ASmall Sorted array code

int count=0;

int[] SArray = new int[10];

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

SArray[i]=i;

System.out.println(i);

count=count+1;

}

System.out.print("code ran " + count);

}

}

Medium array

int count2=0;

int[] MArray = new int[100];

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

MArray[j]=j;

System.out.println(j);

count2=count2 +1;

}

System.out.print("code medium ran " + count2);

}

}

//Large array

int count3=0;

int[] LArray = new int[1000];

for (int L = 0; L < LArray.length; L++) {

MArray[L]=L;

System.out.println(L);

count3=count3 +1;

}

System.out.print("code large ran " + count3);

}

}

UNSORTED FETCH

package a2;

public class A2

{

public static void main(String[] args) {

// Small array

int[] SArray = new int[10];

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

SArray[i] = i;

}

// Unsorted fetch on small array

int index = unsortedFetch(SArray, 5);

System.out.println("Index of 5 in small array: " + index);

// Medium array

int[] MArray = new int[100];

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

MArray[j] = j;

}

// Unsorted fetch on medium array

index = unsortedFetch(MArray, 50);

System.out.println("Index of 50 in medium array: " + index);

// Large array

int[] LArray = new int[1000];

for (int L = 0; L < LArray.length; L++) {

LArray[L] = L;

}

// Unsorted fetch on large array

index = unsortedFetch(LArray, 500);

System.out.println("Index of 500 in large array: " + index);

}

public static int unsortedFetch(int[] array, int value) {

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

if (array[i] == value) {

return i;

}

}

return -1;

}

}

This Java program demonstrates how to find the index of a specific value in an array using an unsorted fetch algorithm. The program creates a small, medium and large array and initializes them with sequential integers and then performs an unsorted fetch of a value in each array using the unsorted fetch algorithm. The unsorted fetch method takes an array and a value as input and returns the index of the value in the array if it is present or -1 if it is not present. The algorithm simply loops through the array and checks each element for the specified value using an if statement. If the value is found the algorithm returns the index of that element. The program then prints out the index of the specified value in each array to the console using the System.out.println method. This unsorted fetch method has a time complexity of O(n) where n is the length of the input array since it requires looping through the entire array to find the value. This algorithm is inefficient for large arrays especially if multiple fetches are needed since each fetch requires looping through the entire array. In such cases a different search algorithm such as a binary search would be more efficient.

UNSORTED INSERT

public class A2

{

public static void main(String[] args) {

// Small array

int[] SArray = new int[10];

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

SArray[i] = i;

}

// Unsorted insert on small array

int[] SArrayInserted = unsortedInsert(SArray, 3, 100);

System.out.println("Small array with 100 inserted at index 3: " + Arrays.toString(SArrayInserted));

// Medium array

int[] MArray = new int[100];

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

MArray[j] = j;

}

// Unsorted insert on medium array

int[] MArrayInserted = unsortedInsert(MArray, 50, 200);

System.out.println("Medium array with 200 inserted at index 50: " + Arrays.toString(MArrayInserted));

// Large array

int[] LArray = new int[1000];

for (int L = 0; L < LArray.length; L++) {

LArray[L] = L;

}

// Unsorted insert on large array

int[] LArrayInserted = unsortedInsert(LArray, 500, 300);

System.out.println("Large array with 300 inserted at index 500: " + Arrays.toString(LArrayInserted));

}

public static int[] unsortedInsert(int[] array, int index, int value) {

int[] newArray = new int[array.length+1];

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

if (i < index) {

newArray[i] = array[i];

} else {

newArray[i+1] = array[i];

}

}

newArray[index] = value;

return newArray;

}

}

This Java program demonstrates how to insert an element at a specific index in an array using an unsorted insert. The program creates three arrays of small, medium, and large. The unsorted insert algo takes an array, an index and a value as input and returns a new array with the specified value inserted at the specified index. First a new array is created with a length one greater than the input array then loops through the input array and copies its values into the new array but shifts the values up by one index if they are after the insert point. Finally the method sets the new value at the insertion point and returns the new array. The program then prints out the original arrays and the new arrays with the inserted values to the console using the Arrays.toString method. This unsorted insert algorithm has a time complexity of O(n) where n is the length of the input array since it requires copying all elements of the input array to the new array. This method can be inefficient for large arrays especially if multiple insertions are needed since each insertion requires copying the entire array.

BINARY SEARCH

package a2;

import java.util.\*;

public class A2

{

public static void main(String[] args) {

// Small array

int[] SArray = new int[10];

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

SArray[i] = i;

}

// Binary search on small array

int indexS = binarySearch(SArray, 3);

System.out.println("Index of 3 in small array: " + indexS);

// Medium array

int[] MArray = new int[100];

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

MArray[j] = j;

}

// Binary search on medium array

int indexM = binarySearch(MArray, 50);

System.out.println("Index of 50 in medium array: " + indexM);

// Large array

int[] LArray = new int[1000];

for (int L = 0; L < LArray.length; L++) {

LArray[L] = L;

}

// Binary search on large array

int indexL = binarySearch(LArray, 500);

System.out.println("Index of 500 in large array: " + indexL);

}

public static int binarySearch(int[] array, int value) {

int low = 0;

int high = array.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

if (array[mid] == value) {

return mid;

} else if (array[mid] < value) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return -1;

}

}

The number of steps that the binary search algorithm takes to find an element in a sorted array of size n is given by O(log n). The number of steps required by the algorithm grows much slower than the size of the array. For example, in an array of size 1000, the binary search algorithm will take at most log2(1000) = 10 steps to find an element. In an array of size 1 million, the algorithm will take at most log2(100) = 6 steps. And in an array size of log2(100)=6 This is why binary search is considered a very efficient algorithm for searching through large sorted arrays.