Grupo 3 Taller Complejidad - Estructura De Datos

Juliana Aguirre Ballesteros
Juan Carlos Santamaria Orjuela

Pontificia Universidad Javeriana
Facultad de ingeniería
Departamento de ingeniería de sistemas
Bogotá D.C.
2025

Pontificia Universidad JAVERIANA Bogotá

2.1 Sort

Datos de los tres algoritmos de ordenamiento el programa que se debe ejecutar 10 veces con diferentes tamaños de arreglo (entre 10000 y 32000)

```
C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions>g++ -std=c++11 -o sort test_sort_inverse.cxx

C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions>sort 32000 31098
32000 1 1 1 1353 551 0

C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions>sort  10000 2000
10000 1 1 1 127 62 0

C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions>sort  10000 20000
10000 1 1 1 125 47 0

C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions>sort  11000 22000
11000 1 1 1 148 63 0

C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions>sort  10360 30000
10360 1 1 1 142 46 16

C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions>25000 32000
"25000" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.

C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions>sort  25000 32000
25000 1 1 1 842 344 0

C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions>sort  26000 32000
26000 1 1 1 888 364 13

C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions>sort  150000 32000
^C
C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions>sort  15000 32000
15000 1 1 1 297 125 0

C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions>sort  17000 27000
17000 1 1 1 389 161 0

C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions>sort  18000 29900
18000 1 1 1 424 188 0

C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions>sort  19000 31800
19000 1 1 1 473 204 0

C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions>sort  21700 31870
21700 1 1 1 612 266 0

C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions>
```

Tabla general

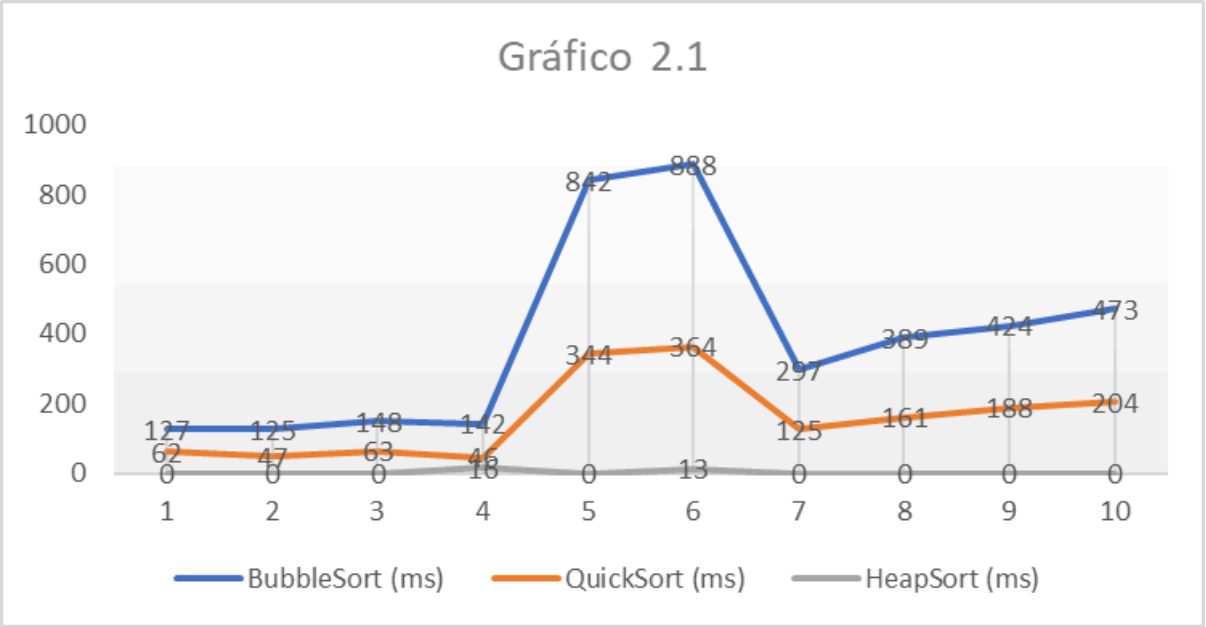
Tamaño del Arreglo	BubbleSort (ms)	QuickSort (ms)	HeapSort (ms)	Ordenado (BubbleSort)	Ordenado (QuickSort)	Ordenado (HeapSort)
10000	127	62	0	1	1	1
10000	125	47	0	1	1	1
11000	148	63	0	1	1	1

10360	142	46	16	1	1	1
25000	842	344	0	1	1	1
26000	888	364	13	1	1	1
15000	297	125	0	1	1	1
17000	389	161	0	1	1	1
18000	424	188	0	1	1	1
19000	473	204	0	1	1	1

Tabla análisis

PUNTOS EN LA GRAFICA	Tamaño del Arreglo	BubbleSort (ms)	QuickSort (ms)	HeapSort (ms)
1	10000	127	62	0
2	10000	125	47	0
3	11000	148	63	0
4	10360	142	46	16
5	25000	842	344	0
6	26000	888	364	13
7	15000	297	125	0
8	17000	389	161	0
9	18000	424	188	0
10	19000	473	204	0

Gráfica



Análisis:

BubbleSort muestra un comportamiento claramente ineficiente con los tiempos de ejecución que crecen significativamente a medida que aumenta el tamaño del arreglo. Por ejemplo para un arreglo de 10,000 elementos, el tiempo de ejecución es de 127 ms pero al aumentar el tamaño a 25,000, el tiempo aumenta dramáticamente a 842 ms. Este aumento en el tiempo de ejecución es exponencial, lo que indica que BubbleSort tiene una complejidad de $O(n$^2) como se esperaba. Este comportamiento es consistente en todas las ejecuciones: a medida que el tamaño del arreglo aumenta, los tiempos se disparan.

QuickSort por su parte muestra un crecimiento mucho más controlado en los tiempos de ejecución. Para el arreglo de 10,000 elementos, QuickSort toma 62 ms mientras que para 25,000 elementos el tiempo de ejecución es de 344 ms. Esto confirma que QuickSort tiene una complejidad de $O(n \log n)$ ya que su tiempo de ejecución crece mucho más lentamente en comparación con BubbleSort. QuickSort por lo tanto es mucho más eficiente para arreglos grandes.

HeapSort muestra tiempos de ejecución aún más bajos. Para arreglos de 10,000 elementos, HeapSort tiene un tiempo de ejecución de 0 ms y para 25,000 elementos, HeapSort también tiene tiempos muy bajos (0 ms). Lo que permite que HeapSort sea muy eficiente, similar a QuickSort, con una complejidad teórica de $O(n \log n)$.

En conclusión nos dimos cuenta que al analizar los gráficos y tiempos, Bubble Sort es el algoritmo menos eficiente para arreglos grandes debido a su complejidad cuadrática en cambio QuickSort y HeapSort son mucho más eficientes pero tenemos una diferencia entre estos dos y es que con QuickSort generalmente es más rápido pero HeapSort siendo más constante y integró en su rendimiento.

Pontificia Universidad JAVERIANA Bogotá

Investigación de algoritmos de complejidad bubble sort, quicksort y heapsort

Bubble Sort

El algoritmo Bubble Sort compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto, repitiendo el proceso hasta que no haya más cambios. Su complejidad en el mejor caso es O(n), lo que ocurre cuando el arreglo ya está ordenado y solo se requiere una pasada de verificación. En el caso promedio y el peor caso, su complejidad es $O(n^2)$ debido a la necesidad de realizar múltiples pasadas y comparaciones en arreglos desordenados o en orden inverso. En cuanto al uso de memoria, Bubble Sort es un algoritmo in-place que requiere O(1) espacio adicional, ya que solo utiliza una variable temporal para el intercambio de elementos (GeeksforGeeks, s.f.).

QuickSort

QuickSort es el algoritmo de ordenación que basa su funcionamiento en el principio de divide y vencerás. El algoritmo comienza seleccionando un elemento del arreglo para utilizarlo como pivote y separando los valores menores a este pivote a un lado y los mayores al otro lado. Tras lo cual, se vuelve a aplicar el mismo proceso sobre cada una de las dos particiones obtenidas, hasta que todo el conjunto queda ordenado. Suponiendo que las divisiones queden equilibradas, es sencillo comprobar que el tiempo de ejecución aproximado sea O(n log n) en el caso promedio y en el mejor de los casos. Si en vez de esto las divisiones fueran muy desiguales, p. ej. elegir siempre el pivote extremo, el rendimiento del algoritmo decaería hasta un límite de $O(n^2)$. En cuanto a la memoria, QuickSort necesitará un orden de O(log n), debido a la gestión de las llamadas recursivas que queda almacenada en la pila (Techie Delight, s.f.).

HeapSort

HeapSort gestiona los datos mediante un montículo binario (heap). En primer lugar, transforma el arreglo en un heap máximo, colocando el valor mayor en la raíz. Luego hace un intercambio de dicho valor con el último del arreglo y, a continuación, ajusta el tamaño del heap. A continuación, repite el proceso hasta que todos los valores están en orden. Su comportamiento es coherente, es decir, en el mejor y en el peor caso, el tiempo de ejecución es O(n log n), puesto que la operación de reajuste del heap es ejecutada n veces y cada uno de los reajustes requiere un tiempo log n. Además, tiene el comportamiento in-place: solo requerirá O(1) de memoria extra sin utilizar estructuras auxiliares significativas.

Análisis de las gráficas con la complejidad teórica de los tres algoritmos:

BubbleSort:

La complejidad de Bubble Sort es $O(n^2)$, este comportamiento se se puede ver en las gráficas ya que podemos ver que los tiempos de ejecución crecen de manera exponencial conforme aumenta el tamaño del arreglo. Por ejemplo tenemos 127 ms para 10,000

elementos a 842 ms para 25,000 elementos, este aumento cuadrático en el tiempo de ejecución confirma la complejidad teórica de BubbleSort.

QuickSort:

QuickSort tiene una complejidad teórica de $O(n \log n)$ y esto se refleja en las gráficas, donde el tiempo de ejecución crece de forma más controlada que en BubbleSort. Por ejemplo, de 62 ms para 10,000 elementos a 344 ms para 25,000 elementos. La gráfica muestra un crecimiento más lento y consistente que el de BubbleSort, lo que coincide con la teoría de su complejidad. QuickSort es, por lo tanto, mucho más eficiente para tamaños grandes de arreglo.

HeapSort:

Al igual que QuickSort, HeapSort tiene una complejidad teórica de $O(n \log n)$ y las gráficas muestran un comportamiento similar: tiempos de ejecución que aumentan lentamente a medida que aumenta el tamaño del arreglo. Lo que destaca de HeapSort es su estabilidad en el tiempo de ejecución, con tiempos que se mantienen más bajos y constantes en comparación con QuickSort. Para el arreglo de 25,000 elementos, HeapSort toma 0 ms, lo que refleja la optimización interna en C + + y su capacidad para manejar grandes arreglos de manera eficiente.

2.2 Búsqueda

Cambios en el código

Cambios del código original:

- Error en la medición del tiempo de la búsqueda binaria:

En el código original el tiempo de ejecución de la búsqueda binaria no estaba siendo calculado correctamente ya que se estaba utilizando la variable de la búsqueda lineal para la medición del tiempo de la búsqueda binaria.

Específicamente, el siguiente fragmento de código: auto time_binaryN = std::chrono::duration_cast<std::chrono::milliseconds>(end_linear - start_linear);

- Corrección: Para corregir este error se cambió el código para medir correctamente el tiempo de la búsqueda binaria. El código corregido es:

auto time_linearN = std::chrono::duration_cast<std::chrono::milliseconds>(end_linearN - start_linearN);
auto time_binaryN = std::chrono::duration_cast<std::chrono::milliseconds>(end_binaryN - start_binaryN);

Ahora las variables de tiempo (end_linearN, start_linearN, end_binaryN, start_binaryN) corresponden correctamente a la búsqueda lineal y la búsqueda binaria, respectivamente.

Esto hace que cada algoritmo de búsqueda (lineal y binaria) tenga su propio tiempo de ejecución sea medido correctamente.

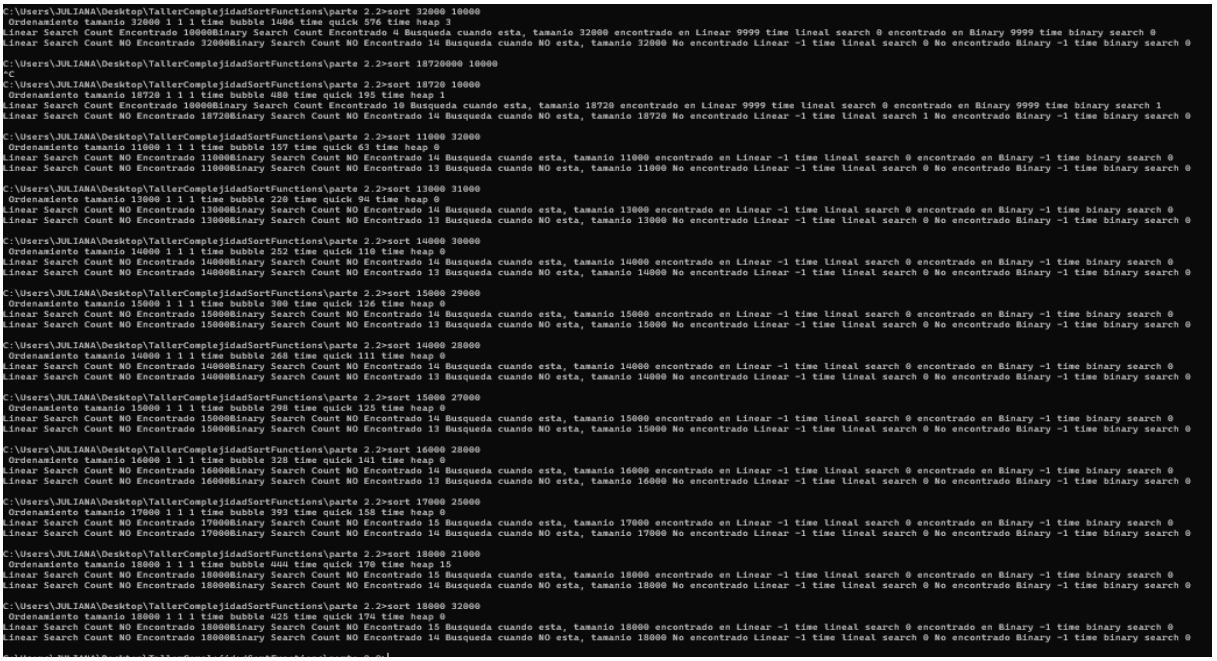
```
Microsoft Windows [Versión 10.0.22631.5472]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\JULIANA>cd C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions\parte 2.2

C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions\parte 2.2>g++ -std=c++11 -o sort test_sort_inverse.cxx

C:\Users\JULIANA\Desktop\TallerComplejidadSortFunctions\parte 2.2>sort 32000 10000
 Ordenamiento tamanio 32000 1 1 1 time bubble 1406 time quick 576 time heap 3
Linear Search Count Encontrado 10000Binary Search Count Encontrado 4 Busqueda cuando esta, tamanio 32000 encontrado en L
inear 9999 time lineal search 0 encontrado en Binary 9999 time binary search 0
Linear Search Count NO Encontrado 32000Binary Search Count NO Encontrado 14 Busqueda cuando NO esta, tamanio 32000 No en
contrado Linear -1 time lineal search 0 No encontrado Binary -1 time binary search 0
```

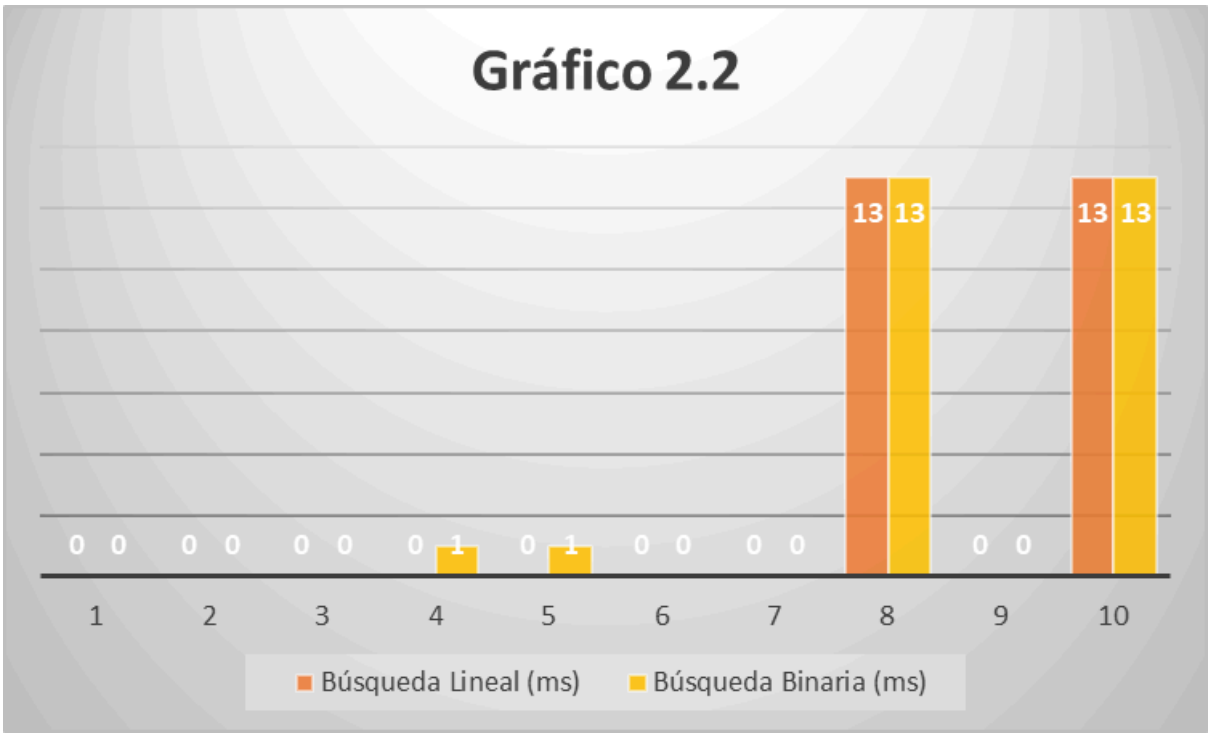
Tabla análisis:



Punto Gráfica	Tamaño del Arreglo	Búsqueda Lineal (ms)	Búsqueda Binaria (ms)
1	10000	0	0
2	11000	0	0
3	11500	0	0
4	18550	0	1
5	18720	0	1
6	21750	0	0
7	27250	0	0

8	28550	13	13
9	32000	0	0
10	30000	13	13

Gráfico



Complejidad Teórica de los Algoritmos de Búsqueda

Búsqueda Lineal

La búsqueda lineal, también conocida como búsqueda secuencial, es un algoritmo que recorre cada elemento de una lista de manera secuencial hasta encontrar el valor objetivo o determinar que no está presente.

- Mejor caso: O(1) — cuando el elemento buscado se encuentra en la primera posición.
- Peor caso: O(n) — cuando el elemento no está presente o se encuentra al final de la lista.
- Caso promedio: O(n) — en promedio, el algoritmo realiza n/2 comparaciones.

La búsqueda lineal es eficiente para listas pequeñas o cuando los datos no están ordenados.

Búsqueda Binaria

La búsqueda binaria es un algoritmo eficiente que encuentra la posición de un valor en una lista ordenada. Compara el valor con el elemento en el medio de la lista y elimina la mitad del conjunto de datos en cada paso.

- Mejor caso: O(1) — cuando el elemento buscado es el del medio.
- Peor caso: O(log n) — cuando el algoritmo reduce el conjunto de datos a la mitad en cada paso.
- Caso promedio: O(log n) — el número de comparaciones crece logarítmicamente con el tamaño de la lista.

La búsqueda binaria es significativamente más rápida que la búsqueda lineal para listas grandes, pero requiere que los datos estén ordenados.

Análisis de la Complejidad Teórica

1. Búsqueda Lineal:

- **Complejidad teórica**: La búsqueda lineal tiene una complejidad de O(n), lo que significa que el tiempo de ejecución debería aumentar linealmente con el tamaño del arreglo. Esto se debe a que, en el peor de los casos, la búsqueda lineal recorre todo el arreglo de principio a fin para encontrar el elemento.

¿Cómo debería comportarse?:

- A medida que el tamaño del arreglo aumenta, el tiempo de ejecución de la búsqueda lineal debería aumentar proporcionalmente.
- Deberíamos ver tiempos que aumentan conforme n (el tamaño del arreglo) se incrementa.

2. Búsqueda Binaria:

- **Complejidad teórica**: La búsqueda binaria tiene una complejidad de O(logn), lo que significa que el tiempo de ejecución crece mucho más lentamente conforme el tamaño del arreglo aumenta. La búsqueda binaria divide el arreglo en mitades repetidamente hasta encontrar el elemento (o determinar que no está presente), lo

que hace que el tiempo de ejecución aumente en una forma logarítmica con el tamaño del arreglo.

¿Cómo debería comportarse?:

- A medida que el tamaño del arreglo aumenta, el tiempo de ejecución de la búsqueda binaria debería aumentar lentamente.

- Deberíamos ver un crecimiento en los tiempos, pero de forma mucho más contenida en comparación con la búsqueda lineal.

Relación de los Datos con la Complejidad Teórica

Comportamiento de la Búsqueda Lineal:

- En la tabla, los tiempos de búsqueda lineal son 0 ms en la mayoría de los casos (para los tamaños más pequeños). Solo cuando el tamaño del arreglo llega a 28,550 y 30,000, el tiempo aumenta a 13 ms.

- Esto es un comportamiento que se espera para la búsqueda lineal. Aunque la búsqueda lineal tiene una complejidad O(n), los tiempos son tan pequeños (0 ms) para los tamaños más pequeños que no se nota una gran diferencia en el tiempo de ejecución.

Comportamiento de la Búsqueda Binaria:

- La búsqueda binaria, como se esperaba con su complejidad O(logn), muestra un comportamiento más contenido.

- Para los tamaños más pequeños, los tiempos de ejecución son 0 ms. Sin embargo, cuando el tamaño aumenta (por ejemplo, para 28,550 y 30,000), el tiempo aumenta a 13 ms.

- Esto es esperado, ya que la búsqueda binaria divide el arreglo en mitades repetidamente, lo que significa que su tiempo de ejecución aumenta mucho más lentamente que el de la búsqueda lineal.

Comparación entre Búsqueda Lineal y Binaria:

- Los tiempos de búsqueda binaria son más estables y pequeños, con un crecimiento más lento conforme aumenta el tamaño del arreglo.

- La búsqueda lineal, aunque en algunos casos tiene tiempos de 0 ms, empieza a mostrar tiempos más grandes (13 ms) cuando el arreglo es más grande. Esto sigue la complejidad esperada O(n).

¿La Tabla Facilita la Relación con la Complejidad Teórica?

Sí, la tabla facilita la relación con la complejidad teórica de los dos algoritmos de búsqueda. A pesar de que los tiempos de ejecución , podemos notar la siguiente relación con la complejidad:

1. **Búsqueda Lineal**:
    - La búsqueda lineal muestra una creciente proporcionalidad en los tiempos a medida que el tamaño del arreglo aumenta, lo que es consistente con su complejidad teórica O(n).
2. **Búsqueda Binaria**:
    - La búsqueda binaria muestra poco crecimiento en los tiempos, lo que es consistente con la complejidad teórica O(logn). El tiempo de ejecución aumenta mucho más lentamente a medida que crece el tamaño del arreglo.

3.TADS, plantillas

Plantilla TAD

TAD R3
Descripción: abstrae un punto del espacio R3, esta se vera el calculo de la distancia de un punto a otro

Conjunto mínimo de datos :

x, entero, contiene la posición en el eje x
y, entero, contiene la posición en el eje y
z, entero, contiene la posición en el eje z

Comportamiento operaciones:

double distancia(const Punto<T> &otro), calcula la distancia al punto recibido como parámetro
setX(x), cambia el valor de x que es recibido como parámetro
setY(y), cambia el valor de y que es recibido como parámetro
setZ(z), cambia el valor de z que es recibido como parámetro
getX(), retorna el valor de x
getY(), retorna el valor de y
getZ(), retorna el valor de z

Archivo R3.h

```
#ifndef R3_H
#define R3_H

template <class T>
class R3 {
private:
   T x, y;
   T z;
public:
   R3();
   R3(T x, T y, T z);
   double distancia(const R3<T> &otro) const;
   T getX() const;
   T getY() const;
   T getZ() const;
   void setX(T nuevoX);
   void setY(T nuevoY);
   void setZ(T nuevoZ);
};

#include "R3.cpp"

#endif
```

Archivo R3.cpp

```
#include "R3.h"
#include <cmath>

template <class T>
R3<T>::R3() {
   x = 0;
   y = 0;
   z = 0;
}


template <class T>
R3<T>::R3(T x, T y, T z) {
   this->x = x;
   this->y = y;
   this->z = z;
}
```

```
template <class T>
void R3<T>::setX(T nuevoX) { x = nuevoX; }
template <class T>
void R3<T>::setY(T nuevoY) { y = nuevoY; }
template <class T>
void R3<T>::setZ(T nuevoZ) { z = nuevoZ; }


template <class T>
T R3<T>::getX() const { return x; }
template <class T>
T R3<T>::getY() const { return y; }
template <class T>
T R3<T>::getZ() const { return z; }

template <class T>
double R3<T>::distancia(const R3<T> &otro) const {
    double dx = otro.x - x;
    double dy = otro.y - y;
    double dz = otro.z - z;
    return sqrt(dx * dx + dy * dy + dz * dz);
}
```

Main.cpp

```
#include <iostream>
#include "R3.h"
using namespace std;

int main() {
    // ===== Puntos con int =====
    R3<int> p1, p2;
    int xi, yi, zi;

    cout << "=== Puntos tipo int ===\n";
    cout << "Ingrese x, y, z para el primer punto: ";
    cin >> xi >> yi >> zi;
    p1.setX(xi);
    p1.setY(yi);
    p1.setZ(zi);

    cout << "Ingrese x, y, z para el segundo punto: ";
    cin >> xi >> yi >> zi;
    p2.setX(xi);
```

```
    p2.setY(yi);
    p2.setZ(zi);

    cout << "Distancia (int): " << p1.distancia(p2) << "\n\n";


    // ===== Puntos con long =====
    R3<long> p3, p4;
    long xl, yl, zl;

    cout << "=== Puntos tipo long ===\n";
    cout << "Ingrese x, y, z para el primer punto: ";
    cin >> xl >> yl >> zl;
    p3.setX(xl);
    p3.setY(yl);
    p3.setZ(zl);

    cout << "Ingrese x, y, z para el segundo punto: ";
    cin >> xl >> yl >> zl;
    p4.setX(xl);
    p4.setY(yl);
    p4.setZ(zl);

    cout << "Distancia (long): " << p3.distancia(p4) << "\n\n";


    // ===== Puntos con double =====
    R3<double> p5, p6;
    double xd, yd, zd;

    cout << "=== Puntos tipo double ===\n";
    cout << "Ingrese x, y, z para el primer punto: ";
    cin >> xd >> yd >> zd;
    p5.setX(xd);
    p5.setY(yd);
    p5.setZ(zd);
  cout << "Ingrese x, y, z para el segundo punto: ";
    cin >> xd >> yd >> zd;
    p6.setX(xd);
    p6.setY(yd);
    p6.setZ(zd);

    cout << "Distancia (double): " << p5.distancia(p6) << "\n";

    return 0;
}
```

Pontificia Universidad JAVERIANA Bogotá

BIBLIOGRAFÍAS

Investigación de algoritmos de complejidad bubble sort, quicksort y heapsort

- GeeksforGeeks. (s.f.). Bubble Sort Algorithm. Recuperado de https://www.geeksforgeeks.org/dsa/bubble-sort-algorithm/
- Techie Delight. (s.f.). Quicksort Algorithm. Recuperado de https://www.techiedelight.com/es/quicksort/
- GeeksforGeeks. (s.f.). Heap Sort. Recuperado de https://www.geeksforgeeks.org/dsa/heap-sort/

Complejidad Teórica de los Algoritmos de Búsqueda Binaria y Lineal:

- W3Schools. (n.d.). *Time complexity of binary search*. Recuperado el 13 de agosto de 2025, de https://www.w3schools.com/dsa/dsa_timecomplexity_binarysearch.php
- PediaExpertos. (n.d.). *Diferencia entre la búsqueda lineal y la búsqueda binaria*. Recuperado el 13 de agosto de 2025, de https://pediaexpertos.com/diferencia-entre-la-busqueda-lineal-y-la-busqueda-binaria/
- Codecademy. (n.d.). *Complexity theory*. Recuperado el 13 de agosto de 2025, de https://www.codecademy.com/articles/complexity-theory