/Creado por Armin Posadas Flores/

Documentación Metodos de ordenamiento

```
template <typename T>
std::vector<T> generateRandomNumbers(typename std::vector<T>::size_type n) {
    std::vector<T> arr(n);
    std::random_device rd;
    std::mt19937 gen(rd());
}
```

Primero se define el template, despues se encarga de generar numeros aleatorios tomando como parametro "n" el cual es el tamaño del vector, despues de inicializa el vector, luego se prepara el generador de numeros random el cual proporciona una fuente de entropía y finalmente se inicializa un Mersenne Twister.

```
if constexpr (std::is_integral<T>::value) {
        std::uniform_int_distribution<T> dis(1, 10000);
        for (typename std::vector<T>::size_type i = 0; i < n; i++) {
            arr[i] = dis(gen);
        }
    } else if constexpr (std::is_floating_point<T>::value) {
        std::uniform_real_distribution<T> dis(0.0, 1.0);
        for (typename std::vector<T>::size_type i = 0; i < n; i++) {
            arr[i] = dis(gen);
        }
    }

    return arr;
```

seguido del codigo anterior se checa si "T" es de tipo integral, en caso de que si se crea una distribucion uniforme que produce integrales del 1 al 100000 a traves de un loop que llena el vector arr.

En caso de que no sea de tipo integral se crea una distribucion que produce una distribución uniforme que produce números de punto flotante entre 0,0 y 1,0 a travez de un loop que llena el vector arr.

Ya por ultimo se envia un valor arr.

```
void bubbleSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                std::swap(arr[j], arr[j + 1]);
            }
        }
    }
}
```

Este es el codigo para el metodo burbuja.

Primero se saca el tamaño del vector volviendo el valor n del mismo que del arr, luego se empieza un loop en donde se itera de 0 a n-1. La variable i representa el paso actual a través de la matriz, durante el loop se compara el valor actual siendo J con el siguiente valor, en caso de que el siguiente valor sea menor que el actual se intercambian de lugar asi hasta que ya no se encuentra un valor menor a travez del loop.

```
void insertionSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

Metodo de incersion.

Primero se saca el tamaño del vector volviendo el valor n del mismo que del arr, luego se empieza un loop en donde se itera de 0 a n-1. Despues se crea la variable "key" para almacenar el elemento actualmente siendo insertado, despues se ejecuta la variable "J" el cual se encuentra antes de la variable "i" y se inicia un loop, durante dicho loop se cambian los elementos en la posicion acomodara de la matrix hacia la derecha hasta que encuentra la posicion correcta de "key" y la inserta en dicha posicion.

```
void selectionSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        std::swap(arr[i], arr[minIndex]);
    }
}
```

Metodo de selection

Primero se saca el tamaño del vector volviendo el valor n del mismo que del arr, luego se empieza un loop en donde se itera de 0 a n-1. Despues a la variable i se le asigna "minindex" el cual indica que es el valor de menor tamaño y luego se pone a buscar por todo el arreglo un valor que sea aun menor y que si lo encuentra a ese valor se le dara el "minindex" para pasar a cambiar lugares, en caso de que no encuentre un valor menor pasara al siguiente valor, se le dara el "minindex" y volver hacer el loop.

```
void shellSort(std::vector<int>& arr) {
    int n = arr.size();
```

```
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}
```

Metodo de shell.

Primero se saca el tamaño del vector volviendo el valor n del mismo que del arr, despues el bucle inicializa el valor de "gap" y lo reduce hasta que sea cero, luego al valor actual se almacena en "temp" mientras que el loop cambia elementos de "gap" hasta que se encuentra la posicion correcta para el valor en "temp" para al final colocarlo en dicha posicion y continuar el loop con el siguiente valor siendo enviado a "temp".

```
void heapify(std::vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }
    if (largest != i) {
        std::swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
```

Metodo de amontonamiento.

primero se inicializa "largest, left, right" en donde largest es "i", left es el valor a la izquierda de "i" y right es el valor de la derecha de "i", luego va a checar si el valor left es mayor a largest, en caso de que si lo sea left se convierte en largest, en caso de que no lo sea se va a checar right y usando la misma logica checara si el valor de right es mayor en caso de que si lo sea este se volvera el nuevo largest, al final en caso de que "i" no sea la raiz esta cambiara con largest y asi por todo el loop.

```
void heapSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
    for (int i = n - 1; i >= 0; i--) {
        std::swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
```

```
}
```

Metodo Heapsort

Primero se saca el tamaño del vector volviendo el valor n del mismo que del arr, luego se empieza un loop en donde se itera de 0 a n-1, luego a "arr, n, i" son llamados a cada nodo asegurandose que el subarbol con raiz a "i" satisfaca la propiedad maxima de heap, despues empieza desde el ultimo elemento y lo mueve al principio, despues se mueve la raiz actual al final del arreglo para que al final se llame al heapify al heap reducido para mantener la propiedad del max-heap.

```
void merge(std::vector<int>& arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    std::vector<int> L(n1), R(n2);
    for (int i = 0; i < n1; i++) {
        L[i] = arr[l + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[m + 1 + j];
    }
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }
    while (i < n1) {
        arr[k++] = L[i++];
    }
    while (j < n2) {
        arr[k++] = R[j++];
    }
}
```

Metodo merge

Par este metodo primero se inicializa los valores "arr, i, m, r" luego a n1 se le consideran todos los elementos de la izquierda y n2 todos los elementos a la derecha, despues se crean vectores temporales para que almacenen los valores de la izquierda y derecha. Luego se crean dos loops en donde cada uno copia los elementos de sus respectivos lados, despues durante el while se usa el arr para almacenas el valor mas pequeño comparando a los dos valores "L" y "R", todo esto continua hasta que todos los elementos del subarray temporal se hallan unido en arr. Por ultimo el ultimo loop se encarga de lidiar con elemento restantes que no se hayan juntado aun.

```
void mergeSort(std::vector<int>& arr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
```

```
    }
```

Metodo mergesort

Para este metodo se checa si "l" es menor a "r" y que en caso de que lo sea divide al array a la mitad luego el metodo llama a ambos lados (izquierda y derecha) del subarray, por ultimo junta ambas mitades para hacer un solo subarray ordenado.

```
int partition(std::vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}
```

Metodo particion

Primero se crean un "array, low, high", despues se escoge al pivote como el elemento mas alto "high" y luego se le da a i al valor que sea menor al pivote para mandarlo para atras, una vez hecho eso se inicia un loop en "if" en donde si el elemento es menor al pivote el elemento en "i" se incrementa, y el elemento actual intercambia con arr[i] asegurandose que todos los elementos se muevan antes del pivote, despues del loop el pivote intercambia con el arr[i+1], poniendo al pivote en la posicion correcta, al final solo se manda el pivote en su posicion correcta.

```
void quickSort(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Metodo quicksort

Primero checa si el valor low < high para ver si ya esta ordenada, luego se llama la funcion "partition" el cual reordena los elementos en el subarray para que el pivote este en la posicion correcta mientras acomoda los elementos menores a la izquierda y mayores a la derecha, por ultimo se utiliza un subarray recursiva para las funciones izquierdas y derechas.

```
void bucketSort(std::vector<float>& arr) {
    int n = arr.size();
    std::vector<std::vector<float>> buckets(n);
    for (int i = 0; i < n; i++) {
        int bi = n * arr[i];
        buckets[bi].push_back(arr[i]);
```

```
    }
    for (int i = 0; i < n; i++) {
        std::sort(buckets[i].begin(), buckets[i].end());
    }
    int index = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < buckets[i].size(); j++) {
            arr[index++] = buckets[i][j];
        }
    }
}
```

Metodo bucketsort

Primero se crea un vector flotante para ser organizados, despues se le da a "n" un valor de tamaño y una cubeta para almacenar cada elemento del input, luego se crea un loop para iterar cada elemento en el array, luego se calcula el index de cada elemento en la cubeta y ya que los elementos estan en el rango de 0,1 al multiplicarlos por "n" se asegura que esten puestos correctamente en la cubeta, luego el arr se agrega a la cubeta correspondiente, despues se crea otro loop que organiza cada cubeta y ya en el ultimo loop se itera cada cubeta y sus elementos copiandolos en el array original mientra los organiza con el index manteniendo su posicion actual en el array original.

```
void countingSort(std::vector<int>& arr, int exp) {
    int n = arr.size();
    std::vector<int> output(n);
    std::vector<int> count(10, 0);
    for (int i = 0; i < n; i++) {
        count[(arr[i] / exp) % 10]++;
    }
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }
    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
    for (int i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}
```

Metodo countingsort

Primero se inicializa "n, output, count", durante el primer loop se itera cada elemento en el input de array, luego se cuenta los digitos en el exponente actual, en el segundo loop se transforma el array count para almacenar el contado de digitos para que cada elemento contenga la posicion del digito en el "output", en el tercer loop se itera sobre el array de input, despues se pone el elemento arr[i] en su posicion correcta en el output basado en el digito actual, luego se decrementa el contado del digito moviendo la siguiente ocurrencia del mismo digito a la izquierda del output, el ultimo loop copia los elementos ordenados de output al array original.

```
void radixSort(std::vector<int>& arr) {
    int max = *std::max_element(arr.begin(), arr.end());
    for (int exp = 1; max / exp > 0; exp *= 10) {
```

```
            countingSort(arr, exp);
        }
    }
```

Primero se inicializa "max" el cual sera el maximo elemento del array hecho para determianar el numero de digitos del numero mayor, despues para el loop se itera cada digito de la posicion empezando desde el menos significante al mas significante, exp sera el exponente que representa la posicion del digito actual, el loop continuara siempre y cuando haya mas digitos a procesar en el numero mas grande para que al final acomode al array basado en el dato que hay en "exp"

```
template <typename T>
bool isSorted(const std::vector<T>& arr) {
    for (size_t i = 0; i < arr.size() - 1; i++) {
        if (arr[i] > arr[i + 1]) {
            return false;
        }
    }
    return true;
}
```

Por ultimo en esta seccion se crea un booliano que indicara si un array ya esta ordenado checando si el digito siguiente es mayor al actual y dara un true, esta funcion sera usada en el siguiente apartado como forma de comprobar que el metodo usado si organizo el array.

Apartado del testeo

Para esta ultima parte todas las funciones funcionan exactamente igual con la unica diferencia siendo el metodo que cada uno manda a llamar.

La funcionalidad basica de estas funciones es el de testear los metodos de ordenamiento usando el comando "TEST" al inicio de esta al igual que un cronometro para ver cuanto tardan cada una, el codigo base va asi:

-Primero se inicia un chronometro automaticamente.

-Luego se inicia el metodo de ordenamiento a usar.

-Una vez que el metodo termina tambien se termina el chronometro.

-Despues para calcular la duracion se resta el tiempo con el que termino, menos, el tiempo con el que empezo.

-Se imprime en la consola el resultado con el input "[Metodo] Time:".

-Por ultimo se verifica que el ordenamiento sea correcto dandole un true.

-Acontinuacion el orden de los metodos en testeo

Testeo para el metodo:

```
TEST(SortingTest, BubbleSortTest) {
    std::vector<int> arr = generateRandomNumbers<int>(100000);
    auto start = std::chrono::high_resolution_clock::now();
    bubbleSort(arr);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Bubble Sort Time: " << elapsed.count() << "s\n";
    ASSERT_TRUE(isSorted(arr));
}
```

Testeo para el metodo: Burbuja

```
TEST(SortingTest, InsertionSortTest) {
    std::vector<int> arr = generateRandomNumbers<int>(100000);
    auto start = std::chrono::high_resolution_clock::now();
    insertionSort(arr);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Insertion Sort Time: " << elapsed.count() << "s\n";
    ASSERT_TRUE(isSorted(arr));
}
```

Testeo para el metodo: Inserción

```
TEST(SortingTest, SelectionSortTest) {
    std::vector<int> arr = generateRandomNumbers<int>(100000);
    auto start = std::chrono::high_resolution_clock::now();
    selectionSort(arr);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Selection Sort Time: " << elapsed.count() << "s\n";
    ASSERT_TRUE(isSorted(arr));
}
```

Testeo para el metodo: Shell

```
TEST(SortingTest, ShellSortTest) {
    std::vector<int> arr = generateRandomNumbers<int>(100000);
    auto start = std::chrono::high_resolution_clock::now();
    shellSort(arr);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Shell Sort Time: " << elapsed.count() << "s\n";
    ASSERT_TRUE(isSorted(arr));
}
```

Testeo para el metodo: Montón

```
TEST(SortingTest, HeapSortTest) {
    std::vector<int> arr = generateRandomNumbers<int>(100000);
    auto start = std::chrono::high_resolution_clock::now();
    heapSort(arr);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Heap Sort Time: " << elapsed.count() << "s\n";
    ASSERT_TRUE(isSorted(arr));
}
```

Testeo para el metodo: Merge

```
TEST(SortingTest, MergeSortTest) {
    std::vector<int> arr = generateRandomNumbers<int>(100000);
    auto start = std::chrono::high_resolution_clock::now();
    mergeSort(arr, 0, arr.size() - 1);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Merge Sort Time: " << elapsed.count() << "s\n";
    ASSERT_TRUE(isSorted(arr));
}
```

Testeo para el metodo: Quick

```
TEST(SortingTest, QuickSortTest) {
    std::vector<int> arr = generateRandomNumbers<int>(100000);
    auto start = std::chrono::high_resolution_clock::now();
    quickSort(arr, 0, arr.size() - 1);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Quick Sort Time: " << elapsed.count() << "s\n";
    ASSERT_TRUE(isSorted(arr));
}
```

Testeo para el metodo: Cubeta

```
TEST(SortingTest, BucketSortTest) {
    std::vector<float> arr = generateRandomNumbers<float>(100000);
    auto start = std::chrono::high_resolution_clock::now();
    bucketSort(arr);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Bucket Sort Time: " << elapsed.count() << "s\n";
    ASSERT_TRUE(isSorted(arr));
}
```

Testeo para el metodo: Por base

```
TEST(SortingTest, RadixSortTest) {
    std::vector<int> arr = generateRandomNumbers<int>(100000);
    auto start = std::chrono::high_resolution_clock::now();
```

```
    radixSort(arr);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Radix Sort Time: " << elapsed.count() << "s\n";
    ASSERT_TRUE(isSorted(arr));
}
```