

Count pairs with difference



Given an array of distinct integer values, count the number of pairs of integers that have difference k . For example, given the array $\{1, 7, 5, 9, 2, 12, 3\}$ and the difference $k = 2$, there are four pairs with difference 2: $(1, 3)$, $(3, 5)$, $(5, 7)$, $(7, 9)$

Solved it

Great! Here's how to solve the problem efficiently:

Problem Statement

Given:

- An array of **distinct integers**
- An integer value k

Task:

- Count the number of **pairs (a, b)** such that $b - a = k$.

Example

Input:

```
plaintext
```

```
array = {1, 7, 5, 9, 2, 12, 3}  
k = 2
```

Valid pairs with difference $k = 2$:

- $(1, 3)$
- $(3, 5)$
- $(5, 7)$
- $(7, 9)$

Output: 4

Efficient Solution (Using Hash Set)

Time complexity: $O(n)$

Space complexity: $O(n)$

🔧 Steps:

1. Add all elements to a hash set for $O(1)$ lookup.
2. For each number `num` in the array:
 - Check if `num + k` exists in the set.
 - If yes, count it as a valid pair.

✅ Python Code:

```
python

def count_pairs_with_diff(arr, k):
    nums = set(arr)
    count = 0

    for num in arr:
        if num + k in nums:
            count += 1

    return count

# Example
arr = [1, 7, 5, 9, 2, 12, 3]
k = 2
print(count_pairs_with_diff(arr, k)) # Output: 4
```

🧠 Notes:

- This method assumes **order of the pair doesn't matter**, i.e., (1, 3) is same as (3, 1) depending on context — but in this case, we only check for `num + k`.
- Since elements are **distinct**, we don't have to worry about duplicate pairs.

Let me know if you want the solution in another language or want to extend it for duplicates or other variations.



c# solution

Here's how you can solve the same problem in **C#** efficiently using a **HashSet**:

✅ C# Solution (Using HashSet)

```
csharp

using System;
using System.Collections.Generic;

class Program
{
    static int CountPairsWithDifference(int[] arr, int k)
    {
        HashSet<int> nums = new HashSet<int>(arr);
    }
}
```

```
int count = 0;

foreach (int num in arr)
{
    if (nums.Contains(num + k))
    {
        count++;
    }
}

return count;
}

static void Main()
{
    int[] arr = { 1, 7, 5, 9, 2, 12, 3 };
    int k = 2;
    int result = CountPairsWithDifference(arr, k);
    Console.WriteLine("Number of pairs with difference " + k + ": " + result);
}
}
```

Output

plaintext

Number of pairs with difference 2: 4

Explanation

- The array is converted to a `HashSet<int>` for $O(1)$ lookup time.
- For every number in the array, we check if `num + k` exists in the set.
- If yes, it's a valid pair, and we increment the counter.

Let me know if you'd like to modify this for:

- Negative differences
- Bidirectional checking (e.g., $|a - b| = k$)
- Returning actual pairs instead of just the count.



what is the complexity of
`HashSet<int> nums = new HashSet<int>(arr);`

and

`nums.Contains(num + k)`

Great question! Let's break down the **time and space complexity** of each line:

✓ 1. `HashSet<int> nums = new HashSet<int>(arr);`

This line:

- **Initializes** a new `HashSet<int>` using the input array.
- Internally, it loops through the array and adds each element to the set.

🕒 Time Complexity:

- Each insertion into a `HashSet` is **$O(1)$** on average.
- For n elements in `arr`, total complexity = **$O(n)$**

🧠 Why?

Because the hash table behind the scenes distributes elements into buckets, and assuming a good hash function with minimal collisions, each insert is constant time.

🧠 Note:

In worst-case (very rare, e.g., massive hash collisions), insertion could degrade to **$O(n^2)$** , but this almost never happens in practice with primitives like `int`.

📁 Space Complexity:

- Also **$O(n)$** , because the `HashSet` stores all n elements.

✓ 2. `nums.Contains(num + k)`

This line:

- Checks whether a particular value exists in the `HashSet`.

🕒 Time Complexity:

- **$O(1)$** on average, due to constant-time lookup in a hash table.
- Again, in worst-case with hash collisions, lookup could be **$O(n)$** , but it's rare.

🧠 Summary

Operation	Average Time Complexity	Space Complexity
<code>new HashSet<int>(arr)</code>	$O(n)$	$O(n)$
<code>nums.Contains(num + k)</code>	$O(1)$	—

Let me know if you want to explore the worst-case scenarios or internal hash mechanics in .NET further!



what is the constant-time lookup in a hash table?

Great question!

✓ What does "constant-time lookup in a hash table" mean?

When we say a hash table provides **constant-time ($O(1)$) lookup**, we mean that **finding whether a value exists** (like using `.Contains()` in `HashSet<T>` or `Dictionary<TKey, TValue>`) takes the **same amount of time regardless of how many items** are stored—**on average**.

But how is this achieved?

🧠 How a Hash Table Works (at a High Level)

1. Hash Function:

- Each key (like an `int` in a `HashSet<int>`) is passed through a **hash function**.
- This function produces a **hash code** (a number), which maps the key to a **bucket** in the internal array.

2. Bucket Indexing:

- The hash code is used to compute an **index** in the internal array where the value is stored.
- Example:

```
csharp
int index = Math.Abs(key.GetHashCode()) % array.Length;
```

3. Lookup:

- When you call `nums.Contains(x)`, it hashes `x`, finds the appropriate bucket, and checks whether `x` is present.
- **If no collisions occur**, this operation takes **constant time**, i.e., does not depend on the size of the collection.

⚡ What Can Affect Constant Time?

1. Hash Collisions:

- When two different keys hash to the same bucket.
- The bucket might store multiple values (e.g., as a linked list or tree), so you may need to scan through them.
- This can degrade lookup from $O(1)$ to $O(n)$ in the worst case.

2. Load Factor:

- Defined as **number of elements / number of buckets**.
- When the load factor gets too high, collisions become more frequent.

- To avoid this, hash tables **resize** and **rehash** the items when they grow.

3. Quality of Hash Function:

- A good hash function distributes values uniformly across buckets.
- In .NET, primitive types like `int`, `string`, etc., have well-optimized hash functions.

Real-World .NET (HashSet<int>) Lookup Example

csharp

```
HashSet<int> set = new HashSet<int> { 10, 20, 30 };  
bool exists = set.Contains(20); // O(1) on average
```

- `20.GetHashCode()` is computed.
- That hash code maps to an index.
- The bucket at that index is checked.
- If `20` is found, `true` is returned.

Summary

Concept	Description
O(1) Lookup	Direct access to an internal array index using a hash function, with minimal computation.
Depends on	Hash function quality, load factor, and collision resolution strategy.
Worst Case	$O(n)$, but very rare in practice for well-designed hash tables and data.

Let me know if you'd like to dive deeper into how .NET's `HashSet<T>` or `Dictionary<TKey, TValue>` handles collisions (like using chaining or probing)!



what is the bucket?

Excellent — let's unpack what a **bucket** is in the context of a **hash table**.

What Is a Bucket?

In a **hash table**, a **bucket** is a **slot or container** in an **internal array** where key-value pairs (or just keys, in the case of a `HashSet`) are stored.

Think of a Hash Table Like This:

Imagine an array:

```
plaintext
```

```
Index:  0    1    2    3    4    5    6
        [ ]  [ ]  [ ]  [ ]  [ ]  [ ]  [ ]
```

Each index in this array is a **bucket**.

When you insert a value (e.g., 42) into a hash table:

1. Hash Code:

Compute a hash code:

```
csharp
int hash = 42.GetHashCode(); // e.g., returns 42
```

2. Index Calculation (Modulus):

```
csharp
int index = Math.Abs(hash) % buckets.Length; // let's say buckets.Length = 7
// index = 42 % 7 = 0
```

3. **Bucket 0** is used to store the value 42.

Why Use Buckets?

Because a hash table doesn't use direct value-based indexing like an array (`arr[42]`), it relies on hash codes to map arbitrary keys (like strings, objects, or numbers) into a **limited range of indices** (the array size). Each of those indices is a bucket.

What Happens When Two Keys Hash to the Same Bucket?

This is called a **hash collision**.

Example:

```
csharp
int index1 = key1.GetHashCode() % bucketCount;
int index2 = key2.GetHashCode() % bucketCount;
```

If `index1 == index2`, both `key1` and `key2` map to the **same bucket**.

The bucket must now hold **multiple values**, typically using:

- A **linked list** (classic method)
- A **balanced tree** (for performance under high collision, like in Java's `HashMap`)
- **Probing** (store the next available slot — used in open addressing)

What's Inside a Bucket in .NET?

In .NET's `HashSet<T>` and `Dictionary<TKey, TValue>`:

- Each bucket **contains the index of the first entry** in a linked structure (like a list or chain) of entries that hash to that bucket.
- The entries themselves are stored in an internal `Entry[]` array (for performance reasons), and each entry has:
 - The actual key or value
 - The full hash code
 - A pointer to the **next** entry in the bucket (if there's a collision)

So the buckets array acts more like a directory pointing to chains of entries.

Summary

Term	Meaning
Bucket	A slot in the hash table array used to store (or reference) entries that share the same hash index.
Why?	To organize entries by hash and handle collisions.
How?	Values are hashed to an index (bucket), and stored or chained if needed.
Used in	<code>HashSet<T></code> , <code>Dictionary<TKey, TValue></code> , etc.

Let me know if you want a visual diagram or want to dive into the internals of how .NET's `HashSet<T>` handles buckets!



when use `HashSet.Contains(20)` what happen internally?

Great question! Let's go step-by-step into what happens **internally** when you call:

```
csharp
HashSet<int> set = new HashSet<int> { 10, 20, 30 };
bool exists = set.Contains(20);
```

What Happens Internally When `HashSet.Contains(20)` is Called?

Step-by-Step:

✓ 1. Get the Hash Code

First, .NET computes the hash code of the value you're checking:


```
csharp
```

```
int hashCode = EqualityComparer<int>.Default.GetHashCode(20);
```

- For an `int`, the hash code is usually just the integer itself:
`20.GetHashCode()` → 20

✓ 2. Find the Bucket Index

The hash code is **mapped to a bucket** in the internal array:

```
csharp
```

```
int bucketIndex = hashCode % buckets.Length;
```

- Let's say the internal bucket array has length 7 (this is managed internally by .NET).
- So:
`bucketIndex = 20 % 7 = 6`

So the value is expected to be in **bucket 6**.

✓ 3. Check the Entries in That Bucket

Each bucket points to an **entry** (or a chain of entries) in the `entries[]` array, like so:

```
csharp
```

```
// Internally:
struct Entry<T>
{
    public int hashCode;
    public T value;
    public int next; // index of next entry in same bucket (-1 if none)
}
```

.NET goes to that bucket and:

- Checks if there's any entry.
- If there is, it compares:
 - **Hash code**: does it match 20's hash code?
 - **Actual value** (via `.Equals()`): does the stored value equal 20?

If there's a **collision** (more than one value hashed to bucket 6), .NET follows the **linked entries** (via the `next` field) and compares each value until it either:

- Finds 20, and returns `true`.
- Reaches the end of the chain (no match), and returns `false`.

✓ 4. Result Returned

- If the value is found: `true`
- If not found: `false`

So:

```
csharp
bool exists = set.Contains(20); // returns true
```

Visual Example (Simplified)

Let's say:

- Internal array size = 7
- You added { 10, 20, 30 }

Hashes:

- $10 \% 7 = 3 \rightarrow$ bucket 3
- $20 \% 7 = 6 \rightarrow$ bucket 6
- $30 \% 7 = 2 \rightarrow$ bucket 2

When you call `set.Contains(20)`, it:

- Goes to **bucket 6**
- Sees an entry with value 20
- Returns `true`

Optimizations Used by HashSet<T>

- Uses a **prime number** for array size to reduce collisions.
- Uses **chaining** (linked list or pointer-style `next` indexing) for collisions.
- Uses `EqualityComparer<T>.Default` for comparing hash codes and actual values, which allows custom comparison for user-defined types.

Summary Table

Step	Operation	Time Complexity
1	Compute hash code	O(1)
2	Map hash code to bucket index	O(1)
3	Check entry and resolve collision	O(1) avg / O(n) worst
4	Return result	O(1)

Let me know if you'd like to see the actual .NET Core source code behind `HashSet.Contains()` or how this works for custom types like `class Person` with overridden `Equals()` and `GetHashCode()`!



what happen when use List.Contains(20)

Great follow-up! Let's now look at what happens internally when you call:

```
csharp
List<int> list = new List<int> { 10, 20, 30 };
bool exists = list.Contains(20);
```

What Happens Internally When List.Contains(20) is Called?

Unlike `HashSet`, which uses a **hash table**, `List<T>` is just a **resizable array** under the hood. So `Contains()` performs a **linear search**.

✓ Step-by-Step Breakdown:

1. Loop Through Items

Internally, the `List<T>.Contains(T item)` method loops through the internal array `_items[]`, checking each element **one by one**:

```
csharp
public bool Contains(T item)
{
    if ((Object)item == null)
    {
        for (int i = 0; i < _size; i++)
            if ((Object)_items[i] == null) return true;
        return false;
    }
    else
    {
        EqualityComparer<T> c = EqualityComparer<T>.Default;
        for (int i = 0; i < _size; i++)
        {
            if (c.Equals(_items[i], item)) return true;
        }
        return false;
    }
}
```

2. Compare Values

For each element, it checks:

```
csharp
```

```
EqualityComparer<T>.Default.Equals(currentItem, itemToCheck)
```

For `int`, this is just a basic `==` comparison.

3. Return Result

- If it finds a match: returns `true`
- If it reaches the end without a match: returns `false`

Time Complexity

Operation	Complexity
<code>List.Contains</code>	$O(n)$

- Because it **checks each item one-by-one** until it finds a match (or not).
- Worst-case: item is not in the list, or is at the end.

Example

```
csharp
```

```
List<int> list = new List<int> { 10, 20, 30 };
bool exists = list.Contains(20); // true
```

- Checks 10 → no match
- Checks 20 → match → returns `true`

Comparison: `List<T>` vs `HashSet<T>`

Feature	<code>List<T></code>	<code>HashSet<T></code>
Lookup (Contains)	$O(n)$ (linear search)	$O(1)$ average (hash table)
Allows Duplicates	✓ Yes	✗ No (unique items only)
Preserves Order	✓ Yes	✗ No
Best for	Small or ordered data	Fast lookups, no duplicates

Summary

- `List.Contains(x)` performs a **linear search**.
- It **compares each element** using `EqualityComparer<T>.Default`.
- Time complexity is **$O(n)$** .

Let me know if you want to look at performance trade-offs between **List**, **HashSet**, and **Dictionary** in real-world usage!