Módulo 6 - Algoritmos II - Teoría

Contenido:

Hasta ahora hemos visto la forma que tienen los algoritmos, sus distintas partes y como abordarlos, entendiendo un algoritmo como un problema complejo que requiere división en partes. Sin embargo, en programación existen pequeños algoritmos que resuelven tareas muy frecuentes y que se repiten una y otra vez en multitud de escenarios y problemas a los que nos enfrentaremos a lo largo de nuestra carrera. En este módulo daremos una breve introducción a dichos algoritmos y a sus versiones más conocidas y útiles.

1. Generación aleatoria

Un problema de gran utilidad que necesitaremos resolver en numerosas ocasiones es la generación aleatoria de números. Hay multitud de procesos que requieren escoger aleatoriamente números: piensa en un casino, en un juego de cartas, en un dado, en la lotería, etc.

Para generar números aleatorios contamos con una potente herramienta en Javascript, el método `random` que pertenece a la librería `Math` de utilidades matemáticas que nos proporciona el lenguaje.

Cada llamada a `random` genera números aleatorios decimales dentro del rango `[0, 1)`. Esto significa, desde el 0 incluido hasta el 1 pero sin incluirlo:

```
console.log(Math.random()); // 0.34310475582803157
console.log(Math.random()); // 0.06647250907121971
console.log(Math.random()); // 0.17012677610315863
...
```

> IMPORTANTE: `Math.random()` es una utilidad de propósito general, pero su resultado no es 100% aleatorio sino pseudo-aleatorio. NO provee números aleatorios con seguridad criptográfica por lo que no debe ser usada para algo relacionado con seguridad. A tal efecto existe un conjunto de utilidades de criptografía que se encuentra en el objeto `window.crypto`.

A. Generación binaria

Una de las formas más sencillas de generación aleatoria se trata cuando hay que escoger entre 2 posibles opciones o valores. Piensa por ejemplo en el "rojo" o "negro" de una ruleta o entre "niño" o "niña" en una embarazada. En definitiva, escoger entre 2 posibilidades al 50%.

¿Como implementar un generador aleatorio binario a partir de `Math.random()`? Sencillo. Todos los resultados que entrega `random` están siempre acotados entre 0 y 1, por tanto, si dividimos ese rango en 2, tendremos probabilidades al 50% que es lo que buscamos. Es decir:

- Probabilidad de que el resultado sea menor que `0.5` --> 50%
- Probabilidad de que el resultado mayor o igual a `0.5` --> 50%

```
0                   0.5                   1
[--------------------|--------------------)
```

Por tanto:

```
var getRandom = (a, b) => (Math.random() < 0.5 ? a : b);

console.log(getRandom("rojo", "negro")); // negro
console.log(getRandom("rojo", "negro")); // rojo
console.log(getRandom("rojo", "negro")); // rojo
console.log(getRandom("rojo", "negro")); // negro
```

Otra alternativa sería utilizar la técnica del redondeo, por lo que los posibles valores resultantes serían `0` o `1`. Por ejemplo:

```
var getRandom = (a, b) => (Math.round(Math.random()) ? a : b);

console.log(getRandom("niño", "niña")); // niña
console.log(getRandom("niño", "niña")); // niña
console.log(getRandom("niño", "niña")); // niña
console.log(getRandom("niño", "niña")); // niño
```

B. Múltiples opciones

Un caso más general de generación aleatoria consiste en elegir entre múltiples opciones con igualdad de probabilidades.

CASO SENCILLO

Empecemos por un caso sencillo, por ejemplo, sacar un número aleatorio del 0 al 9.

Recordemos que `Math.random()` nos ofrece este rango de resultados:

```
Math.random(); // Números aleatorios decimales en el rango [0, 1)
```

```
0                   1
[--------------------)
```

¿Cuantos números enteros hay entre el 0 y el 9 incluyendo ambos? Si los contamos veremos que son 10 números. Si multiplicamos por 10:

```
Math.random() * 10; // Números aleatorios decimales en el rango [0, 10)
```

```
0    1    2    3    4    5    6    7    8    9   10
[----|----|----|----|----|----|----|----|----|----)
```

Podemos clasificar los resultados decimales que nos dará `Math.random() * 10` de modo que:

- Todos los resultados entre 0 y 1 los clasificamos como 0.
- Todos los resultados entre 1 y 2 los clasificamos como 1.
- Todos los resultados entre 2 y 3 los clasificamos como 2.
- etc.

Es decir, aproximamos siempre al entero más cercano por debajo. A esto se le llama `floor` y tenemos una función en `Math` para ello:

```
Math.floor(Math.random() * 10); // Números aleatorios enteros en el rango [0, 9]
```

CASO GENERAL

Ahora un caso cuyo mínimo no sea un 0. En lugar del 0 al 9, busquemos un aleatorio entre 50 y 100 por ejemplo.

¿Cuantos números hay entre 50 y 100 incluidos ambos? 51, o lo que es lo mismo, el rango, que se calcula como la diferencia 100 - 50 más 1 (porque están incluidos ambos extremos).

Aplicando la fórmula anterior tendremos números aleatorios enteros entre 0 y 50

```
const range = 100 - 50 + 1;
Math.floor(Math.random() * range); // Números aleatorios enteros en el rango [0, 50]
```

Nos faltaría por tanto, sumar a nuestro resultado el mínimo que queremos, que en nuestro caso es 50, por tanto:

```
Math.floor(Math.random() * range) + 50; // Números aleatorios enteros en el rango [50, 100]
```

Por tanto la fórmula general podría expresarse en pseudo código como:

```
const range = max - min + 1;
Math.floor(Math.random() * range) + min; // Números aleatorios enteros en el rango [min, max]
```

C. Distintas probabilidades

Ver ejercicio práctico "Dado trucado".

2. Algoritmos de Búsqueda

A. Búsqueda Secuencial o Lineal

Es el algoritmo más sencillo de búsqueda posible y que todos sabréis aplicar sin dificultad dado un array de elementos desordenados. En este algoritmo, iteramos de forma secuencial, elemento a elemento, desde el primero hasta el último, por todos los elementos del array hasta encontrar el resultado dado. Una vez encontrado, devolvemos su posición en el array, o bien `-1` en caso de no encontrarlo:

```
var array = [1, 3, 4, 5, 2, 9, 6, 7, 8];

var search = (array, target) => {
  for (var i = 0; i < array.length; i++) {
    if (array[i] === target) return i;
  }
  return -1;
};

console.log(search(array, 2)); // 4
```

En Javascript contamos con una utilidad en los arrays para hacernos la vida más fácil que ya implementa esta búsqueda binaria. Se trata del método `indexOf()`, que acepta como argumento el elemento a buscar y nos devuelve el índice del **primer elemento** que encuentre, si es que lo encuentra, o `-1` en caso de no encontrarlo:

```
var array = [1, 3, 4, 5, 2, 9, 6, 7, 8];

console.log(array.indexOf(2)); // 4
```

Recuerda que, en caso de elementos duplicados, devuelve siempre el primero.

Este método también está disponible en los strings, ya que no dejan de ser un array de carácteres:

```
var myString = "casa";
myString.indexOf("a"); // 1
```

B. Búsqueda Binaria

Sin embargo, cuando tenemos la certeza que nuestro array está ordenado, existe una forma más eficiente de buscar elementos y se llama búsqueda binaria. Este algoritmo se basa en la idea de que si los elementos están ordenados nos da una pista de por dónde debe ubicarse un elemento dado. Por ejemplo, en un array como este:

```
[12, 14, 23, 25, 30, 36, 43, 62, 67, 72, 88];
```

Para buscar el elemento 72 no hace falta empezar por el principio. Por contra, si buscamos el 14, sí que sabemos que debe estar al comienzo del array. Podríamos hacer lo siguiente:

1. Tomar el punto medio como referencia, el `36` y preguntarnos si lo que buscamos es mayor o menor.
2. Si es mayor, tendré que buscar a la derecha,
3. Si es menor a la izquierda.
4. Si es igual, ya lo he encontrado.

Si repito en bucle hasta encontrarlo, me acercaré a la solución final más rápido que buscando de forma lineal.

Por favor, abre en tu navegador el siguiente enlace para ver un GIF explicativo de la búsqueda binaria vs búsqueda secuencial: https://dojo.stuycs.org/resources/_images/binary-and-linear-search.gif

Por tanto, en pseudocódigo, nuestro algoritmo quedaría como:

1. Buscar los índices máximos y mínimos del array como `min = 0` y `max = length - 1`.
2. Computar el punto medio como la media entre min y max redondeada hacia abajo (debe ser un índice entero).
3. Si `array[punto medio] === target`, paramos el algoritmo, lo hemos encontrado.
4. Si `array[punto medio] < target`, movemos el mínimo `min = punto medio + 1`.
5. En otro caso, movemos el máximo a `max = punto medio - 1`.
6. Volvemos al `paso 2`.

Y finalmente, su implementación:

```
var binarySearch = (array, target) => {
  var min = 0;
  var max = array.length - 1;
  while (min <= max) {
    var mid = min + Math.floor((max - min) / 2);
    if (array[mid] === target) return mid;
    else if (array[mid] < target) min = mid + 1;
    else max = mid - 1;
  }
  return -1;
};

var sortedArray = [11, 15, 32, 34, 36, 37, 75, 79, 80, 89];
console.log(binarySearch(sortedArray, 34));
```

3. Algoritmos de ordenación

Los algoritmos de ordenación nos sirven para ordenar una lista o colección de elementos donde éstos están colocados al azar o simplemente desordenados.

A pesar de que existen muchas formas de abordar este problema, cada una con sus ventajas e inconvenientes, vamos a introducir tres de estas técnicas de ordenación, las más conocidas. Serán por tanto 3 algoritmos que resuelven el mismo problema.

Para nuestros ejemplos, partiremos siempre de la misma lista desordenada de 6 elementos:

```
array = [3, 5, 1, 8, 7, 2];
```

De manera que iremos aplicando cada una de las técnicas al array anterior para que nos devuelva como resultado la misma lista, pero con sus elementos ordenandos de **menor a mayor**. Con una simple modificación, también se podrían ordenar en sentido contrario, de mayor a menor.

A. Bubble Sort u ordenación por burbuja

En el método por burbuja, iremos tomando grupos de elementos de dos en dos, preguntando cuál de esos dos valores es mayor.

Es decir, en nuestra la lista de ejemplo `[3, 5, 1, 8, 7, 2]`:

- Tomaremos la primera pareja, el grupo `[3, 5]`. Vemos que el 3 es menor que 5, por lo que no hay que hacer ningún cambio. El orden dentro de la pareja es correcto, de forma que quedaría `[3, 5, 1, 8, 7, 2]`.
- Repetimos el proceso con la segunda pareja (segundo y tercer elemento), es decir, `[5, 1]`. Como 5 es mayor que 1 debemos intercambiar los valores para que queden bien ordenados de menor a mayor. Ahora la lista quedaría `[3, 1, 5, 8, 7, 2]`.
- Seguimos con la siguiente pareja, el tercer y cuarto elemento, `[5, 8]`. Tampoco hay que hacer nada ya que el orden es correcto. Obtenemos `[3, 1, 5, 8, 7, 2]`.
- La siguiente pareja es `[8, 7]` que necesita hacer intercambio para respetar el orden, y nos da por tanto `[3, 1, 5, 7, 8, 2]`
- Finalmente, la última pareja `[8, 2]` a la que también hay que aplicar un intercambio, dando como resultado final de la iteración `[3, 1, 5, 7, 2, 8]`.

Así, habremos terminado la primera iteración, sin embargo la lista todavía no está ordenada. Si nos fijamos, lo que hemos conseguido con la primera iteración es llevar hacia la derecha el elemento más grande de nuestra lista, el `8`. Con cada iteración vamos colocando un solo elemento en su posición definitiva.

Si volvemos a dar otra iteración sobre nuestro array siguiendo el mismo proceso, nos quedaría como resultado `[1, 3, 5, 2, 7, 8]`. Hemos colocado el 7 en su posición final.

Si continuamos repitiendo este proceso tantas veces como elementos hay en el array, acabaremos con todos los elementos ordenados de menor a mayor: `[1, 2, 3, 5, 7, 8]`

Implementación

```
var swap = (array, a, b) => {
    // Intercambiamos el contenido de los índices a y b
    var temp = array[a];
    array[a] = array[b];
    array[b] = temp;
};

var bubbleSort = array => {
    var size = array.length; // Calculamos su tamaño

  // Bucle externo
    for (var index = 1; index < size; index++) {
    // Bucle interno
        for (var left = 0; left < size - index; left++) {
            var right = left + 1;
            if (array[left] > array[right]) {
                swap(array, left, right);
            }
        }
    }

    return array; // Devuelve el array ordenado
};

console.log(bubbleSort([3, 5, 1, 8, 7, 2]));
```

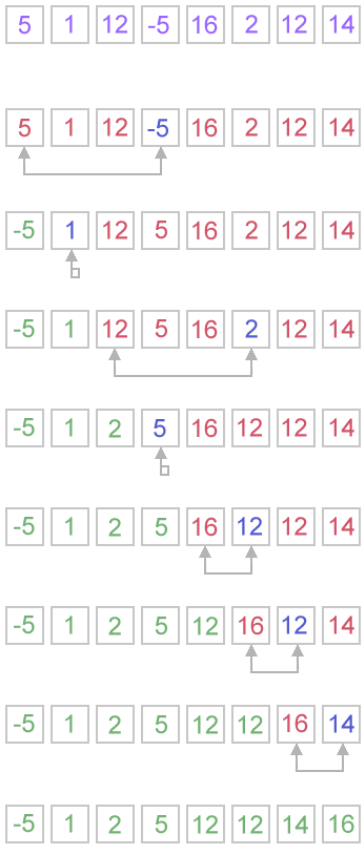
Pasamos un array a nuestra función y dentro de ella, lo primero que hacemos es calcular su tamaño, el número de elementos del array.

Después creamos un bucle externo que recorra nuestro array tantas veces como elementos tenga menos uno, ya que son las veces máximas necesarias para que quede completamente ordenado en el peor caso.

Internamente, creamos otro bucle que compara en parejas, cada valor con el siguiente. Si el de la izquierda es mayor que el de la derecha, los intercambia con la función `swap`, donde `a` y `b` son los índices del primer y segundo elemento a intercambiar.

Por último, nos devuelve el array ordenado.

B. Selection Sort u ordenación por selección



Este algoritmo consiste en buscar el mínimo elemento entre una posición i y el final de la lista.

Es decir, en nuestra la lista de ejemplo `[3, 5, 1, 8, 7, 2]`:

- Se empieza por recorrer la lista desde el índice 0 hasta encontrar el menor elemento. En este caso el menor elemento es el 1. De manera que se intercambia con el dato que están en la primera posición, el 3, quedando la lista como `[1, 5, 3, 8, 7, 2]`.
- En la siguiente iteración, se comienza recorriendo la lista desde el índice 1, por lo que se encuentra el elemento 2 que se intercambia por el elemento que está en la segunda posición, el 5. Quedando la lista `[1, 2, 3, 8, 7, 5]`.
- En la siguiente iteración, se comenzaría desde el índice 2 de la lista y vemos que está el elemento 3. Como es el menor elemento de la lista no se realiza ningún cambio.
- Continuamos con la iteración, ahora empezaríamos desde el índice 3 y encontramos como menor elemento el número 5, que se cambiaría por el elemento de la cuarta posición: `[1, 2, 3, 5, 7, 8]`.
- Continuamos iterando y como están de menor a mayor, no se producirán más cambios. La lista está ordenada.

Implementación

```
var swap = (array, a, b) => {
  // Intercambiamos el contenido de los índices a y b
  var temp = array[a];
  array[a] = array[b];
  array[b] = temp;
};

var selectionSort = array => {
  var minIndex;
  var size = array.length; // Calculamos su tamaño;
  for (var s = 0; s < size; s++) {
    // s => selection
    // Bucle externo.
    minIndex = s;
    for (var i = s + 1; i < size; i++) {
      // Bucle interno.
      if (array[i] < array[minIndex]) minIndex = i;
    }
```

```
      swap(array, minIndex, s);
    }
    return array;
};

console.log(selectionSort([3, 5, 1, 8, 7, 2]));
```

C. Insertion Sort u ordenación por inserción

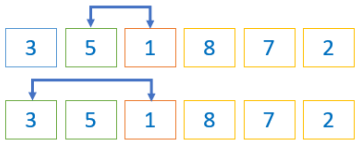
Este algoritmo consiste en comparar un elemento con todos los ubicados a su izquierda e ir realizando cambios hasta que encuentre su posición en la lista.

> Por favor, abre en tu navegador el siguiente enlace para ver un GIF explicativo de la ordenación por inserción: https://upload.wikimedia.org/wikipedia/commons/9/9c/Insertion-sort-example.gif

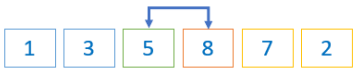
Es decir, en nuestra la lista de ejemplo `[3, 5, 1, 8, 7, 2]`:

- Se selecciona el segundo valor como clave, en este caso el 5, y se compara con los valores a su izquierda, es decir el 3. Como es mayor, no ocurre ningún cambio.

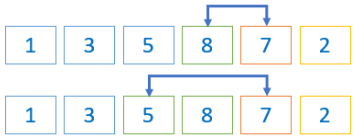
- Se selecciona el siguiente número como clave, el 1, se compara con los valores a su izquierda hasta que encuentre un elemento menor que él o finalice y se inserta en el lugar correspondiente: `[1, 3, 5, 8, 7, 2]`.



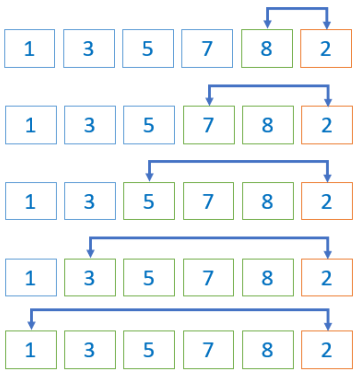
- Se selecciona el siguiente número como clave, el 8 y se repite el proceso para todos los valores anteriores. Como el 8 es mayor, no se produce ningún cambio.



- En la siguiente iteración, se selecciona el 7 y se compara con todos los valores de su izquierda hasta que encuentre uno menor que él para insertarse en el lugar correspondiente quedando `[1, 3, 5, 7, 8, 2]`.



- Finalmente, se selecciona como clave el siguiente valor, el 2, y se compara con todos los valores a su izquierda realizando cambios hasta que encuentra uno menor que él (el 1).



- Quedando la lista ordenada:

Implementación

```
var swap = (array, a, b) => {
  // Intercambiamos el contenido de los índices a y b
  var temp = array[a];
  array[a] = array[b];
  array[b] = temp;
};

var insertionSort = array => {
  var size = array.length; // Calculamos su tamaño.
  var sortedListLastIndex = 0;
  // Bucle externo.
  for (var item = 1; item < size; item++) {
    var current = array[item];
    var currentIndex = item;
    var swapIndex = sortedListLastIndex;
    // Bucle interno.
    while (current < array[swapIndex] && swapIndex >= 0) {
      swap(array, currentIndex--, swapIndex--);
    }
    sortedListLastIndex++;
  }

  return array;
};

console.log(insertionSort([3, 5, 1, 8, 7, 2]));
```