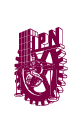
**INSTITUTO POLITÉCNICO NACIONAL**

UNIDAD PROFESIONAL INTERDISCIPLINARIA DE INGENIERÍA Y CIENCIAS SOCIALES ADMINISTRATIVAS

(UPIICSA)

**TAREA**

Probar los algoritmos de ordenamiento.

**3NM41**

Algoritmos Computacionales.

VÁZQUEZ JUÁREZ BRANDON.

Prof. SERGIO FUENLABRADA VELAZQUEZ

Índice

Contenido

[Solución 4](#_Toc101470799)

[Consideraciones iniciales 4](#_Toc101470800)

[P015-Algoritmo de la burbuja 8](#_Toc101470801)

[Descripción 8](#_Toc101470802)

[Código 8](#_Toc101470803)

[Medición del tiempo 9](#_Toc101470804)

[Análisis del comportamiento 10](#_Toc101470805)

[P016-Ordenamiento de burbuja bidireccional 11](#_Toc101470806)

[Descripción 11](#_Toc101470807)

[Código 11](#_Toc101470808)

[Medición del tiempo 12](#_Toc101470809)

[Análisis del comportamiento 13](#_Toc101470810)

[P017-Ordenamiento por inserción 14](#_Toc101470811)

[Descripción 14](#_Toc101470812)

[Código 14](#_Toc101470813)

[Medición del tiempo 15](#_Toc101470814)

[Análisis del comportamiento 16](#_Toc101470815)

[P018-Ordenamiento por casilleros 16](#_Toc101470816)

[Descripción 16](#_Toc101470817)

[Código 17](#_Toc101470818)

[Medición del tiempo 19](#_Toc101470819)

[Análisis del comportamiento 20](#_Toc101470820)

[P019- Ordenamiento por cuentas 21](#_Toc101470821)

[Descripción 21](#_Toc101470822)

[Código 21](#_Toc101470823)

[Medición del tiempo 22](#_Toc101470824)

[Análisis del comportamiento 23](#_Toc101470825)

[P020-El ordenamiento por mezcla 23](#_Toc101470826)

[Descripción 24](#_Toc101470827)

[Código 24](#_Toc101470828)

[Medición del tiempo 26](#_Toc101470829)

[Análisis del comportamiento 27](#_Toc101470830)

[P021-Método Árbol binario 27](#_Toc101470831)

[Descripción 27](#_Toc101470832)

[Código 28](#_Toc101470833)

[Medición del tiempo 29](#_Toc101470834)

[Análisis del comportamiento 32](#_Toc101470835)

[P22-Método Radix 32](#_Toc101470836)

[Descripción 32](#_Toc101470837)

[Código 33](#_Toc101470838)

[Medición del tiempo 34](#_Toc101470839)

[Análisis del comportamiento 35](#_Toc101470840)

[Resumen de comportamiento 36](#_Toc101470841)

# Solución

## Consideraciones iniciales

Se creó el algoritmo que genera números aleatorios para crear los archivos de entrada (código 1),

|  |
| --- |
| Genera archivos |
| for (i=0;i<r;i++){  v=inicio + rand () %(f+1 - inicio);  entrada.valor=v;  fprintf(archivo1, "%i\n", entrada.valor);  } |

Código 1 Algoritmo que genera los archivos de entrada para la medición del tiempo

Se generaron los archivos con los números aleatorios para los seis casos de prueba solicitados, los nombres de los archivos de entrada se muestran en la tabla 1.

|  |
| --- |
| num10 |
| num100 |
| num1000 |
| num10000 |
| num100000 |
| num1000000 |

Tabla 1 Nombres de los archivos con números aleatorios para cada bloque de pruebas.

Como comprobación que el algoritmo de ordenamiento realiza correctamente su función se genera un archivo de salida que contiene el arreglo ordenado. Los nombres de los archivos de salida se muestran en la tabla 2.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Burbuja | Burbuja balanceado | Inserción | Casilleros | Cuentas | Mezcla | Árbol binario | Radix |
| bur10 | bal10 | in10 | NULL | cue10 | mez10 | bin10 | rad10 |
| bur100 | bal100 | in100 | NULL | cue100 | mez100 | bin100 | rad100 |
| bur1000 | bal1000 | in1000 | NULL | cue1000 | mez1000 | bin1000 | rad1000 |
| bur10000 | bal10000 | in10000 | NULL | cue10000 | mez10000 | bin10000 | rad10000 |
| Bur100000 | Bal100000 | in100000 | NULL | cue100000 | mez100000 | bin100000 | rad100000 |
| Bur500000 | Bal500000 | in500000 | NULL | cue500000 | NULL | NULL | rad500000 |
| Bur1000000 | Bal1000000 | in1000000 | NULL | cue1000000 | mez1000000 | NULL | rad1000000 |

Tabla 2 Nombres de los archivos de salida ordenados

Notas del comportamiento de los algoritmos de ordenamiento

Los primeros 3 algoritmos en los últimos 2 archivos tardaban demasiado era algo extremo realmente sin embargo el algoritmo de casilleros no funcionaba en el metodo ordenar saltaban errores en cuanto al metodo árbol habían instrucciones comentadas en los últimos dos datos habían tantos datos que ni con 1 hora terminaba el proceso.

Se creó un código que incluye tres funciones para unificar la medición del tiempo para los algoritmos de ordenamiento que utilicen un arreglo:

1. **LeeArchivo**: que lee el archivo que se indica para construir el arreglo. Ver el código 2.
2. **Ordenamiento**: en este espacio de código se agrega el algoritmo de ordenamiento correspondiente. Ver el código 3.
3. **GrabaArchivo**: que crea un archivo de salida que contiene el arreglo ordenado, como mecanismo de comprobación de que fue ordenado como se indica. Ver el código 4.

|  |
| --- |
| Función LeeArchivo |
| int LeeArchivo(){  FILE \*archivo1;  string valorInt;  printf("\n Teclea el nombre del archivo de entrada: ");  cin >> nombrearchivo;  nombrearchivo+= ".txt"; // le agregarmos la extension ".txt"  archivo1 = fopen(nombrearchivo.c\_str(), "r"); // Abrimos archivo  if(archivo1== NULL ){  printf(" No se puede abrir el archivo ");  exit(1);  }  else{  printf(" Se abrio el archivo correctamente %s\n ", nombrearchivo.c\_str() );  }  cn=0;  for (int i=0; !feof(archivo1); i++) {  fscanf (archivo1, "%i", &entrada.valor);  a[i]=entrada.valor; // crea el arreglo de acuerdo al contenido del archivo  cn++;  }  printf("valor de n es: %i\n",cn-1);  salvacn=cn;  fclose(archivo1);  return (0);  } |

Código 2 Función LeeArchivo

|  |
| --- |
| Función GrabaArchivo |
| int GrabaArchivo(){  FILE \*archivo2;  string valorInt;  printf("\n Teclea el nombre del archivo de salida: ");  cin >> nombrearchivo1;  nombrearchivo1+= ".txt"; // le agregarmos la extension ".txt"  archivo2 = fopen(nombrearchivo1.c\_str(), "w"); // Abrimos archivo  if(archivo2== NULL ){  printf(" No se puede abrir el archivo ");  exit(1);  }  else{  printf(" Se abrio el archivo correctamente %s\n ", nombrearchivo1.c\_str() );  }  valorsalva=a[0];  cn=salvacn;  for (i=1; i<=cn-1; i++) {  if (valorsalva!=a[i]){  entrada1.valor1=valorsalva;  valorsalva=a[i];  fprintf(archivo2, "%i\n", entrada1.valor1);  }  }  printf("valor de n es: %i\n",cn-1);  fclose(archivo2);  return (0);  } |

Código 3 Función GrabaArchivo

|  |
| --- |
| Función Ordenamiento |
| Ordenamiento(){  printf("Ordenamiento nombre del algoritmo de ordenamiento\n");  Agregar el código de ordenamiento que corresponda  } // fin del ordenamiento |

Código 4 Función Ordenamiento

Para medir el tiempo de ejecución en el main se salva el tiempo antes y después de invocar la función de Ordenamiento (), adicionalmente se da la opción para ordenar otro archivo. Ver código 5.

|  |
| --- |
| Función main |
| main(){  si=1;  do{  LeeArchivo();  // Inicia area de medicion de tiempo  t\_ini = clock(); // almacena tiempo inicial  Ordenamiento(); // realiza el ordenamiento  t\_fin = clock(); // almacena tiempo final  // Termina area de medicion de tiempo  secs = (double)(t\_fin - t\_ini) / (double)CLOCKS\_PER\_SEC; // determina los milisegundo utilizados  printf("%.16g milisegundos\n", secs \* 1000.0); // muestra el tiempo utilizado  system("pause");  GrabaArchivo();  printf("Deseas probar con otro archivo (Si=1, NO=0) ");  cin >> si;  }while(si==1);  return(0);  } |

Código 5 Función main que lleva el control y medición del tiempo de ejecución del ordenamiento

# P015-Algoritmo de la burbuja

## Descripción

Este es el algoritmo de ordenamiento se considera uno de los más sencillo. Permite adquirir el conocimiento de ordenar una serie de datos. Consiste en realizar un ciclo que compare los elementos adyacentes de dos en dos, a través de todo el arreglo. Si el elemento es mayor que el que le sigue, se realiza un proceso de intercambio, lo que permite hacer que el valor mayor llegue a las últimas posiciones del arreglo.

## Código

El código del algoritmo de la burbuja es el que se muestra en el código 6[[1]](#footnote-1).

|  |
| --- |
| Ordenamiento de la burbuja |
| Ordenamiento(){  printf("Ordenamiento burbuja\n");  for(i=1;i<cn;i++){  for(j=0;j<cn-i;j++){  if(a[j]>a[j+1]){  k=a[j+1]; a[j+1]=a[j]; a[j]=k;  }  }  }  } // fin del ordenamiento |

Código 6 Algoritmo de ordenamiento por el método de la burbuja

## Medición del tiempo

Se ejecutó el código 6, como lo muestra la ilustración 1, para determinar el tiempo de ejecución, donde resalta el crecimiento exponencial que se presentó.

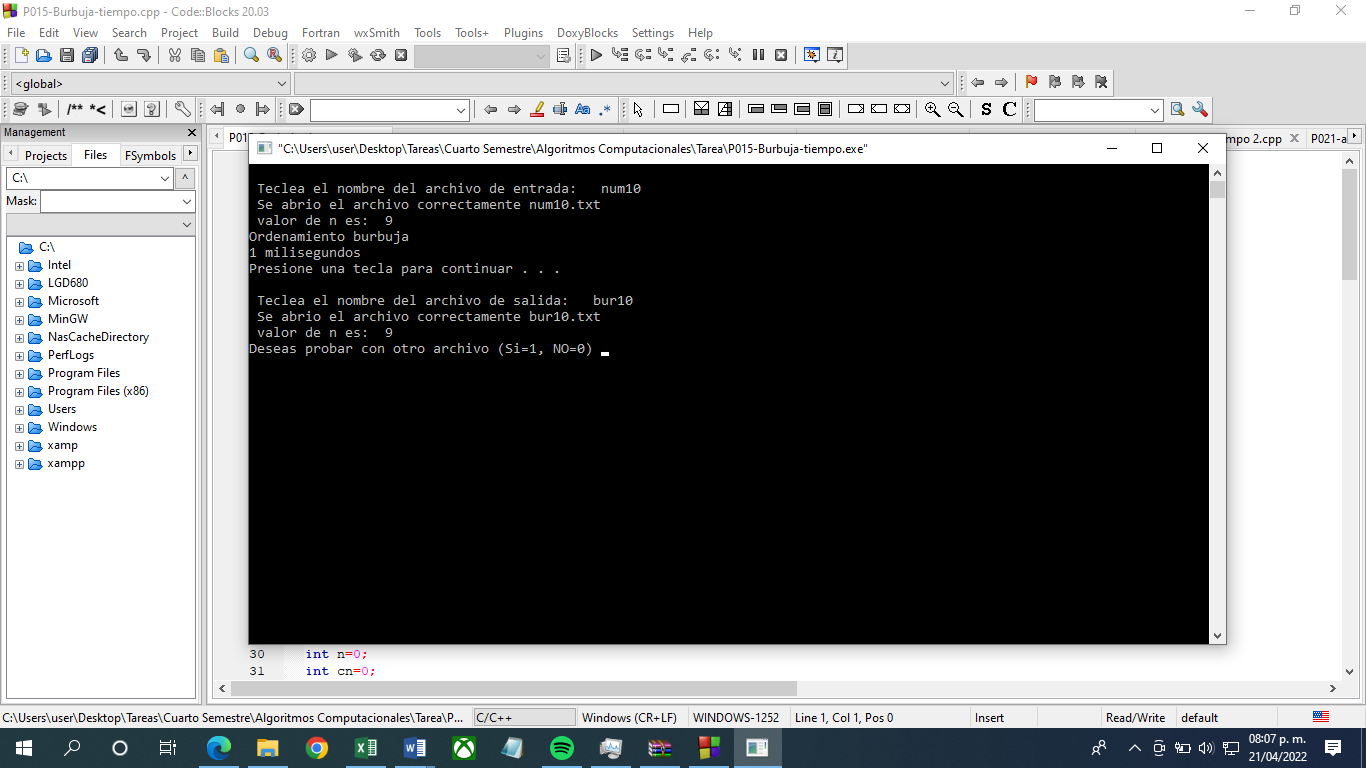


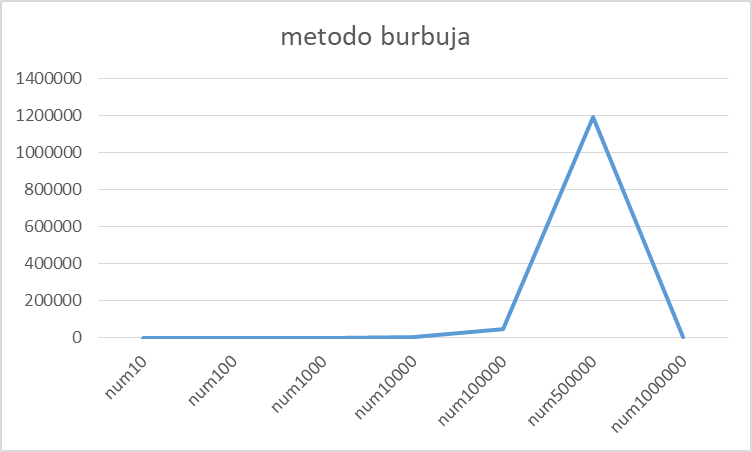
Ilustración 1 Pantalla de la ejecución del ordenamiento de burbuja

En resumen, los tiempos de ejecución en milisegundos los muestra la tabla 3.

|  |  |
| --- | --- |
|  | Burbuja |
| N | O(n2) |
| 10 | 2 |
| 100 | 2 |
| 1,000 | 5 |
| 10,000 | 1999 |
| 100,000 | 47557 |
| 1,000,000 | 1188318 |

Tabla 3 Tiempos de ejecución del ordenamiento de burbuja.

En la gráfica 1, se muestra el crecimiento exponencial al incrementarse el número de elementos a ordenar.



Gráfica 1 Comportamiento de ordenamiento de burbuja

## Análisis del comportamiento

Éste es el análisis para la versión no optimizada del algoritmo se muestra en la tabla 4.

|  |  |
| --- | --- |
| [Estabilidad](http://c.conclase.net/orden/?cap=introduccion#estabilidad) | En todos los tamaños de archivos probados, se obtuvo el resultado, se considera que su comportamiento es estable. |
| [Requerimientos de Memoria](http://c.conclase.net/orden/?cap=introduccion#requerimientos) | Entre más datos más desordenados era un proceso más lento El tamaño de memoria esta en relación al número de elementos (registros) que contenga el archivo. |
| [Tiempo de Ejecución](http://c.conclase.net/orden/?cap=introduccion#tiempo_ejecucion) | Su ejecución está en términos de la función O(n2) por lo que a mayor número de elementos a ordenar, el crecimiento del tiempo aumenta sin embargo mientras más ordenados estén los archivos menor es su tiempo de ejecución. |
| Ventajas | El algoritmo es simple, y sin muchos datos teniéndolo medianamente ordenados es práctico de usar. |
| Desventajas | si el número de elementos a ordenar es muy grande y están muy desordenados se puede presentar el uso de mucho tiempo para su ejecución |

Tabla 4 Análisis de comportamiento del ordenamiento de burbuja.

# P016-Ordenamiento de burbuja bidireccional

## Descripción

El algoritmo de ordenamiento de burbuja bidireccional también llamado ordenamiento cocktail es un intento de mejorar el rendimiento del ordenamiento de burbuja al incorporar el recorrido de comparación en ambas direcciones, de manera que se pueda realizar más de un intercambio por pasada (iteración).

## Código

El código del algoritmo de la burbuja bidireccional es el que se muestra en el código 7[[2]](#footnote-2).

|  |
| --- |
| Ordenamiento de la burbuja |
| Ordenamiento(){  printf("Ordenamiento burbuja bidireccional\n");  izq=0;  der=cn=salvacn-2;  do{  for(j=izq+1;j<=der;j++){  // burbuja a la izquierda los menores van a la izquierda  if(a[j-1]>a[j]){  k=a[j]; a[j]=a[j-1]; a[j-1]=k;  }  }  der=der-1;  for(i=der;i>izq;i--){  if(a[i-1]>a[i]){  k=a[i-1]; a[i-1]=a[i]; a[i]=k;  }  }  izq=izq+1;  }while (der>=izq);  } // fin del ordenamiento |

Código 7 Algoritmo de ordenamiento por el método de la burbuja bidireccional

## Medición del tiempo

Se ejecutó el código 7, para determinar el tiempo de ejecución, como lo muestra la ilustración 2, donde resalta el crecimiento exponencial y una breve mejora en el tiempo.

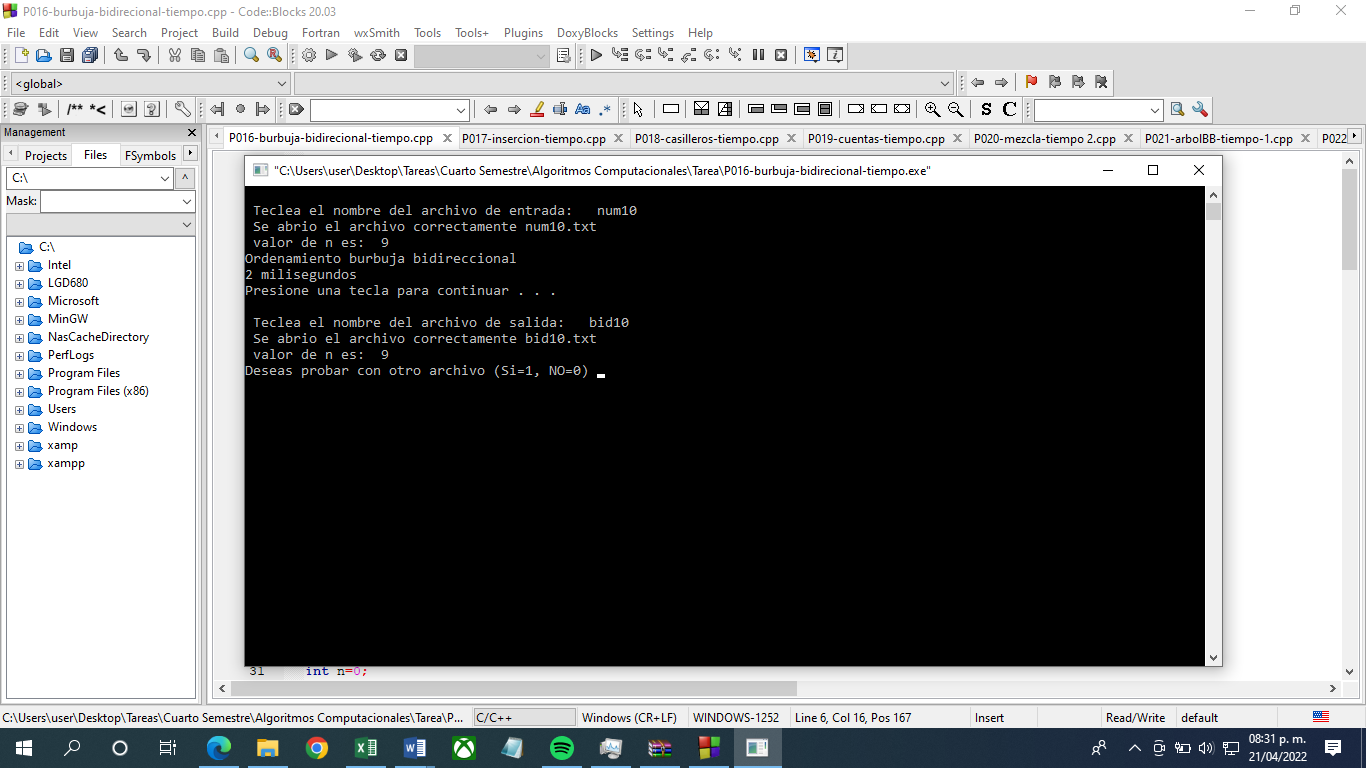


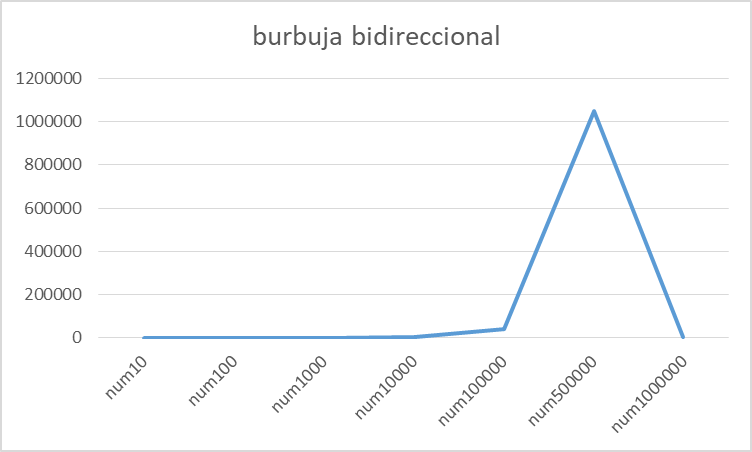
Ilustración 2 Pantalla de la ejecución del ordenamiento de burbuja bidireccional

En resumen, los tiempos de ejecución en milisegundos los muestra la tabla 4.

|  |  |
| --- | --- |
|  | Burbuja bidireccional |
| n | O(n2) |
| 10 | 2 |
| 100 | 3 |
| 1,000 | 7 |
| 10,000 | 2348 |
| 100,000 | 41831 |
| 1,000,000 | 1049348 |

Tabla 5 Tiempos de ejecución del ordenamiento de burbuja bidireccional.

En la gráfica 2, se muestra el crecimiento exponencial al incrementarse el número de elementos a ordenar.



Gráfica 2 Comportamiento de ordenamiento de burbuja bidireccional

## Análisis del comportamiento

Éste es el análisis para la versión del algoritmo se muestra en la tabla 6.

|  |  |
| --- | --- |
| [Estabilidad](http://c.conclase.net/orden/?cap=introduccion#estabilidad) | El programa arroja los mismos resultados por lo que es estable. |
| [Requerimientos de Memoria](http://c.conclase.net/orden/?cap=introduccion#requerimientos) | El programa puede llegar a tener problemas con archivos demasiados grandes y desordenados. |
| [Tiempo de Ejecución](http://c.conclase.net/orden/?cap=introduccion#tiempo_ejecucion) | Puede a llegar a tardar demasiado si los archivos son muy grandes y están desordenados |
| Ventajas | El código es versátil y es bidireccional es una mejora al algoritmo anterior. |
| Desventajas | El programa puede llegar a tardar mucho tiempo con los archivos muy grandes y muy desordenados. |

Tabla 6 Análisis de comportamiento del ordenamiento de burbuja bidireccional.

# P017-Ordenamiento por inserción

## Descripción

El ordenamiento por inserción se considera que es una manera muy natural de ordenar para un ser humano, ya que consiste en ir insertando un elemento de un arreglo en la parte ordenada de la misma, asumiendo que el primer elemento es la parte ordenada, se ira comparando un elemento de la parte desordenada contra los elementos de la parte ordenada, insertando el elemento en la posición correcta, y así sucesivamente hasta obtener la lista ordenada.

## Código

El código del algoritmo de ordenamiento por inserción es el que se muestra en el código 8[[3]](#footnote-3).

|  |
| --- |
| Ordenamiento por inserción |
| Ordenamiento(){  printf("Ordenamiento por insercion\n");  cn=salvacn-1;  for (i=0; i < cn; i++) {  k = a[i]; // k es el indice  j = i-1;  while (j >= 0 && a[j] > k) {  a[j + 1] = a[j];  j--;  }  a[j+1] = k;  }  } // fin del ordenamiento |

Código 8 Algoritmo de ordenamiento por el método de inserción

## Medición del tiempo

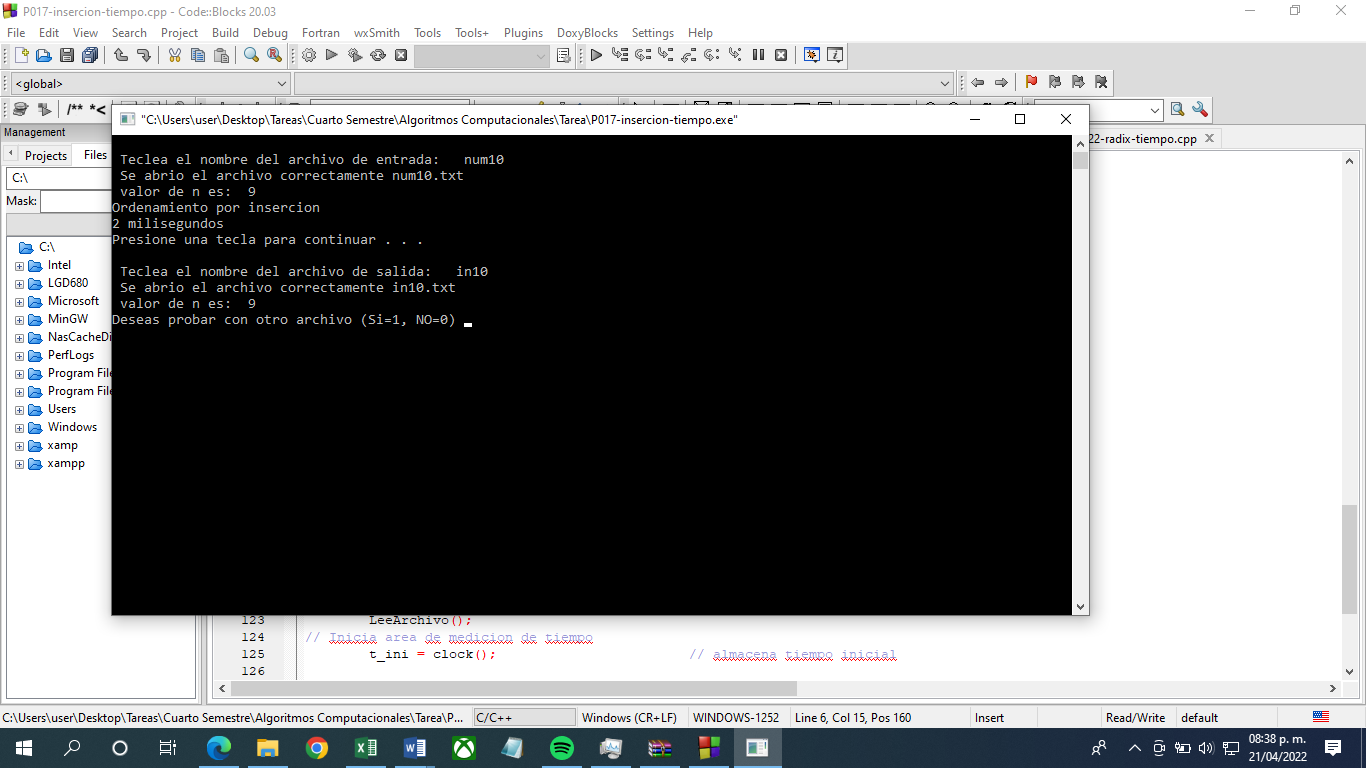
Se ejecutó el código 8, para determinar el tiempo de ejecución, como lo muestra la ilustración 3, donde resalta una disminución del crecimiento exponencial y una mejora en el tiempo.

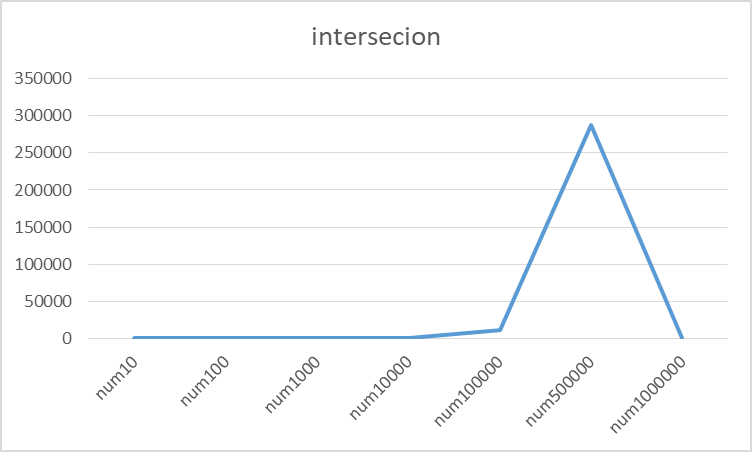
Ilustración 3 *Pantalla de la ejecución del ordenamiento por el método de inserción*

En resumen, los tiempos de ejecución en milisegundos los muestra la tabla 7.

|  |  |
| --- | --- |
|  | Inserción |
| N | O(n2) |
| 10 | 2 |
| 100 | 2 |
| 1,000 | 3 |
| 10,000 | 2 |
| 100,000 | 11638 |
| 1,000,000 | 287099 |

Tabla 7 Tiempos de ejecución del ordenamiento por inserción.

En la gráfica 3, se muestra el crecimiento exponencial al incrementarse el número de elementos a ordenar, pero con una gran disminución con respecto a los ordenamientos de burbuja y burbuja bidireccional.



Gráfica 3 Comportamiento de ordenamiento por inserción

## Análisis del comportamiento

Éste es el análisis para la versión del algoritmo se muestra en la tabla 8.

|  |  |
| --- | --- |
| [Estabilidad](http://c.conclase.net/orden/?cap=introduccion#estabilidad) | El algoritmo es estable arroja los mismos resultados en los archivos. |
| [Requerimientos de Memoria](http://c.conclase.net/orden/?cap=introduccion#requerimientos) | Al tener archivos muy grandes y poco ordenados puede haber problemas al momento de realizar el algoritmo. |
| [Tiempo de Ejecución](http://c.conclase.net/orden/?cap=introduccion#tiempo_ejecucion) | El tiempo de ejecución puede llegar a ser demasiado alto si se cuenta con un archivo muy grande y demasiado desordenado. |
| Ventajas | Es un algoritmo sencillo y practico que comprueba todos los lugares anteriores |
| Desventajas | Si hay un número muy pequeño al final el algoritmo debe recorrer todas las posiciones y comprobar si es menor esto en archivos muy largos puede ser muy tardado. |

Tabla 8 Análisis de comportamiento del ordenamiento por inserción.

# P018-Ordenamiento por casilleros

## Descripción

El ordenamiento por casilleros también conocido por bucket sort o bin sort, es un  [ordenamiento](https://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento) que distribuye todos los elementos a ordenar entre un número finito de casilleros. Cada casillero sólo puede contener los elementos que cumplan unas determinadas condiciones, que deben ser excluyentes entre sí, para evitar que un elemento pueda ser clasificado en dos casilleros distintos. Después cada uno de esos casilleros se ordena individualmente aplicando recursivamente este algoritmo para obtener casilleros con menos elementos

## Código

El código del algoritmo de casillero es el que se muestra en el código 9, y el ordenamiento se presenta cuando se invoca la función addNode(mybucket, arr[i]/range, arr[i]); invocada desde main.

|  |
| --- |
| Ordenamiento por casilleros |
| // A structure to represent a node.  struct Node{  int value;  struct Node\* next;  };  // A structure to represent a Head Bucket Node of the bucket list.  struct Bucket{  // Pointer to head node of Bucket.  struct Node \*head;  };  struct BucketList{  int V;  struct Bucket \* array;  };  // A utility function to create a new node for a particular entry in a bucket.  struct Node\* newNode(int value){  struct Node\* newnode = new Node;  newnode->value = value;  newnode->next = NULL;  return newnode;  }  // A utility function that creates a list of the bucket over the range of input data.  struct BucketList\* createBucket(int V){  int i;  struct BucketList\* bl = new BucketList;  bl->V = V;  bl->array = new Bucket[V];  // Initialize each Bucket list as empty by making head as NULL.  for(i = 0; i < V; i++)  bl->array[i].head = NULL;  return bl;  }  // A function to Insert the nodes to corresponding Buckets.  void addNode(struct BucketList\* bl, int bckt, int value){  // Creating new data node.  struct Node \*newnode = newNode(value);  struct Node \*temp = new Node;  if(bl->array[bckt].head != NULL) {  temp = bl->array[bckt].head;  // Sorting.  // If the head node value is lesser than the newnode value, then add node at beginning.  if(temp->value > newnode->value){  newnode->next = bl->array[bckt].head;  bl->array[bckt].head = newnode;  }  else {  // Search for the node whose value is more than the newnode value.  while(temp->next != NULL) {  if((temp->next)->value > newnode->value)  break;  temp = temp->next;  }  // Insert newnode after temp node.  newnode->next = temp->next;  temp->next = newnode;  }  }  else {  // Assign head of the Bucket as newnode since bucket head is NULL.  bl->array[bckt].head = newnode;  }  }  // A function to print the result as sorted Data.  void printBuckets(struct BucketList \*bl){  int v;  struct Node\* pCrawl = new Node;  for(v = 0; v < bl->V; v++)  {  // To view the data in individual bucket remove next line from comment.  // cout<<"\n\t bucket "<<v+1;  pCrawl = bl->array[v].head;  while (pCrawl != NULL) {  cout<<"->"<< pCrawl->value;  pCrawl = pCrawl->next;  }  }  } |

Código 9 Algoritmo de ordenamiento por el método de casillerros

## Medición del tiempo

Se ejecutó el código 9, para determinar el tiempo de ejecución, como lo muestra la ilustración 4, donde resalta un comportamiento de O(n), con una mejora en el tiempo, pero con un gran uso de memoria, de tal suerte que, para el caso de 1,000,000 de números se desbordo la pila (stack) presentado el error de APPCRASH. Por lo que se probó con un archivo de 500,000 números.

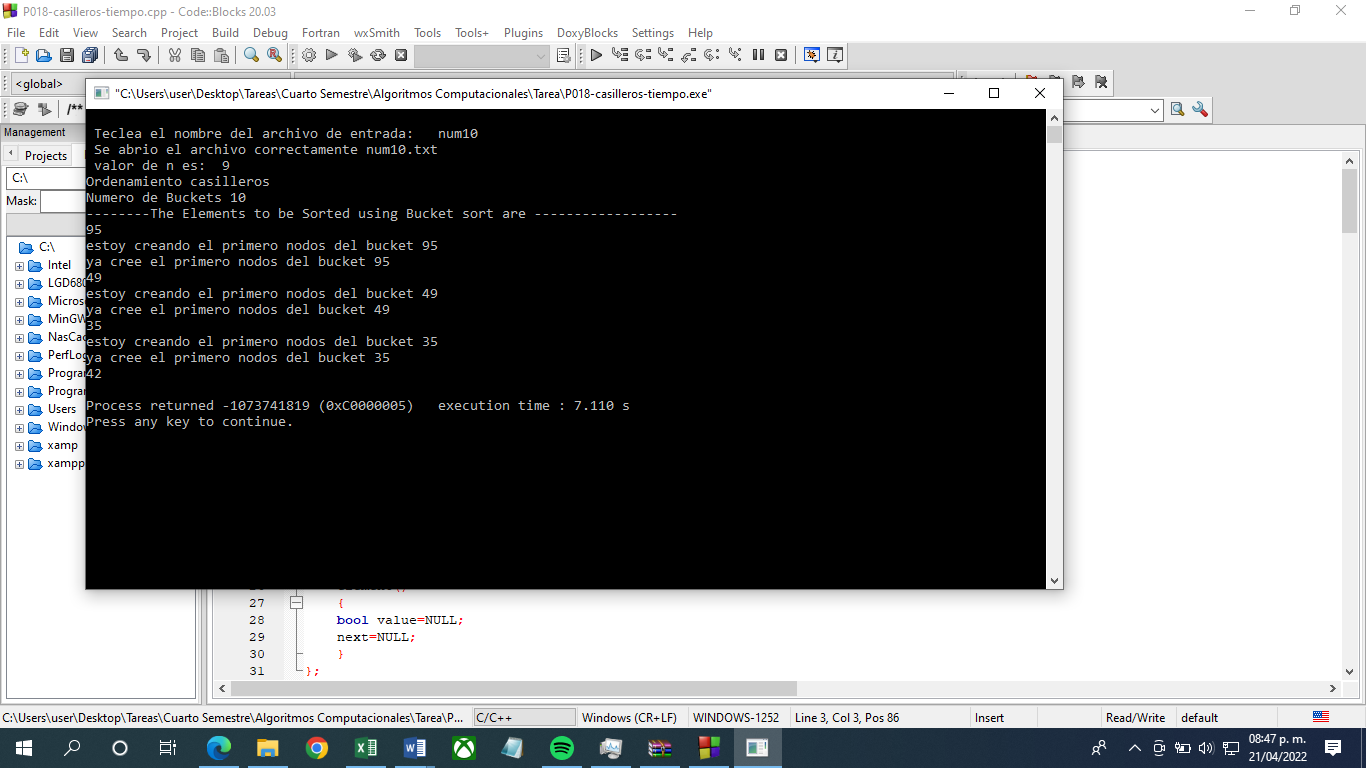


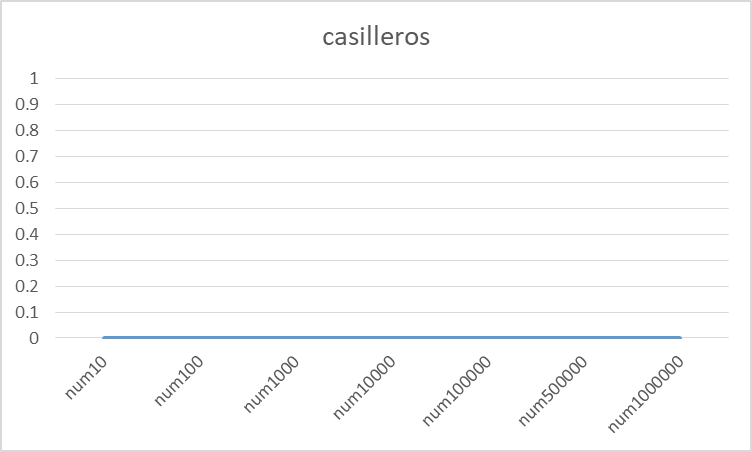
Ilustración 4 Pantalla de la ejecución del ordenamiento por el método de casilleros

En resumen, los tiempos de ejecución en milisegundos los muestra la tabla 9.

|  |  |
| --- | --- |
|  | Casilleros |
| N | O(n) |
| 10 | 0 |
| 100 | 0 |
| 1,000 | 0 |
| 10,000 | 0 |
| 100,000 | 0 |
| 500,000 | 0 |
| 1,000,000 | 0 |

Tabla 9 Tiempos de ejecución del ordenamiento por casillero.

En la gráfica 4, se muestra el crecimiento lineal que se incrementa al crecer el número de elementos a ordenar.



Gráfica 4 Comportamiento de ordenamiento por casilleros

## Análisis del comportamiento

Éste es el análisis para la versión del algoritmo se muestra en la tabla 10.

|  |  |
| --- | --- |
| [Estabilidad](http://c.conclase.net/orden/?cap=introduccion#estabilidad) | ERROR DE LOGICA |
| [Requerimientos de Memoria](http://c.conclase.net/orden/?cap=introduccion#requerimientos) | ERROR DE LOGICA |
| [Tiempo de Ejecución](http://c.conclase.net/orden/?cap=introduccion#tiempo_ejecucion) | ERROR DE LOGICA |
| Ventajas | ERROR DE LOGICA |
| Desventajas | ERROR DE LOGICA |

Tabla 10 Análisis de comportamiento del ordenamiento por casillero.

# P019- Ordenamiento por cuentas

## Descripción

El ordenamiento por cuentas o counting sort consiste en contar el número de elementos de cada clase para luego ordenarlos. Sólo puede ser utilizado para ordenar elementos numéricos. Se crea un vector de números enteros con tantos elementos como valores existen en el intervalo (valores mínimo y máximo), se recorre el arreglo de los elementos a ordenar y se cuenta el número de apariciones de cada elemento utilizando el vector de números enteros. Para finalizar, se recorre el vector para obtener todos los elementos ordenados.

## Código

El código del algoritmo de cuentas es el que se muestra en el código 10[[4]](#footnote-4). Cabe mencionar que, si se desea reducir el tiempo de ordenamiento más, se puede eliminar de la función ordenamiento el regresar los elementos ordenados al arreglo.

|  |
| --- |
| Ordenamiento por cuentas |
| Ordenamiento(int a[], int cn, int rango, int mini){  printf("Ordenamiento por cuentas \n");  int counter[rango+mini] = {0};  // Cuenta el numero de ocurrencias de cada elemento  for(i=0; i<cn; i++){  j=a[i]-mini;  counter[j]=counter[j]+1;  }  j=0;  // colocando los elementos nuevamente en el arreglo  for (i=0; i < rango+1; i++){  f=0;  for (k=counter[i]; k>0; k--){  a[j]= f+i+mini;  f++;  j++;  }  } |

Código 10 Algoritmo de ordenamiento por el método de cuentas

## Medición del tiempo

Se ejecutó el código 10, para determinar el tiempo de ejecución, como lo muestra la ilustración 5, donde resalta un comportamiento O(n+k), donde n el número de elementos a ordenar y k es el tamaño del vector auxiliar (máximo-mínimo).

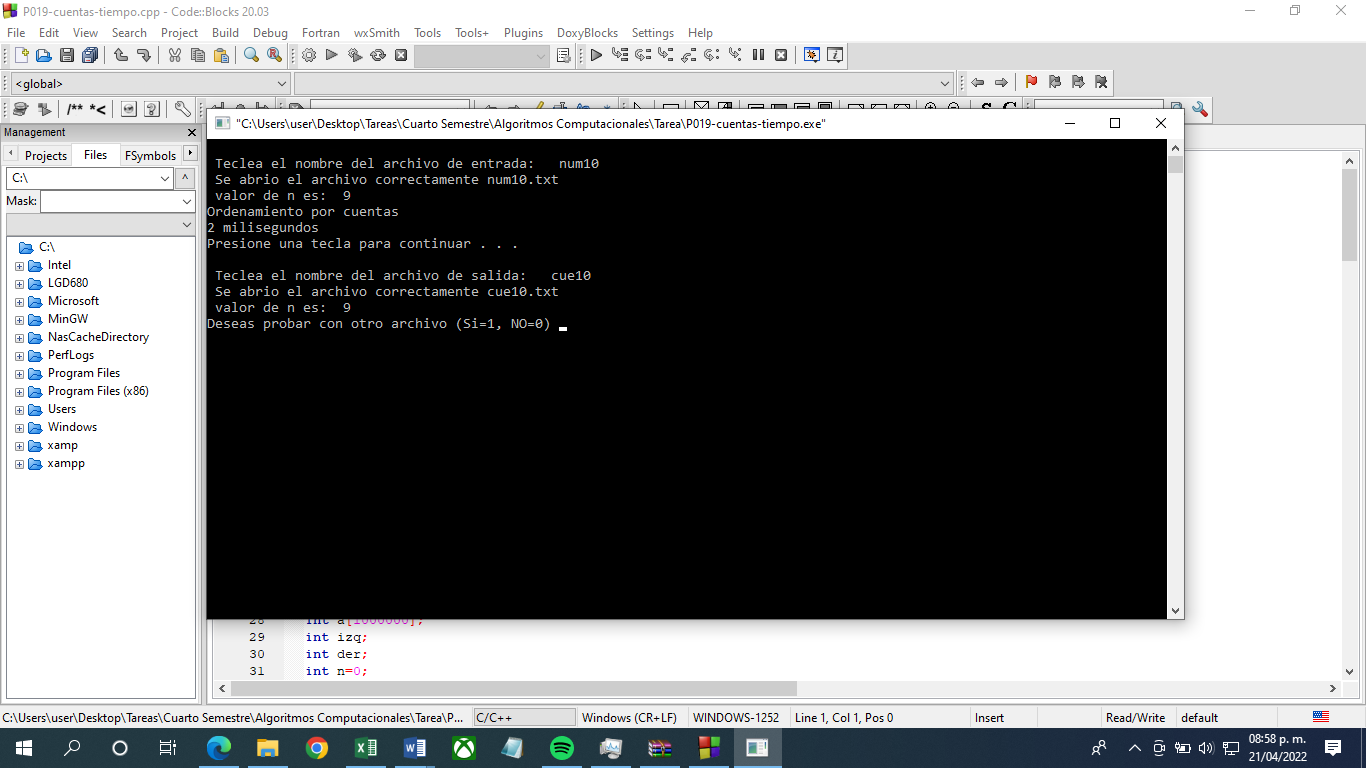


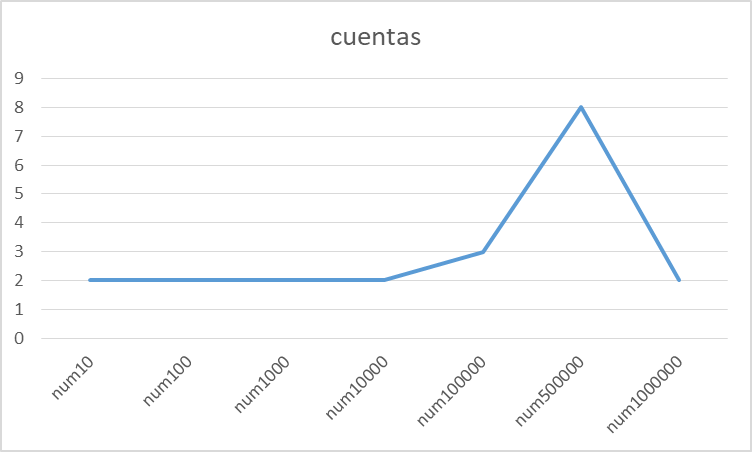
Ilustración 5 Pantalla de la ejecución del ordenamiento por el método de casilleros

En resumen, los tiempos de ejecución en milisegundos los muestra la tabla 11.

|  |  |
| --- | --- |
|  | Cuentas |
| N | O(n+k) |
| 100 | 2 |
| 1,000 | 2 |
| 10,000 | 2 |
| 100,000 | 2 |
| 1,000,000 | 3 |

Tabla 11 Tiempos de ejecución del ordenamiento por cuentas.

En la gráfica 5, se muestra el crecimiento lineal que se incrementa al crecer el número de elementos a ordenar.



Gráfica 5 Comportamiento de ordenamiento por cuentas

## Análisis del comportamiento

Éste es el análisis para la versión del algoritmo se muestra en la tabla 12.

|  |  |
| --- | --- |
| [Estabilidad](http://c.conclase.net/orden/?cap=introduccion#estabilidad) | Es muy estable todos los archivos tienen orden similar. |
| [Requerimientos de Memoria](http://c.conclase.net/orden/?cap=introduccion#requerimientos) | Muy bajo requerimiento de memoria pues con archivos muy grandes dio resultados muy rápidos. |
| [Tiempo de Ejecución](http://c.conclase.net/orden/?cap=introduccion#tiempo_ejecucion) | Muy velos aun con archivos muy grandes y muy desordenados logro hacer tiempos extremadamente rápidos a comparación con otros archivos |
| Ventajas | Su velocidad y bajo consumo de requisitos de memoria. |
| Desventajas | Este tiene un código más complejo. |

Tabla 12 Análisis de comportamiento del ordenamiento por cuentas.

# P020-El ordenamiento por mezcla

## Descripción

El ordenamiento por mezcla se basa en una heurística de divide y vencerás, es un algoritmo recursivo que divide continuamente una lista por la mitad. Si la lista está vacía o tiene un solo ítem, se ordena por definición (el caso base). Si la lista tiene más de un ítem, dividimos la lista e invocamos recursivamente un ordenamiento por mezcla para ambas mitades. Una vez que las dos mitades están ordenadas, se realiza la operación fundamental, denominada mezcla. La mezcla es el proceso de tomar dos listas ordenadas más pequeñas y combinarlas en una sola lista nueva y ordenada.

## Código

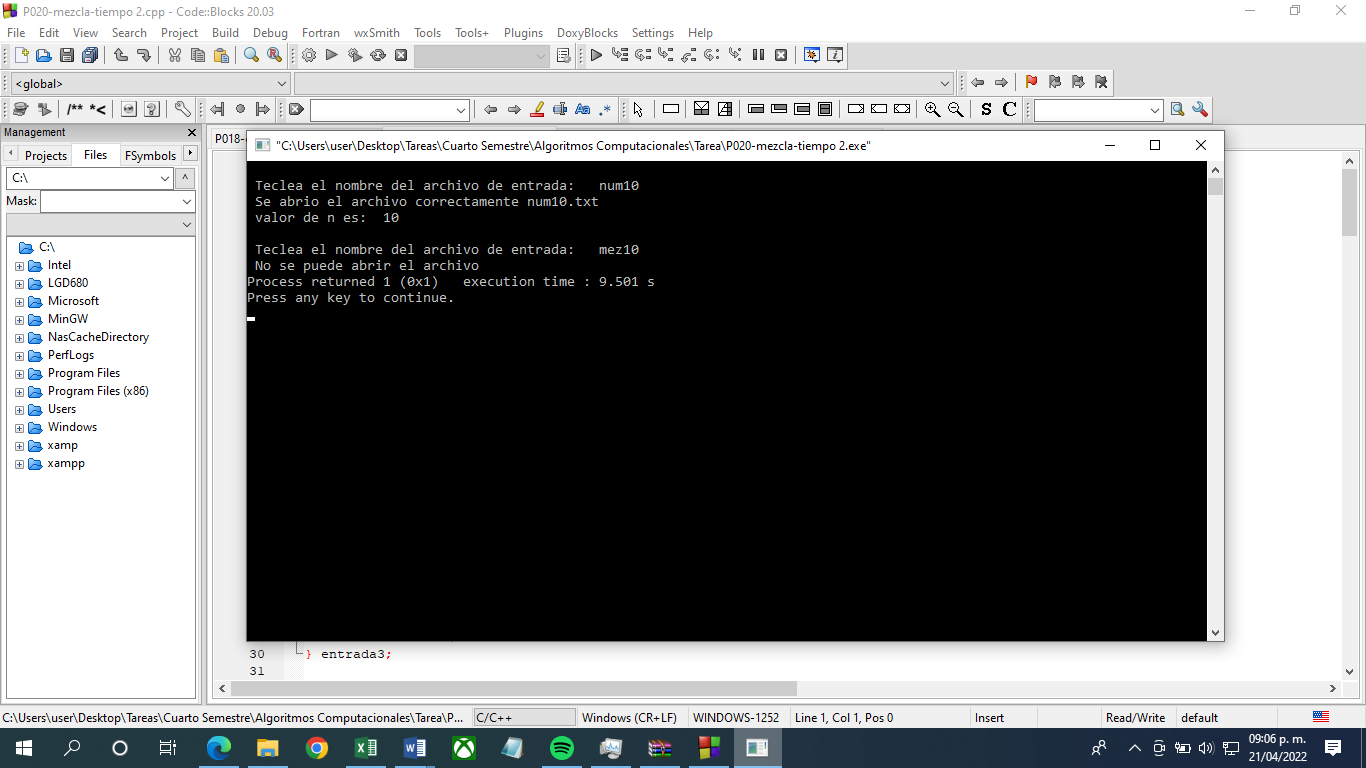
El código del algoritmo de mezcla es el que se muestra en el código 11[[5]](#footnote-5). Cabe mencionar que, este es un algoritmo recursivo y que para ordenar 1,000,000 de números desbordo la pila (stack) presentando el error APPSCRASH.

|  |
| --- |
| Ordenamiento por mezcla |
| // A function to merge the two half into a sorted data.  void Merge(int \*a, int low, int high, int mid){  // We have low to mid and mid+1 to high already sorted.  int i, j, k, temp[high-low+1];  i = low;  k = 0;  j = mid + 1;  // Merge the two parts into temp[].  while (i <= mid && j <= high){  if (a[i] < a[j]){  temp[k] = a[i];  k++;  i++;  }  else{  temp[k] = a[j];  k++;  j++;  }  }  // Insert all the remaining values from i to mid into temp[].  while (i <= mid){  temp[k] = a[i];  k++;  i++;  }  // Insert all the remaining values from j to high into temp[].  while (j <= high){  temp[k] = a[j];  k++;  j++;  }  // Assign sorted data stored in temp[] to a[].  for (i = low; i <= high; i++){  a[i] = temp[i-low];  }  }  // A function to split array into two parts.  void Ordenamiento(int \*a, int low, int high){  int mid;  if (low < high){  mid=(low+high)/2;  // Split the data into two half.  Ordenamiento(a, low, mid);  Ordenamiento(a, mid+1, high);  // Merge them to get sorted output.  Merge(a, low, high, mid);  }  } |

Código 11 Algoritmo de ordenamiento por el método de mezcla

## Medición del tiempo

Se ejecutó el código 11, para determinar el tiempo de ejecución, como lo muestra la ilustración 6, donde resalta un comportamiento O(n lg n), donde n el número de elementos a ordenar y k es el tamaño del vector auxiliar (máximo-mínimo).

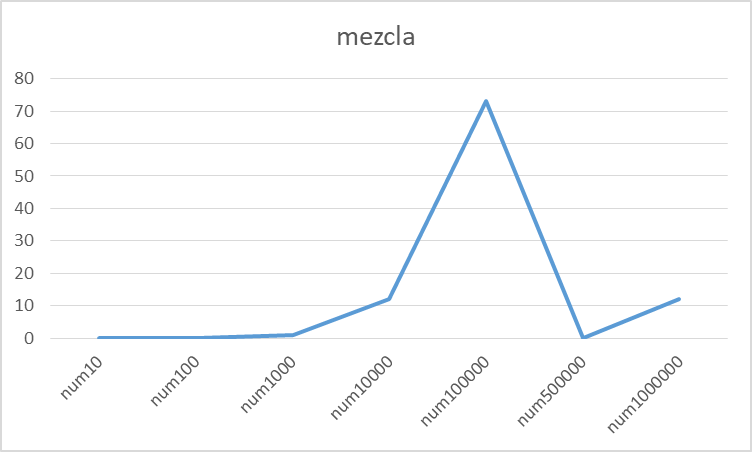


*Ilustración 6 Pantalla de la ejecución del ordenamiento por el método de mezcla*

En resumen, los tiempos de ejecución en milisegundos los muestra la tabla 13.

|  |  |
| --- | --- |
|  | Mezcla |
| n | O(n lg n) |
| 10 | 0 |
| 100 | 0 |
| 1,000 | 1 |
| 10,000 | 12 |
| 100,000 | 73 |
| 500,000 | 0 |
| 1,000,000 | 12 |

Tabla 13 Tiempos de ejecución del ordenamiento por mezcla.En la gráfica 6, se muestra el crecimiento O(n lg n).



Gráfica 6 Comportamiento de ordenamiento por mezcla

## Análisis del comportamiento

Éste es el análisis para la versión del algoritmo se muestra en la tabla 14.

|  |  |
| --- | --- |
| [Estabilidad](http://c.conclase.net/orden/?cap=introduccion#estabilidad) | Muy poco estable el algoritmo deja de funcionar con algunos archivos y cantidades sinceramente creo que está mal codificado, |
| [Requerimientos de Memoria](http://c.conclase.net/orden/?cap=introduccion#requerimientos) | Baja sin embargo con algunos archivos falla. |
| [Tiempo de Ejecución](http://c.conclase.net/orden/?cap=introduccion#tiempo_ejecucion) | El tiempo de ejecución es muy bajo sin embargo en algunos archivos el programa falla. |
| Ventajas | Se usa una idea muy buena el segmentar los elementos del arreglo o más bien del archivo para irlos reordenando por partes. |
| Desventajas | En este caso que no funcionó correctamente con el archivo de 500,000. |

Tabla 14 Análisis de comportamiento del ordenamiento por mezcla.

# P021-Método Árbol binario

## Descripción

Un árbol binario es un tipo de árbol en que cada vértice máximo puede tener dos hijos; su nodo raíz está enlazado a dos subárboles binarios disjuntos denominados subárbol izquierdo y subárbol derecho. Los árboles binarios no son vacíos ya que como mínimo tienen el nodo raíz.

En creación (adición de un nodo) del árbol binario se determina si debe estar del lado izquierdo o del lado derecho del nodo que se considerada raíz, lo que implica que el árbol al crearse esta ordenado. Una de las características del árbol binario es que no adiciona un nodo si este ya existe (eliminación de duplicados).

El recorrido en un árbol binario consiste en visitar todos sus nodos, de tal manera que cada vértice se visite una sola vez. El recorrido toma en cuenta la posición de la raíz y siempre se debe ejecutar primero el hijo izquierdo y luego el derecho.

El recorrido INORDEN primero recorre el subárbol izquierdo, segundo visita la raíz y, por último, va al subárbol derecho. En síntesis: hijo izquierdo — raíz — hijo derecho. Este recorrido debe mostrar por el valor de nodo en orden ascendente.

## Código

El código del algoritmo de árbol binario es el que se muestra en el código 12[[6]](#footnote-6). Cabe mencionar que, este es un algoritmo recursivo y realiza el ordenamiento al invocar el método ArbolInt.Insertar(nodo); en la instrucción marcada en rojo determina la existencia de un nodo a insertar.

|  |
| --- |
| Ordenamiento por árbol binario |
| void ArbolABB::Insertar(const int dat)  {  Nodo \*padre = NULL;  actual = raiz;  // Buscar el int en el árbol, manteniendo un puntero al nodo padre  while(!Vacio(actual) && dat != actual->dato) {  padre = actual;  if(dat > actual->dato) actual = actual->derecho;  else if(dat < actual->dato) actual = actual->izquierdo;  }  // Si se ha encontrado el elemento, regresar sin insertar  if(!Vacio(actual)) return;  // Si padre es NULL, entonces el árbol estaba vacío, el nuevo nodo será  // el nodo raiz  if(Vacio(padre)) raiz = new Nodo(dat);  // Si el int es menor que el que contiene el nodo padre, lo insertamos  // en la rama izquierda  else if(dat < padre->dato) padre->izquierdo = new Nodo(dat);  // Si el int es mayor que el que contiene el nodo padre, lo insertamos  // en la rama derecha  else if(dat > padre->dato) padre->derecho = new Nodo(dat);  } |

Código 12 Algoritmo de ordenamiento por árbol binario

## Medición del tiempo

Se ejecutó el código 12, para determinar el tiempo de ejecución, como lo muestra la ilustración 7, donde resalta un comportamiento O(log2 n), donde n el número de elementos a ordenar. También se observa que el número de nodos es menor a los de entrada.

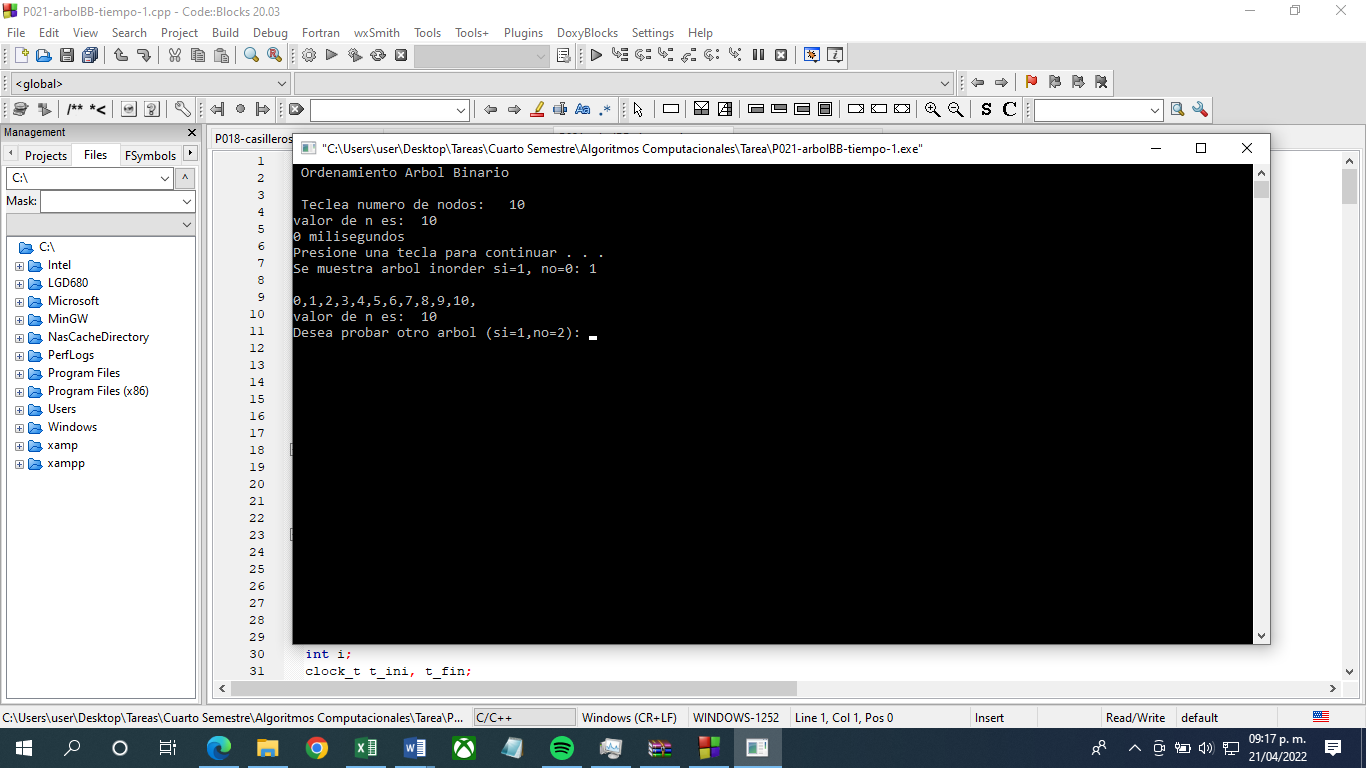


Ilustración 7 Pantalla de la ejecución del ordenamiento por árbol binario

Dado que la creación del árbol binario elimina los duplicados se modificó el algoritmo para que en la función obtención en lugar de obtener los datos de un archivo genere los números en el rango establecido (ver código 13) y así considerará el número de elementos indicado en el ejercicio, se tomó en cuenta que el peor de los casos era que el arreglo estuviera ordenado de mayor a menor. Los nuevos tiempos se muestran en la ilustración 8.

|  |
| --- |
| Función Obtención del algoritmo de árbol binario |
| // Obtencion: Obtiene arreglo ordenado a la inversa  int ArbolABB::Obtencion(){  int j;  int tnodo;  printf("\n Teclea numero de nodos: ");  cin >> tnodo;  for (j=tnodo+1; j>0 ; j--) {  arreglo[j]=j;  n++;  }  salvan=n;  } |

Código 13 Función de obtención del algoritmo de árbol binario modificada para obtener número en orden inverso

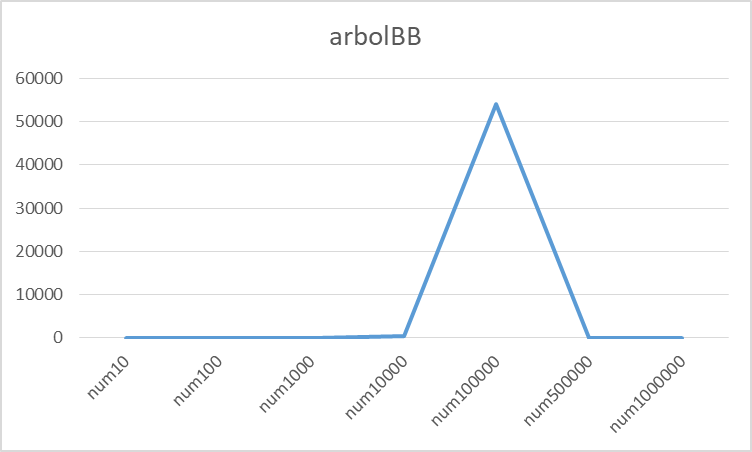


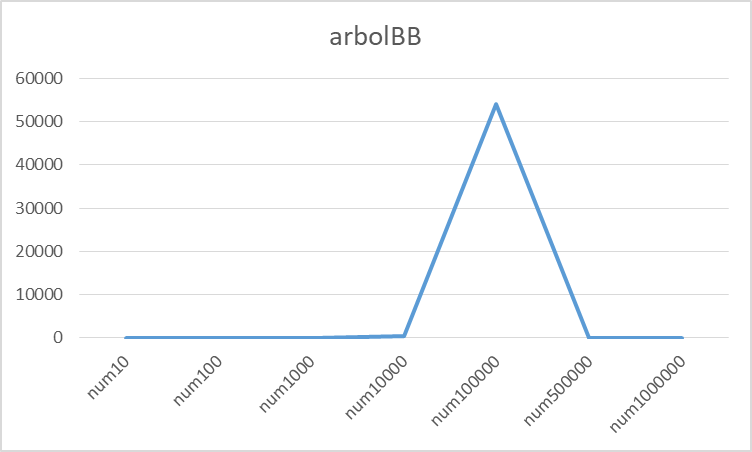
Ilustración 8 Pantalla de la ejecución del ordenamiento por árbol binario modificado para cumplir con el número de datos de entrada

En resumen, los tiempos de ejecución en milisegundos los muestra la tabla 15, se muestra el número de nodos creados que reduce significativamente. Se modificó el código para que generara números consecutivos en orden inverso (peor de los casos) en el número de datos de entrada establecidos, ver el código 13.

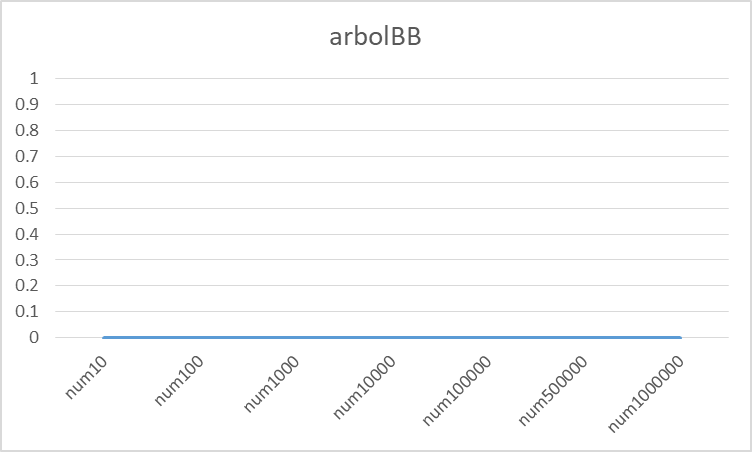
|  |  |  |  |
| --- | --- | --- | --- |
| n (entrada) | N (nodos creados) | Árbol binario | Árbol binario completo |
| O(log2 n) | O(log2 n) |
| 10 | 10 | 0 | 0 |
| 100 | 100 | 0 | 0 |
| 1,000 | 1,000 | 5 | 0 |
| 10,000 | 10,000 | 480 | 0 |
| 100,000 | 100,000 | 54154 | 0 |
| 1,000,000 | 1,000,000 | ind | 0 |

Tabla 15 Tiempos de ejecución del ordenamiento árbol binario,

En la gráfica 7, se muestra el crecimiento O(log2 n) del árbol binario que elimina los duplicados, en la columna n (nodos creados) de la tabla 15 son los nodos no duplicados que se crearon el árbol. En la gráfica 8 se nuestra el resultado de la generación de nodos en los tamaños n indicados en la tabla 15.



Gráfica 7 Comportamiento de ordenamiento por árbol binario con eliminación de duplicados



Gráfica 8 Comportamiento de ordenamiento por árbol binario modificado

## Análisis del comportamiento

Éste es el análisis para la versión del algoritmo se muestra en la tabla 16.

|  |  |
| --- | --- |
| [Estabilidad](http://c.conclase.net/orden/?cap=introduccion#estabilidad) | Muy mala está mal echo el programa. Pero debería ser muy estable |
| [Requerimientos de Memoria](http://c.conclase.net/orden/?cap=introduccion#requerimientos) | Baja con valores pequeños y extrema con valores muy grandes. Aunque debería ser muy bueno y rápido. |
| [Tiempo de Ejecución](http://c.conclase.net/orden/?cap=introduccion#tiempo_ejecucion) | Es un tiempo de ejecución muy bueno el algoritmo. |
| Ventajas | Puede ordenar datos tal cual los recibe tiene buen rendimiento y es estable |
| Desventajas | Tiene una complejidad un poco mayor que los anteriores. |

Tabla 16 Análisis de comportamiento del ordenamiento de árbol binario

# P22-Método Radix

## Descripción

El ordenamiento por el método Radix es para ordenar claves formadas por dígitos, para lograrlo toma los dígitos de las claves (enteros) y los clasifica usando cada uno de sus dígitos, de menor peso a más. A cada ordenación de las claves por un dígito se llama iteración. Se necesita un arreglo auxiliar del mismo tamaño del arreglo a ordenar.

Para el ordenamiento en cada iteración puede utilizar cualquier método de ordenación para ordenar un digito de las claves.

## Código

El código del algoritmo por el método Radix es el que se muestra en el código 14[[7]](#footnote-7). Cabe mencionar que, este es un algoritmo iterativo y que presentaba para el caso de 1,000,000 de números un desbordamiento de la pila (stack), lo que se presentaba por utiliza un arreglo dinámico de trabajo. Con lo cual se eliminó el desbordamiento.

|  |
| --- |
| Ordenamiento del método Radix |
| // Cuenta sort para a[].  void countSort(int a[], int cn, int k)  {  // Count [i] array contará el número de valores de matriz que tienen ese dígito 'i' en su (k) lugar.  int i, count[10] = {0};  // Cuente la cantidad de veces que cada dígito ocurrió en (k) en el lugar de cada entrada.  for (i = 0; i < cn; i++){  count[(a[i] / k) % 10]++;  }  // Calculando el acumulativo en count.  for (i = 1; i < 10; i++){  count[i] += count[i-1];  }  // Insertar valores de acuerdo con el dígito '(a [i] / k)% 10' obtenido en el recuento [(a [i] / k)% 10].  for (i = cn - 1; i >= 0; i--){  output[count[(a[i] / k) % 10] - 1] = a[i];  count[(a[i] / k) % 10]--;  }  // Asignando el resultado a un puntero de main().  for (i = 0; i < cn; i++){  a[i] = output[i];  }  }  Ordenamiento(){  printf("Ordenamiento por Radix\n");  cn=salvacn-1;  // llamando a countSort () para el dígito en (k) ne el lugar de cada entrada.  for (k = 1; cn/k > 0; k \*= 10){  countSort(a, cn, k);  }  } // fin del ordenamiento |

Código 14 Algoritmo de ordenamiento por el método Radix

## Medición del tiempo

Se ejecutó el código 14, para determinar el tiempo de ejecución, como lo muestra la ilustración 9, donde resalta un comportamiento O(n\*d), donde n el número de elementos a ordenar y d es el tamaño del arreglo auxiliar (n=d).

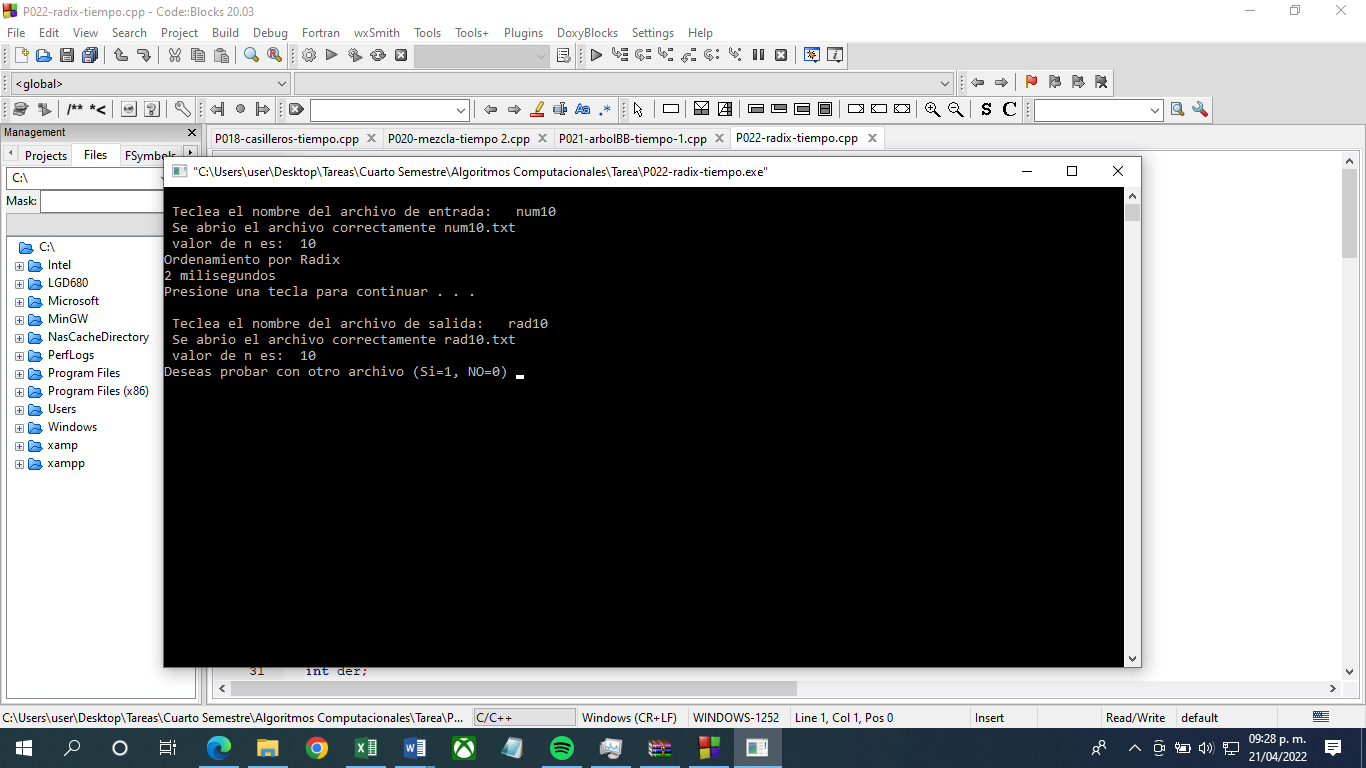


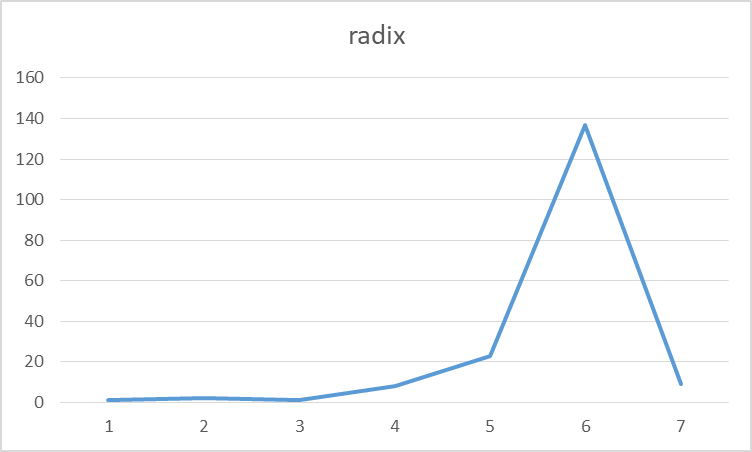
Ilustración 9 *Pantalla de la ejecución del ordenamiento por el método Radix*

En resumen, los tiempos de ejecución en milisegundos los muestra la tabla 17.

|  |  |
| --- | --- |
|  | Radix |
| n | O(n\*d) |
| 10 | 1 |
| 100 | 2 |
| 1,000 | 1 |
| 10,000 | 8 |
| 100,000 | 23 |
| 500,000 | 137 |
| 1,000,000 | 9 |

Tabla 17 Tiempos de ejecución del ordenamiento el método Radix.

En la gráfica 9, se muestra el crecimiento O(n\*d) con un incremento poco creciente.



Gráfica 9 Comportamiento de ordenamiento por el método Radix

## Análisis del comportamiento

Éste es el análisis para la versión del algoritmo se muestra en la tabla 18.

|  |  |
| --- | --- |
| [Estabilidad](http://c.conclase.net/orden/?cap=introduccion#estabilidad) | El código es muy estable pues asigna la posición según su bit más significativo. |
| [Requerimientos de Memoria](http://c.conclase.net/orden/?cap=introduccion#requerimientos) | Baja puesto que puede simplemente clasificar los números según su número de significancia. |
| [Tiempo de Ejecución](http://c.conclase.net/orden/?cap=introduccion#tiempo_ejecucion) | El tiempo de ejecución es muy bajo a comparación con otros algoritmos sin embargo es muy veloz y eficaz |
| Ventajas | Es un algoritmo rápido y estable que puede ordenar grandes cantidades de datos. |
| Desventajas | Puede llegar a tardar un poco con archivos muy grandes y desordenados. |

Tabla 18 Análisis de comportamiento del ordenamiento por el método Radix.

# Resumen de comportamiento

Se muestra los comportamientos de los algoritmos de ordenamiento clásicos iterativos en la tabla 19 y en los recursivos en la tabla 20 que desbordaron la memoria en el caso de 1,000,0000 de elementos

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | num10 | num100 | num1000 | num10000 | num100000 | num500000 | num1000000 |
| metodo burbuja | 2 | 2 | 5 | 1999 | 47557 | 1188318 | 1994 |
| burbuja bidireccional | 2 | 3 | 7 | 2348 | 41831 | 1049348 | 2342 |
| intercesión | 2 | 2 | 3 | 2 | 11638 | 287099 | 2 |
| casilleros | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| cuentas | 2 | 2 | 2 | 2 | 3 | 8 | 2 |
| mezcla | 0 | 0 | 1 | 12 | 73 | 0 | 12 |
| arbolBB | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| radix | 1 | 2 | 1 | 8 | 23 | 137 | 9 |

Tabla 19 Comportamiento de los algoritmos de ordenamientos iterativos

|  |  |  |
| --- | --- | --- |
|  | O(n) | O(n lg n) |
|  | Casilleros | Mezcla |
| 10 | 0 | 0 |
| 100 | 0 | 0 |
| 1,000 | 0 | 1 |
| 10,000 | 0 | 12 |
| 100,000 | 0 | 73 |
| 500,000 | 0 | 0 |
| 1,000,000 | 0 | 12 |

Tabla 20 Comportamiento de los algoritmos de ordenamiento recursivo

En las gráficas siguientes se muestra un comparativo de los algoritmos iterativos (gráfica 10) y recursivos (gráfica 11).

Gráfica 10 Comparación de los algoritmos de ordenamiento iterativos

Grafica 11 Comparación de los algoritmos de ordenamiento recursivos

1. El código se obtuvo de <https://beastieux.com/2008/01/05/codigo-c-ordenamiento-burbuja/> agosto 31, 2018 [↑](#footnote-ref-1)
2. El código se obtuvo de <https://juncotic.com/ordenamiento-de-burbuja-bidireccional-algoritmos-de-ordenamiento/> agosto 31, 2018 [↑](#footnote-ref-2)
3. El código se obtuvo de https://www.ecured.cu/Ordenamiento\_por\_Inserci%C3%B3n agosto 31, 2018 [↑](#footnote-ref-3)
4. Se obtuvo el código de <https://www.sanfoundry.com/cpp-program-implement-counting-sort/> el 30 de agosto de 2018, cabe mencionar que el código tiene algunos errores y omisiones de código pero permitió recordar el algoritmo. [↑](#footnote-ref-4)
5. Se obtuvo el código de <https://www.sanfoundry.com/cpp-program-implement-merge-sort/> el 30 de agosto de 2018. [↑](#footnote-ref-5)
6. Se obtuvo el código de <http://c.conclase.net/edd/?cap=ejemplos#inicio> el 30 de agosto de 2018. [↑](#footnote-ref-6)
7. Se obtuvo el código de <https://www.sanfoundry.com/cpp-program-implement-radix-sort/> el 30 de agosto de 2018. [↑](#footnote-ref-7)