Find intersection of two arrays

We do not know the range of numbers. and if the range is very large w.r.t the elements in the array, the binary tree logic works fine.

Method 1)

Prepare a hash table of Array A1 and then search for elements of Array A2 in this table.

Time - O(n)

Space - O(range) // range>>n

Method 2)

Prepare BST of A1 and search for elements of A2 in A1.

Time - O(nLgn)

Space - O(n)

1. sort one array

2. each element from other array binary search in sorted array

o(nlog n

Expand: Union and Intersection of two sorted arrays

For example, if the input arrays are:

arr1[] = {1, 3, 4, 5, 7}

arr2[] = {2, 3, 5, 6}

Then your program should print Union as {1, 2, 3, 4, 5, 6, 7} and Intersection as {3, 5}.

Algorithm Union(arr1[], arr2[]):

For union of two arrays, follow the following merge procedure.

1) Use two index variables i and j, initial values i = 0, j = 0

2) If arr1[i] is smaller than arr2[j] then print arr1[i] and increment i.

3) If arr1[i] is greater than arr2[j] then print arr2[j] and increment j.

4) If both are same then print any of them and increment both i and j.

5) Print remaining elements of the larger array.

#include<stdio.h>

/\* Function prints union of arr1[] and arr2[]

m is the number of elements in arr1[]

n is the number of elements in arr2[] \*/

int printUnion(int arr1[], int arr2[], int m, int n)

{

int i = 0, j = 0;

while(i < m && j < n)

{

if(arr1[i] < arr2[j])

printf(" %d ", arr1[i++]);

else if(arr2[j] < arr1[i])

printf(" %d ", arr2[j++]);

else

{

printf(" %d ", arr2[j++]);

i++;

}

}

/\* Print remaining elements of the larger array \*/

while(i < m)

printf(" %d ", arr1[i++]);

while(j < n)

printf(" %d ", arr2[j++]);

}

/\* Driver program to test above function \*/

int main()

{

int arr1[] = {1, 2, 4, 5, 6};

int arr2[] = {2, 3, 5, 7};

int m = sizeof(arr1)/sizeof(arr1[0]);

int n = sizeof(arr2)/sizeof(arr2[0]);

printUnion(arr1, arr2, m, n);

getchar();

return 0;

}

Time Complexity: O(m+n)

Algorithm Intersection(arr1[], arr2[]):

For Intersection of two arrays, print the element only if the element is present in both arrays.

1) Use two index variables i and j, initial values i = 0, j = 0

2) If arr1[i] is smaller than arr2[j] then increment i.

3) If arr1[i] is greater than arr2[j] then increment j.

4) If both are same then print any of them and increment both i and j.

#include<stdio.h>

/\* Function prints Intersection of arr1[] and arr2[]

m is the number of elements in arr1[]

n is the number of elements in arr2[] \*/

int printIntersection(int arr1[], int arr2[], int m, int n)

{

int i = 0, j = 0;

while(i < m && j < n)

{

if(arr1[i] < arr2[j])

i++;

else if(arr2[j] < arr1[i])

j++;

else /\* if arr1[i] == arr2[j] \*/

{

printf(" %d ", arr2[j++]);

i++;

}

}

}

/\* Driver program to test above function \*/

int main()

{

int arr1[] = {1, 2, 4, 5, 6};

int arr2[] = {2, 3, 5, 7};

int m = sizeof(arr1)/sizeof(arr1[0]);

int n = sizeof(arr2)/sizeof(arr2[0]);

printIntersection(arr1, arr2, m, n);

getchar();

return 0;

}

Time Complexity: O(m+n)

Please write comments if you find any bug in above codes/algorithms, or find other ways to solve the same problem.

Expand (From leetcode): Find the intersection of two sorted arrays.

Let’s called array1 as A and array2 as B, each with size m and n.

The obvious brute-force solution is to scan through each element in A, and for each element in A, scan if that element exist in B. The running time complexity is O(m\*n). Not good! Can we do better? Absolutely!

First, we know that both arrays are sorted. Can we somehow use this information to our advantage?

We can apply binary search to search if an element of A exist in B. So, the only modification from the brute-force approach is modifying linear search to binary search. This seems like a good improvement, we manage to reduce the complexity to O(m\*lg(n)).

Of course, you know you can trade space for running time by using a hash table. Is it really useful? We can definitely hash each element in B to an array index (takes O(n) time). Therefore, to find if an element of A exist in B, it would require just O(1) time. The complexity improves to O(m+n).

But there is a problem, what if n is very big? (ie, n is one billion!). We have a problem here. The hash table will either requires a large amount of memory space, or there will be lots of collision in the table, which makes access time no longer O(1) time. Therefore, using a hash table is not a good general solution to this problem. Besides, using hash table DO NOT require that the array being sorted in the first place.

Here is the most important observation in order to solve this problem. Both arrays ARE sorted. This provides a very important clue. We must make full use of this information that they ARE in fact sorted.

We can have two index, which both starts at zero. Compare the two first elements of A and B. If A[0] is greater than B[0], we increase index of B by one. If B[0] is greater than A[0], we increase index of A by one. If they are equal, we know an intersection has occurred, so add it to the list and increment index of A and B by one. Once either index reaches the end of A or B, we have found all the intersections of A and B.

The complexity of this approach is still O(m+n), but it does not requires any extra space that a hash table requires. The complexity is O(m+n) because in the worse case, there would be no intersection between the two arrays, and we need to increment first index a total of m times and increment second index a total of n times, which is a total of m+n times.

Below is the C++ code for this approach:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

vector<int> findIntersection(vector<int> A, vector<int> B) {

vector<int> intersection;

int n1 = A.size();

int n2 = B.size();

int i = 0, j = 0;

while (i < n1 && j < n2) {

if (A[i] > B[j]) {

j++;

} else if (B[j] > A[i]) {

i++;

} else {

intersection.push\_back(A[i]);

i++;

j++;

}

}

return intersection;

}

Do you think that this approach always work better? Not necessarily… Think what happens when n is very large, say one billion…

Compare this approach with the binary search approach.

O(m+n) and O(m\*lg(n))

lg(n) is much smaller than n when n is very big. However, this does not necessarily means binary search is better in this case. In fact, binary search approach is only better when m << n (m is very small compared to n). For example, when m = 1 and n = one billion, which method will you choose? Binary search is definitely the winner here.

All of our above approaches assume that we have enough space to load both arrays to the memory. Here are some interesting questions to ponder about:

i) What if elements of array B is stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

ii) How will the complexity change in this case? Are there any factors you need to consider?

iii) How do you change your solution to adapt to this situation?

Expand (From tekmarathon): Two arrays are unsorted

Here we pick one of the array and load it into hash implemented data structure, i.e, HashSet and then proceeds further to find intersection of elements.

Since hashed data structure’s complexity is O(1), the total complexity to find intersection of elements the complexity would become

Algorithm Time Complexity: O(m) + O(n)\*O(1)

Algorithm Space Complexity: O(m)

1

2

3

4

5

6

7

8

9

10

11

12

public void intersect1(int[] a, int[] b) {

HashSet<Integer> hs = new HashSet<Integer>();

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

hs.add(b[i]);

}

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

if(hs.contains(a[i])) {

System.out.println(a[i]+" is present in both arrays");

}

}

}

pros:

best algorithm when compared to all others provided one implements appropriate hashcode method

cons:

when the size of the data structure grows too high, it might lead to hash collisions