***מיונים***

# Selection Sort – O(N2)

נמצא כל פעם את המספר הקטן ביותר ונשים אותו בתחילת המערך.

void selection\_sort(int arr[])

    {

        int n = arr.length;

        // One by one move boundary of unsorted subarray

        for (int i = 0; i < n-1; i++)

        {

            // Find the minimum element in unsorted array

            int min\_idx = i;

            for (int j = i+1; j < n; j++)

                if (arr[j] < arr[min\_idx])

                    min\_idx = j;

            // Swap the found minimum element with the first

            // element

            int temp = arr[min\_idx];

            arr[min\_idx] = arr[i];

            arr[i] = temp;

        }

    }

# Insertion Sort – O(N2)

עבור על כל מספר ומיין אותו ביחס למספרים שמיינו עד כה.

void insertion\_sort(int arr[])

    {

        int n = arr.length;

        for (int i=1; i<n; ++i)

        {

            int key = arr[i];

            int j = i-1;

            /\* Move elements of arr[0..i-1], that are

               greater than key, to one position ahead

               of their current position \*/

            while (j>=0 && arr[j] > key)

            {

                arr[j+1] = arr[j];

                j = j-1;

            }

            arr[j+1] = key;

        }

# Bubble Sort – O(N2)

המיון הקלאסי ביותר, ניקח כל מספר ומיין אותו למקומו במערך.

 void bubbleSort(int arr[])

    {

        int n = arr.length;

        for (int i = 0; i < n-1; i++)

            for (int j = 0; j < n-i-1; j++)

                if (arr[j] > arr[j+1])

                {

                    // swap arr[j+1] and arr[i]

                    int temp = arr[j];

                    arr[j] = arr[j+1];

                    arr[j+1] = temp;

                }

    }

# Merge Sort – O(N\*log(N))

נפצל את הטיפול במערך לחלקים קטנים ככל הניתן ואז נחבר.

 // Merges two subarrays of arr[].

    // First subarray is arr[l..m]

    // Second subarray is arr[m+1..r]

    void merge(int arr[], int l, int m, int r)

    {

        // Find sizes of two subarrays to be merged

        int n1 = m - l + 1;

        int n2 = r - m;

        int L[] = new int [n1];

        int R[] = new int [n2];

        /\*Copy data to temp arrays\*/

        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];

        /\* Merge the temp arrays \*/

        int i = 0, j = 0;

        // Initial index of merged subarry array

        int k = l;

        while (i < n1 && j < n2)

        {

            if (L[i] <= R[j])

            {

                arr[k] = L[i];

                i++;

            }

            else

            {

                arr[k] = R[j];

                j++;

            }

            k++;

        }

        /\* Copy remaining elements of L[] if any \*/

        while (i < n1)

        {

            arr[k] = L[i];

            i++;

            k++;

        }

        /\* Copy remaining elements of R[] if any \*/

        while (j < n2)

        {

            arr[k] = R[j];

            j++;

            k++;

        }

    }

    void merge\_sort(int arr[], int l, int r)

    {

        if (l < r)

        {

            // Find the middle point

            int m = (l+r)/2;

            // Sort first and second halves

            sort(arr, l, m);

            sort(arr , m+1, r);

            // Merge the sorted halves

            merge(arr, l, m, r);

        }

    }

# Heap Sort – O(N\*log(N))

נכניס את הנתונים לערימת מקסימום וכל פעם ניקח את הערך המקסימלי.

public void heap\_sort(int arr[])

    {

        int n = arr.length;

        // Build heap (rearrange array)

        for (int i = n / 2 - 1; i >= 0; i--)

            heapify(arr, n, i);

        // One by one extract an element from heap

        for (int i=n-1; i>=0; i--)

        {

            // Move current root to end

            int temp = arr[0];

            arr[0] = arr[i];

            arr[i] = temp;

            // call max heapify on the reduced heap

            heapify(arr, i, 0);

        }

    }

    // To heapify a subtree rooted with node i which is

    // an index in arr[]. n is size of heap

    void heapify(int arr[], int n, int i)

    {

        int largest = i; // Initialize largest as root

        int l = 2\*i + 1; // left = 2\*i + 1

        int r = 2\*i + 2; // right = 2\*i + 2

        // If left child is larger than root

        if (l < n && arr[l] > arr[largest])

            largest = l;

        // If right child is larger than largest so far

        if (r < n && arr[r] > arr[largest])

            largest = r;

        // If largest is not root

        if (largest != i)

        {

            int swap = arr[i];

            arr[i] = arr[largest];

            arr[largest] = swap;

            // Recursively heapify the affected sub-tree

            heapify(arr, n, largest);

        }

    }

# Quick Sort – O(N\*log(N))

נקח את האיבר הראשון נבצע פיצול לפיו ונמיין את שני חלקי המערך מימין ומשמאל.

int partition(int arr[], int low, int high)

    {

        int pivot = arr[high];

        int i = (low-1); // index of smaller element

        for (int j=low; j<high; j++)

        {

            // If current element is smaller than or

            // equal to pivot

            if (arr[j] <= pivot)

            {

                i++;

                // swap arr[i] and arr[j]

                int temp = arr[i];

                arr[i] = arr[j];

                arr[j] = temp;

            }

        }

        // swap arr[i+1] and arr[high] (or pivot)

        int temp = arr[i+1];

        arr[i+1] = arr[high];

        arr[high] = temp;

        return i+1;

    }

    /\* The main function that implements QuickSort()

      arr[] --> Array to be sorted,

      low  --> Starting index,

      high  --> Ending index \*/

    void QuickSort (int arr[], int low, int high)

    {

        if (low < high)

        {

            /\* pi is partitioning index, arr[pi] is

              now at right place \*/

            int pi = partition(arr, low, high);

            // Recursively sort elements before

            // partition and after partition

            sort(arr, low, pi-1);

            sort(arr, pi+1, high);

        }

    }

# Counting Sort – O(N + K)

נבצע מספור של כל איבר, נחבר את הסכומים (מיצג את האינדקס שלהם) ולבסוף נעתיק.

void countSort(int[] arr)

    {

        int max = Arrays.stream(arr).max().getAsInt();

        int min = Arrays.stream(arr).min().getAsInt();

        int range = max - min + 1;

        int count[] = new int[range];

        int output[] = new int[arr.length];

        for (int i = 0; i < arr.length; i++)

        {

            count[arr[i] - min]++;

        }

        for (int i = 1; i < count.length; i++)

        {

            count[i] += count[i - 1];

        }

        for (int i = arr.length - 1; i >= 0; i--)

        {

            output[count[arr[i] - min] - 1] = arr[i];

            count[arr[i] - min]--;

        }

        for (int i = 0; i < arr.length; i++)

        {

            arr[i] = output[i];

        }

    }

# Radix Sort – O(D\*N) (D – number of digits)

נבצע מיון מספור (Counting) שהטווח הוא 10 כל פעם על ספרה אחרת, פעם לפי האחדות, ואז עשרות, מאות וכו'...

    // A function to do counting sort of arr[] according to

    // the digit represented by exp.

    static void countSort(int arr[], int n, int exp)

    {

        int output[] = new int[n]; // output array

        int i;

        int count[] = new int[10];

        Arrays.fill(count,0);

        // Store count of occurrences in count[]

        for (i = 0; i < n; i++)

            count[ (arr[i]/exp)%10 ]++;

        // Change count[i] so that count[i] now contains

        // actual position of this digit in output[]

        for (i = 1; i < 10; i++)

            count[i] += count[i - 1];

        // Build the output array

        for (i = n - 1; i >= 0; i--)

        {

            output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];

            count[ (arr[i]/exp)%10 ]--;

        }

        // Copy the output array to arr[], so that arr[] now

        // contains sorted numbers according to current digit

        for (i = 0; i < n; i++)

            arr[i] = output[i];

    }

    // The main function to that sorts arr[] of size n using

    // Radix Sort

    static void radixsort(int arr[], int n)

    {

        // Find the maximum number to know number of digits

        int m = getMax(arr, n);

        // Do counting sort for every digit. Note that instead

        // of passing digit number, exp is passed. exp is 10^i

        // where i is current digit number

        for (int exp = 1; m/exp > 0; exp \*= 10)

            countSort(arr, n, exp);

    }

# Bucket Sort – O(N)

מיון המיועד למספרים בין 0 ל1.

public static void sort(Integer[] array, int bucketSize) {

if (array.length == 0) {

return;

}

// Determine minimum and maximum values

Integer minValue = array[0];

Integer maxValue = array[0];

for (int i = 1; i < array.length; i++) {

if (array[i] < minValue) {

minValue = array[i];

} else if (array[i] > maxValue) {

maxValue = array[i];

}

}

// Initialise buckets

int bucketCount = (maxValue - minValue) / bucketSize + 1;

List<List<Integer>> buckets = new ArrayList<List<Integer>>(bucketCount);

for (int i = 0; i < bucketCount; i++) {

buckets.add(new ArrayList<Integer>());

}

// Distribute input array values into buckets

for (int i = 0; i < array.length; i++) {

buckets.get((array[i] - minValue) / bucketSize).add(array[i]);

}

// Sort buckets and place back into input array

int currentIndex = 0;

for (int i = 0; i < buckets.size(); i++) {

Integer[] bucketArray = new Integer[buckets.get(i).size()];

bucketArray = buckets.get(i).toArray(bucketArray);

InsertionSort.sort(bucketArray);

for (int j = 0; j < bucketArray.length; j++) {

array[currentIndex++] = bucketArray[j];

}

}