Finding Best Duo In Sports Analytics

18CSS202J- Design Analysis and Algorithms

Report

Submitted by

V.R.Rishendra-(RA2111026010221)
J.Mohith -RA2111026010241)

Submitted to

Dr.Sivashankar G

Assistant Professor, Department of Computational Intelligence

in partial fulfilment for the award of the degree of

BACHELOR OF TECHNOLOGY

In

COMPUTER SCIENCE ENGINEERING WITH SPECIALIZATION IN ARTIFICIAL INTELLIGENCE AND MACHINE LEARING

SCHOOL OF COMPUTING
DEPARTMENT OF COMPUTATIONAL INTELLIGENCE
COLLEGE OF ENGINEERING AND TECHNOLOGY
SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR- 603 203

April 2023

SRM INSTITUTION OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that this Course Project Report titled **“Finding Best Duo In Sports Analytics”** is the bonafide work done by V.R.Rishendra-(RA2111026010221),J.Mohith-RA2111026010241)who carried out under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other work.

SIGNATURE	**HEAD OF THE DEPARTMENT**
Faculty In-Charge	**Dr. R Annie Uthra**
Dr.Sivashankar G	Professor and Head ,
Assistant Professor	Department of Computational Intelligence,
Department of CINTEL	SRM Institute of Science and Technology
SRM Institute of Science and Technology	Kattankulathur Campus, Chennai
Kattankulathur Campus, Chennai	

Table of Contents

Problem statement:

Problem: Given an array of integers, find two numbers such that they add up to a specific target number.

The problem of finding two numbers in an array that add up to a specific target number is a common problem in computer science and programming. The problem statement is as follows:

Given an array of integers and a target number, find two numbers in the array that add up to the target number. If such numbers do not exist, return null or an empty result.

To solve this problem, various algorithmic techniques can be used, such as brute force search, hashing, and binary search.

Hashing :

Hashing is used to index and retrieve items in a database because it is faster to find the item using the shortest hashed key than to find it using the original value. It is also used in many encryption algorithms.A hash code is generated by using a key, which is a unique value. Hashing is a technique in which given key field value is converted into the address of storage location of the record by applying the same operation on it.The advantage of hashing is that allows the execution time of basic operation to remain constant even for the larger side.

Binary search:

Binary search method is a searching algorithm that follows the divide and conquer technique. This is based on the idea of ordered searching where the algorithm divides the sorted list into two halves and performs the searching. If the key value to be searched is lower than the mid-point value of the sorted list, it performs searching on the left half; otherwise it searches the right half. If the array is unsorted, linear search is used to determine the position.

Sample Input and Output:

Hashing Approach:

Input:
Array: [1, 4, 45, 6, 10, 8]
Target: 16

Output:
Yes

Explanation:

In the given input array, the pair (6, 10) adds up to the target value of 16. Therefore, the output is "Yes".

Binary search Approach:

Input:
Array: [2, 5, 8, 12, 16, 23]
Target: 20

Output:
Yes

Explanation:
In the given input array, the pair (8, 12) adds up to the target value of 20. Therefore, the output is "Yes".

Problem Explanation

Finding the best duo or combination of players is a common problem in sports analytics, especially in team sports like basketball, football, and soccer. By analyzing the statistics of individual players and looking for combinations that add up to a specific value or meet certain criteria, teams can identify optimal player pairings and improve their overall performance.

In basketball, finding the best duo or combination of players is an important problem in sports analytics. Coaches and analysts use statistical analysis to identify player combinations that work well together and improve team performance.

One common approach is to look for players who complement each other's skills. For example, a team might look for a player who is strong in defense to pair with a player who excels in offense. By combining these skills, the team can create a more balanced and effective lineup.

Another approach is to look for players whose statistics add up to a specific value. For example, a team might look for two players whose points, rebounds, and assists add up to a certain value. This can help to identify the best duo on the court and maximize the team's scoring potential.

To solve this problem, various algorithmic techniques can be used, such as linear programming, integer programming, and machine learning. Linear programming involves optimizing a linear objective function subject to linear constraints. Integer programming involves solving optimization problems where the decision variables are restricted to integer values. Machine learning algorithms can be used to learn patterns in the data and identify optimal player combinations based on historical performance.

Overall, finding the best duo or combination of players is a challenging problem in sports analytics. However, with the right data and analytical tools, teams can identify optimal player pairings and improve their overall performance.

Design technique – Hashing:

In this approach, we use a hash map to store the difference between the target and the current number. Then, we iterate over the array, and for each number, we check if the difference between the target and the current number is already present in the hash map. If it is present, we return the current number and the difference. If it is not present, we add the current number to the hash map.

This problem can be solved efficiently by using the technique of hashing. Use a hash_map to check for the current array value x(let), if there exists a value target_sum-x which on adding to the former gives target_sum. This can be done in constant time.

Algorithm:

```
function findTwoNumbersWithTarget(array, target) {
    for (let i = 0; i < array.length; i++) {
        for (let j = i + 1; j < array.length; j++) {
            if (array[i] + array[j] === target) {
                return [array[i], array[j]];
            }
        }
    }
    return null;
}
```

- Initialize an empty hash table s.
- Do the following for each element A[i] in A[]
- If s[x – A[i]] is set then print the pair (A[i], x – A[i])
- Insert A[i] into s.

Illustration:

arr[] = {0, -1, 2, -3, 1}
sum = -2

Now start traversing:

Step 1: For ‘0’ there is no valid number ‘-2’ so store ‘0’ in hash_map.
Step 2: For ‘-1’ there is no valid number ‘-1’ so store ‘-1’ in hash_map.
Step 3: For ‘2’ there is no valid number ‘-4’ so store ‘2’ in hash_map.
Step 4: For ‘-3’ there is no valid number ‘1’ so store ‘-3’ in hash_map.
Step 5: For ‘1’ there is a valid number ‘-3’ so answer is 1, -3

```
unordered_set s

for(i=0 to end)

  if(s.find(target_sum – arr[i]) == s.end)

    insert(arr[i] into s)

  else

    print arr[i], target-arr[i]
```

Code:

```
// C program to check if given array
// has 2 elements whose sum is equal
// to the given value

#include <stdio.h>
#define MAX 100000

// NOTE: Works only if range elements is limited
// target - arr[i] >= 0 && target - arr[i] < MAX

void printPairs(int arr[], int arr_size, int target)
{
	int i, temp;

	/*initialize hash set as 0*/
	int s[MAX] = { 0 };

	for (i = 0; i < arr_size; i++) {
		temp = target - arr[i];
		if (s[temp] == 1) {
			printf("Yes");
			return;
		}
		s[arr[i]] = 1;
	}

	printf("No");
}

/* Driver Code */
int main()
{
	int A[] = { 1, 4, 45, 6, 10, 8 };
	int target = 16;
	int arr_size = sizeof(A) / sizeof(A[0]);

	printPairs(A, arr_size, target);

	getchar();
	return 0;
}
```

Output:

```
Yes
```

Time complexity:

O(N), As the whole array is needed to be traversed only once.

Design technique-Binary search:

In this approach, we first sort the array in ascending order. Then, we iterate over the array, and for each number, we perform a binary search on the rest of the array to find the number that adds up to the target with the current number.

Algorithm:

```
function binarySearch(array, start, end, target) {
   while (start <= end) {
      const mid = Math.floor((start + end) / 2);
      if (array[mid] === target) {
         return mid;
      }
      if (array[mid] < target) {
         start = mid + 1;
      } else {
         end = mid - 1;
      }
   }
   return -1;
}


function findTwoNumbersWithTarget(array, target) {
   array.sort((a, b) => a - b);
   for (let i = 0; i < array.length; i++) {
      const difference = target - array[i];
      const j = binarySearch(array, i + 1, array.length - 1, difference);
      if (j !== -1) {
         return [array[i], array[j]];
      }
   }
   return null;
}
```

- Sort the array in non-decreasing order.
- Traverse from 0 to N-1
- Initialize searchKey = sum – A[i]
- If(binarySearch(searchKey, A, i + 1, N) == True
- Return True
- Return False

Code:

```
// C++ program to check if given array
// has 2 elements whose sum is equal
// to the given value

#include <bits/stdc++.h>
using namespace std;

bool binarySearch(int A[], int low, int high, int searchKey)
{

	while (low <= high) {
		int m = low + (high - low) / 2;

		// Check if searchKey is present at mid
		if (A[m] == searchKey)
			return true;

		// If searchKey greater, ignore left half
		if (A[m] < searchKey)
			low = m + 1;

		// If searchKey is smaller, ignore right half
		else
			high = m - 1;
	}

	// if we reach here, then element was
	// not present
	return false;
}



bool checkTwoSum(int A[], int arr_size, int sum)
{
	int l, r;

	/* Sort the elements */
	sort(A, A + arr_size);

	// Traversing all element in an array search for
	// searchKey
	for (int i = 0; i < arr_size - 1; i++) {

		int searchKey = sum - A[i];
		// calling binarySearch function
		if (binarySearch(A, i + 1, arr_size - 1, searchKey)
			== true) {
			return true;
		}
	}
	return false;
}
```

```
/* Driver program to test above function */
int main()
{
	int A[] = { 1, 4, 45, 6, 10, -8 };
	int n = 14;
	int arr_size = sizeof(A) / sizeof(A[0]);

	// Function calling
	if (checkTwoSum(A, arr_size, n))
		cout << "Yes";
	else
		cout << "No";

	return 0;
}
```

Output:

```
Yes
```

Time Complexity:

O(NlogN)

Complexity Analysis:

Hashing:

The hashing approach works well for this problem as it provides a time complexity of O(n) which is optimal and efficient. It involves iterating through the input array and using a hash table to store the elements already traversed. As we traverse the array, we check if the target minus the current element exists in the hash table. If it does, then we have found a pair of elements that add up to the target value. Otherwise, we insert the current element into the hash table. The space complexity of this approach is O(n) since we are storing all the elements in the hash table. The hashing approach works well for this problem when the range of elements is limited.

- The initialization of the hash set takes constant time, which can be denoted as O(1).
- The for loop executes n times, where n is the size of the input array. Therefore, the time complexity of the for loop is O(n).
- The operations inside the for loop, such as calculating the difference between the target and the current array element and checking if the difference exists in the hash set, take constant time. Therefore, the time complexity of these operations is O(1).
- Since the for loop has a time complexity of O(n) and the operations inside the for loop have a time complexity of O(1), the overall time complexity of the program is O(n).

Therefore, the step by step time complexity of the program can be expressed as:

$T(n) = O(1) + O(n) * O(1)$
$= O(1) + O(n)$
$= O(n)$

Binary Search:

The binary search approach involves sorting the array and then iterating through it to find the pair of elements that add up to the target value. Since binary search requires a sorted array, we first need to sort the input array, which takes O(n log n) time complexity. After sorting, we perform binary search for each element in the array to find the other element that will add up to the target value. The time complexity of binary search is O(log n), so performing binary search for each element in the array takes O(n log n) time complexity. The space complexity of the binary search approach is O(1), since we are not using any additional data structures to store the array elements.

- The time complexity of the given program is O(n log n) in terms of T(n), where n is the size of the input array. The main steps contributing to this time complexity are:
- Sorting the array using the C++ std::sort function, which has a time complexity of O(n log n).
- Traversing the array and for each element, performing a binary search on the remaining elements to find the second element that sums up to the target. This step has a time complexity of O(n log n) as well because the binary search algorithm takes log n time for each search, and it is performed n times in the worst case scenario.

Therefore, the overall time complexity of the program is O(n log n) because it is dominated by the sorting and searching steps.

Conclusion:

In conclusion, both the hashing and binary search approaches are efficient and work well for solving the given problem. However, the hashing approach has a slightly better time complexity of O(n), making it faster than the binary search approach. Therefore, the hashing approach is the more efficient approach for this problem, especially when the range of elements is limited.

Reference:

https://www.javatpoint.com/
https://www.tutorialspoint.com/index.htm
https://www.geeksforgeeks.org/