

Técnicas de Programación

CFP Programador full-stack

Recursividad (Repaso)

Recursión

Repaso

- Permite que un método se invoque a si mismo para realizar una determinada tarea
- Siempre hay una condición que debe cortar la recursión



Recursión

Repaso

- **Ventajas**

- Soluciona problemas recurrentes
- Permite solucionar problemas complejos con pocas líneas de código

- **Desventajas**

- Puede ser difícil de entender el código
- Produce excesivas demandas de memoria o tiempo de ejecución



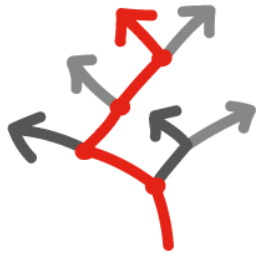
Técnicas de Programación

CFP Programador full-stack

Algoritmos Básicos (Conceptos)

Algoritmos Básicos

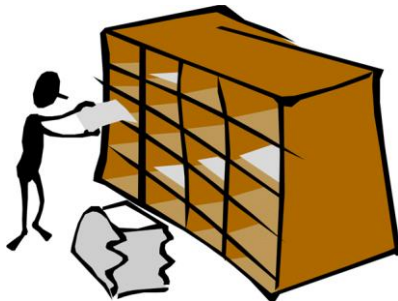
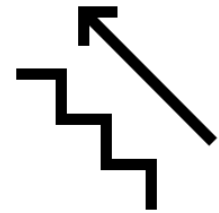
Muchas aplicaciones requieren contar con métodos básicos para brindar funcionalidad útil y de valor:



Recorrido



Búsqueda



Ordenamiento



Algoritmos Básicos

Ranking de Facturación

- Una empresa quiere saber quienes fueron los clientes que más facturaron en un mes
- let dos arreglos, uno para los nombres de los clientes y otro para los montos de facturación (enteros)
- La cantidad de clientes es fija (10)
- Mostrar por pantalla los 5 clientes que más facturaron y los montos
- Pensar en como cargar la información para facilitar la escritura del ranking



Algoritmos Básicos

Ranking de Facturación



```
let readlineSync = require('readline-sync');

let cantidad = 10;
let clientes = new Array(cantidad);
let facturacion = new Array(cantidad);
//Carga ordenado, uno por uno
console.log ("Cargando los arreglos de forma ordenada" );
let cliente;
let fact;
let numCliente;
let i, j;
```

Algoritmos Básicos

Ranking de Facturación



```
for (numCliente = 0; numCliente < cantidad; numCliente++) {
```

```
    cliente = readlineSync.question("Cliente ", numCliente + 1, ": ");
```

```
    fact = readlineSync.question("Facturacion ", numCliente + 1, ": ");
```

```
    i = 0
```

```
    while (i < numCliente && facturacion[i] > fact) {  
        i++;
```

```
    }
```

```
    for (j = numCliente; j <= i; j++) {  
        clientes[j] = clientes[j-1];  
        facturacion[j] = facturacion[j-1];  
    }
```

```
    clientes[i] = cliente ;  
    facturacion[i] = fact ;
```

```
}
```

Busco la posición donde tengo que agregar al cliente en los arreglos según su facturación

Corro los elementos desde el ultimo cliente agregado hasta la posición donde lo tengo que insertar

Agrego al cliente y su facturación en la posición que mantiene el orden de los arreglos

Algoritmos Básicos

Ranking de Facturación



Para mostrar el ranking, solamente tengo que recorrer el arreglo de 0 a 4 porque los clientes están ordenados por facturación

```
for (posicion = 0; posicion < 4; posicion++) {  
    console.log ("("posicion,") ",clientes[posicion],  
                "["facturacion[posicion],"] ");  
}
```

Algoritmos Básicos

Ranking de Facturación

- Y si tenemos los arreglos ya cargados?
- Podemos aplicar la técnica de insertar ordenado?
- Que pasa si los datos vienen desordenados y no podemos hacer nada para cambiar eso?

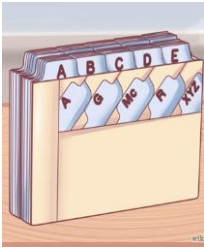


Algoritmos de Ordenamiento

Objetivo y Alternativas



- Permiten dar un orden a los elementos de una estructura, por ejemplo:



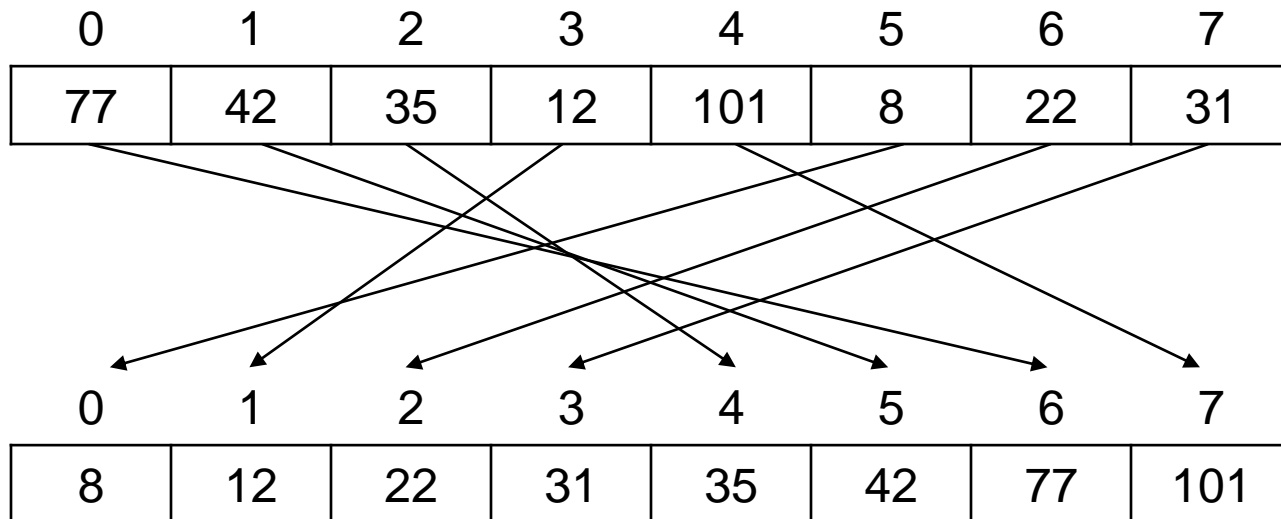
- Orden alfabético descendente (de la Z a la A)
- Orden numérico ascendente (0 a infinito)
- Existen diferentes variantes, que dependen de su complejidad temporal y espacial, así también de su simplicidad a la hora de programar

Algoritmos de Ordenamiento

Lineamientos del Código

↓
A
Z

- Tienen como **entrada** una **estructura** (arreglo)
- Tienen como **salida** la misma **estructura ordenada**
- Saben como **comparar** e **intercambiar** los elementos



Algoritmos de Ordenamiento

Tipos de Algoritmos



- Pueden ser iterativos o recursivos
- Pueden tardar **más o menos** según:
 - La cantidad de veces que recorren la estructura
 - La cantidad de comparaciones que hacen
 - La cantidad de veces que intercambian valores
- Clasificados por su desempeño promedio, el mejor y el peor caso
- Algoritmos:
 - Burbuja (bubble-sort)
 - Selección (selection-sort)
 - Mezclado (merge-sort)
 - Rápido (quick-sort)
 - Muchos más...

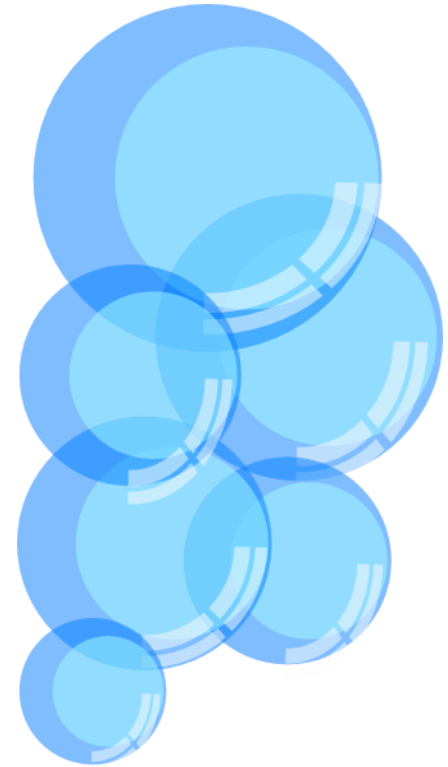


Algoritmos de Ordenamiento

Burbuja (bubble-sort)



- Se **comparan** los elementos **adyacentes** y se simula un burbujeo, donde las burbujas más grandes se cambian con las más chicas
- Se **intercambian** los elementos solamente si los elementos no están en el **orden incorrecto**
- Es uno de los algoritmos de ordenamiento más **simples** de programar porque solo hace **comparaciones** entre **vecinos**



Algoritmos de Ordenamiento

Burbuja (video)



<https://www.youtube.com/watch?v=lyZQPjUT5B4>

Algoritmos de Ordenamiento

Burbuja (razonamiento)



Como se codifica:

- Dos bucles (con índices i y j)
- El primero itera la cantidad de veces que tenemos que burbujear
- El segundo delimita desde donde empieza y donde termina el burbujeo
- El burbujeo consiste en comparar $a[j]$ y $a[j + 1]$ y darlos vuelta si corresponde
- Tener en cuenta a medida que burbujeamos los elementos al final del arreglo empiezan a estar ordenados

Algoritmos de Ordenamiento

Burbuja (código)



Este método permite cargar un arreglo “arreglo” de dimensión “cantidad” y llenarlo de valores generados al azar entre 0 y “numAzar” (parámetro)

```
function cargar(arreglo, cantidad, numAzar)
    let i;
    for (i = 0 ; i<cantidad; i++ ) {
        arreglo[i] = Azar(numAzar);
    }
}
```

Algoritmos de Ordenamiento

Burbuja (código)



Este método permite mostrar un arreglo “arreglo” de dimensión “cantidad” en una única línea, separando los valores con un espacio

```
function escribirEnUnaLinea(arreglo, cantidad) {  
    let i;  
    for (i = 0 ; i<cantidad; i++) {  
        console.log arreglo[i], " ";  
    }  
    console.log '\n';  
}
```

Algoritmos de Ordenamiento

Burbuja (código)



Este método permite intercambiar los valores en las posiciones “i” y “j” de un arreglo “arreglo” utilizando una variable auxiliar

```
function intercambiar(arreglo, i, j) {  
    let aux;  
    aux = arreglo[i] ;  
    arreglo[i] = arreglo[j] ;  
    arreglo[j] = aux ;  
}
```

Algoritmos de Ordenamiento

Burbuja (código)



Este método permite comparar los valores en las posiciones “i” y “j” del arreglo “arreglo”

Devuelve 0 si son iguales, 1 si lo que hay en “i” es mayor a lo que hay en “j” y -1 si lo que hay en “i” es menor a lo que hay en “j”

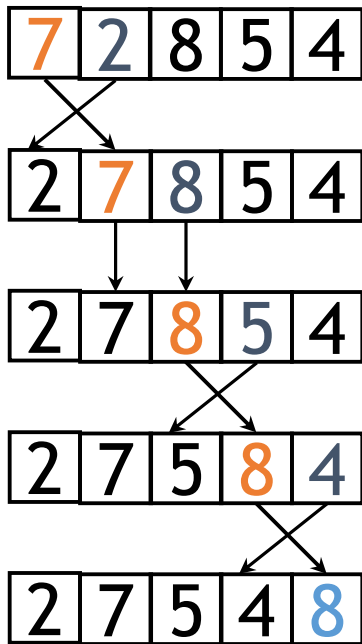
```
function comparar(arreglo, i, j) {  
  let comparacion;  
  if (arreglo[i] == arreglo[j]) {  
    comparacion = 0;  
  } else if (arreglo[i] < arreglo[j]) {  
    comparacion = -1;  
  } else {  
    comparacion = 1;  
  }  
  return comparacion;  
}
```

Algoritmos de Ordenamiento

Burbuja (ejemplo)

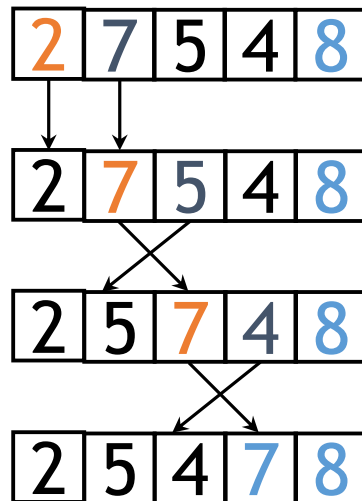


$i=2$



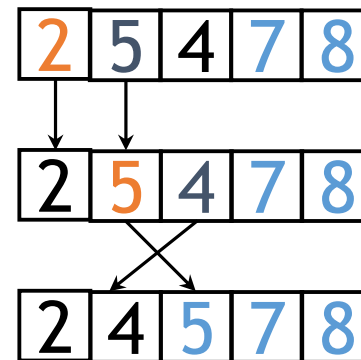
$j=3 \quad j+1=4$

$i=3$



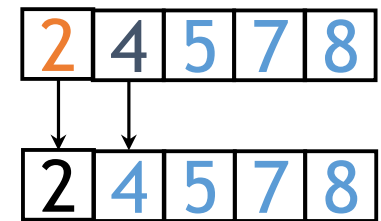
$j=2 \quad j+1=3$

$i=4$



$j=1 \quad j+1=2$

$i=5$



$j=0 \quad j+1=1$

$N=5$

1er Ciclo: $i = 2$ Hasta 5

2do Ciclo: $j = 0$ Hasta $5-i$

Algoritmos de Ordenamiento

Burbuja (código)



```
function burbuja(arreglo, cantidad)
```

```
  let i, j;
```

```
  for (i = 2 ; i < cantidad; i++) {
```

```
    for (j = 0 ; j < cantidad; j++) {
```

```
      if (comparar(arreglo, j, j+1) == 1) {
```

```
        intercambiar(arreglo, j, j+1);
```

```
      }
```

```
    }
```

```
  }
```

```
}
```

Desde 2 hasta n (el primer elemento esta ordenado en la ultima vuelta)

Desde 0 hasta n – i (vamos achicando el rango a medida que se ubican los valores al final del arreglo)

Si los adyacentes j y j + 1 no están ordenados, intercambiarlos

Algoritmos de Ordenamiento

Burbuja (código)



//Algoritmo Orden

```
let lim = 10;  
let a = new Array(lim);  
cargar(a, lim, 100);  
escribirEnUnaLinea(a, lim);  
burbuja(a, lim);  
escribirEnUnaLinea(a, lim);
```

Algoritmos de Ordenamiento

Burbuja (eficiencia)



- Complejidad: n^2 (dos loops)



- Mejor caso: todo ordenado de antemano



- Peor caso: ordenado en sentido inverso

Algoritmos de Ordenamiento

Selección (selection-sort)

- Permite **ordenar** un estructura de forma **natural**
- Funciona **buscando** el elemento que corresponde en una ubicación y moviéndolo al **lugar correcto** (es decir, ordenado)
- Ejemplo para orden ascendente:
 - se localiza el mínimo de un arreglo y se lo coloca en el primer lugar
 - se localiza el segundo mínimo y se lo coloca en el segundo,
 - y así hasta que no queden elementos que colocar
- Es **ligeramente mejor** que “burbuja” porque **intercambia menos** valores





25

Algoritmos de Ordenamiento

Selección (razonamiento)



Como se codifica:

- Dos bucles (con índices i y j)
- El primero itera por la cantidad de elementos en el arreglo, y el índice i denota la posición que se esta buscando ordenar
- El segundo delimita las posiciones que todavía no han sido ordenadas
- Se busca el mínimo/máximo valor en el arreglo en el rango del segundo bucle (índice j)
- Al terminar el segundo bucle, intercambiamos lo que haya en la índice i con lo que haya en la posición con el valor mínimo/máximo

Algoritmos de Ordenamiento

Selección (código)



```

function seleccion(arreglo, cantidad) {
  let i, j, posicion;
  for (i = 0; i < cantidad - 1; i++) {
    posicion = i;
    for (j = i + 1; j < cantidad - 1; j++) {
      if (comparar(arreglo, posicion, j) == 1) {
        posicion = j;
      }
    }
    intercambiar(arreglo, i, posicion);
  }
}
  
```

Desde 0 hasta $n-2$ (el ultimo elemento queda ordenado al final del ciclo)

Desde $i+1$ hasta $n-1$ (vamos moviendo el rango izquierdo a medida que se ubican los valores al comienzo del arreglo)

Si el valor en el índice “j” es menor/mayor que el que hay en “posicion”, actualizar “posicion” con “j”

Una vez que encontré el valor en el índice “posicion” que corresponde en el índice “i”, intercambiarlos

Algoritmos de Ordenamiento

Selección (código)



```
//Algoritmo Orden  
let lim = 10;  
let a = new Array(lim);  
cargar(a, lim, 100);  
escribirEnUnaLinea(a, lim);  
//burbuja(a, lim)  
seleccion(a, lim);  
escribirEnUnaLinea(a, lim);
```

Algoritmos de Ordenamiento

Selección (eficiencia)



- Complejidad: n^2 (dos loops)



- Mejor y peor caso: siempre hace la misma cantidad de comparaciones



Algoritmos de Búsqueda

Búsqueda de Clientes y Facturación

- La misma empresa ahora quiere buscar la facturación de un cliente dado
- Leer el nombre del cliente que interesa
- Pensar como optimizar la búsqueda con el ordenamiento



Algoritmos de Búsqueda

Búsqueda de Clientes y Facturación



Ya conocemos la búsqueda tradicional:

```
function buscarTradicional (valorBuscado, a, lim) {  
  let i;  
  let posicion = -1;  
  i = 0;  
  while (i < lim - 1 && posicion == -1) {  
    if (a[i] == valorBuscado) {  
      posicion = i;  
    }  
    i++;  
  }  
  return posicion;  
}
```

Asumo que no lo encuentre
(posicion = -1)

Recorro el arreglo mientras que
queden más elementos y no lo
haya encontrado

Cuando encuentro el
valor buscado, registro
su índice en la variable
posición



Algoritmos de Búsqueda

Búsqueda de Clientes y Facturación



Agregamos algunos métodos para organizar mejor el programa

```
function cargarClientes(clientes, facturacion, cantidad) {  
  let cliente, fact, numCliente;  
  for (numCliente=0; numCliente<cantidad; numCliente++) {  
    cliente = readlineSync.question("Cliente ", numCliente + 1, ": ");  
    fact = readlineSync.question("Facturacion ", numCliente + 1, ": ");  
    clientes[numCliente] = cliente;  
    facturacion[numCliente] = fact;  
  }  
}
```



Algoritmos de Búsqueda

Búsqueda de Clientes y Facturación



Agregamos algunos métodos para organizar mejor el programa

```
function imprimirCliente (posicion, buscado, clientes, facturacion) {  
  if (posicion == -1) {  
    "El cliente ", buscado, " no pudo ser encontrado");  
  } else {  
    console.log ("El cliente ", clientes[posicion],  
      " con facturacion ", facturacion[posicion],  
      " esta en la posicion ", posicion);  
  }  
}
```



Algoritmos de Búsqueda

Búsqueda de Clientes y Facturación



```
//Algoritmo Busqueda
```

```
let cantidad = 10;
```

```
let posicion;
```

```
let clientes = new Array(cantidad);
```

```
let facturacion = new Array(cantidad);
```

```
let buscado = "Ale";
```

```
//Busqueda tradicional
```

```
console.log("Busqueda tradicional");
```

```
cargarClientes(clientes, facturacion, cantidad);
```

```
escribirEnUnaLinea(clientes, facturacion, cantidad);
```

```
posicion = buscarTradicional(buscado, clientes, cantidad);
```

```
imprimirCliente(posicion, clientes, facturacion);
```

```
...
```

Que pasa si ordenamos los arreglos por nombre de cliente?

Es necesario mirar todo el arreglo?



Algoritmos de Búsqueda

Búsqueda Binaria

- Conocida como búsqueda de “intervalo medio”
- **Aprovecha la estructura ordenada** para evitar buscar en lugares donde no se encuentra el elemento
- Realiza solamente $\log_2(n)$ comparaciones
- Es **fácil** de programar con **recursión** (usando 2 métodos)

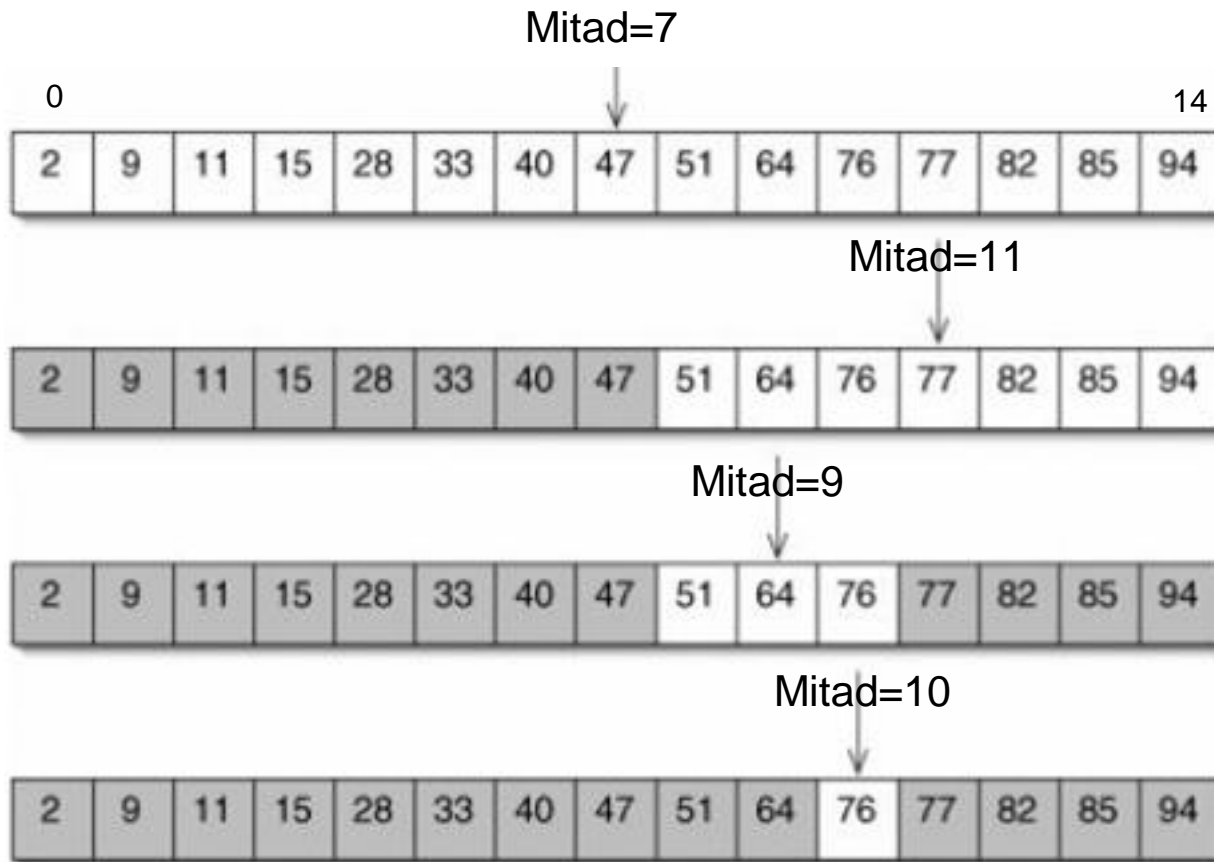
Razonamiento:

- Comienza por comparar el elemento en el centro del arreglo con el valor buscado
- Si el valor buscado es menor, la búsqueda continúa en la primer mitad del arreglo
- Si el valor buscado es mayor, la búsqueda continúa en la segunda mitad del arreglo



Algoritmos de Búsqueda

Búsqueda Binaria



Búsqueda: 76
N: 15

Algoritmos de Búsqueda

Búsqueda de Clientes y Facturación



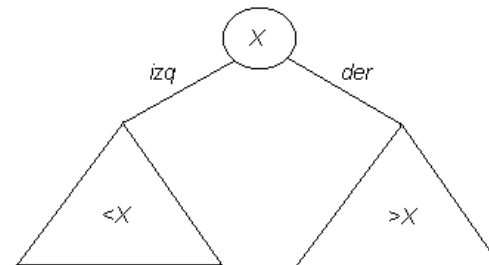
```

function buscarRekursivo(valorBuscado, a, izq, der) {
  let posicion;
  if (izq <= der) {
    let medio;
    medio = Math.floor ((izq + der) / 2);
    if (valorBuscado == a[medio]) {
      posicion = medio;
    } else if (valorBuscado < a[medio]) {
      posicion = buscarRekursivo(valorBuscado, a, izq, medio - 1);
    } else {
      posicion = buscarRekursivo(valorBuscado, a, medio + 1, der);
    }
  } else {
    posicion = -1;
  }
  return posicion;
}
  
```

Busco entre "izq" y "der", cortando cuando los índices se cruzan

Determino el centro del arreglo entre "izq" y "der" y miro si el valor en el medio coincide con lo que busco

Si el valor buscado es menor, buscar en la mitad a la izquierda
En caso contrario buscar en la mitad a la derecha



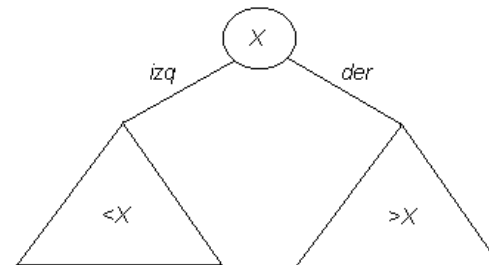
Algoritmos de Búsqueda

Búsqueda de Clientes y Facturación



Creamos un método que tiene la responsabilidad de hacer el primer llamado recursivo con los parámetros correctos

```
function buscarBinario(valorBuscado, a, lim) {  
  let posicion;  
  posicion = buscarRecursivo(valorBuscado, a, 0, lim - 1);  
  return posicion;  
}
```



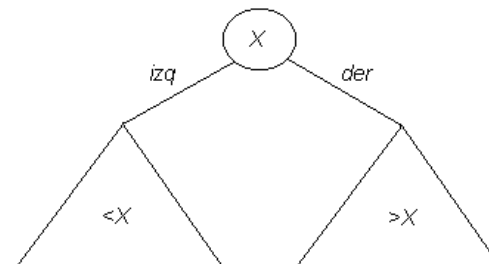
Algoritmos de Búsqueda

Búsqueda de Clientes y Facturación



```
function burbuja(clientes, facturacion, cantidad) {
  let i, j;
  Para (i = 2; i < cantidad; i++) {
    Para (j = 0; j < cantidad - i; j++) {
      if (comparar(clientes, j, j+1) == 1) {
        intercambiar(clientes, j, j+1);
        intercambiar(facturacion, j, j+1);
      }
    }
  }
}
```

Se compara el arreglo de clientes, y se intercambia tanto clientes como facturación



Algoritmos de Búsqueda

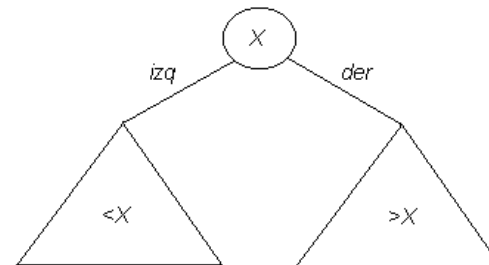
Búsqueda de Clientes y Facturación



En el algoritmo principal podemos agregar la búsqueda binaria de esta forma:

```
console.log("Busqueda mejorada con ordenamiento");  
burbuja(clientes, facturacion, cantidad);  
escribirEnUnaLinea(clientes, facturacion, cantidad);  
posicion = buscarBinario(buscado, clientes, cantidad);  
imprimirCliente(posicion, clientes, facturacion);
```

El ordenamiento burbuja
tiene que ser por facturación?

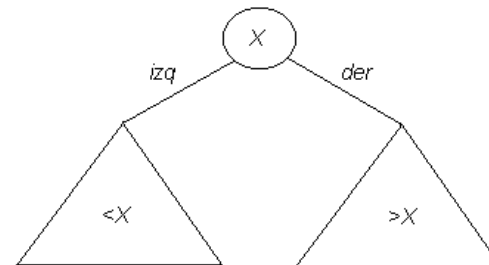


Algoritmos de Búsqueda

Búsqueda de Clientes y Facturación



```
//Algoritmo Busqueda
let cantidad = 10;
let posicion;
let clientes = new Array(cantidad);
let facturacion = new Array(cantidad);
//Busqueda tradicional
console.log("Busqueda tradicional");
cargarClientes(clientes, facturacion, cantidad)
escribirEnUnaLinea(clientes, facturacion, cantidad)
posicion = buscarTradicional("Ale", clientes, cantidad)
imprimirCliente(posicion, clientes, facturacion)
//Busqueda binaria
console.log("Busqueda mejorada con ordenamiento");
burbuja(clientes, facturacion, cantidad)
escribirEnUnaLinea(clientes, facturacion, cantidad)
posicion = buscarBinario("Ale", clientes, cantidad)
imprimirCliente(posicion, clientes, facturacion)
```



Técnicas de Programación

CFP Programador full-stack

Algoritmos Básicos (Ejercicios)

Algoritmos de Ordenamiento

Ordenar por Dos Criterios

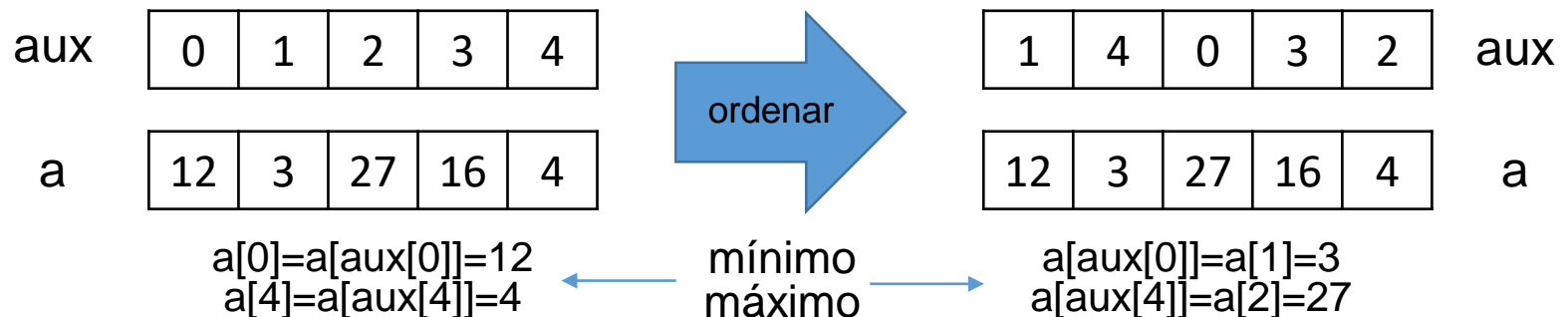
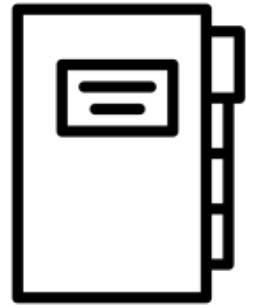
- Dados un arreglo de texto y dos arreglos de enteros de tamaño n :
 - nombres Como Texto
 - años Como Entero y altura Como Entero
- Ordénelos los tres vectores a la vez según los años, y en caso que haya un empate, utilice la altura para desempatar
- Tener en cuenta que los intercambios tienen que cambiar los elementos de los tres vectores a la vez



Algoritmos de Ordenamiento

Ordenar con Arreglo Auxiliar

- Desarrollar un programa que permita ordenar un arreglo "a" de tamaño "n" sin modificarlo, es decir, sin hacer los intercambios sobre la estructura "a"
- Utilizar un arreglo auxiliar "aux" cargado con los índices del arreglo "a" (de 0 a n)
- El ordenamiento tiene que hacerse mirando los valores de "a" pero haciendo los intercambios en "aux"
- Crear un método que permita imprimir ordenado que reciba como parámetros "a", "aux" y "n"



Algoritmos de Ordenamiento

Ordenar Matriz por Fila

- let un algoritmo que permita ordenar las filas de una matriz de $n \times m$ en orden descendente según la suma de todas sus elementos (es decir, todas las columnas)
- Tener en cuenta que la comparación se hace entre filas (y no entre elementos puntuales de la matriz)
- Considerar que el intercambio tiene que mover filas enteras (en vez de un solo número)

