrigues Sepulveda Marques

Realizado en: Departamento de Matemáticas e Informática

Barcelona, 20 de Junio de 2021

Índice

Abstract **3**

Resum **4**

Resumen **5**

Palabras clave **6**

1. Introducción **7**

2. Trabajos relacionados **8**

2.1. Análisis y Conclusión 10

3. Motivación y definición de los objetivos **11**

3.1. Algoritmos de ordenación secuenciales 12

3.1.1. Bubble Sort 12

3.1.2. Insertion Sort 13

3.1.3. Selection Sort 14

3.1.4. Merge Sort 15

3.1.5. Quick Sort 16

3.2. Algoritmos de ordenación paralelos 17

3.2.1. Ordenación Par-Impar 17

3.2.2. Merge Sort Paralelo 18

3.2.3. Quick Sort Paralelo 19

3.3. Algoritmos basados en árboles 20

3.3.1. Binary Search Tree 20

3.3.2. Breadth First Search 21

3.3.3. Depth First Search 22

4. Desarrollo **23**

4.1. Diseño 23

4.2. Lenguajes, librerías y frameworks 26

4.3. Comparación de Librerías JavaScript 28

4.3.1. Conclusión 29

4.4 Características generales de la herramienta de visualización 30

4.4.1 Detalles de implementación de los componentes 30

4.4.2 Funcionalidad “Drag and Drop” 32

4.5. Implementación de la visualización interactiva de los distintos algoritmos 33

4.5.1. Bubble Sort 34

4.5.2. Insertion Sort 36

4.5.3. Selection Sort 37

4.5.4. Merge Sort 38

4.5.5. Quick Sort 40

4.5.6. Ordenación Par-Impar 41

4.4.7. Merge Sort Paralelo 43

4.5.8. Quick Sort Paralelo 45
4.5.9. Binary Search Tree 46
4.4.10. Breadth First Search 49
4.4.11. Depth First Search 51

5. Discusión de los Resultados y Limitaciones 52

6. Conclusiones 54

7. Bibliografía 55

8. Anexo 57

Abstract

According to a survey of computing educators, loops and arrays are two of the three programming topics of major difficulty for novice programming students [2]. In the early phases, the correct understanding of how the algorithms work is important for the development of the student. For this reason, sorting algorithms are presented as educational material. This is the main motivation for which I consider important tools that favor the understanding of these algorithms.

In this project I intend to report the implementation process and the result obtained after creating a web-based didactic visualization tool for sequential and parallel algorithms and tree algorithms. To do this, i will carry out two phases:

- The first phase consists of making a review of the material related to the one I want to program, as well as the tools available for its creation. I will explore existing viewers if there are any and compare programming languages and libraries in order to decide which is the optimal method.
- The second phase will consist of the implementation of the code to create the tool, together with many algorithms to visualize and a template which any user can use to implement their own.

The reason I decided to make my tool web-based is the ease of access it provides. Users positively value the possibility of running the tool easily and quickly from their personal computer [7]. Additionally, the no need to install applications natively is a remarkable feature. As the objective of this project is not only provide an algorithm visualization, but also a template for users who are interested in developing their own visualization, I will use an easy, crear, and generalized structure as possible without using specific frameworks that limit the programming to their language, causing that whoever develops with the documents that I will provide can do so from a didactic spirit using HTML, CSS, JavaScript and simple libraries of these languages.

The tool will have different functionalities so that the user can interact with it and learn in its process. It will be web-based to achieve a familiar and friendly appearence with easy and intuitive navigability.

Resum

Segons els resultats d'una enquesta realitzada per educadors de computació, els vectors i els bucles són dos dels tres temes de programació més complicats segons els estudiants de graus on s'imparteixen assignatures de programació [2]. En les fases inicials, la correcta comprensió del funcionament d'algorismes és important per al desenvolupament de l'estudiant. Per això, els algorismes d'ordenació es presenten com a material didàctic. Aquesta és la principal motivació per la qual considero importants les eines que afavoreixen la comprensió d'aquests algorismes.

En aquest treball pretenc reportar el procés d'implementació i el resultat obtingut després de realitzar una eina didàctica de visualització d'algorismes seqüencials i paral·lels basada en web. Per a això, duré a terme dues fases:

- La primera fase consisteix en realitzar una revisió del material relacionat al que vull programar, així com les eines disponibles per a la seva creació. Exploraré visors existents en cas d'haver-los i compararé llenguatges de programació i llibreries amb la finalitat de decidir quin és el mètode òptim.
- La segona fase consistirà en la implementació del codi per a crear l'eina, juntament amb diversos algorismes a visualitzar i una plantilla perquè l'usuari pugui implementar el seu propi.

EL motiu pel qual decideixo que la meva eina sigui basada en web és la facilitat d'accés que això proporciona. Els usuaris valoren positivament la possibilitat d'executar fàcil i ràpidament l'eina des de l'ordinador de la seva casa [7]. Addicionalment, la no necessitat d'instal·lar aplicacions és una característica a remarcar. Com l'objectiu d'aquest projecte no és solament proporcionar una visualització d'algorismes, sinó també una plantilla per als usuaris que tinguin interès per desenvolupar la seva pròpia visualització, usaré una estructura fàcil, clara i el més generalitzada possible sense comptar amb frameworks específics que limitin la programació al seu llenguatge, amb la finalitat que aquell qui desenvolupi amb els documents que proporcionaré pugui fer-ho des d'una posició amb ànim didàctic usant HTML, CSS, Javascript i llibreries simples d'aquests llenguatges.

L'eina comptarà amb diferents funcionalitats perquè l'usuari pugui interactuar amb ella i aprendre en el seu procés. Serà basada en web per a aconseguir una aparença familiar i amigable, amb una navegabilitat fàcil i intuïtiva.

Resumen

Según los resultados de una encuesta realizada por educadores de computación, los vectores y los bucles son dos de los tres temas de programación más complicados según los estudiantes de grados donde se imparten asignaturas de programación [2]. En las fases iniciales, la correcta comprensión del funcionamiento de algoritmos es importante para el desarrollo del estudiante. Por eso, los algoritmos de ordenación se presentan como material didáctico. Esta es la principal motivación por la que considero importantes las herramientas que favorezcan la comprensión de estos algoritmos.

En este trabajo pretendo reportar el proceso de implementación y el resultado obtenido tras realizar una herramienta didáctica de visualización de algoritmos secuenciales y paralelos basada en web. Para ello, llevaré a cabo dos fases:

- La primera fase consiste en realizar una revisión del material relacionado al que quiero programar, así como las herramientas disponibles para su creación. Exploraré visores existentes en caso de haberlos y compararé lenguajes de programación y librerías con el fin de decidir cuál es el método óptimo.
- La segunda fase consistirá en la implementación del código para crear la herramienta, junto a varios algoritmos a visualizar y una plantilla para que el usuario pueda implementar el suyo propio.

El motivo por el que decido que mi herramienta sea basada en web es la facilidad de acceso que esto proporciona. Los usuarios valoran positivamente la posibilidad de ejecutar fácil y rápidamente la herramienta desde el ordenador de su casa *[7]*. Adicionalmente, la no necesidad de instalar aplicaciones nativamente es una característica a remarcar. Como el objetivo de este proyecto no es solamente proporcionar una visualización de algoritmos, sino también una plantilla para los usuarios que tengan interés en desarrollar su propia visualización, usaré una estructura fácil, clara y lo más generalizada posible sin contar con frameworks específicos que limiten la programación a su lenguaje, con el fin de que aquel quien desarrolle con los documentos que proporcionaré pueda hacerlo desde una posición con ánimo didáctico usando HTML, CSS, JavaScript y librerías simples de estos lenguajes.

La herramienta contará con diferentes funcionalidades para que el usuario pueda interactuar con ella y aprender en su proceso. Será basada en web para conseguir una apariencia familiar y amigable, con una navegabilidad fácil e intuitiva.

Palabras clave

web-based tool, learning, algorithm visualization.

1. Introducción

La etapa estudiantil en la que se aprenden los algoritmos de ordenación es importante para desarrollar la lógica y entender la gestión de datos por elementos como vectores. Aprender el funcionamiento de estos algoritmos no solo aporta el simple comprendimiento de los mismos, sino que generan una dinámica de comprensión adecuada y útil para futuros algoritmos más complejos. Tiene tal relevancia que es sumamente importante la correcta comprensión.

Una buena técnica para realizar un proceso de aprendizaje adecuado consiste en descomponer un proceso en pasos pequeños, claros y bien definidos [3]. Por esta razón, la creación de una herramienta donde se muestre el funcionamiento de ciertos algoritmos con funcionalidades para visualizar el proceso de manera pausada y colorida [8] puede ser de gran ayuda para los usuarios en este proceso de aprendizaje.

Es bien sabido que la tecnología avanza a pasos agigantados, por lo que son cada vez mejores las prestaciones al alcance de cualquiera. Por ese motivo, nuevas técnicas se aplican para mejorar la eficiencia de los procesos, como por ejemplo, las ejecuciones en paralelo. En el caso de los algoritmos de ordenación, hay versiones secuenciales y equivalentes en paralelo de muchos de ellos. Por eso, si ya podían ser de gran ayuda las herramientas de visualización de algoritmos secuenciales, con estas nuevas tendencias considero importante disponer de herramientas que incluyan la visualización de estos algoritmos ejecutados en paralelo. Dada esta premisa, creo apropiado e innovador incluir la visualización de algunos algoritmos de ordenación ejecutados en paralelo equivalentes a algunos de los que haya implementado de secuenciales en mi herramienta. Adicionalmente, para completar la herramienta y aportar más contenido útil, implementaré algoritmos basados en árboles. Esta decisión tiene un componente personal, ya que durante mi recorrido estudiantil, me resultó tediosa la comprensión del funcionamiento de ciertos algoritmos. Personalmente, me hubiera sido de gran ayuda contar con una herramienta en la que poder visualizar estos algoritmos. Consecuentemente, la herramienta deberá ser lo suficientemente flexible como para poder presentar estos tres tipos de algoritmos en un mismo formato.

2. Trabajos relacionados

Uno de los motivos por el que decido realizar este proyecto es por mi voluntad de aumentar mi conocimiento sobre herramientas con esta funcionalidad y contribuir activamente en mejorar las funciones y el aspecto artístico de estas. En este apartado haré una búsqueda y mención de contenido relacionado.

El primer visualizador de algoritmos a mencionar es el *Sorting Algorithms Animations* del portal *toptal.com.* Su zona de visualización tiene la siguiente apariencia:

Figura 1. Visualizador de *toptal.com.*

En la visualización de la Figura 1 los elementos visuales durante la ejecución ayudan a la comprensión. Es útil para comparar algoritmos entre ellos gracias a la opción de ejecutar varios al mismo tiempo, pero no da ningún dato sobre el funcionamiento de estos algoritmos que, para mi herramienta didáctica, considero parte importante.

Seguidamente, mencionaré un portal donde se visualizan todo tipo de algoritmos. Nuevamente una herramienta basada en web siguiendo la tendencia del anterior. Se trata de *Visualgo.* Una herramienta didáctica elaborada por un grupo de más de 10 personas desarrollada y mejorada a lo largo de 6 años para la Universidad de Singapur. Su apariencia es la siguiente:

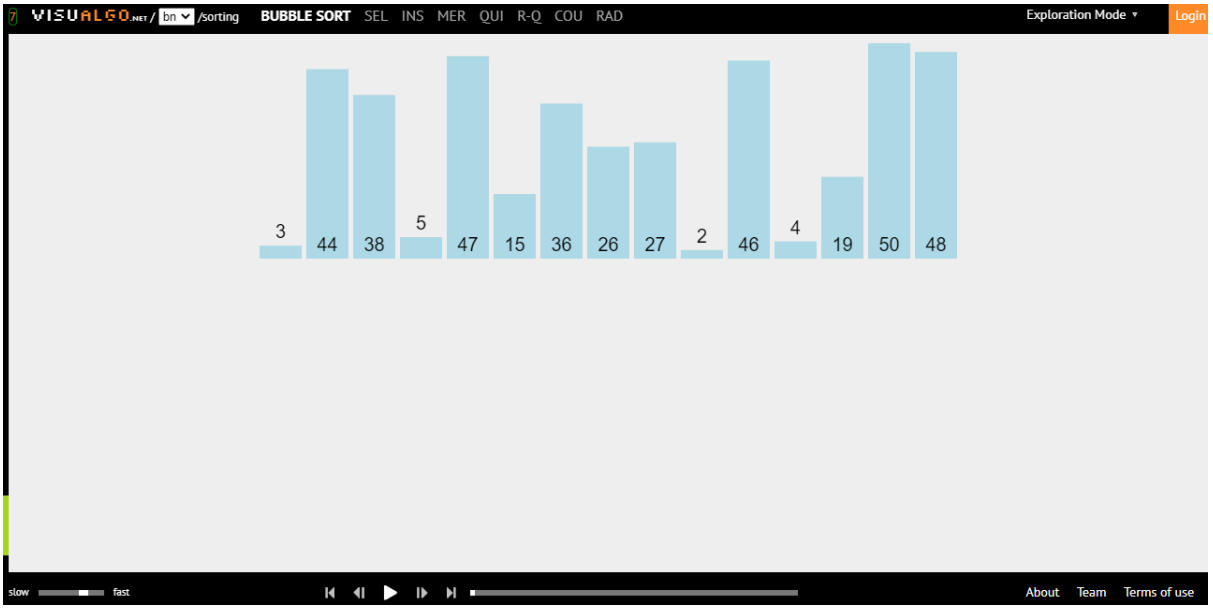


Figura 2. Apariencia de Visualgo.

El visor de la Figura 2 tiene características útiles, las cuales quiero implementar de igual manera en mi herramienta de visualización. Trabaja en formato SVG, por lo que los elementos son dinámicos y tienen movilidad clara y entendedora. Presenta únicamente algoritmos secuenciales, ya que el formato en el que muestra los datos puede ser limitante para mostrar de manera clara procesos en paralelo. Sin duda, será una fuente de inspiración para mi herramienta y un estándar a alcanzar.

Finalmente, un estudio sobre el diseño y la evaluación de un visualizador de algoritmos basado en web llamado DAVE [7] aportó esta herramienta, también con una finalidad didáctica.

Figura 3. Visualización de binary search algorithm en la herramienta DAVE.

Vemos que la herramienta de la Figura 3 donde se visualizan algoritmos tiene una estética pobre y poco entendedora.

2.1. Análisis y Conclusión

Seguidamente, citaré de manera sintetizada las limitaciones principales que tienen estas herramientas web ya existentes.

toptal.com

- Falta de información sobre la ejecución de los algoritmos.
- Poca flexibilidad para escoger valores en los vectores.
- Velocidad alta y sin posibilidad de cambiarla.

Visualgo.com

- Falta de elementos visuales.
- Limitado a idioma inglés.
- Limitación de algoritmos comunes.

DAVE

- Apariencia obsoleta.
- Representación de datos poco clara.
- Estructuras poco entendibles.

Cabe destacar la ausencia de algoritmos paralelos en estas herramientas así como la dificultad de su implementación, lo cual no favorece a ampliarlas programando algoritmos diferentes.

Concluímos con que hay una clara tendencia en realizar las herramientas de visualización de algoritmos basadas en web. Proyectos encontrados de visualización de algoritmos no basados en web tienen aspecto anticuado y funciones escasas, por lo que, mencionadas las ventajas de desarrollar herramientas basadas en web, no plantearé la posibilidad de una implementación que no se destinada a web.

3. Motivación y definición de los objetivos

Vistos los ejemplos citados anteriormente y sus limitaciones, una de mis motivaciones es contribuir activamente en la mejora estética y funcional de las herramientas de visualización actuales con el propósito de enseñar con la muestra de datos y la posible modificación del código a modo de práctica, implementando nuevos algoritmos o visualizaciones. Por eso, mi código estará abierto de manera pública alojado en el portal web Github en la URL: https://github.com/macsimdl/visualizador-de-algoritmos , el cual contendrá un directorio llamado *template* donde se encontrará una plantilla para el desarrollo autónomo de nuevos algoritmos siguiendo la estructura de la herramienta.

Realizando este proyecto quiero demostrar competencias obtenidas durante el grado de ingeniería informática tales como la capacidad de diseñar, desarrollar, evaluar y asegurar la accesibilidad, ergonomía y usabilidad de sistemas, servicios y aplicaciones informáticas, así como de la información que gestionan, conocer las técnicas algorítmicas avanzadas que permiten abordar el desarrollo de programas correctos y eficientes para resolver problemas computacionales no triviales, como por ejemplo, la implementación de algoritmos paralelos escritos de manera secuencial, escoger el paradigma y el lenguaje de programación más adecuado, conocer estrategias algorítmicas que puedan resolver un problema o la capacidad comunicativa dado que es una herramienta didáctica pensada para ser usada por usuarios. Estas competencias las he ido adquiriendo durante mi trayectoria universitaria en asignaturas como Algorítmica Avanzada, Estructura de Datos, Factores Humanos y Computación, Ingeniería del Software, Computación Orientada al Web o Diseño del Software.

3.1. Algoritmos de ordenación secuenciales

Algunos de los algoritmos que describiré a continuación no tienen una única versión. Cada código cuenta con un paradigma propio, pero la manera de ejecutarlo puede cambiar según las preferencias del usuario [11] [12].

3.1.1. Bubble Sort

```
Bubble Sort(lista)
  for i = 0 hasta lista.tamaño - 1
    for j = 0 hasta lista.tamaño - 1 - i
      if(lista[j] > lista[j + 1])
        intercambia(lista[j] y lista[j + 1])
```

Algoritmo 1. Bubble Sort

En el Algoritmo 1 vemos el algoritmo Bubble sort. Tiene una lista como elemento entrante pasada por referencia y seguidamente realiza dos iteraciones anidadas. La primera marcará el número de elementos que se han colocado y la segunda itera por los elementos hasta llegar al límite marcado por i y el tamaño de la lista. Después de una comprobación de cada elemento con el siguiente, los intercambia de posición en caso de ser mayor el primero.

Es un algoritmo simple, pero considerado poco eficiente. Es un buen algoritmo para usarse como material didáctico o en implementaciones donde esté limitado el uso de recursión

3.1.2. Insertion Sort

```
InsertionSort(lista)
    for i = 1 hasta última posición de lista
        j = i
        while j > 0 y lista[j - 1] > lista[j]
            intercambia (lista[j], lista[j - 1])
            j = j - 1
```

Algoritmo 2. Insertion Sort

En el Algoritmo 2 vemos el algoritmo Insertion Sort. Entra por referencia una lista y se itera desde el segundo elemento hasta el último de la lista. Se inicializa una variable j con el valor de i y se decrementa en cada ciclo, en los que en el caso de que el elemento en la posición j sea menor que el elemento en la posición anterior se intercambian sus posiciones.

Este algoritmo también es simple y con complejidad poco eficiente.

3.1.3. Selection Sort

```
SelectionSort(lista)
    for j = 0 hasta última posición de lista
        mínimo = j
        for i = j + 1 hasta última posición de lista
            if (lista[i] < lista[mínimo])
                mínimo = i
        if(mínimo != j)
            intercambiar(lista[j], lista[mínimo])
```

Algoritmo 3. Selection Sort

En el Algoritmo 3 vemos el algoritmo Selection Sort. En este algoritmo tenemos como entrada por referencia una lista y seguidamente una iteración para recorrer todos los elementos de esta. Se inicializa una variable que almacena la posición del elemento mínimo. Después se realiza otra iteración en la que, en caso de haber un elemento menor que el de la posición mínimo, mínimo tomar el valor de la posición del elemento. Finalmente, hace una comprobación sobre mínimo, si ha cambiado el valor con el que se inició, cambia las posiciones del elemento desde el que se empezó y la del actual mínimo.

3.1.4. Merge Sort

```
MergeSort(lista)
   if lista.tamaño = 1
      return lista[0]

   listaIzquierda = crearLista[ lista[0] … lista[mitad] ]
   listaDerecha = crearLista[ lista[mitad + 1] … lista[lista.tamaño] ]
   listaIzquierda = mergeSort(listaIzquierda)
   listaDerecha = mergeSort(listaDerecha)
   return merge(listaIzquierda, listaDerecha)

merge(listaIz, listaDer)
   listaUnida = []

   while (listaIz y listaDer tienen elementos)
      if (listaIz[0] > listaDer[0])
         añadir listaDer[0] al final de listaUnida
         quitar listaDer[0] de listaDer
      else
         añadir listaIz[0] al final de listaUnida
         quitar listaIz[0] de listaIz

   //una lista tiene elementos y la otra no

   while (listaIz tiene elementos)
      añadir listaIz[0] al final de listaUnida
      quitar listaIz[0] de listaIz
   while (listaDer tiene elementos)
      añadir listaDer[0] al final de listaUnida
      quitar listaDer[0] de listaDer


return listaUnida
```

Algoritmo 4. Merge Sort.

En Algoritmo 4 vemos el algoritmo Merge Sort. Entra una lista la cual se retornará en caso de tener un único elemento. Si tiene más de uno, se generarán dos listas, una con los elementos desde el primer índice hasta la mitad y otra con los restantes. A cada lista se le aplicará el método Merge Sort llamándose recursivamente. Finalmente, se hará *merge* de estas listas. La función merge retorna una lista donde, entradas dos listas se les irán restando elementos añadiéndolos a la lista resultante de manera ordenada.

3.1.5. Quick Sort

```
Quicksort( lista, posición menor, posición mayor)
   if (menor < mayor)
      pivote = Partición(lista, menor, mayor)
      Quicksort(lista, menor, pivote)
      Quicksort(lista, pivote + 1, mayor)


Partición(lista, posición menor, posición mayor)
   pivote = lista[menor]
   limiteIzquierda = menor

   for i = menor + 1 hasta mayor
      if (lista[i] < pivote)
         intercambia (lista[i], lista[limiteIzquierda])
         limiteIzquierda = limiteizquierda + 1

   intercambia (lista[posiciónPivote], lista[limiteIzquierda])
   return limiteizquierda
```

Algoritmo 5. Quick Sort

En Algoritmo 5 vemos el algoritmo Quick Sort. Entra una lista, un entero que marca la posición menor y otro entero que marca la posición mayor. En caso de que el entero *menor* sea mayor a *mayor* no realizará el proceso. En caso contrario, primeramente se definirá un pivote con la función Partición, que tiene las mismas entradas que Quick Sort. En esta función se asigna el elemento en la posición *menor* de la lista entrante a un pivote, guardando esta posición en una variable llamada *limiteIzquierda*. Una iteración recorre los elementos desde la posición siguiente a *menor* hasta la posición *mayor,* y va intercambiando la posición de los elementos que sean menores al pivote incrementando el valor de *limiteizquierda* en uno. Finalmente se intercambia la posición donde haya acabado el pivote con la posición de *limiteIzquierda,* de esta manera, resulta en una lista donde, entre las posiciones *menor* y *mayor,* se encuentran en la parte izquierda los elementos menores al que inicialmente se encontraba en primera posición, y en la parte derecha los mayores. Una vez realizada esta función, sigue la ejecución realizando el método QuickSort de nuevo con la lista entre las posiciones de *menor* hasta el pivote, y con los elementos en posiciones mayores al pivote hasta *mayor*.

La versión que vemos en Algoritmo 5 no es la única versión de Quick Sort. Encontramos diferentes versiones de este algoritmo donde el pivote se escoge de manera aleatoria.

3.2. Algoritmos de ordenación paralelos

[6]

3.2.1. Ordenación Par-Impar

```
OrdenaciónParImpar(lista)
   ordenado = false
   while(not ordenado)
      lista intercambios
      hacer en paralelo:
         intercambios[0] = IntercambioCondicional(lista, 0)
         intercambios[2] = IntercambioCondicional(lista, 2)
         (lista.tamaño/2 veces)
      hacer en paralelo:
         intercambios[1] = IntercambioCondicional(lista, 1)
         intercambios[3] = IntercambioCondicional(lista, 3)
         (lista.tamaño/2 veces)
      if (todosFalse(intercambios))
         ordenado = true;

IntercambioCondicional(lista, i)
   if (lista[i] > lista[i+1])
      intercambia(lista[i], lista[i+1])
      return true
   return false
```

Algoritmo 6. Ordenación Par Impar.

En el Algoritmo 6 vemos el algoritmo de ordenación par impar en el que entra una lista, se inicializa una variable booleana con valor falso y mientras no tome valor de verdadera se ejecuta un bucle. En este, de manera paralela se escogen los elementos en posiciones pares y se comparan con el siguiente, en el caso de que el primero sea mayor se intercambian sus posiciones. Seguidamente se hace lo mismo con las posiciones impares. Cuando se intercambian dos posiciones se devuelve true y se almacena en un vector de valores booleanos. Si este vector tiene algún valor true, la lista puede aún no estar ordenada, en caso contrario se considera la lista ordenada y el proceso se acaba.

3.2.2. Merge Sort Paralelo

```
ParallelMergeSort(lista)
    if lista.tamaño = 1
        return lista[0]

    listaIzquierda = crearLista[ lista[0] … lista[mitad] ]
    listaDerecha = crearLista[ lista[mitad + 1] … lista[lista.tamaño] ]
    hacer en paralelo:
        listaIzquierda = mergeSort(listaIzquierda)
        listaDerecha = mergeSort(listaDerecha)
    return mergeParalelo(listaIzquierda, listaDerecha)

merge(listaIz, listaDer)
    listaUnida = []
    hacer en paralelo:
        añadir elementos de listaIz y listaDer a listaUnida
    return listaUnida
```

Algoritmo 7. Merge Sort Paralelo.

En Algoritmo 7 vemos el algoritmo de Merge Sort Paralelo. Entra una lista y en caso de no tener más de un elemento, se retorna sin modificar. En caso contrario se crean dos listas, una con los elementos hasta la posición mitad y otra con los restantes. De manera paralela, se realiza el método Merge Sort Paralelo en cada lista. En cuanto se han separado los elementos en grupos de un elemento se hace *merge* recursivamente de las listas de manera paralela. Esto es posible gracias a un método que calcula la posición final de los elementos de cada lista. [9] Por ejemplo, Teniendo las dos listas a unir A y B:

Lista A = [1, 3, 12, 28]
Lista B = [2, 10, 15, 21]

Cada posición, mediante binary search por ejemplo, determina en qué posición estaría en la otra lista y la suma a la posición en la que está. El elemento 12 de la Lista A, está en la posición 2, en la Lista B tomaría la posición 2 de igual manera. Así que en la lista ordenada resultante estará en la posición 4 (2+2)

[1, 3, 12, 28] [2, 10, 15, 21]
 0 2 4   7   1 3   5   6

[1, 2, 10, 12, 15, 21, 28]

3.2.3. Quick Sort Paralelo

```
QuickSortParalelo(lista)
    if (lista.tamaño == 1) return lista[0]
    pivote = posición aleatoria de lista
    listas = listasParalelo(lista,pivote)
    listaIzquierda = listas[0]
    listaDerecha = listas[1]
    hacer en paralelo:
        listaIzquierda = QuickSortParalelo(listaIzquierda)
        listaIzquierda = QuickSortParalelo(listaDerecha)
    return listaIzquierda + listaDerecha

listasParalelo(lista,pivote)
    listaDeListas listas
    hacer en paralelo:
        añadir elementos de lista <= pivote a listas[0]
        añadir elementos de lista > pivote a listas[1]
    return listas
```

Algoritmo 8. Quick Sort Paralelo

En el Algoritmo 8 vemos el algoritmo Quick Sort Paralelo en el que nuevamente entra una lista y en caso de tener únicamente un elemento no realiza ningún cambio a esta. En caso contrario se escogerá un pivote de manera aleatoria y se ejecutará la función listasParalelo donde entra la lista y el pivote seleccionado. De manera paralela se crean dos listas, una en la que se añaden los elementos menores o igual al pivote y otra en la que se añaden los restantes. Una vez retornadas estas listas se llama a la función Quick Sort Paralelo por cada lista. Cuando se han separado los elementos en listas de un único elemento se unen de manera recursiva.

La función listasParalelo que vemos en Algoritmo 8, es posible hacerlo en paralelo gracias a un método en el que se escanean los elementos de la lista marcando con un flag(1 o 0) los elementos menores y mayores al pivote, después sumando a lo largo del vector de flags. [9] Por ejemplo, teniendo una lista entrante que se quiere separar en dos:

[3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 9] pivote = 3, menores a pivote = 1, mayores = 0
[1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0]
[1, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5] número menores o igual a pivote 5, el último número

La lista resultante tendrá tamaño 5, correspondiente al último elemento. Cada cambio de valor corresponde a un elemento menor o igual al pivote. El valor determina su posición en el vector.

3.3. Algoritmos basados en árboles

[4]

3.3.1. Binary Search Tree

```
BST(árbol, valor)
  actual = árbol.padre
  while(true)
    if(valor == actual)
      return true
    else if(valor < actual)
      if(actual.hijoIzquierdo existe)
        actual = actual.hijoIzquierdo
      else
        return false
    else
      if(actual.hijoDerecho existe)
        actual = actual.hijoDerecho
      else
        return false
```

Algoritmo 9. Árbol Binario de Búsqueda.

En el Algoritmo 9 vemos el algoritmo de Árbol Binario de Búsqueda en el que entra un árbol y un valor que desea buscar. Empezando desde el nodo padre, se entra en un bucle infinito. Si el valor que buscamos es igual al nodo actual (nodo padre), se retorna verdadero. En caso de ser menor, se comprueba si el nodo actual tiene hijo Izquierdo, si lo tiene el nodo actual será este hijo Izquierdo. En caso de ser mayor se realizará el mismo proceso que con el hijo izquierdo pero esta vez con el hijo derecho.

3.3.2. Breadth First Search

```
BFS(árbol)
   visitados.añadir(árbol.primerNodo)
   resultado.añadir(arbol.primerNodo)
   continua = true
   while(continua)
      if(visitados[0].hijoizquierdo existe)
         visitados.añadir(visitadoes[0].hijoIzquierdo)
         resultado.añadir(visitados [0].hijoIzquierdo)
      if(visitado[0].hijoDerecho existe)
         visitados.añadir(visitadoes[0].hijoDerecho)
         resultado.añadir(visitados [0].hijoDerecho)
      visitados.eliminaPrimerElemento
      if(visitados.vacío)
         continua = false
   return resultado
```

Algoritmo 10. BFS.

En el Algoritmo 10 vemos el algoritmo BFS, en el que entrado un árbol se crean dos vectores añadiendo el nodo padre y se inicializa una variable booleana que marcará si el bucle siguiente debe seguir o no. En este bucle se comprueba primero si el primer elemento en el vector de visitados tiene el hijo izquierdo, en caso de tenerlo lo añadirá al final del vector de resultados y de visitados. Hace el mismo proceso con el hijo derecho. Finalmente, elimina el primer elemento del vector de visitados explorando el siguiente. Cuando este se queda sin elementos, significa que se han explorado todos los elementos. Se retorna el vector de resultados.

3.3.3. Depth First Search

```
DFS(árbol)
   visitados.añadir(árbol.primerNodo)
   if(visitados[0].hijoIzquierdo existe)
      visitados.añadir( DFS(subArbol(arbol.primerNodo.hijoIzquierdo)) )

   if(visitados[0].hijoDerecho existe)
      visitados.añadir( DFS(subArbol(arbol.primerNodo.hijoDerecho)) )

   return visitados
```

Algoritmo 11. DFS

En el Algoritmo 11 vemos el algoritmo Depth First Search el cual entra un árbol e inicializa una lista con el primer nodo de este. Si el primer nodo tiene nodo hijo izquierdo, se añade a la lista el subárbol generado a partir del hijo izquierdo como nodo padre. El mismo proceso se aplica con el hijo derecho.

4. Desarrollo

4.1. Diseño

Como hemos visto en los diseños anteriormente mencionados, la estética es un factor muy importante y con gran impacto en una labor didáctica. Tanto en un entorno analógico como en un entorno digital, los elementos artísticos son de gran ayuda para explicar algoritmos de ordenación a usuarios iniciados en el campo de la algoritmia *[8]*. Por este motivo, es prioritario añadir elementos con claros componentes visuales (colores, formas…) en el diseño de la herramienta.

Primeramente, al ser un instrumento de visualización basado en web, debe contar con elementos básicos presentes en un diseño estándar de una página para otorgarle funcionalidades útiles para su navegabilidad y uso.

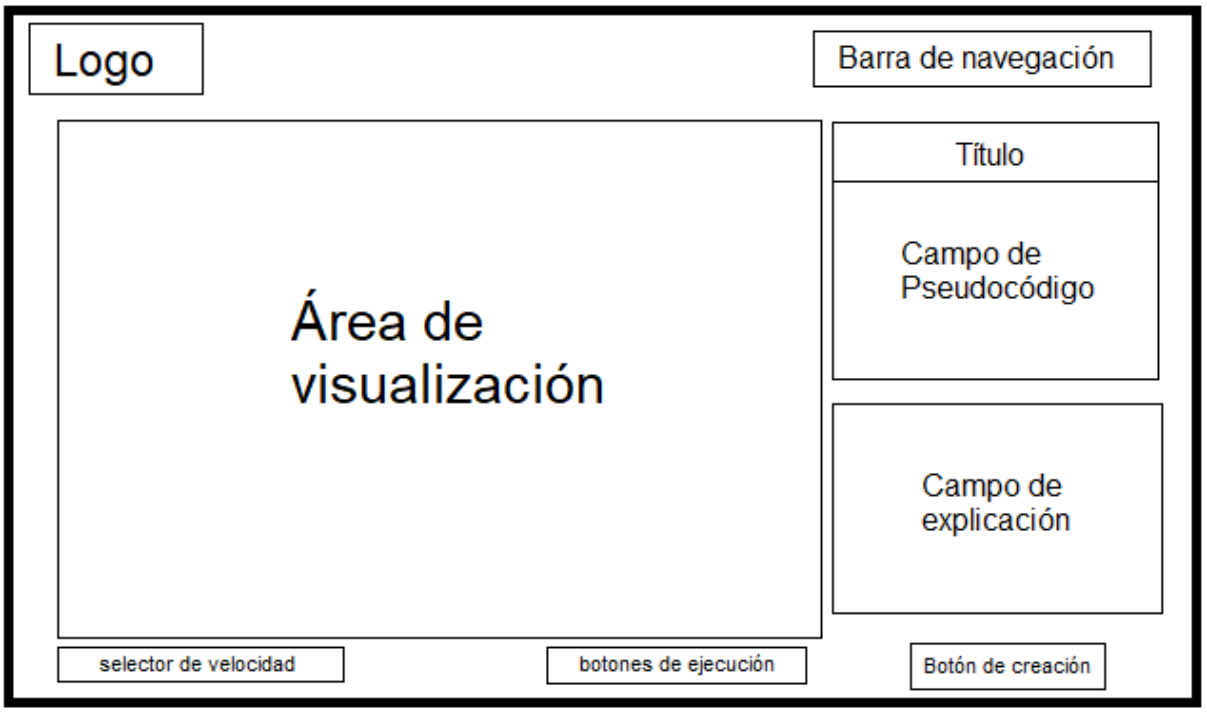


Figura 4. Diseño principal de la herramienta.

En la Figura 4 podemos identificar distintos componentes:

- Logo: Otorga tanto la identidad como la exclusividad del portal. Cuenta con una fuente con estilo pixelado comunicando indirectamente la temática que se trata. Como vemos en la Figura 4, se encuentra en el extremo superior izquierdo de la herramienta.
- Barra de navegación: Esencial para navegar entre los diferentes menús y explorar los distintos algoritmos. Cuenta con un desplegable en cada apartado, los cuales diferencian los tres principales grupos de algoritmos a visualizar. Lo vemos en la parte superior derecha.
- Título del algoritmo: Presenta e indica en qué menú te encuentras en cada momento. Se encuentra presente en la parte derecha de la ventana.
- Campo de pseudocódigo: Una zona donde se presenta el pseudocódigo del algoritmo visualizado. Cuenta con una funcionalidad la cual destaca la/s

línea/s de código que se estaría/n ejecutando en una ejecución normal. Lo vemos debajo del título como vemos en la Figura 4.

- Campo de explicación: En la parte inferior del campo de pseudocódigo, un diálogo explicativo donde encontramos unas indicaciones del funcionamiento del algoritmo en cada fase de ejecución.
- Botón de creación: En los algoritmos que lo permiten, otorga la posibilidad de insertar elementos de manera manual separados por comas para generar un vector con elementos elegidos por el usuario.
- Botones de ejecución: Realizan las funciones de “reproducir/detener”, “pausar y reiniciar” y “avanzar un paso”. Cuentan con un icono indicativo de tu función en lugar de texto reduciendo la fatiga visual. Su diseño es altamente reconocido asociando una funcionalidad a este.
- Selector de velocidad: Un deslizador que aumentará o disminuirá la velocidad de ejecución según su posición.
- Área de visualización: El espacio más amplio en pantalla, desplazado ligeramente a la izquierda, donde se muestran los elementos representativos del algoritmo a visualizar.

Todos los elementos deben estar distribuidos por la pantalla según su prioridad. Tienen una proporción acorde a una estética moderna, beneficiando la visualización del contenido dibujado en el centro. De igual manera, la selección de colores no ha sido aleatória. El tono predominantemente oscuro sigue la tendencia y adaptación actual de muchas aplicaciones, programas y webs al conocido “night mode” (modo noche) o “dark mode” (modo oscuro), el cual oscurece la pantalla para reducir la fatiga visual y la exposición a luz azul, la cual puede causar sequedad ocular, molestia, dolor de cabeza, modifica el ritmo circadiano y perjudica la calidad del sueño en casos de exposición prolongada *[16]*. Concretamente, el color escogido es el RGB(34, 17, 69) para el color del contorno, presente también en un margen en la parte izquierda. En contraste, los textos, iconos y líneas sobre este, serán de color blanco (con el componente rojo y verde sin maximizar para darle un ligero tono azulado). De esta tonalidad clara también será el fondo del área de visualización, donde se encontrarán los elementos inicialmente con un tono más oscuro que el fondo con la finalidad de destacar y ser fácilmente reconocibles. A medida que vaya avanzando la ejecución, estos irán actualizando su color para aportar información visual a la ejecución del algoritmo.

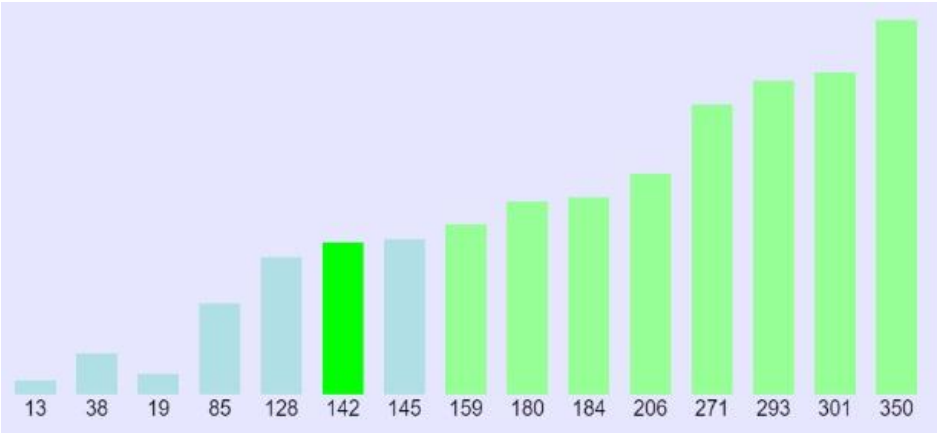


Figura 5. Elementos del área de visualización coloreados.

Según el algoritmo, los elementos usarán los colores de la figura para marcar su estado:

- RGB(176, 224, 230): Marcar elementos sin evaluar.
- RGB(0, 255 ,0): Marcar elementos siendo modificados actualmente.
- RGB(150, 255, 150): Elementos ya evaluados y modificados.

En la medida de lo posible, todos los algoritmos contarán con esta paleta de colores para los distintos estados de sus elementos visuales para mantener una coherencia y una estética lineal.

En cuanto al área de pseudocódigo, presenta un color de fondo amarillo oscuro RGB(240, 230, 140) para hacer que destaque respecto al resto de elementos dada su importancia en cuanto a entender el código.

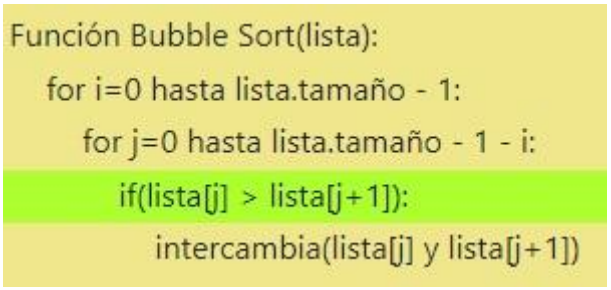


Figura 6. Área de pseudocódigo coloreada.

A lo largo de la ejecución, se irán destacando líneas del pseudocódigo para llevar un control de la instrucción ejecutándose. En ese momento, la zona donde se encuentre el texto de esta instrucción se señalará con el color verde dado por RGB(173, 255, 47) como vemos en la Figura 6, dejando claramente una zona destacada.

Es destacable el amplio surtido de dispositivos al alcance de todos los usuarios, y con ellos diferentes resoluciones de visualización. Para afrontar este requisito, la aplicación debe contar con un diseño *responsive [14]*. Para ello, los elementos estarán situados en la pantalla porcentualmente al tamaño de la ventana. De esta manera no variarán las proporciones dadas diferentes resoluciones.

```
#lefbar{
        width: 5%;
        background-color: #214;
        height: 100%;
}
```

Aunque la herramienta no es compatible para resoluciones tan pequeñas como la de dispositivos móviles, cubre las resoluciones más comunes con las que funcionan los ordenadores a día de hoy. La funcionalidad ha sido probada en dispositivos de resolución 1366x768 y de 1920x1080.

4.2. Lenguajes, librerías y frameworks

Antes de justificar el lenguaje escogido, debo justificar la decisión de la plataforma a la que va dirigida la herramienta. Inicialmente, planteé la posibilidad de desarrollar una herramienta ejecutable basada en Java o en C, pero no la veía suficientemente versátil. Dado el avance tecnológico y el alcance de dispositivos compatibles con tecnología web vi adecuado desarrollar la aplicación basada en web, por lo tanto, con una estructura HTML. Son numerosas las ventajas que aporta esta arquitectura: entre otras, los usuarios valoran positivamente la posibilidad de ejecutar fácil y rápidamente la herramienta desde el ordenador de su casa *[7].* Aunque esta herramienta no haya sido publicada, al estar desarrollada basada en web, existe la posibilidad de publicarla en línea en un futuro para poder ser usada con un simple browser. Mientras tanto se encontrará alojada en el portal Github en la URL: https://github.com/macsimdl/visualizador-de-algoritmos .

Dado el ánimo didáctico de la herramienta y la posibilidad de crear visualizaciones propias del usuario, debe estar desarrollada en un lenguaje fácil de entender por el usuario. Por este motivo, aunque existen frameworks que pueden facilitarme la tarea de desarrollo de la herramienta, no contaré con su ayuda. Asumo que el modelo de usuario para el que puede ser útil mi herramienta, dado que usa la herramienta para aprender sobre el funcionamiento de estos algoritmos, a priori, sin un elevado grado de complejidad, asumo que no tiene un manejo del lenguaje de programación destinado a web muy avanzado. Por lo tanto, me parece una decisión acertada no usar frameworks tales como Vue.js, React, Angular, o Node.js.

Acompañando la estructura HTML, está presente una hoja de estilo CSS para otorgarle la apariencia que previamente mencioné junto a los colores.

Por otro lado, para incorporar elementos mencionados en el apartado anterior como los botones es recomendable usar el framework de JavaScript *Bootstrap.* Mediante sus componentes y plantillas, este nos proporciona elementos dinámicos fácilmente aplicables y modificables para diseños óptimos. El elemento a destacar por el que catalogo Bootstrap como imprescindible es la barra de navegación superior. Esta presenta un espacio para un Logo clicable y unos desplegables con botones para seleccionar el menú al que queremos acceder. Estos componentes se incorporan fácilmente al código HTML. Debo mencionar también el set de iconos *FontAwesome* el cual proporciona una colección de iconos muy amplia, de la que doy uso a los botones de la barra de botones de ejecución.

Al ser una herramienta didáctica, he mencionado la importancia de su apariencia. También quiero mencionar la efectividad a la hora de aprender que tienen las herramientas con apariencias de juegos y con elementos divertidos:

> *"The trend of employing game mechanisms and techniques in non-game contexts, gamification, has dramatically increased in recent years. Gamification can be viewed as a new paradigm for enhancing brand awareness and loyalty, innovation, and online user engagement."[15]*

Para añadir esta técnica, los elementos en la zona de visualización deben tener un aspecto dinámico, deben ser móviles y cambiantes. Consecuentemente, usaremos JavaScript y, en concreto, la librería p5.js. De esta manera, en cada menú constará un fichero HTML estilizado por una hoja de estilo CSS que reproducirá un script escrito en JavaScript. De esta manera, gran parte del código de cada fichero puede ser reutilizado para cada algoritmo. Este código común será el usado para crear el template para el desarrollo autónomo de una visualización propia, estará adjunto con el código. El menú principal que se mostrará como portada será el del algoritmo Bubble Sort, ya que la herramienta no pretende tener la misma navegabilidad propia de una web y carece de un menú principal selector o un menú inicial explicativo.

4.3. Comparación de Librerías JavaScript

JavaScript es un lenguaje que cuenta con una gran cantidad de librerías diferentes que ofrecen todo tipo de funcionalidades y características. Desde recursos que manipulan tanto la estructura de la página como su estilo visual de forma dinámica hasta transiciones y manipulaciones de la apariencia de elementos, las librerías son esenciales para desarrollar una página con las características propias que se exigen a día de hoy. Para escoger las librerías que quiero usar debo conocer las funcionalidades que ofrece cada una y las características que requiere mi herramienta. Primeramente, haré un cribado previo con las posibles candidatas que puedan ajustarse al propósito que busco separándolas por categoría, por este motivo, debo detectar las categorías que se adecuen a mis necesidades.

Mi herramienta se centra en la visualización de algoritmos de ordenación, los cuales modifican los datos de uno o más vectores. Para representar el vector, inspirado por la revisión de los trabajos relacionados citados anteriormente, usaré formas rectangulares para representar cada uno de sus elemento con el valor que le pertoque. Al ser una apariencia parecida a la de un gráfico de barras, haré una revisión de bibliotecas centradas en la representación de gráficos:

- *Highcharts*: Permite crear gráficas interactivas. Una API de JavaScript con una gran cantidad de funciones y gráficos diferentes.
- *Morris.js*: Ofrece una Api para renderizar gráficos de barras, entre otros. Es una de las librerías más empleadas para la representación de gráficos.
- *Google Charts*: Ofrece gran variedad de gráficos y documentación al respecto, con una apariencia minimalista y limpia.

Por otro lado, al querer conseguir una apariencia dinámica y animada, también revisaré librerías con funcionalidades de *motion graphics* para aplicar a los elementos a visualizar:

- *Mo.js*: Ofrece animaciones sorprendentes. Ofrece una herramienta con una línea de tiempo para modificar las animaciones fácilmente. Cuenta con numerosos tutoriales para aprender sobre su funcionamiento.
- *Anime.js*: Una de las librerías mejor valoradas en el campo de la animación en JavaScript. Usada desde una única y potente API, se usa para animar elementos HTML, CSS, JavaScript, entre otros.

Por último, la zona de visualizado puede presentar figuras dibujadas sobre un lienzo para conseguir visualizaciones y elementos más flexibles. Para ello, haré una revisión de las bibliotecas más adecuadas para mi caso:

- *Fabric.js*: Una librería JavaScript que funciona sobre un canvas para modelar objetos de manera interactiva.
- *D3*: Proporciona la posibilidad de representar datos de manera visual usando HTML, SVG y CSS. Una de las más populares en esta categoría.
- *P5.js*: Pretende ser accesible para artistas, diseñadores y educadores. Tiene buenas referencias y documentación. Proporciona un canvas fácilmente modificable.

4.3.1. Conclusión

Una vez reunidas las librerías que considero más adecuadas para mi proyecto, comparo sus funcionalidades para concluir con qué opción usaré.

Todas las librerías mencionadas pueden aportar características útiles a mi herramienta, pero después de probar las que pertenecen a la categoría relacionada con gráficos, veo que no son suficientemente flexibles como para presentar los datos de manera clara y entendedora. Aunque proporcionan plantillas y herramientas para crear gráficos de barras semejantes a la apariencia objetivo a la que quiero llegar, su estilo puede resultar confuso.

Respecto a la siguiente categoría; animación, consigo un resultado muy visual y satisfactorio, sobre todo con Anime.js, pero la modificación de elementos visuales dependen de demoras temporales que deberán aplicarse a todos los rectángulos creando una cola de eventos que se ejecutarán en orden. Esto dificulta la actualización dinámica de los mismos. La parametrización de variables para su correcto funcionamiento es tediosa y artificiosa, por lo cual optaré por la última categoría mencionada: las librerías relacionadas con el dibujo.

Después de probar las tres listadas anteriormente de esta índole, la librería que más se amolda a mis necesidades es P5.js. Dadas sus instrucciones Intuitivas y sus componentes fácilmente parametrizables, es una opción válida para el desarrollo. Consiste en una función *setup()* en la que se inicializan las variables y toman valor. Seguidamente, se ejecutará la función *draw()* de manera cíclica a la velocidad que se haya definido con la función *frameRate(int).* Esta librería nos proporciona formas 2D y modificadores de apariencia para generar escenarios a voluntad.

Como es una librería para pintar sobre un canvas que funciona con el bucle *draw()*, se deberán modificar todos los algoritmos que se deseen implementar para que funcionen teniendo una estructura secuencial dentro de un bucle sin contar con recursión.

El modo de empleo que mantendremos a en todos los menús consistirá en:

- Un fichero HTML con el nombre del algoritmo en el que nos encontramos.
- Un fichero CSS personalizado para cada menú.
- Un fichero JS que cuenta con el script con el que se dibuja.

Para añadir el lienzo en el que se dibuja a la estructura HTML del proyecto, simplemente debemos añadir el apartado <main></main> en la sección que queramos dentro del *body.*

4.4 Características generales de la herramienta de visualización

Cada elemento que he implementado en la herramienta interactúa directamente con el script de P5.js creando las funcionalidades deseadas en un principio. De igual manera, he implementado otras características para ser usadas sobre el lienzo de visualización sin necesitar un elemento extra que la haga funcionar.

4.4.1 Detalles de implementación de los componentes

Son varios los componentes presentes en la herramienta, como vemos en la Figura 4, y cada uno tiene su propia implementación para su correcto funcionamiento. Muchos de ellos interactúan y actualizan valores y funciones del script. Haré mención de las funciones y variables más relevantes y constantes en el código de cada algoritmo.

La **barra deslizadora** de la parte inferior izquierda funciona como selector de velocidad. Cada vez que se modifica el valor del slider, este llamara una función llamada *setVelo(int, int)* el cual cambiará el valor de dos variables en el script. Contamos con un contador que irá incrementándose en uno en cada ciclo de *draw().* En cuanto el contador llegue al valor de esta variable se reiniciará tomando el valor de 0. En cuanto el contador tome este valor, se actualizarán los valores. Como el *framerate* es constante y no lo hacemos cambiar, la velocidad a la que se incrementa el contador siempre es la misma, pero el límite que alcanzar para reiniciarse se modifica, llegando a este antes o después variando el tiempo de ciclo. Los dos límites definidos son *velocidadCiclo* y *velocidadBucle*. El segundo se usa según en qué ocasiones para alargar el tiempo entre dos ciclos.

El siguiente elemento, la **barra de botones de ejecución** consta de tres botones con diferentes funciones. Son los encargados de administrar la ejecución de la visualización. El primer botón ejecuta la función de *botonPlay()*, la cual actualizará una variable llamada *ejecutado*. De esta variable dependerá que los datos se actualicen o no. La función *draw()* no dejará de ejecutarse siempre que esté la página cargada, pero la llamada de la función *actualizaValores()* dependerá del valor booleano que tome *ejecutado.* Por lo que solo en el momento en que esta variable tome el valor True los valores se actualizarán. Inicialmente esta variable se inicializa en False para que sea decisión del usuario cuándo empezar la ejecución.

El botón play empezará el proceso con un simple cambio del valor de la variable. En cuanto el script se esté ejecutando, el icono de *play* que aparece en el botón cambiará al de *pause* y en este momento la función será la inversa a la anterior.

El segundo botón, el de stop, reinicia la ejecución generando un vector con valores nuevos. Internamente, todos los valores deben volver a tener el valor definido en la función *setup()*.

En cuanto al tercero, el botón de *step forward*, avanzará la ejecución un único ciclo. Esta vez, no actualizará el valor de *ejecutado,* sino que ejecutará la función *actualizaValores()* sin cambiar el valor de la variable.

El siguiente elemento, el botón de creación, despliega un campo de introducción de texto junto a una explicación donde especifica el formato que deben tener los números introducidos, los cuales tienen que estar separados por comas. Este componente permite crear un vector personalizado con los valores entrados manualmente. Su funcionamiento es similar al botón *stop*, reiniciando los valores por defecto inicialmente pero cargando esta vez los valores decididos por el usuario.

El campo de explicación cuenta con un seguido de elementos <h2> los cuales explicarán el funcionamiento de la ejecución en lenguaje verbal. El contenido de los mismos se irá actualizando con el modificador *.innerHTML* en un script en el mismo documento HTML.

De la misma manera, el campo de pseudocódigo cuenta con campos de texto <p>, pero esta vez no se actualizará el texto que contiene, sino el color de fondo de los mismos. Todos estos campos de textos se encuentran dentro de un ocupando un porcentaje del mismo definido en el fichero de *style.*

Este código define que cada que se encuentre dentro del elemento con id=”codigo” tendrá una altura de un 10% de este. El funcionamiento es el mismo en los dos últimos componentes. Definiendo el espacio que ocupa, si modificamos el su color de fondo mediante el modificador *.style.backgroundColor* del elemento seleccionado por id.

La barra de navegación superior tiene definido su funcionamiento por Bootstrap, por lo que simplemente deberemos cambiar el contenido de los títulos así como los menús a los que dirige cada botón.

La zona de visualización se gestiona mediante el script del documento que definimos. Está ubicada dentro de una estructura de flotantes como vemos en la Figura 7:

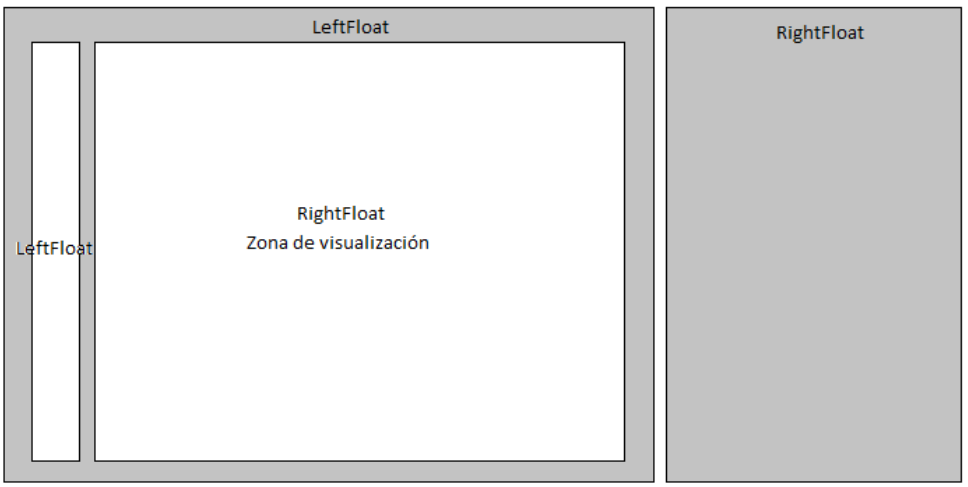


Figura 7. Estructura de flotantes en la que se encuentra la zona de visualización.

4.4.2 Funcionalidad “Drag and Drop”

Una funcionalidad innovadora respecto a otros trabajos relacionados es la funcionalidad Drag and Drop, que consiste en poder seleccionar un elemento del vector con el cursor manteniendo el clic presionado sobre él y arrastrarlo a la posición deseada.

Para ello usaremos funciones proporcionadas por la biblioteca p5.js para detectar el evento de ratón que corresponde con la acción realizada. En cuanto el botón primario del ratón se mantiene presionado, en caso de que el proceso de visualización no se esté ejecutando se comprobará que la posición del ratón corresponde a la vertical de alguno de los elementos. Si es así, lo seleccionará haciendo una copia de este para pintarla en cada iteración en la nueva posición del ratón y cambiando el color del elemento seleccionado por el color de fondo, dando la sensación de que este ha desaparecido.

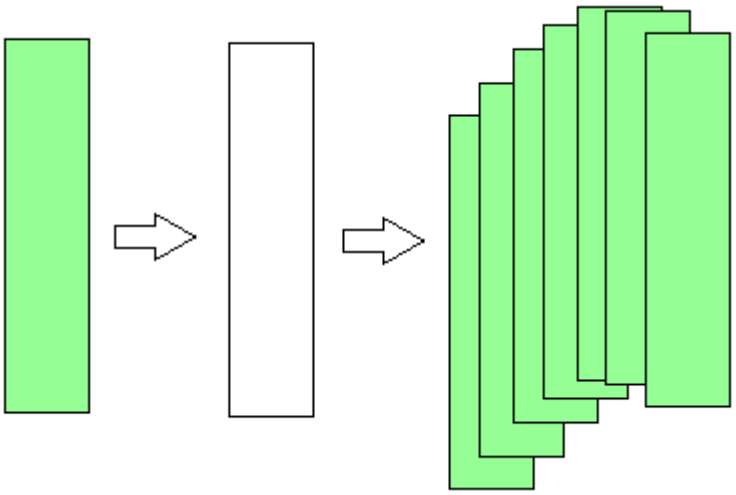


Figura 8. Secuencia de selección de un elemento del vector al moverlo.

Como vemos en la Figura 8, detecta la posición del elemento en la que se encuentra el cursor para gestionar los demás y ordenarlos de la manera conveniente, dejando el espacio “vacío” en la posición del cursor y los demás reordenados a su alrededor. Dependiendo del algoritmo del que se trate, el proceso se reanudará desde un punto lógico.

Una vez soltado el botón del ratón, se devuelve el color al elemento incoloro.

4.5. Implementación de la visualización interactiva de los distintos algoritmos

Primeramente, es necesario definir el modelo de los datos que se usan, común en los diferentes algoritmos:

- **vector:** Es la estructura central a mostrar, se definen un número determinado de elementos con un valor numérico aleatorio el cual define la altura del elemento (en caso de los algoritmos de ordenación). Se muestran en pantalla de izquierda a derecha los elementos de índice menor a los de mayor índice. Excepto en los algoritmos basados en árboles, que el orden de los valores no determina la posición en la que se muestran.
- **elemento del vector:** Cada índice del vector tiene un elemento con diferentes apartados:
    - *num*: El valor numérico generado aleatoriamente.
    - *col*: Define el color del que se pintará su interior.
    - *posx*: La coordenada en el eje X en la que se encuentra.
    - *posy*: La coordenada en el eje Y en la que se encuentra.
- **canvas:** La superficie donde se pintará el procedimiento. Por defecto, el tamaño está definido en 1400 píxeles de anchura por 800 de altura.
- **numeroRectangulos:** La cantidad de elementos que se generan inicialmente.
- **anchoRectangulos:** La anchura en píxeles de los rectángulos a representar.
- **separacionRectangulos:** La distancia en píxeles entre el final de un rectángulo y el principio del siguiente.
- **contador:** La variable que se incrementa cada ciclo para gestionar la velocidad de ejecución.
- **ejecutado:** La variable que determina si se actualizan los valores o no.
- **velocidadCiclo/velocidadBucle:** Límite para gestionar la velocidad de ciclo y de ejecución.

Durante la ejecución del procedimiento, cada ejecución de la función *draw()* llamará a *actualizaValores()* en caso de que el la variable *ajecutado* tenga el valor True. Posteriormente se ejecutará la función *drawrectangulos()* que dibuja los elementos en el orden pertinente con el color que contenga en su campo *col*.

En la función *actualizaValores()* se llevan a cabo las modificaciones en el caso de que el *contador* tenga valor 0. De no ser así, se incrementará hasta llegar al valor de *velocidadCiclo*.

Cada algoritmo cuenta con variables locales y variaciones en el código, las cuales citaré a continuación.

4.5.1. Bubble Sort

Este algoritmo, al ser secuencial y no contar con recursión, su implementación es simple. Se definen los valores de los elementos en el setup usando *Math.floor(Math.random() * 356)* en una función *for* que recorre todas las posiciones del vector. Las variables que iteran son *n* y *m.*

Inicialmente se marca el elemento en primera posición. Para ello se le cambia el valor del atributo *col* a color verde (color(0,255,0)) como podemos ver en la Figura 9.

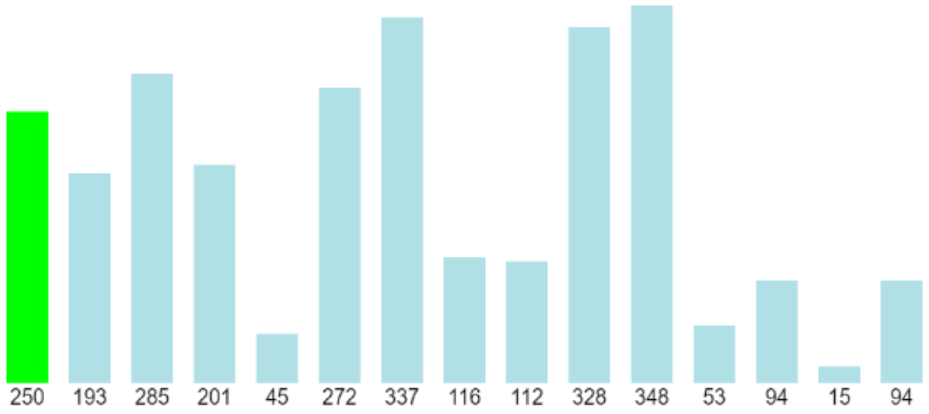


Figura 9. Selección del primer elemento.

Después, se compara su valor numérico con el siguiente elemento. En el caso de que el primero sea mayor al segundo los cambiará de posición, o ese será el efecto visual, ya que los elementos no cambian de posición, sino que al segundo elemento se le otorga el valor numérico y el color del primero y viceversa, dando la sensación de que cambian su posición, como se observa en la Figura 10. En el caso contrario, el valor de n se incrementará seleccionando el siguiente elemento como vemos en la Figura 11.

Figura 10. Elemento mayor al siguiente

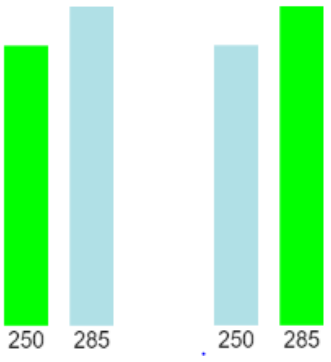


Figura 11. elemento menor al siguiente

De esta manera el mayor elemento será seleccionado para llevarlo al límite superior. Como hay elementos del código que se tienen que destacar, hay ocasiones en las que no basta con un solo ciclo, ya que se requieren más para destacar diferentes líneas de pseudocódigo, unas en cada ciclo. Para ello, se usan variables booleanas para hacer avanzar el código a ejecutar, ya que si avanzara de manera normal no se

llegarían a ejecutar numerosos comandos necesarios para la correcta visualización. Esto lo podemos ver, por ejemplo, en la comprobación de si el valor del siguiente elemento es mayor al actualmente seleccionado:

```
if(lista[j] > lista[j+1]){

    intercambia(lista[j], lista[j+1])

}
```

En el caso de que la condición no sea cierta, el código avanzará en el siguiente ciclo, resaltando únicamente la línea donde se encuentra la instrucción *if.* En el caso de que la condición sea cierta, primero se resalta la misma línea donde se encuentra la instrucción *if,* para seguidamente resaltar la línea con la función *intercambia()* antes de seguir con la ejecución normalmente.

Una vez el mayor valor esté en el mayor índice del vector, tomará un color igualmente verde pero más suavizado (color(150,255,150)) y se selecciona nuevamente el primero para repetir el procedimiento, incrementando *m* y reiniciando *n* a 0 como observamos en la Figura 12.

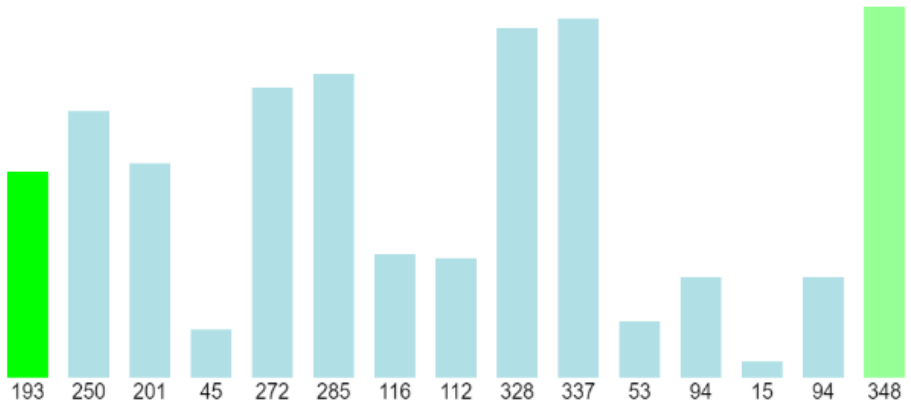


Figura 12. Selecciona el siguiente elemento a ordenar.

4.5.2. Insertion Sort

Nuevamente, este algoritmo es secuencial y sin presencia de recursividad, por lo que su implementación no será complicada. El ritmo de ejecución vuelve a estar pautado por variables booleanas las cuales harán avanzar la ejecución del código. La variable llamada *inicial* determinará si la ejecución se encuentra al principio de un nuevo bucle. De ser así selecciona un nuevo elemento, el cual es comparado con el anterior e intercambiado en caso de tener un menor valor. Como empieza desde la segunda posición, en un estado más avanzado de la ejecución, siempre que un elemento compare su valor con el de la posición anterior y tenga un valor mayor implica que no hay ningún elemento con una posición menor a la suya con un valor menor.

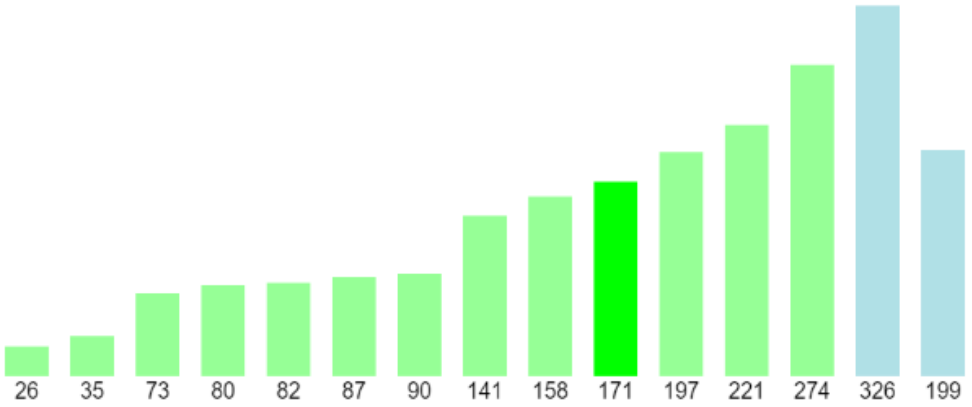


Figura 13. Proceso de ejecución de Insertion Sort.

A efectos prácticos, durante la ejecución, como vemos en la Figura 13, los elementos descienden por el vector hasta encontrar su posición.

4.5.3. Selection Sort

Este algoritmo es el último de los algoritmos secuenciales que no cuenta con la presencia de recursividad siendo, de igual manera que los dos anteriores, de fácil implementación. Para ello, solo hace falta una pequeña reinterpretación del algoritmo original: Al ejecutarse actualizando sus valores en cada llamada de la función *draw(),* debemos sustituir los dos bucles *for* del algoritmo original por un único *while* i gestionar el proceso con variables booleanas, como vemos en el Código 1.

```
while(continua){
      if(inicial){
            m = 0;
            minimo=m;
            inicial = false;
            n = m+1;
      }
      else{
            if(n<values.length){
                  if(values[n].num < values[minimo].num){
                        minimo = n;
                  }
                  n++;
            }
            else{
                  if(minimo != m){
                        swap(minimo, m)
                  }
                  if(m<values.length){
                        m++;
                        n = m+1;
                        minimo=m;
                  }
            }
      }
      if(values.sorted){
            continua = false;
      }
}
```

Código 1. Reinterpretación de Selection Sort.

La función *swap(int, int)* intercambia el valor numérico y el color a los elementos del vector con los índices de entrada.

4.5.4. Merge Sort

Esta vez, tenemos un algoritmo basado en la recursividad el cual debe ser implementado de manera secuencial y sin recursión como he explicado en la sección 4.2. Para ello, implementé un método basado en dos vectores auxiliares, uno para almacenar los límites entre los que se debe evaluar, y un segundo para gestionar el lado que le toca (izquierdo o derecho).

Inicialmente, el array de límites tendrá las posiciones de los dos subvectores a analizar y el vector de lados se inicializará con valor 1, como vemos en el Código 2.

```
limites[0] = 0;
limites[1] = Math.floor(values.length/2) - 1;
limites[2] = Math.floor(values.length/2);
limites[3] = values.length - 1;

lado[0] = 1;
```

Código 2. Inicialización del vector *limites* y *lado*.

La secuencia de estados del vector *lado* corresponde a: 1 si se debe analizar la parte izquierda, 0 si se debe analizar la parte derecha, -1 si se han analizado ambas partes.
Para tener una gestión apropiada, se usa una variable booleana llamada *expandido* que nos indica si ya ha sido desarrollado el lado que toca antes de actualizar el valor. La ejecución se mantendrá en funcionamiento mientras el vector *lado* tenga elementos.
Siguiendo con la ejecución, como el último valor de *lado* es 1 y la variable *expandido* tiene el valor False, significa que debemos analizar la parte izquierda entre los límites y expandirla.

```
posiAux = limites[limites.length-4];
posiAux2 = limites[limites.length-3];
var calc = posiAux2 + 1 - posiAux;
if(calc>2){
      limites[limites.length] = posiAux;
      limites[limites.length]    =    Math.floor(calc/2)-1    +
posiAux;
      limites[limites.length] = limites[limites.length-1] + 1;
      limites[limites.length] = posiAux2;
      lado[lado.length] = 1;
}
```

Código. Añadiendo nueva partición a evaluar en Merge Sort.

La variable *calc* guarda cuántos elementos contiene esta parte izquierda, ya que en caso de haber más de dos elementos se debe seguir haciendo particiones al vector.

Una vez que la partición contiene 1 o 2 elementos, se comparan y se ordenan de menor a mayor.

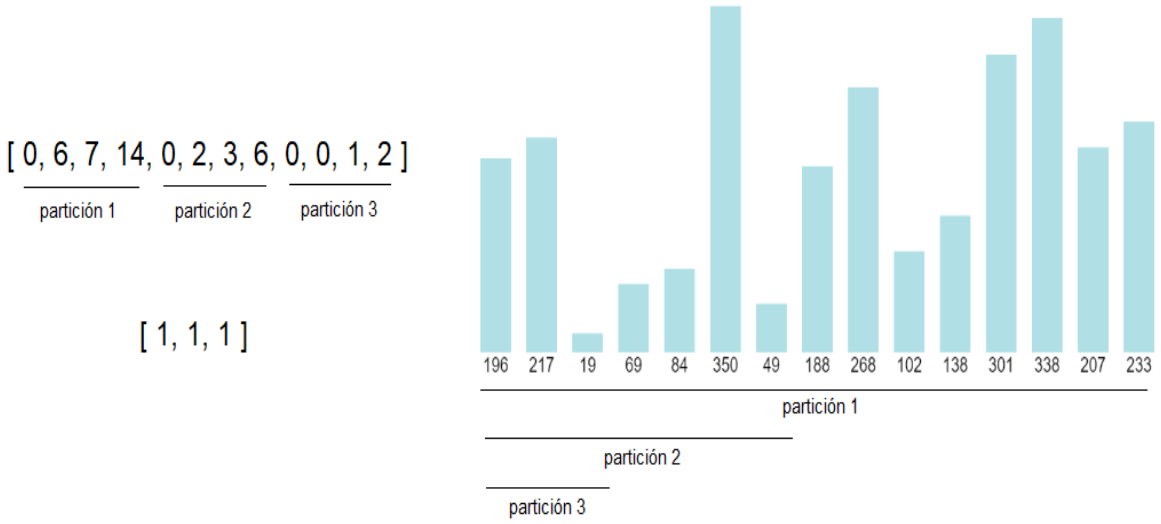


Figura 14. Gestión de particiones a analizar

Posteriormente se analiza la parte derecha de la última partición añadida, cambiando el último valor del vector *lado* a 0. En este caso se realiza exactamente el mismo procedimiento que en el lado izquierdo, pero esta vez cogiendo los dos últimos valores del vector de límites, quedando una estructura como vemos en la Figura 14.

Una vez analizados ambos lados, el elemento en el vector *lado* toma valor -1, por lo que se ordenan los elementos entre el menor de la posición izquierda y el mayor en la posición derecha. Después, se elimina el último valor de *lado* así como los cuatro últimos valores del vector *limites,* reanudando la evaluación de la anterior partición.

Durante su ejecución, se irá resaltando la partición que actualmente se está manipulando como podemos observar en la Figura 15.

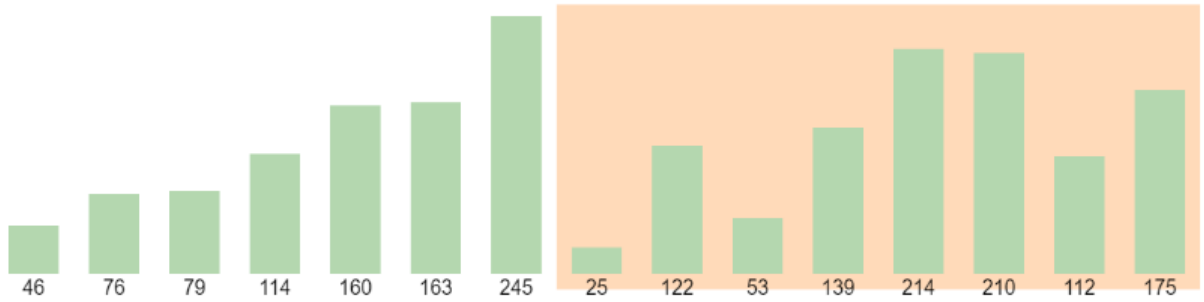


Figura 15. Marca visual de la partición a manipular.

4.5.5. Quick Sort

Este algoritmo sigue la misma técnica que Merge Sort descrito en el apartado anterior. Cuenta con una lista llamada *lado* que gestiona la qué lado de la partición es necesario expandir, también cuenta con una lista llamada *limites* que almacena los límites entre los que se tienen que gestionar los datos. Como vemos en la Figura 16, una vez se ejecuta, se marca de color los límites entre los que se gestionan.

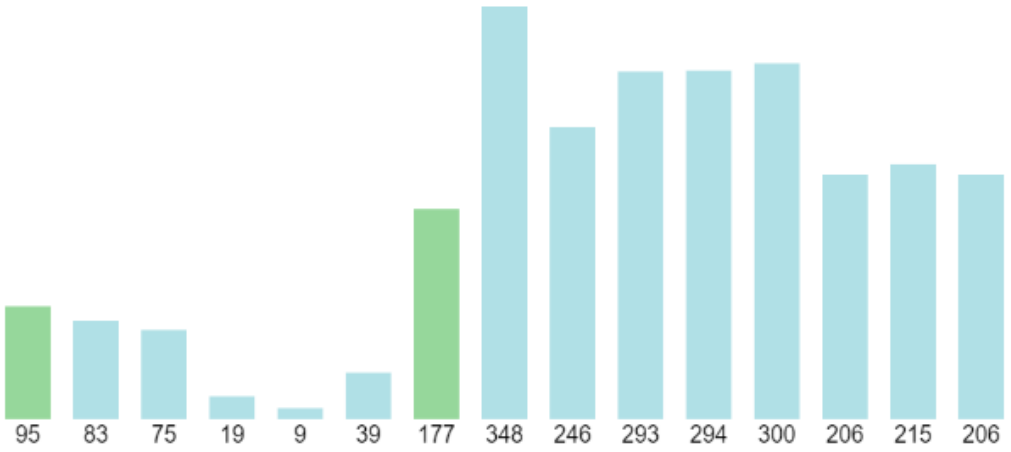


Figura 16. Marca visual de los límites durante la ejecución de Quick Sort

Durante la ejecución, se distinguen dos estados principales, uno en el que aun no se ha definido el pivote y el otro en el que se ha definido el pivote y se han gestionado los elementos. Con dos variables marcando el límite menor y otra marcando el límite mayor, inicialmente inicializadas entre las posiciones 0 y la última.

Primeramente, al no haberse establecido un pivote, entra en la fase que pertoca. En un ciclo se selecciona el primer elemento de la división entre el límite menor y el mayor como pivote. En el siguiente ciclo se sitúan a la izquierda de la sublista los elementos menores al pivote y a la derecha los mayores, dejando paso a que el siguiente ciclo entre en la fase en la que el pivote se ha seleccionado. En esta fase, dependiendo del valor que se encuentre en la lista *lado,* gestionará el vector de límites para definir los nuevos valores.

4.5.6. Ordenación Par-Impar

El primer algoritmo en paralelo. El funcionamiento es semejante al de *Bubble Sort* pero gestionando varias comprobaciones en paralelo al mismo tiempo. Mi implementación es una reinterpretación de este algoritmo programado secuencialmente para dar la sensación de que los procesos pasan de manera paralela, ayudándome del refresco del *canvas* proporcionado por p5.js. El proceso lo divido en tres fases:

- fase 0: Se encuentra el vector principal y no existen procesos en paralelo en este momento, los cambios causados por los que hayan habido anteriormente se han aplicado.
- fase 1: Se crean procesos en paralelo en las posiciones adecuadas.
- fase 2: Se gestionan los procesos en paralelo anteriormente creados ordenando los elementos en caso de ser necesario.

Mediante una variable llamada *oddeven* se gestiona si los procesos en paralelo se crean en las posiciones pares o en las impares. Como crea procesos con pares de elementos, gestiona de manera adecuada el número de elementos para no crear procesos con posiciones nulas.

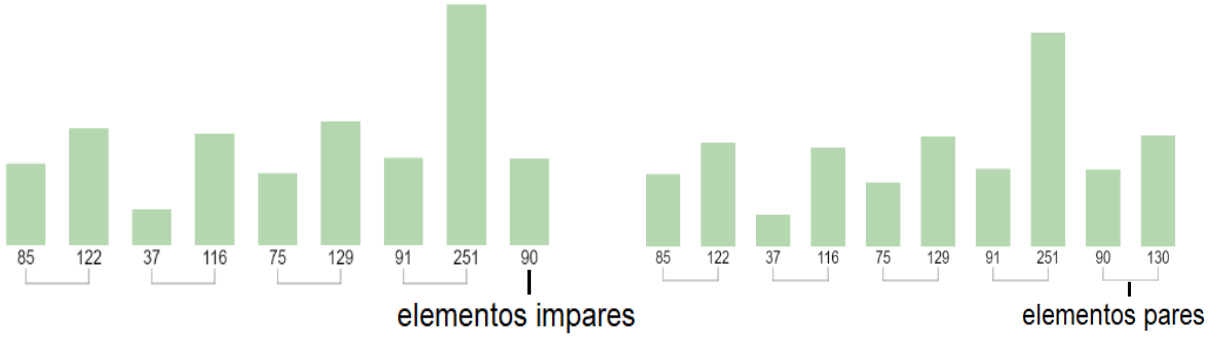


Figura 17. Comparación de la gestión con elementos pares con la gestión con elementos impares

Como vemos en la Figura 17, tanto en la fase 1 como en la fase 2 se marcará mediante líneas qué elementos se agrupan entre sí. Estas empezarán en la mitad del rectángulo desde el que empieza el método y llegarán a la mitad del siguiente rectángulo, acompañadas por dos líneas más pequeñas verticales a modo de marca visual clara.

Distinguimos dos procesos diferentes: uno en el que se actualizan los valores y otro en el que se pintan las líneas y los rectángulos secundarios. Durante la fase 0, una variable booleana llamada *inicio* marca si la ejecución se encuentra al principio y de ser así no ordenará ningún elemento pasando a la siguiente fase. Seguidamente, en la fase 1 y 2 no se actualizará ningún valor, pero se pintarán los rectángulos

seleccionados por pares en la parte inferior de los rectángulos del vector principal, como vemos en la Figura 18.

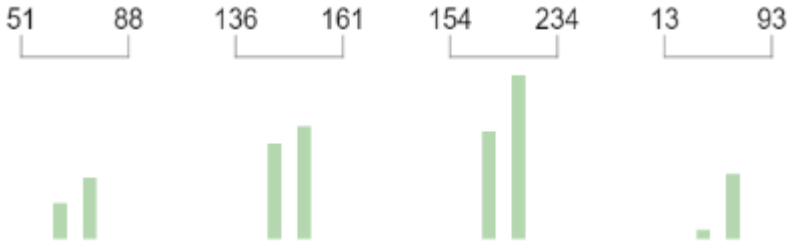


Figura 18. Rectángulos secundarios pintados en la parte inferior.

Aunque en la fase 1 y 2 no se actualicen los valores, sí que se pintan en posiciones diferentes si conviene. En la fase 1 se muestran en el mismo orden en el que se encuentran en el vector principal, en cambio, en la fase 2, se pintarán con el rectángulo con mayor valor a la derecha simulando que en los procesos en paralelo los elementos se ordenan. una secuencia de ordenación pasando por las tres fases tiene la apariencia que vemos en la Figura 19.

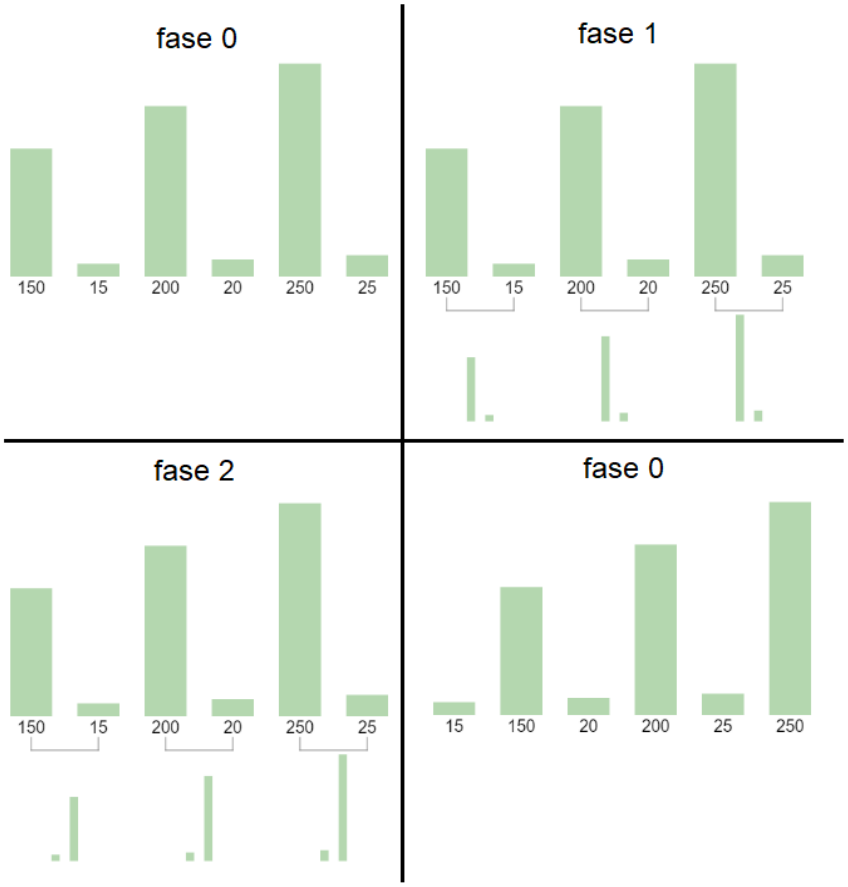


Figura 19. Secuencia de fases.

En cada ciclo, se comprobará si los elementos del vector principal están ordenados. De ser así, los elementos se pintarán de verde indicando que la ejecución acabó.

4.4.7. Merge Sort Paralelo

Como se explica en el apartado 3.2., este algoritmo funciona de la misma manera que su versión secuencial pero ejecutando todas las particiones en procesos en paralelo. Por eso, la visión más fiel a la realidad corresponde a la que se visualiza. El vector principal no se va modificando, son los subprocesos los que gestionan los datos y aplicando *merge* se irán reduciendo hasta "volcar" los datos al vector principal.

Como no se modifican los valores del vector principal, será en la función de dibujo donde se gestionará la posición de los rectángulos secundarios. Nuevamente, los subprocesos se representarán como grupos de rectángulos de tamaño más reducido que los del vector principal. Serán agrupados representando el número de subprocesos generados. De igual manera, como marca visual se generan líneas para delimitar los elementos del subproceso, como vemos en la Figura 20.

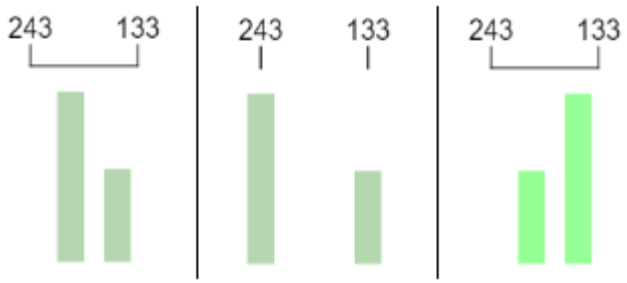


Figura 20. Representación de la creación de subprocesos y merge.

Como vemos en la figura anterior, se crean subprocesos hasta que los elementos de cada subproceso es igual a 1.
Primeramente, se calcula el número de estados que habrán durante la ejecución y se inicializa un contador de estados a 0. Respecto a la ejecución, podemos distinguir dos partes. La primera parte de la ejecución consiste en dividir los elementos en subvectores y la segunda consiste en hacer *merge* de todos los subprocesos en el mismo orden en el que se han separado. Para realizarlo se evalúa el valor del contador de estados y en caso de ser menor a la mitad del número de estados totales se ejecutará la primera parte pintando los rectángulos secundarios del mismo color que los del vector principal. Una vez pasada la mitad de procesos, se ejecuta la segunda parte, donde se volverán a reagrupar los elementos de manera ordenada, esta vez pintados de verde indicando que están ordenados. Una vez llegue al último estado desaparecen los rectángulos secundarios y las líneas como vemos en la Figura 21.

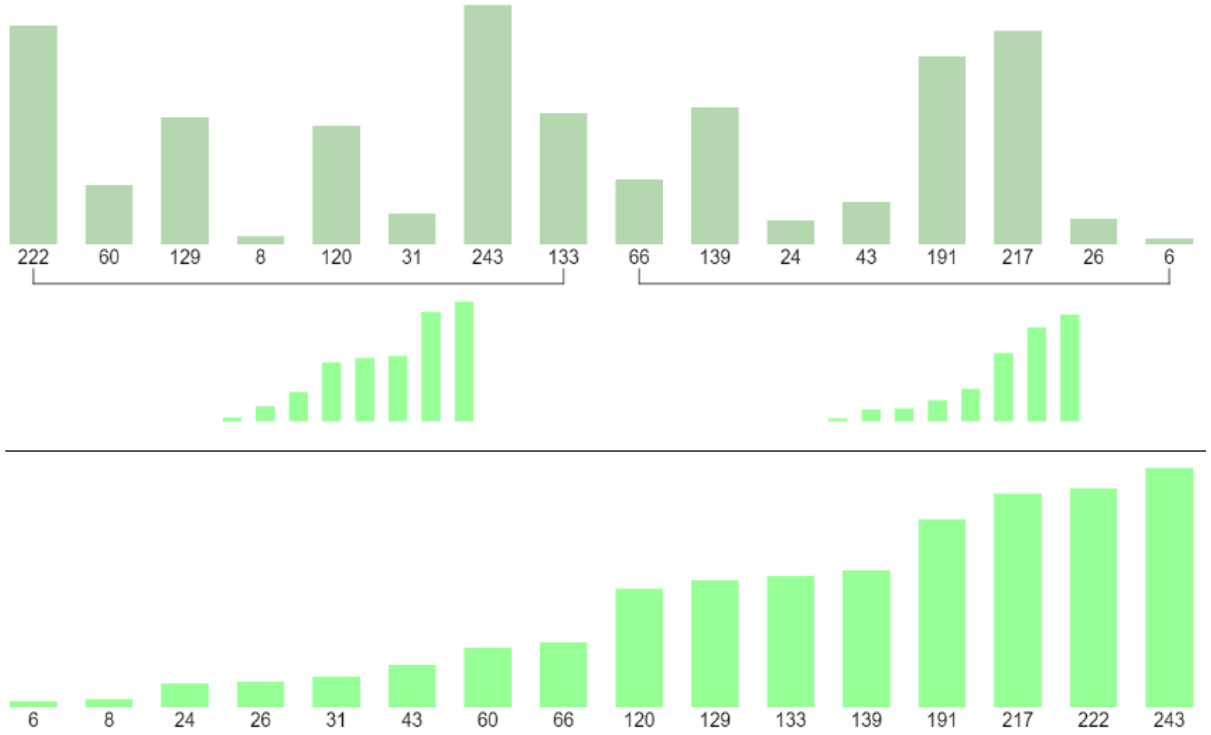


Figura 21. Subprocesos uniéndose al vector principal.

Se calcula el número de subprocesos que le pertoca al estado en el que se encuentra la ejecución y se crean vectores auxiliares donde se crea una copia de los elementos en ese sector. Para cada grupo, se calcula el punto correspondiente a la mitad del espacio total que ocupan los rectángulos en su posición en el vector principal, a partir de ese punto se añadirán los rectángulos secundarios.
Una vez llegado al estado final, para simular que al vector principal se le añaden los elementos del proceso paralelo donde están todos los elementos ordenados, se ordenan todos los elementos del vector para mostrarlo en su fase final ordenada.

Este algoritmo prescinde de la función de crear. El número de rectángulos inicial es óptimo para que la ejecución sea clara y entendedora. Siendo otra cantidad, el proceso se realizaría de manera dispar y unos procesos se desarrollarían mientras otros habrían finalizado. Como el objetivo de esta visualización es didáctico, prefiero que todos los procesos se ejecuten a la vez para que el concepto de paralelismo sea claro.

4.5.8. Quick Sort Paralelo

De igual manera que los algoritmos paralelos anteriores, este algoritmo es la versión en paralelo de Quick Sort secuencial definido anteriormente. Aunque esta versión cuenta con una diferencia: el pivote que en la versión secuencial se escogía mediante un algoritmo, en la versión en paralelo se escoge de manera aleatoria para cada partición. Se pueden distinguir dos partes distintas en este procedimiento, en la primera se definen los pivotes y en la segunda se ordenan los elementos. Para realizar la primera parte, se analizará el color de los elementos. Cuando un elemento es escogido como pivote, su color cambia a RGB(72,61,139), por lo que para saber si un elemento es pivote o no simplemente se comprueba el valor del canal azul en su color RGB (su valor inicial es 175), vemos la diferencia en la Figura 22.

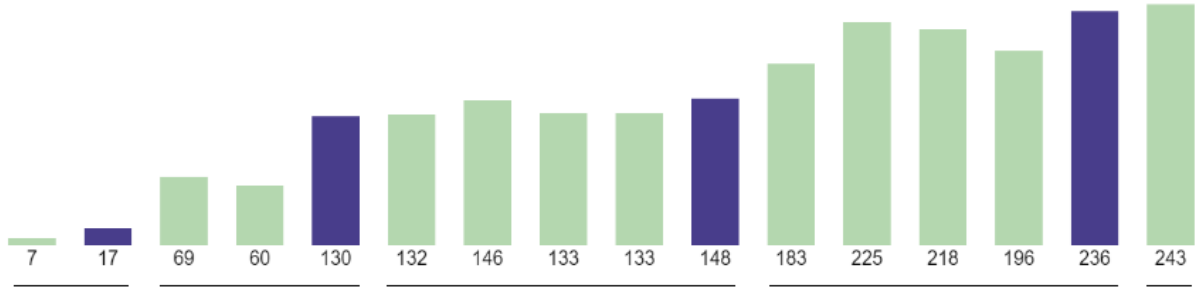


Figura 22. Gestión de las diferentes particiones.

Un vector de pivotes gestionará el número de pivotes que presenta el proceso. Para cada partición, si hay más de un elemento que no sea pivote, se escogerá aleatoriamente un pivote. Este proceso se realizaría de manera paralela, así que realizaré todas las actualizaciones de pivotes en el mismo ciclo, al igual que la posterior ordenación de los elementos.
Una vez escogidos los pivotes se inicia la siguiente parte. Se crea una copia del vector principal y se recorre todo el vector añadiendo en un vector auxiliar los elementos menores al pivote eliminandolos de este vector copia. Posteriormente, se añade el pivote y todos los elementos restantes, que serán los de mayor valor que el pivote, como vemos en la Figura 23. Este proceso se realiza tantas veces como pivotes hayan, más una vez adicional teniendo en cuenta los posibles elementos mayores al último pivote. Repitiendo este proceso, se escogerá un pivote siempre que haya más de un elemento no pivote junto, reordenando los elementos hasta que el vector queda completamente ordenado.

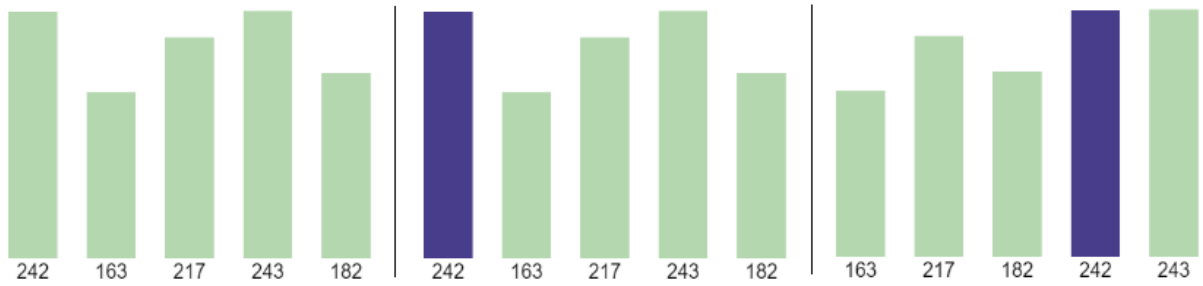


Figura 23. Secuencia de selección de pivote y ordenación.

4.5.9. Binary Search Tree

Encabezando los algoritmos basados en una estructura de árbol, en Binary Search Tree implemento un buscador donde, entrado un valor numérico, recorre los nodos del árbol para encontrarlo.
Esta vez, al cambiar la estructura de los elementos, también cambiarán sus atributos. Siguen teniendo un valor numérico, un color, posición en el eje horizontal y posición en el eje vertical, adicionalmente se añaden los atributos:

- padre: la posición en el vector principal *values* donde se encuentra el nodo padre. Toma valor -1 en caso de no tener nodo padre.
- iz: la posición en el vector principal *values* donde se encuentra el nodo hijo izquierdo. Toma valor -1 en caso de no tener nodo hijo izquierdo.
- der: la posición en el vector principal *values* donde se encuentra el nodo hijo derecho. Toma valor -1 en caso de no tener nodo hijo derecho.
- diam: el diámetro que tendrá el círculo pintado para representar el elemento.

Como vemos, los elementos se siguen almacenando en un vector principal, ahora no son ordenados según su posición, en su lugar cada elemento contará con el índice de la posición del vector donde se alojan los elementos relacionados. En el caso de crearse un elemento con un valor numérico más pequeño o igual al actual, se situará en posición izquierda de este. En caso de ser mayor, se posiciona en la posición derecha. Esa es la premisa de un árbol binario de búsqueda.

Para generar el árbol inicial se generará un valor aleatorio entre 1 y 250 que se convertirá en el nodo padre. Seguidamente se generarán tantos valores aleatorios como marque la variable *numeroCirculos* que limita la cantidad de elementos. Una vez generado un nuevo valor, empezando desde el nodo padre situado en la posición 0 del vector principal se compara su valor numérico con el generado, en caso de que el generado sea menor se comprueba si el nodo tiene un índice válido en su atributo *iz*, de ser así, exploramos ese nodo izquierdo almacenado en la posición values[nodoPadre.iz]. El proceso se realiza de la misma manera en caso de que el valor sea mayor, con la diferencia que en lugar de evaluar el nodo izquierdo evalúa el derecho. Seguiremos el proceso hasta encontrar un elemento que no contenga un nodo hijo en la posición en la que debería alojarse el valor. Será en esa posición donde se cree un elemento calculando sus atributos de posición y se inicializar los de color, número, padre, hijo izquierdo, hijo derecho, y diámetro. El tamaño del círculo depende del valor numérico, pero se inicializa con un valor mínimo de 40px al que se le suma una quinta parte del valor. Podemos ver la posición resultante y el tamaño de los elementos en la Figura 24.

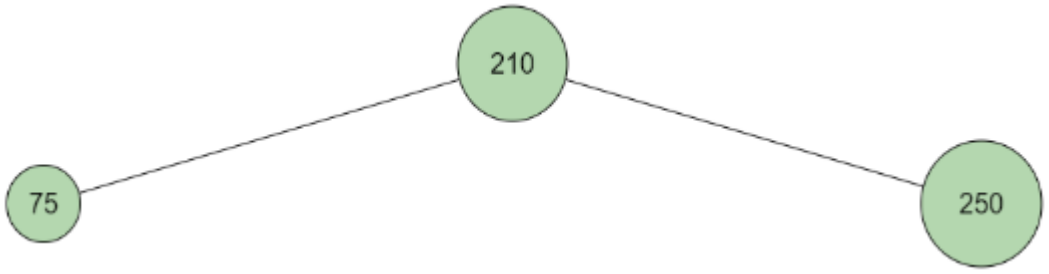


Figura 24. Estructura de los elementos en el árbol binario de búsqueda.

Para calcular la posición del elemento generado, se calcula la posición horizontal y la vertical por separado. Se registra mediante un contador la profundidad a la que ha llegado el elemento. Este contador se incrementará en uno cada vez que se explore un nodo hijo. De esta manera se añadirán 100 pixeles a la altura por cada nivel de profundidad. Por otro lado, para calcular la posición horizontal también se tiene en cuenta la profundidad. Depende de los niveles que haya descendido en el árbol, se calculan tantas particiones como elementos podrían haber si el árbol estuviera completo de elementos en esa profundidad, como vemos en la Figura 25.

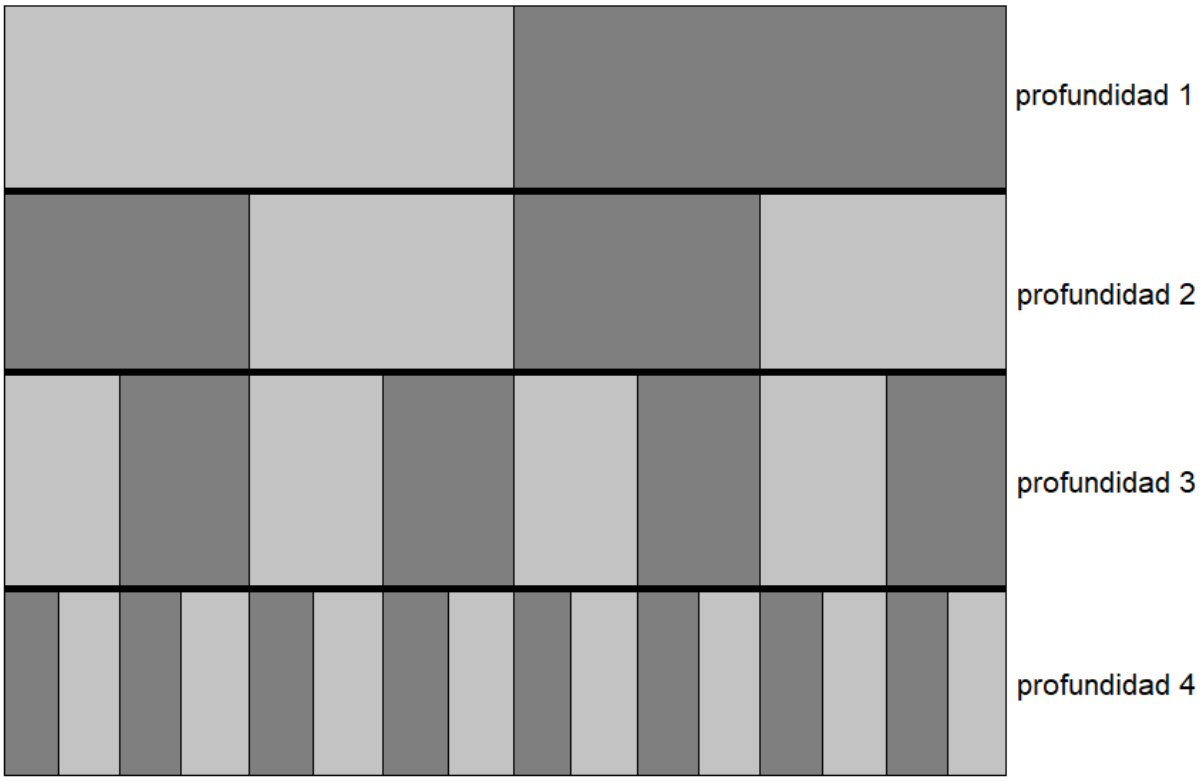


Figura 25. Cálculo de la posición horizontal según la profundidad.

Mediante un nuevo contador iniciado en 0, cada vez que explore el nodo izquierdo se restará 1, pero si el nodo explorado es el derecho se le sumará 1. Llegado a la posición donde se genera, su posición horizontal final será al valor absoluto de este contador desplazado al lado que pertoque como se ve en la Figura 26.

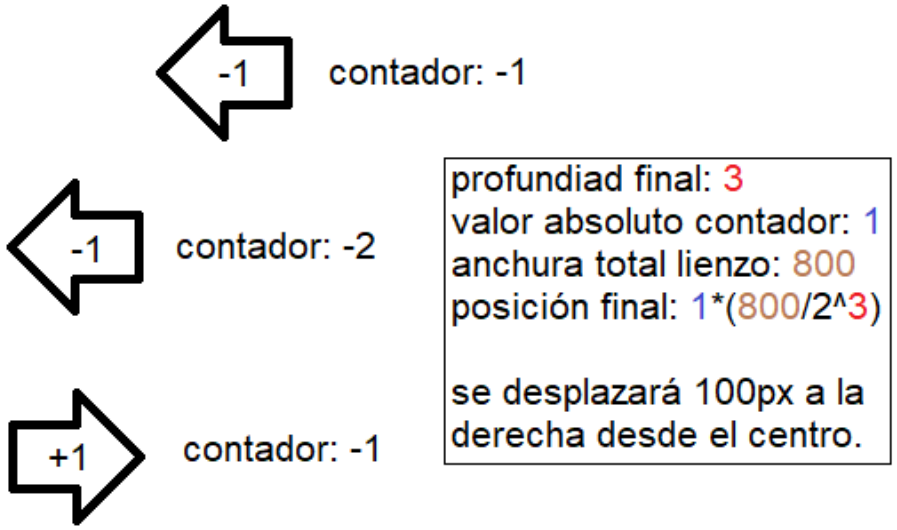


Figura 26. Cálculo de la posición horizontal.

Con este procedimiento, el árbol se generará de manera aleatoria cada vez que es cargado.

Este algoritmo se iniciará como ejecutado, a diferencia de los demás, ya que la función que hará iniciar la ejecución será la entrada de un valor en el campo “Buscar Valor” situado en la posición donde antes se encontraba el botón de creación de lista. Una vez entrado un valor y presionado el botón, se muestra un texto indicativo del número que se buscará. A lo largo de la ejecución donde recorrerá el árbol comparando el valor entrado con el valor numérico de cada elemento explorando los nodos hijos, en caso de no coincidir, siguiendo el mismo criterio de creación del árbol, se creará un vector donde se guardarán pares de números. Estos pares corresponden a los índices de los elementos alojados en el vector principal desde los cuales se ha recorrido su camino. El primer número indica el elemento de partida y el segundo número indicará el elemento destino. Recorriendo este vector se dibuja el camino total recorrido marcándolo de color verde, como vemos en la Figura 27.

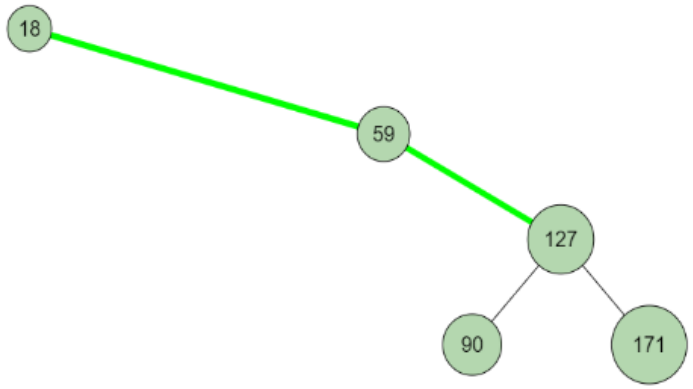


Figura 27. Marca del camino recorrido.

4.4.10. Breadth First Search

La estructura del árbol generado y el método para pintarlo en el lienzo son similares a los del algoritmo explicado anteriormente “Binary Search Tree”. La única diferencia es que, en este caso, no es necesario que su posición se determine por el valor numérico del elemento, sino que esta será aleatoria. Para poder generar números aleatoriamente y que estos estén unidos al árbol sin crear nodos inconexos se genera primeramente un árbol binario de la misma manera que en el caso de “Binary Search Tree”, con elementos menores en la posición izquierda y elementos mayores en la posición derecha. Posteriormente, se recorren todos los elementos del vector cambiando el atributo *num* por un número generado aleatoriamente, y en consecuencia su diámetro. De esta manera tendremos un árbol con elementos completamente aleatorios, como vemos en la Figura 28.

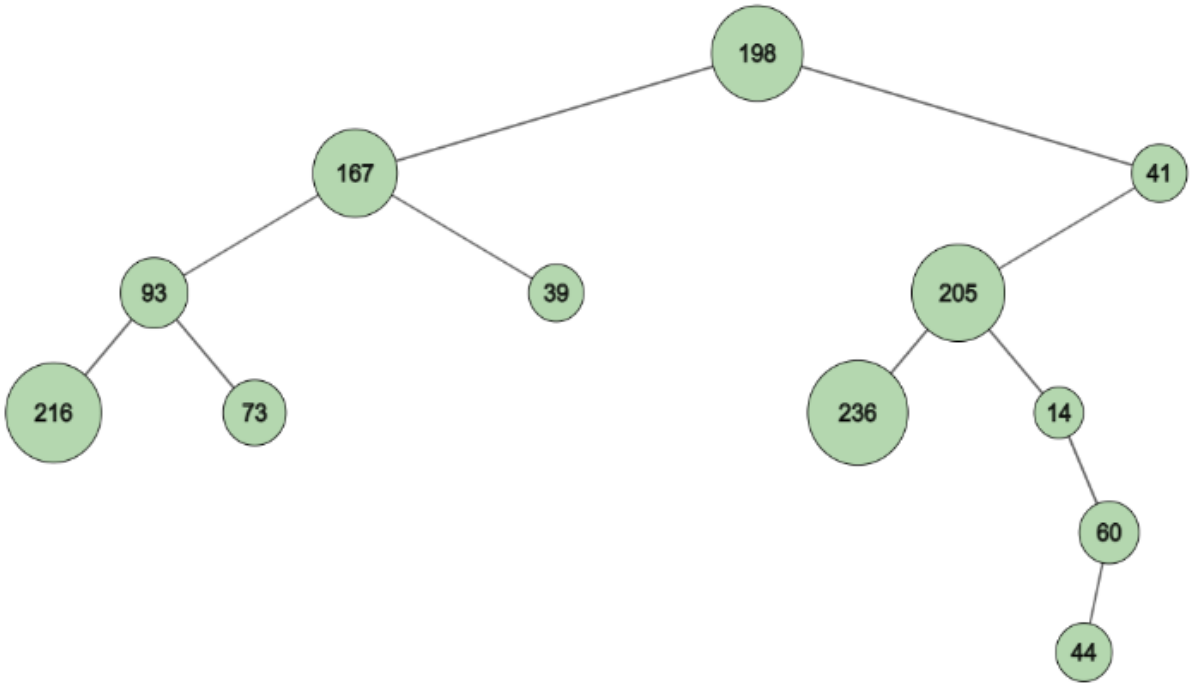


Figura 28. Apariencia del árbol aleatorio.

Para considerar el proceso como terminado, deben haber sido visitados todos los elementos. Como retorno, un texto en la parte superior indica qué nodos han sido visitados.

Para la ejecución del algoritmo, se genera un vector con los nodos a explorar. Como en el caso anterior, este vector guardará los índices de la posición de los elementos en el vector principal *values.* Inicialmente, el vector contiene únicamente el primer nodo, el nodo padre. Cuando se explore, se añadirá al vector el hijo izquierdo y el hijo derecho, en caso de tener. Una vez añadidos los posibles hijos, se elimina del vector de nodos a explorar y se añade a otro vector de nodos visitados, procediendo así a la exploración del siguiente nodo (uno de sus hijos, o un nodo anterior en caso

de no tener hijos). Un sistema de *flags* coordinará cuándo explorar el hijo derecho, el izquierdo o cuándo eliminarlo de la lista.

Igual que en el algoritmo del apartado anterior, el camino que se recorre se irá marcando de verde con el mismo sistema que ofrece el vector de caminos anterior. Pero al no ser un camino continuo, sino que va explorando los elementos separadamente, solo se marcará de color verde el último camino que recorre, los recorridos anteriormente se marcan con un tono más apagado para explicar visualmente que no es el camino actual. De igual manera, se marcan de verde los nodos a explorar y de un tono más apagado los nodos que ya han sido explorados, como vemos en la Figura 29.

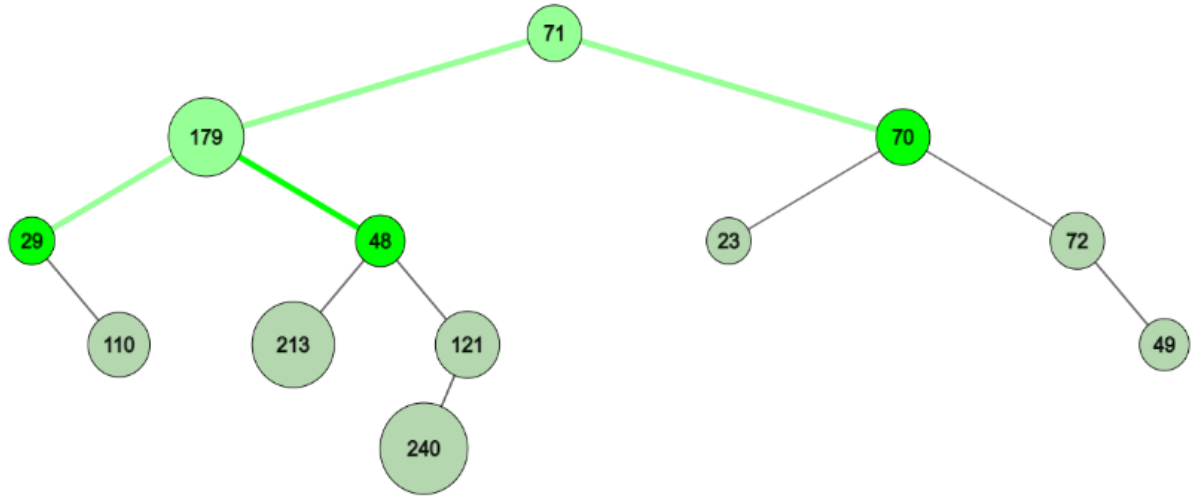


Figura 29. Colores de los caminos según el estado de ejecución.

4.4.11. Depth First Search

Este algoritmo es muy similar al anterior, la única diferencia es que a la hora de explorar nodos prioriza la profundidad a la anchura, de ahí su nombre. Hay tres estrategias para devolver los datos:

- preorder: Se devuelve el elemento antes de expandirlo, después se expanden los posibles hijos.
- inorder: Se devuelve el elemento una vez se ha expandido el subárbol generado por el posible hijo izquierdo.
- postorder: Se devuelve el elemento una vez se han expandido ambos hijos en caso de existir.

La estrategia que sigue mi código es postorder, y en ese orden devolverá los elementos como vemos en la Figura 30.

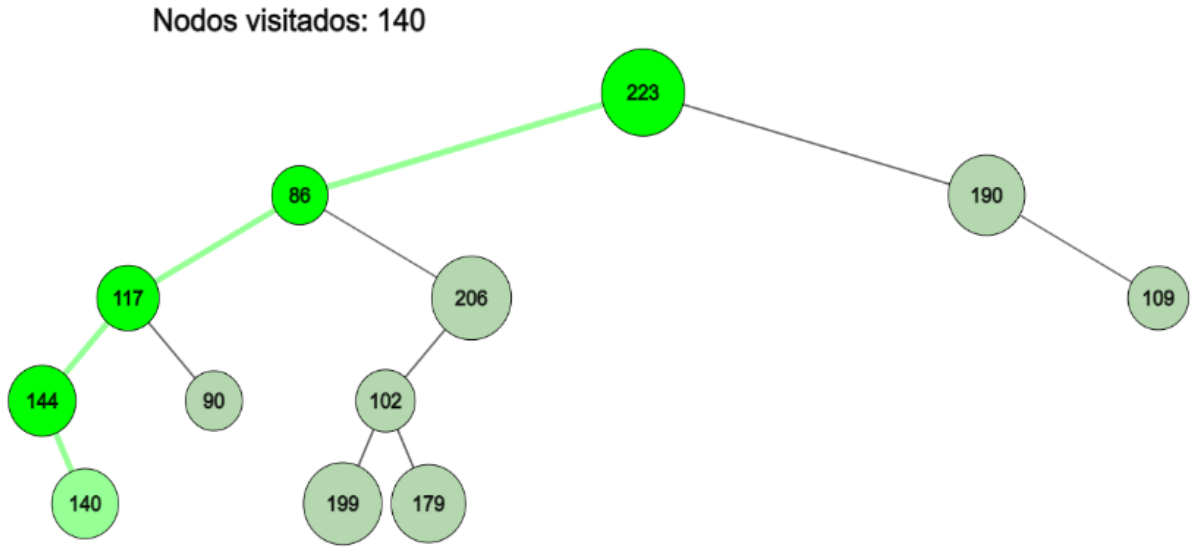


Figura 30. Orden de exploración de los nodos.

Para implementar esta estrategia, esta vez se añadirá el elemento a explorar al final del vector *explorados*, que será el elemento el cual se evalúe cada ciclo. Una vez expandido completamente se eliminará del vector, funcionando como una pila en lugar de como una cola.

5. Discusión de los Resultados y Limitaciones

JavaScript ofrece una fácil implementación de elementos dinámicos. El proyecto ha resultado correcto. La combinación de características y funcionalidades hace de este visor una herramienta útil para la visualización de diferentes algoritmos. Las limitaciones de procesamiento y memoria que supone la creación de la herramienta basada en web no resultan en absoluto condicionantes.

En cuanto a la apariencia, la elección de la librería p5.js sí que condiciona en algún aspecto:

- No ofrece transiciones dinámicas de movimiento de elementos. En diseños webs modernos, son comunes los elementos con movimientos dinámicos. Estos son de gran utilidad a la hora de visualizar de manera clara y entendedora la trayectoria de los datos.
- Al ser una librería que ofrece funciones con las que trabajar en un canvas, al pintarse elementos sobre el canvas, estos no se visualizan tan nítidamente como si se trabajara con elementos HTML.

No obstante, aun teniendo estas desventajas, el balance beneficio/pérdida sale favorable teniendo en cuenta tanto los inconvenientes como las ventajas:

- La flexibilidad para pintar elementos visuales por todo el canvas resulta de gran ayuda para marcar elementos en cada fase de ejecución, aportando comprensión al proceso.
- La facilidad de crear elementos intuitivamente, con atributos bien definidos en las especificaciones de la biblioteca, resulta en la posibilidad de poder incorporar algoritmos más complejos que con una biblioteca que gestione elementos HTML.
- Gracias a la tasa de refresco del canvas y la actualización de datos entre refrescos de lienzo, resulta en una buena representación de elementos sucediendo en procesos paralelos.

El resultado de la estructura HTML es responsive, adaptándose a diferentes resoluciones de pantalla, pero como los elementos con texto tienen un tamaño fijo, la herramienta se muestra de manera incorrecta a partir de cierto zoom. En la Tabla 1 se muestra el zoom máximo que puede tomar el browser dependiendo de varias resoluciones sin que la herramienta pierda su correcta visualización.

Resolución	zoom máximo
1280 x 600	x90
1280 x 768	x125
1366 x 768	x125
1920 x 1080	x150

Tabla 1. Zoom máximo con varias resoluciones.

Sin embargo, la herramienta está diseñada para mostrarse en pantalla completa a una resolución mínima de 1366x768 con un zoom del 100%

Todas las librerías que uso en la herramienta son versiones alojadas en línea, CDN. De esta manera, la herramienta puede usarse desde cualquier ordenador con un browser instalado y conexión a internet. Al no contener librerías descargadas, el conjunto de documentos resulta poco pesado facilitando el acceso a él y su posible descarga de manera rápida.

Adjuntaré ejemplos de ejecución en el Anexo, donde se puede ver la apariencia de la herramienta durante la ejecución.

6. Conclusiones

Vistos los resultados obtenidos creando esta herramienta y el proceso que esto ha conllevado saco la conclusión de que la implementación hubiera sido más fácil usando uno de los framework populares hoy en día, pero la ausencia de estos ha resultado en una herramienta fácil de entender sin dejar atrás la complejidad de los algoritmos.

Dado que la visualización de algoritmos de ordenación ejecutados en paralelo es una de las mayores motivaciones de la realización de este trabajo, los beneficios que otorga la librería p5.js a la hora de visualizar estos algoritmos son decisivos para concluir en que és una buena elección para esta tarea en particular.

Con el conocimiento adquirido tras la realización de este trabajo, volvería a escoger esta librería para la realización de una herramienta similar. Ya que esta herramienta va dedicada a los usuarios en posición de aprender estos algoritmos, no dudaría en seguir la misma metodología para añadir más algoritmos que pudieran ser útiles para este grupo de usuarios. Por falta de información sobre más algoritmos esenciales para el aprendizaje de este modelo de usuario no implementé más algoritmos ya que podía resultar en una herramienta difícil de entender por el exceso de información. Por ello, con la implementación de los presentes, pretendo demostrar que pueden ser implementados más algoritmos de la misma índole.

7. Bibliografía

[1] VASSILIOS DAGDILELIS, MAYA SATRATZEMI, GEORGIOS EVANGELIDIS (2004). *Introducing Secondary Education Students to Algorithms and Programming*. Department of Educational and Social Policy, University of Macedonia.
URL: https://link.springer.com/content/pdf/10.1023/B:EAIT.0000027928.94039.7b.pdf

[2] Michael Eagle, Tiffany Barnes (Junio 2008). *Wu's castle: teaching arrays and loops in a game*. University of North Carolina at Charlotte.
URL: https://dl.acm.org/doi/abs/10.1145/1384271.1384337

[3] Michal FORIŠEK (2015). *Towards a Better Way to Teach Dynamic Programming*. Comenius University, Bratislava, Slovakia.
URL: http://ioi.te.lv/oi/files/volume9.pdf#page=47

[4] Steven HALIM, Zi Chun KOH, Victor Bo Huai LOH, Felix HALIM (2012). *Learning Algorithms with Unified and Interactive Web-Based Visualization*. School of Computing, National University of Singapore.
URL: https://ioinformatics.org/journal/INFOL099.pdf

[5] Ladislav Végh (2016). *Creating Interactive JavaScript Animations for Demonstrating Algorithms on One-Dimensional Arrays*. Acta Didactica Napocensia.
URL: https://files.eric.ed.gov/fulltext/EJ1110308.pdf

[6] Bilal Jan, Bartolomeo Montrucchio , Carlo Ragusa , Fiaz Gul Khan, Omar Khan (2012). *FAST PARALLEL SORTING ALGORITHMS ON GPUS.*Dipartimento di Automatica e Informatica, Politecnico di Torino. URL:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.403.5244&rep=rep1&type=pdf

[7] Euripides Vrachnos, Athanassios Jimoyiannis (2014) *Design and evaluation of a web-based dynamic algorithm visualization environment for novices*. Department of Social and Educational Policy, University of Peloponnese, Korinthos, Greece.
URL: https://pdf.sciencedirectassets.com/280203/1-s2.0-S1877050914X00025/

[8] Sarah Douglas, Donna McKeown, Christopher Hundhausen. *Exploring Human Visualization of Computer Algorithms*. Psychology Dept. University of Oregon. URL:
https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.25.5948&rep=rep1&type=pdf

[9] *High Performance Computing*. curso de URL: https://www.udacity.com/

[10] Libro. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009) *Introduction to Algorithms. 3rd edition*. Massachusetts Institute of Technology.

[11] Libro. Robert Sedgewick, Kevin Wayne (2011) *Algorithms.* Addison-Wesley, 4th Edition.

[12] Libro. Thomas H. Cormen (2013) *Algorithms unlocked.* Massachussetts Institute of Technology.

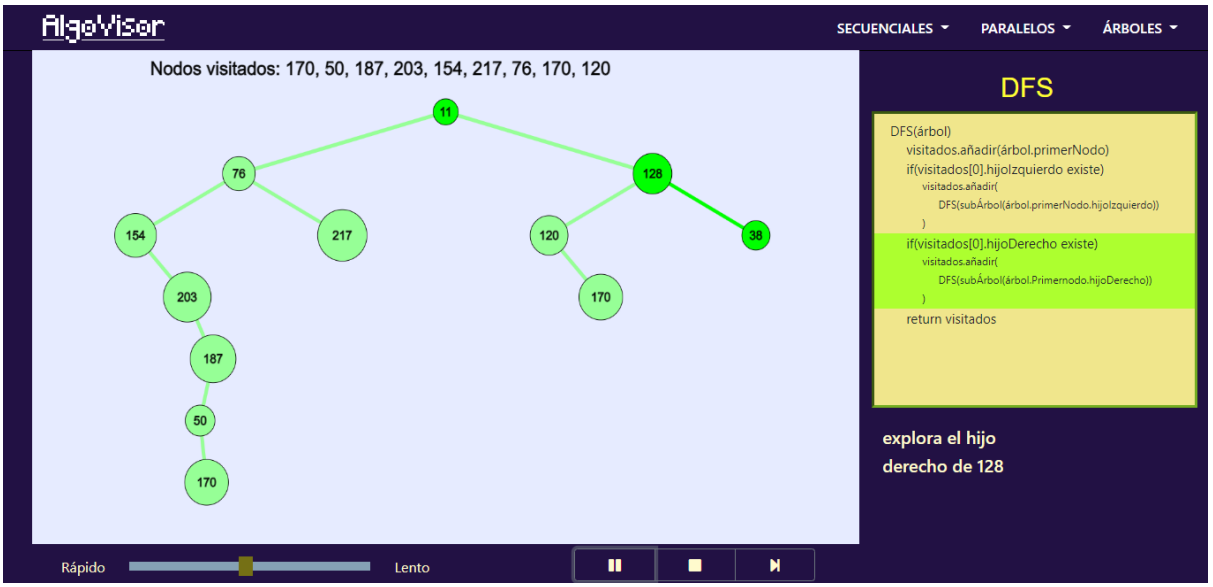
[13] Libro. Selim G. Akl (1985) *Parallel Sorting Algorithms*. Academic Press Inc.

[14] Pallavi Yadav, Paras Nath Barwal (November 2014) *Designing Responsive Websites Using HTML And CSS.* Internal journal of scientifc & Technology research Vol.3. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1091.1360&rep=rep1&type=pdf

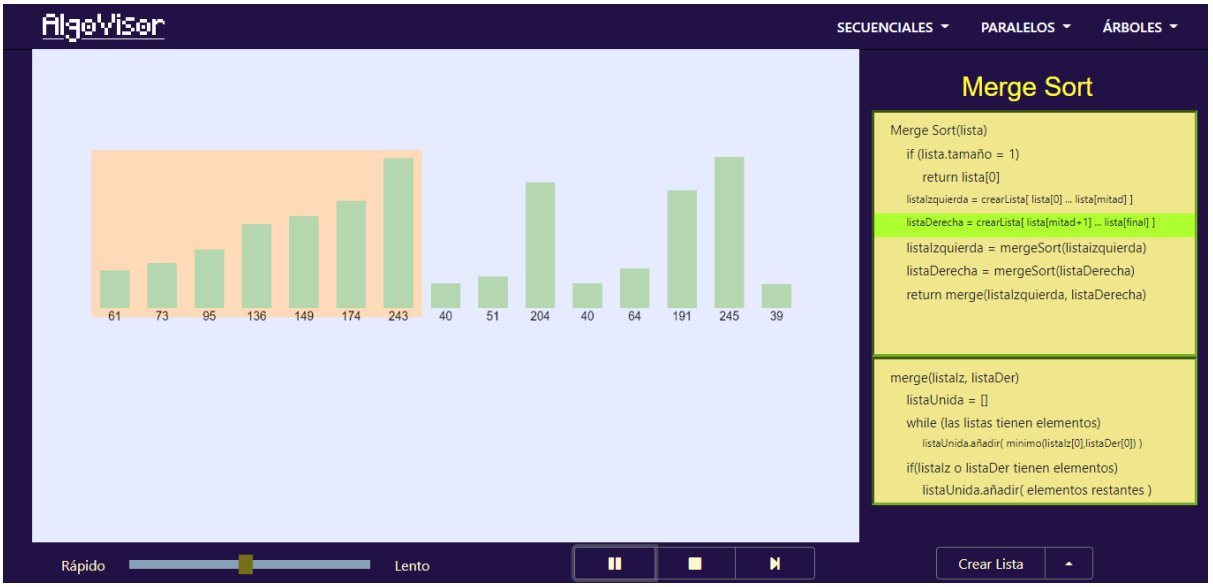
[15] Mahdieh Taher, Seung Hyun Kim, Huseyin Cavusoglu, Atreyi Kankanhalli (2012). *GAMIFICATION: A NEW PARADIGM FOR ONLINE USER ENGAGEMENT*. Thirty third International Conference of Information Systems. Orlando
URL: https://core.ac.uk/download/pdf/301358719.pdf

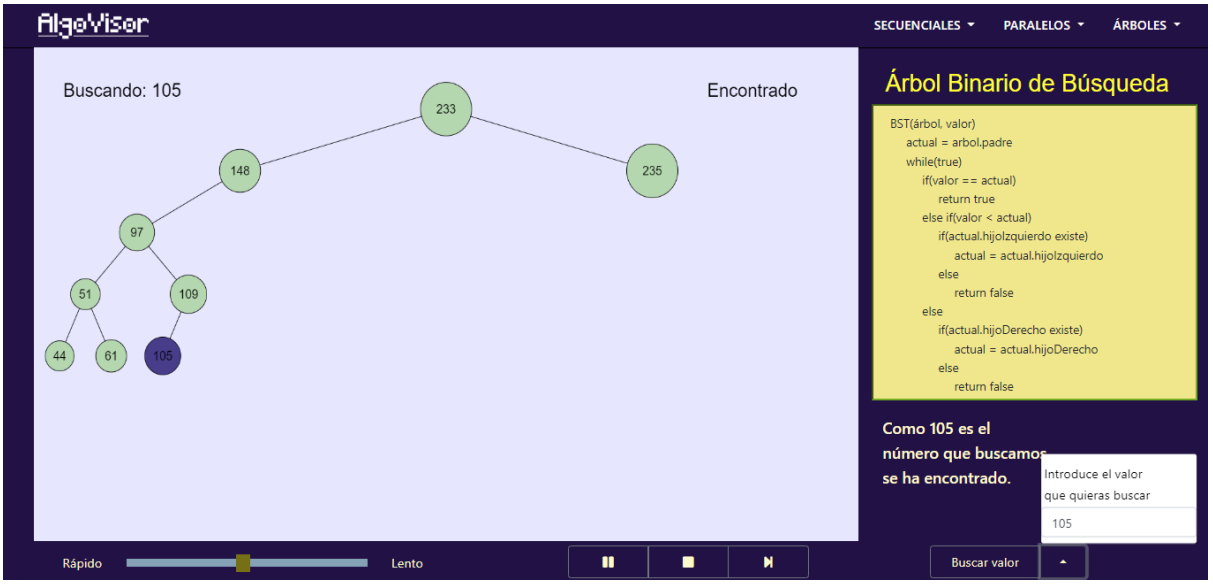
[16] JB O'Hagan, M Khazova, LLa Price (2016) *Low-energy light bulbs, computers, tablets and the blue light hazard*. Centre for Radiation, Chemical and Environmental Hazards, Public Health England. Macmillan Publishers Limited
URL: https://www.nature.com/articles/eye2015261.pdf

8. Anexo


AlgoVisor
SECUENCIALES
PARALELOS
ÁRBOLES
Nodos visitados: 170, 50, 187, 203, 154, 217, 76, 170, 120
11
76
128
154
217
120
38
203
170
187
50
170
DFS
DFS(árbol)
visitados.añadir(árbol.primerNodo)
if(visitados[0].hijoIzquierdo existe)
visitados.añadir(
DFS(subÁrbol(árbol.primerNodo.hijoIzquierdo))
)
if(visitados[0].hijoDerecho existe)
visitados.añadir(
DFS(subÁrbol(árbol.Primernodo.hijoDerecho))
)
return visitados
explora el hijo
derecho de 128
Rápido
Lento


AlgoVisor
SECUENCIALES
PARALELOS
ÁRBOLES
Bubble Sort
Bubble Sort(lista)
for i=0 hasta lista.tamaño - 1
for j=0 hasta lista.tamaño - 1 - i
if(lista[j] > lista[j+1])
intercambia(lista[j] y lista[j+1])
37 46 64 102 142 28 37 165 166 73 11 180 206 254 295
Compara el valor seleccionado 142 con
el siguiente valor 28.
La comprobación es cierta.
Rápido
Lento
Crear Lista


AlgoVisor
SECUENCIALES
PARALELOS
ÁRBOLES
61 73 95 136 149 174 243 40 51 204 40 64 191 245 39
Merge Sort
Merge Sort(lista)
if (lista.tamaño = 1)
return lista[0]
listaIzquierda = crearLista[lista[0] ... lista[mitad]]
listaDerecha = crearLista[lista[mitad+1] ... lista[final]]
listaIzquierda = mergeSort(listaizquierda)
listaDerecha = mergeSort(listaDerecha)
return merge(listaIzquierda, listaDerecha)
merge(listaIz, listaDer)
listaUnida = []
while (las listas tienen elementos)
listaUnida.añadir(minimo(listaIz[0],listaDer[0]))
if(listaIz o listaDer tienen elementos)
listaUnida.añadir(elementos restantes)
Rápido
Lento
Crear Lista


AlgoVisor
SECUENCIALES
PARALELOS
ÁRBOLES
Buscando: 105
Encontrado
233
148
235
97
51
109
44
61
105
Árbol Binario de Búsqueda
BST(árbol, valor)
actual = arbol.padre
while(true)
if(valor == actual)
return true
else if(valor < actual)
if(actual.hijoIzquierdo existe)
actual = actual.hijoIzquierdo
else
return false
else
if(actual.hijoDerecho existe)
actual = actual.hijoDerecho
else
return false
Como 105 es el número que buscamos se ha encontrado.
Introduce el valor que quieras buscar
105
Rápido
Lento
Buscar valor


AlgoVisor
SECUENCIALES
PARALELOS
ÁRBOLES
20 200 25 250 30 300
Selection Sort
SelectionSort(lista)
for j = 0 hasta última posición de lista
mínimo = j
for i = j + 1 hasta última posición de lista
if (lista[i] < lista[mínimo])
mínimo = i
if(mínimo != j)
intercambiar(lista[j], lista[mínimo])
Ordena los elementos de la lista recorriendo todas las posiciones empezando por la de índice más bajo y acabando en indice, haciéndolos
Introduce valores separados por comas
20,200,25,250,30,300
Rápido
Lento
Crear Lista