Cache Misses

Description

Modern computers use a **cache** to store a **small amount of data** in a high-speed memory, significantly improving overall system performance. Even though programs frequently access **large amounts of data**, by keeping only a **subset of the main memory** in the cache, access time can be greatly reduced.

When a program runs, it makes a sequence of **memory requests**:

$$\{\boldsymbol{r1}, \boldsymbol{r2}, \dots, \boldsymbol{rn}\}$$

Each request corresponds to a **specific data element** that the program needs to access. If the requested element is already in the cache, it is considered a **cache hit**, meaning it is accessed instantly. However, if the requested element is not in the cache, it results in a **cache miss**, requiring the system to retrieve it from the main memory, which is significantly slower.

Since the cache can only hold **k elements at a time**, when it is full and a new request arrives, the system must decide **which element to evict** to make space for the new one. The **goal of the cache management strategy is to minimize the total number of cache misses** over the entire request sequence. An example of a program that accesses four distinct elements **{A, B, C, D}** might generate the following request sequence:

D, B, D, B, D, A, C, D, B, A, C, B

If the **cache size is k = 3**, each time a new element is requested, the system must determine which elements to **keep in the cache** and which one to **evict**, ensuring that future requests have the highest chance of being cache hits.

Typically, caching is considered an **on-line problem**, meaning the system must make real-time decisions about which elements to retain in the cache **without knowledge of future requests**. However, in this problem, we consider an **off-line caching version**, where the **entire sequence of requests** is **known in advance**. This allows us to use an optimal **cache replacement strategy**, where elements are evicted **based on their future use**, ensuring the **fewest possible cache misses**.

A crucial rule in this problem is that **every request must be cached if it is not already present**. It is **not allowed** to ignore or skip caching a request, even if it will never be used again. If the cache is full, an eviction **must** take place before adding a new request. The **eviction strategy should** minimize the cache misses.

Your task is to design an **efficient algorithm** to solve this offline caching problem to **minimize** the cache misses?

Given:

- An integer cacheSize (**K**), representing the **maximum number of elements** the cache can store.
- An array of **N** requests, representing the **sequence of memory requests**.

Required:

- The **minimum number of cache misses** that occurs while executing the requests.

Complexity

complexity of your algorithm should be **less than O($K \times N^2$);**

Function: **Implement it!**

```
static public int RequiredFunction(int cacheSize, string[] requests)
```

`PROBLEM_CLASS.cs includes this method.`

Examples

Test Case 1:

- Cache Size: 2
- Requests: ["R0", "R1", "R2", "R1", "R2", "R2", "R1"]

Step-by-Step Execution

1. R0 is not in the cache → Miss → Cache = [R0]
2. R1 is not in the cache → Miss → Cache = [R0, R1]
3. R2 is not in the cache → Miss → Evict R0 → Cache = [R1, R2]
4. R1 is already in the cache → Hit
5. R2 is already in the cache → Hit
6. R2 is already in the cache → Hit
7. R1 is already in the cache → Hit

✅ Min Cache Misses = 3

Test Case 2:

- Cache Size: 3
- Requests: ["R10", "R11", "R12", "R10", "R11", "R12"]

Step-by-Step Execution

1. R10 is not in the cache → Miss → Cache = [R10]
2. R11 is not in the cache → Miss → Cache = [R10, R11]
3. R12 is not in the cache → Miss → Cache = [R10, R11, R12]
4. R10 is in the cache → Hit
5. R11 is in the cache → Hit
6. R12 is in the cache → Hit

✅ Min Cache Misses = 3

Test Case 3:

- Cache Size: 3
- Requests: ["R20", "R21", "R22", "R23", "R24", "R25", "R26"]

Step-by-Step Execution

1. R20 is not in the cache → Miss → Cache = [R20]
2. R21 is not in the cache → Miss → Cache = [R20, R21]
3. R22 is not in the cache → Miss → Cache = [R20, R21, R22]
4. R23 is not in the cache. Cache is full → Evict R20 → Cache = [R21, R22, R23] → Miss
5. R24 is not in the cache. Cache is full → Evict R21 → Cache = [R22, R23, R24] → Miss
6. R25 is not in the cache. Cache is full → Evict R22 → Cache = [R23, R24, R25] → Miss
7. R26 is not in the cache. Cache is full → Evict R23 → Cache = [R24, R25, R26] → Miss

✅ Min Cache Misses = 7

Test Case 4:

- Cache Size: 3
- Requests: ["R30", "R31", "R32", "R30", "R33", "R34", "R30", "R31", "R32"]

Step-by-Step Execution

1. R30 is not in the cache → Miss → Cache = [R30]
2. R31 is not in the cache → Miss → Cache = [R30, R31]
3. R32 is not in the cache → Miss → Cache = [R30, R31, R32]
4. R30 is in the cache → Hit
5. R33 is not in the cache. Cache is full → Evict R32 → Cache = [R30, R31, R33] → Miss
6. R34 is not in the cache. Cache is full → Evict R33 → Cache = [R30, R31, R34] → Miss
7. R30 is in the cache → Hit
8. R31 is in the cache → Hit
9. R32 is not in the cache. Cache is full → Evict R34 → Cache = [R30, R31, R32] → Miss

✅ Min Cache Misses = 6

Test Case 5:

- Cache Size: 2
- Requests: ["R40", "R41", "R42", "R41", "R41", "R40", "R40", "R41", "R41", "R42"]

Step-by-Step Execution

1. R40 is not in the cache → Miss → Cache = [R40]
2. R41 is not in the cache → Miss → Cache = [R40, R41]
3. R42 is not in the cache. Cache is full → Evict R40 → Cache = [R41, R42] → Miss
4. R41 is in the cache → Hit
5. R41 is in the cache → Hit

6. R40 is not in the cache. Cache is full → Evict R42 → Cache = [R41, R40] → Miss
7. R40 is in the cache → Hit
8. R41 is in the cache → Hit
9. R41 is in the cache → Hit
10. R42 is not in the cache. Cache is full → Evict R40 → Cache = [R41, R42] → Miss

✅ Min Cache Misses = 5

Test Case 6:

- Cache Size: 4
- Requests: ["R50", "R51", "R52", "R53", "R50", "R54", "R55", "R56", "R50", "R57", "R56", "R54", "R51", "R57", "R50"]

Step-by-Step Execution

✅ Min Cache Misses = 9

C# Help

Getting the size of 1D array

```
int size = array1D.GetLength(0);
```

Getting the size of 2D array

```
int size1 = array2D.GetLength(0);

int size2 = array2D.GetLength(1);
```

Creating 1D array

```
int [] array1D = new int [size]
```

Creating 2D array

```
int [,] array2D = new int [size1, size2]
```

Sorting single array

Sort the given array "items" in ascending order

```
Array.Sort(items);
```

Sorting parallel arrays

Sort the first array "master" and re-order the 2nd array "slave" according to this sorting

```
Array.Sort(master, slave);
```