# Ice Cream Parlor

locked

by dheeraj

Editorial by AllisonP

## Naive Solution

1. Run a nested loop where the first (outer) loop corresponds to the first flavor and the second (inner) loop corresponds to the second flavor.
2. Once the first valid solution is found (i.e., $c_i + c_j = m$), break the loop and print your answer.

```java
// Naive Java Solution
import java.util.*;

public class Solution {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int t = scan.nextInt();

        while(t-- > 0) {
            int m = scan.nextInt();
            int n = scan.nextInt();
            int[] menu = new int[n];

            for(int i = 0; i < n; i++) {
                menu[i] = scan.nextInt();
            }

            for(int i = 0; i < n; i++) {
                for(int j = i + 1; j < n; j++) {
                    if(menu[i] + menu[j] == m) {
                        System.out.println( (i + 1) + " " + (j + 1) );
                        break;
                    }
                }
            }
        }
        scan.close();
    }
}
```

The complexity for this is $O(n^2)$, which will time out if the data sets are large.

## $O(n)$ solution

Let $cost[i]$ denote the cost, $c_i$, of the flavor with ID $i$, and let $cost[j]$ denote the cost, $c_j$, of the flavor with ID $j$ where $1 \leq i, j \leq n$ and $i \neq j$.
Let $frequency[c_i]$ denote the number of times that $c_i$ occurs in the $cost$ array.

1. Loop over the $cost$ array.

## Statistics

Difficulty: Easy
Time Complexity: $O(n)$ Required
Knowledge: Array, Loop
Publish Date: Aug 16 2014

2. Consider two distinct flavors having costs $c_i$ and $c_j$. Because Sunny and Johnny want to spend all $m$ dollars during each trip to the store, we know that $c_i + c_j = m$ for the correct solution. From this, we can say that if we have some first flavor with cost $c_i$, it can only be paired with flavor $c_j = m - c_i$. If we find the $c_j$ satisfying that equation in the $cost$ array, then we have our answer. We determine this by checking if $c_j$ exists in the $frequency$ array; if it does, then we have our answer. If $c_j$ does not exist in the $frequency$ array, then we simply check the next possible $c_i$.

**Note:** The cost of the two flavors may be the same! In that case, you must make sure that the cost occurs *at least* twice (i.e., $frequency[c_i] \geq 2$).

Set by nabila_ahmed

Problem Setter's code :

C++

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

int frequency[10004];
int price[10004];

int main() {
    int test, n, m, c_i, c_j;

    cin >> test;

    for (int t = 1; t <= test; t++) {

        memset(frequency, 0, sizeof(frequency));

        cin >> m >> n;

        for (int i = 1; i <= n; i++) {
            scanf("%d", &price[i]);
            frequency[price[i]]++;
        }

        int c_i_index;

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

            c_i = price[i];

            // Decrease the frequency to avoid counting value twice
            frequency[c_i]--;

            int tmp_c_j = m - price[i];

            if (frequency[tmp_c_j] > 0) {
                c_j = tmp_c_j;
                c_i_index = i;
                printf("%d", i);
                break;
            }

            frequency[c_i]++;  //Restore original frequency
        }

        // Search for c_j in the rest of the array
        for (int i = c_i_index + 1; i <= n; i++) {
            if (price[i] == c_j) {
                printf(" %d\n", i);
                break;
            }
        }
    }
}
```

Tested by AllisonP

Problem Tester's code :

Java

```java
import java.util.*;

class IceCream implements Comparable{
        public int cost;
        public int id;

        public IceCream(int cost, int index) {
                this.cost = cost;
                this.id = index;
        }

        @Override
        public int compareTo(Object o) {
                IceCream iceCream = (IceCream) o;

                return this.cost - iceCream.cost;
        }

        @Override
        public boolean equals(Object o){
                IceCream iceCream = (IceCream) o;

                return iceCream.cost == this.cost;
        }

}

class Solution {
    public IceCream[] menu;
    public int n;
    public int m;

    public Solution(IceCream[] menu, int n, int m) {
        this.menu = menu;
        Arrays.sort(this.menu);
        this.n = n;
        this.m = m;
    }

    public int binarySearch(int min, int max, int search) {

        int middle = (min + max) >> 1;

        while(min <= max) {
            // Search value is found
            if( menu[middle].cost == search ) {
                if(max - min <= 1 ) {
                    return menu[middle].id;
                }

                // else, continue searching
                max = middle;
            }
            // Else, continue looking for search value
            else {
                if ( menu[middle].cost < search ) {
                    // Continue searching right
                    min = middle + 1;
                }
                else {
                    // Continue searching left
                    max = middle - 1;
                }
            }
            // Set new middle at halfway point
            middle = (min + max) >> 1;

        } // END WHILE, first > last

        // No price matching 'search' exists in the menu
        return -1;
    }

    public void solve() {
        // Search menu for a valid pair of prices
        for(int i = 0; i < n - 1 ; i++) {
            // Set desired price that will match the cost at index i
            int search = m - menu[i].cost;

            // If search < menu[i], then no match exists for that cost beca
use the menu array is sorted
            if(search >= menu[i].cost) {

                // Search for the desired value starting at the first index
to the right of the flavor at index i
                int index = binarySearch(i + 1, n - 1, search);

                // Index of valid second flavor was returned by binary sear
ch
                if( index != -1 ) {
                    System.out.println( Math.min(menu[i].id, index) + " " +
Math.max(menu[i].id, index));
                    break;
                }
                // Else, continue looping and check the next value.
            }
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int t = scanner.nextInt();
        while(t-- > 0) {
            int m = scanner.nextInt();
            int n = scanner.nextInt();
            IceCream[] menu = new IceCream[n];

            // Fill flavor menu and sort
            for (int i = 0; i < n; i++) {
                menu[i] = new IceCream(scanner.nextInt(), i + 1);
            }

            Solution solution = new Solution(menu, n, m);
            solution.solve();
        }
        scanner.close();
    }
}
```

# Sherlock and Anagrams

locked

by darkshadows

Editorial by pkacprzak

Two string are anagrams if and only if for every letter occurring in any of them the number of its occurrences is equal in both the strings.

This definition is crucial and will lead to the solution. Since the only allowed letters are lowercase English letters, from $a$ to $z$, the alphabet size is constant and its size is $26$. This allows us to assign a constant size signature to each of the substring of $s$.

A signature of some string $w$ will be a tuple of $26$ elements where the $i$-th element denotes the number of occurrences of the $i$-th letter of the alphabet in $w$.

So, for example, if $w = "\mathrm{mom}"$ then its signature is $[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$, so the only non-zero elements are the ones corresponding to letter $m$ with value of 2 and letter $o$ with value of 1.

Notice, that any string that is an anagram of $"\mathrm{mom}"$ will have the same signature as $"\mathrm{mom}"$, and every string that is not an anagram of $"\mathrm{mom}"$ will definitely have a different signature.

This concept of signatures allows the following approach.

Let's iterate over all substrings of $s$ and for each fixed substring let's compute its signature and add that signature to signatures hashmap, where **`signatures[sig]`** denotes the number of substrings of $s$ with a signature **`sig`**.

Finally, the only remaining thing to do is to get the number of pairs of substrings of $s$ that are anagrams. It's easy to do having our hashmap. Notice that if there are $n$ substrings of $s$ with signature **`sig`**, then they can form $n \cdot (n-1)/2$ pairs of substrings with signature **`sig`**, so we can just iterate over all values in the hashmap and for each value $n$ add $n \cdot (n-1)/2$ to the final result.

The below, commented code, in Python, illustrates this exact approach.

The time complexity is $O(|s|^3)$ since we iterate over all $O(|s|^2)$ substrings of $s$ and for each substring we compute its signature in $O(|s|)$ time. It's worth to mention that each operation on hashmap has constant running time since our signatures have a constant size, i.e. $26$ which is the size of our alphabet. Otherwise, if the alphabet size is not constant, this approach will have $O(|s|^3 \cdot ALPHABET\_SIZE)$ time complexity.

```
import string

q = int(raw_input())
```

## Statistics

| | |
|---|---|
| Difficulty: | Medium |
| Time Complexity: | O(\|s\|^3) Required |
| Knowledge: | Strings |
| Publish Date: | Dec 03 2014 |

```
ALPHABET = string.ascii_lowercase

for _ in xrange(q):
    s = raw_input()
    signatures = {}

    signature = [0 for _ in ALPHABET]
    for letter in s:
        signature[ord(letter)-ord(ALPHABET[0])] += 1

    # iterate over all substrings of s
    for start in range(len(s)):
        for finish in range(start, len(s)):
            # initialize substring signature
            signature = [0 for _ in ALPHABET]
            for letter in s[start:finish+1]:
                signature[ord(letter)-ord(ALPHABET[0])] += 1
            # tuples are hashable in contrast to lists
            signature = tuple(signature)
            signatures[signature] = signatures.get(signature, 0) + 1

    res = 0
    for count in signatures.values():
        res += count*(count-1)/2
    print res
```

# Two Strings

locked

by zxqfd555

Problem Submissions Leaderboard Discussions Editorial

## Editorial by AllisonP

There are two concepts involved in solving this challenge:

1. Understanding that a single character is a valid substring.
2. Deducing that we only need to know that the two strings *have* a common substring — we don't need to know what that substring is.

Thus, the key to solving this challenge is determining whether or not the two strings share a common character because if they have a common character then they have a common substring of lengh 1.

To do this, we create two sets, $a$ and $b$, where each set contains the unique characters that appear in the string it's named after. Because sets don't store duplicate values, we know that the size of our sets will never exceed the $26$ letters of the English alphabet. In addition, the small size of these sets makes finding the intersection very quick.

If the intersection of the two sets is empty, we print `NO` on a new line; if the intersection of the two sets is *not* empty, then we know that strings $a$ and $b$ share one or more common characters and we print `YES` on a new line.

```
# Complete the twoStrings function below.
def twoStrings(s1, s2):
    # create sets of unique characters
    # and test for intersection
    if set(s1)&set(s2):
        return "YES"
    else:
        return "NO"
```

## Statistics

| | |
|---|---|
| Difficulty: | Easy |
| Time Complexity: | $O(n)$ Required |
| Knowledge: | Set theory, Strings, Characters |
| Publish Date: | Nov 14 2014 |

## Tested by AllisonP

Problem Tester's code :

```
import java.util.*;

public class Solution {
    static Set<Character> a;
    static Set<Character> b;

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
        for(int i = 0; i < n; i++) {
            a = new HashSet<Character>();
            b = new HashSet<Character>();
            //creating the set of string1
            for(char c : scan.next().toCharArray()) {
                a.add(c);
            }
            //creating the set of string2
            for(char c : scan.next().toCharArray()) {
                b.add(c);
            }

            // store the set intersection in set 'a'
            a.retainAll(b);

            System.out.println( (a.isEmpty()) ? "NO" : "YES" );
        }
        scan.close();
    }
}
```