

Username: Pralay Patoria **Book:** Coding Interviews: Questions, Analysis & Solutions. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Space-Time Trade-Off

In many situations, time and space performance cannot be improved at the same time and a choice has to be made between time efficiency and space efficiency. The space-time trade-off is a situation where the memory consumption can be reduced at the cost of slower program execution or, conversely, the execution time can be reduced at the price of more memory consumption.


In the last couple of decades, the RAM space and hard drive space have been getting cheaper at a much faster rate than other computer components, including CPUs. Therefore, it is more often preferred to sacrifice some space for time-efficiency optimization.

There are many strategies to reduce execution time at the cost of more memory use, and the most common one is to utilize lookup tables in order to avoid recalculations. A table (an array or a 2D matrix for most cases) is utilized that reduces execution time but increases the amount of memory needed.

As discussed in the section *Dynamic Programming and Greedy Algorithms*, arrays or 2D matrices are utilized for many dynamic programming algorithms to avoid recalculations on overlapping subproblems. Similarly, ugly numbers already found are stored to get next ugly numbers. Ugly numbers are discussed in detail in the following section.

It costs $O(n)$ time to sequentially search a character in a string with length n . If a hash table with ASCII values as keys is utilized, it only costs $O(1)$ time to get a character. It accelerates character searches with little more memory consumption because there are only 256 ANSI characters in total. Many coding problems in the interview can be solved with this strategy, as discussed in the section *Hash Tables for Characters*.

Ugly Numbers

 **Question 75** If a number only has factors 2, 3, and 5, it is an *ugly number*. For example, 6 and 10 are two ugly numbers, but 14 is not because it has a factor of 7. Usually 1 is considered to be the first ugly number. What is the arbitrary k^{th} ugly number?

Check Every Number One by One

According to the definition of ugly numbers, they have factors 2, 3, and 5. If a number has a factor 2, it is divided by 2 continuously. If it has a factor 3, it is divided by 3 continuously. Similarly, it is divided by 5 continuously if it has a factor 5. If the number becomes 1 finally after divisions, it is an ugly number, as shown in [Listing 7-10](#).

Listing 7-10. C# Code to Check an Ugly Number

```
bool IsUgly(int number) {
    while (number % 2 == 0)
        number /= 2;
    while (number % 3 == 0)
        number /= 3;
    while (number % 5 == 0)
        number /= 5;

    return (number == 1) ? true : false;
}
```

The % operator is used to check whether a number n has a factor m . If n has a factor m , the result of $n \% m$ is 0.

Numbers are verified one by one with the method `IsUgly`, as shown in [Listing 7-11](#).

Listing 7-11. C# Code to Check an Ugly Number (Version 1)

```
int GetUglyNumber_Solution1(int index) {
    if (index <= 0)
        return 0;

    int number = 0;
    int uglyFound = 0;

    while (uglyFound < index) {
        ++number;

        if (IsUgly(number)) {
            ++uglyFound;
        }
    }

    return number;
}
```

The solution in this code looks straightforward, but it is not efficient because it wastes time on non-ugly numbers. Let's try to find a more efficient solution.

Store Found Ugly Numbers into an Array

The new solution only spends time on ugly numbers. According to the definition of ugly numbers, the solution gets a bigger ugly number if it multiplies an ugly number with 2, 3, or 5. An array is created to store found ugly numbers. Any elements in the array except the first one are the multiplication results of an ugly number on the left side by 2, 3, or 5. The first element in the array is initialized as 1.

The key to this solution is how to keep the array sorted. Suppose that there are already some ugly numbers in the array and the greatest one is M . Let's analyze how to get the next ugly number.

The next ugly number should be the multiplication result of an existing ugly number by 2, 3, or 5. First, it multiplies all existing numbers by 2. Some multiplication results may be less than or equal to M , and they should be already in the array, so they are discarded. There may be several multiplication results greater than M , but only the first one is important and others will be recalculated later. The first multiplication result by 2 greater than M is defined as M_2 .

Similarly, it also multiplies all existing ugly number by 3 and 5 and defines the first numbers greater than M as M_3 and M_5 respectively. The next ugly number is the minimum one among M_2 , M_3 , and M_5 .

Is it necessary to multiply all existing ugly numbers by 2, 3, and 5? Actually, it is not. Since the existing ugly numbers are sorted, there is a special ugly number T_2 in the array. The multiplication results of numbers before T_2 by 2 are less than M . It stores the index of T_2 and updates the index when a new ugly number is found. Similarly, there are also T_3 and T_5 for factors 3 and 5. In each round, it starts from T_2 , T_3 , and T_5 to find the next ugly number.

Listing 7-12 shows the sample code of this solution.

Listing 7-12. C# Code to Check an Ugly Number (Version 2)

```
int GetUglyNumber_Solution2(int index) {
    if (index <= 0)
        return 0;

    int[] uglyNums = new int[index];
    uglyNums[0] = 1;
    int nextUglyIndex = 1;

    int index2 = 0;
    int index3 = 0;
    int index5 = 0;

    while (nextUglyIndex < index) {
        int min = Math.Min(uglyNums[index2] * 2, uglyNums[index3] * 3);
        min = Math.Min(min, uglyNums[index5] * 5);

        uglyNums[nextUglyIndex] = min;

        while (uglyNums[index2] * 2 <= uglyNums[nextUglyIndex])
            ++index2;
        while (uglyNums[index3] * 3 <= uglyNums[nextUglyIndex])
            ++index3;
        while (uglyNums[index5] * 5 <= uglyNums[nextUglyIndex])
            ++index5;

        ++nextUglyIndex;
    }

    int ugly = uglyNums[nextUglyIndex - 1];
    return ugly;
}
```

In this code, the local variables `index2`, `index3` and `index5` are the corresponding indexes of T_2 , T_3 , and T_5 .

The second solution is more efficient from the perspective of execution time because it only takes time on ugly numbers. However, it consumes more memory than the first solution because it creates an array to store known ugly numbers. If it tries to find the 1500th ugly number, it creates an array for 1500 integers and the size is 6KB. This amount of memory is not a big problem in most scenarios. In general, it sacrifices a little space for much better time efficiency.

Source Code:


075_UglyNumbers.cs

Test Cases:

- Functional Test Cases (Input 2, 3, 4, 5, 6, ...)

- Boundary Test Cases (Input 0 and 1)
- Performance Test Cases (Input a large index, such as 1500)

Hash Tables for Characters

 **Question 76** Implement a function to find the first character in a string that only appears once. For example, the output is the character 'l' when the input is "google".

Our naive solution for this problem involves scanning the input string from its beginning to end. The current scanned character is compared with every one behind it. If there is no duplication after it, it is a character appearing once. Since it compares each character with $O(n)$ ones behind it, the overall time complexity is $O(n^2)$ if there are n characters in a string.

In order to get the number of occurrences for each character in a string, a data container is needed. We need to get and update the occurrence number of each character in a string, so the data container is used to assign an occurrence number to a character. Hash tables fulfill this kind of requirement, in which keys are characters and values are their occurrence numbers in a string.

It is necessary to scan strings twice. When a character is visited, we increase the corresponding occurrence number in the hash table during the first scanning. In the second round of scanning, whenever a character is visited, we also check its occurrence number in the hash table. The first character with occurrence number 1 is the required output.

Hash tables are complex, and they are not implemented in the C++ standard template library. Therefore, we have to implement one by ourselves.

Characters have eight bits, so there are only 256 variances. We can create an array with 256 numbers, in which indexes are ASCII values of all characters and numbers are their occurrence numbers in a string. That is to say, we have a hash table whose size is 256, with ASCII values of characters as keys.

It is time for coding after the solution is clear. [Listing 7-13](#) demonstrates some sample code.

Listing 7-13. C++ Code for First Character Appearing Once in a String

```
char FirstNotRepeatingChar(char* pString) {
    if(pString == NULL)
        return '\0';

    const int tableSize = 256;
    unsigned int hashTable[tableSize];

    for(unsigned int i = 0; i < tableSize; ++i)
        hashTable[i] = 0;

    char* pHashKey = pString;
    while(*(pHashKey) != '\0')
        hashTable[*pHashKey++]++;

    pHashKey = pString;
    while(*pHashKey != '\0') {
        if(hashTable[*pHashKey] == 1)
            return *pHashKey;

        pHashKey++;
    }

    return '\0';
}
```

In this code, it costs $O(1)$ time to increase the occurrence number for each character. The time complexity for the first scanning is $O(n)$ if the length of string is n . It takes $O(1)$ time to get the occurrence number for each character, so it costs $O(n)$ time for the second scanning. Therefore, the overall time it costs is $O(n)$.

In the meantime, an array with 256 numbers is created, whose size is 1K. Since the size of the array is constant no matter how long the input string, the space complexity of this algorithm is $O(1)$.

Source Code:

076_FirstNotRepeatingChar.cpp

Test Cases:

- Functional Test Cases (One or more or no characters in a string appear only once)
- Boundary Test Cases (There is only one character in a string; the string is empty)
- Robust Test Cases (The pointer of the input string is `NULL`)

Question 77 Implement a function to find the first character in a stream that only appears once at any time while reading the stream. For example, when the first two characters “go” are read from a stream, the first character which appears once is the character ‘g’. When the first six characters “google” are read, the first character appearing only once is ‘l’.

Characters are read one by one from a stream. A container is needed to store the indexes of characters in the stream. The first time the character is inserted into the container, its index is stored. If the character has been inserted into the container before, it appears multiple times in the stream and it can be ignored, so its index is updated to a special value (a negative value).

In order to improve performance, it is necessary to check whether a character has been inserted before in $O(1)$ time. Inspired by the solution of the preceding problem, the container is a hash table, which can be implemented as an array. The index of each element in the array is the ASCII value of a character, and the corresponding value is the index of the character in the stream. [Listing 7-14](#) contains sample code.

Listing 7-14. C++ Code for First Character Appearing Once in a Stream

```
class CharStatistics {
public:
    CharStatistics() : index (0) {
        for(int i = 0; i < 256; ++i)
            occurrence[i] = -1;
    }

    void Insert(char ch) {
        if(occurrence[ch] == -1)
            occurrence[ch] = index;
        else if(occurrence[ch] >= 0)
            occurrence[ch] = -2;

        index++;
    }

    char FirstAppearingOnce() {
        char ch = '\0';
        int minIndex = numeric_limits<int>::max();
        for(int i = 0; i < 256; ++i) {
            if(occurrence[i] >= 0 && occurrence[i] < minIndex) {
                ch = (char)i;
                minIndex = occurrence[i];
            }
        }

        return ch;
    }

private:
    int occurrence[256];

    int index;
};
```

In this code, an element in the array `occurrence[i]` is for the character with ASCII value i . Every element in the array is initialized to -1. When the character with ASCII value i is read from the stream and inserted into the hash table for the first time, `occurrence[i]` is updated as the index of the character in the stream. If the character has been inserted before (the value of `occurrence[i]` is greater than or equal to 0), `occurrence[i]` is updated to -2.

If we are going to get the first character that appears only once in the deserialized stream so far, it is only necessary to scan the array `occurrence` to get the minimum index, which is greater than or equal to 0, as implemented in the function `FirstAppearingOnce`.


Source Code:

077_FirstCharAppearingOnce.cpp

Test Cases:

- Functional Test Cases (Input an arbitrary stream of characters)

- Boundary Test Cases (There is only one character in the input stream; all characters in the input stream are unique/duplicated)

 **Question 78** Given two strings, how do you delete characters contained in the second string from the first string? For example, if all characters in the string “aeiou” are deleted from the string “We are students.”, the result is “W r stdnts.”.

Suppose that we are going to delete from a string s_1 the characters that are contained in another string s_2 . Characters in s_1 are scanned one after another. When a character is visited, we have to check whether it is in s_2 . If we search in s_2 sequentially, it costs $O(m)$ time when the length of s_2 is m . Therefore, the overall time complexity is $O(mn)$ if the length of s_1 is n .

Inspired by the solution of the preceding problems, a hash table built from s_2 can be utilized to improve efficiency. The hash table is implemented as an array with length 256. The i^{th} element is for the character with ASCII value i . If the character with ASCII value i is contained in s_2 , the i^{th} element is set as 1; otherwise, it is set as 0. This solution can be implemented with the code shown in [Listing 7-15](#).

Listing 7-15. C Code to Delete Characters Contained in Another String

```
void DeleteCharacters(char* pString, char* pCharsToBeDeleted) {
    int hashTable[256];

    const char* pTemp = pCharsToBeDeleted;

    char* pSlow = pString;
    char* pFast = pString;

    if(pString == NULL || pCharsToBeDeleted == NULL)
        return;

    memset(hashTable, 0, sizeof(hashTable));

    while (*pTemp != '\0') {
        hashTable[*pTemp] = 1;
        ++ pTemp;
    }

    while (*pFast != '\0') {
        if(hashTable[*pFast] != 1) {
            *pSlow = *pFast;
            ++ pSlow;
        }
        ++ pFast;
    }

    *pSlow = '\0';
}
```


It only costs $O(1)$ time to check whether a character is in s_2 or not with a hash table, so the overall time complexity is $O(n)$ if the length of s_1 is n .

Source Code:

078_DeleteCharacters.c

Test Cases:

- Functional Test Cases (Some or all characters of s_1 are contained in s_2 ; no characters of s_1 are contained in s_2)
- Boundary Test Cases (s_1 and/or s_2 are empty)
- Robust Test Cases (The pointers to s_1 and/or s_2 are `NULL`)

 **Question 79** Please implement a function to delete all duplicated characters in a string and keep only the first occurrence of each character left. For example, if the input is string “google”, the result after deletion is “gole”.

All characters in a string are scanned. When a character is visited, we have to know whether it appeared in the string before. It costs $O(n)$ time to search sequentially in a string with length n . A hash table can be utilized to facilitate character search. Values corresponding to all characters (keys) in the hash table are initialized as 0. When a character is visited for the first time, its corresponding value in the hash table is updated to 1. When the value of a character is already 1, it indicates that the character is duplicated and should be deleted. It costs only $O(1)$ time to check whether a character is duplicated with such a hash table.

The sample code is shown in [Listing 7-16](#).

Listing 7-16. C Code to Delete Duplicated Characters in a String

```

void DeletedDuplication(char* pString) {

    int hashTable[256];

    char* pSlow = pString;

    char* pFast = pString;

    if(pString == NULL)

        return;

    memset(hashTable, 0, sizeof(hashTable));

    while (*pFast != '\0') {

        *pSlow = *pFast;

        if(hashTable[*pFast] == 0) {

            ++ pSlow;

            hashTable[*pFast] = 1;

        }

        ++pFast;

    }

    *pSlow = '\0';

}

```

Source Code:

079_DeleteDuplicatedCharacters.c

Test Cases:

- Functional Test Cases (All/Some/No characters in the input string are duplicated)
- Boundary Test Cases (The input string is empty)
- Robust Test Cases (The pointer to the input string is `NULL`)



Question 80 If two English words have the same characters and the occurrence number of each character is also identical respectively, they are anagrams. The only difference between a pair of anagrams is the order of characters. For example, “silent” and “listen”, “evil” and “live” are two pairs of anagrams. Please implement a function to verify whether two words are a pair of anagrams.

Two strings of a pair of anagrams have the same set of characters. The only difference in them is the order of characters. Therefore, they will become the same string if they are sorted. For example, both “silent” and “listen” become “eilnst” after they are sorted. It costs $O(n \log n)$ time to sort a string with n characters. Let’s explore more efficient solutions.

A data container is used to store the occurrence number of each character. Each record in the container is composed of a character and its occurrence number. To solve this problem, it scans all characters in a string one by one. It checks the existence of the scanned character in the container. If the character already exists, its occurrence number increases; otherwise, a new record about the scanned character is inserted, with occurrence number as 1. There are two requirements of the data container: (1) each record maps a character to an integer number, and (2) each record can be accessed and updated efficiently. A hash table fulfills these requirements.

In order to solve the preceding problems in C/C++, a hash table is implemented as an array in which the index of an element is the key and the element is the value. If the problem is solved in Java, the solution might be simpler because there is a type `HashMap` for hash tables. The code in [Listing 7-17](#) is based on

`HashMap` in Java.

Listing 7-17. Java Code for Anagrams

```

boolean areAnagrams(String str1, String str2) {

    if(str1.length() != str2.length())

        return false;

    Map<Character, Integer> times = new HashMap<Character, Integer>();

    for(int i = 0; i < str1.length(); ++i) {

        Character ch = str1.charAt(i);

        if(times.containsKey(ch))

```

```

        times.put(ch, times.get(ch) + 1);
    }
    else
        times.put(ch, 1);
}

for(int i = 0; i < str2.length(); ++i) {
    Character ch = str2.charAt(i);
    if(!times.containsKey(ch))
        return false;

    if(times.get(ch) == 0)
        return false;

    times.put(ch, times.get(ch) - 1);
}

return true;
}

```

In this code, it increases the occurrence numbers when it scans the string `str1` and decreases the occurrence numbers when it scans the string `str2`. If two strings compose a pair of anagrams, all occurrence numbers in the hash map `times` should be 0 eventually.

It scans both strings `str1` and `str2` once. When a character is scanned, it accesses a record in the hash table and updates it. Both operations cost $O(1)$ time. Therefore, it costs $O(n)$ time if the length of strings is n . It allocates some auxiliary space to accommodate the hash map. If strings only contain ASCII characters, there are 256 characters at most, so there are 256 records in the hash map at most. Therefore, the space efficiency is $O(1)$.

Source Code:

080_Anagram.java

Test Cases:

- Functional Test Cases (Pairs of strings are/are not anagrams)

Reversed Pairs in Array



Question 81 If an element at the left side is greater than another element at the right side, they form a *reversed pair* in an array. How do you get a count of reversed pairs?

For example, there are five reversed pairs in the array $\{7, 5, 6, 4\}$, which are $(7, 5)$, $(7, 6)$, $(7, 4)$, $(5, 4)$, and $(6, 4)$.

The brute-force solution is to find the number of reversed pairs while scanning the array. A scanned number is compared with all numbers behind it. If the number behind is less than the currently visited number, a reversed pair is found. Since it compares a number with $O(n)$ numbers in an array with size n , the time complexity is $O(n^2)$. Let's try to improve the performance.

Since it costs too much time to compare a number with all numbers on its right side, it may improve efficiency if every two adjacent numbers are compared. Let's take the array $\{7, 5, 6, 4\}$ as an example, as shown in [Figure 7-2](#).

The new solution splits the whole array into two sub-arrays of size 2 ([Figure 7-2\(a\)](#)) and continues to split the sub-arrays till the size of sub-arrays is 1 ([Figure 7-2\(b\)](#)). Then it merges the adjacent sub-arrays and gets number of reversed pairs in them. The number in the sub-array $\{5\}$ is less than the number in the sub-array $\{7\}$, so these two numbers compose a reversed pair. Similarly, numbers in the sub-arrays $\{4\}$ and $\{6\}$ also compose a reversed pair. After the number of reversed pairs is found between every two adjacent sub-arrays with size 1, they are merged to form a set of sub-arrays with size 2 ([Figure 7-2\(c\)](#)). Sub-arrays are sorted when they are merged in order to avoid counting the reversed pairs again inside the merged arrays.

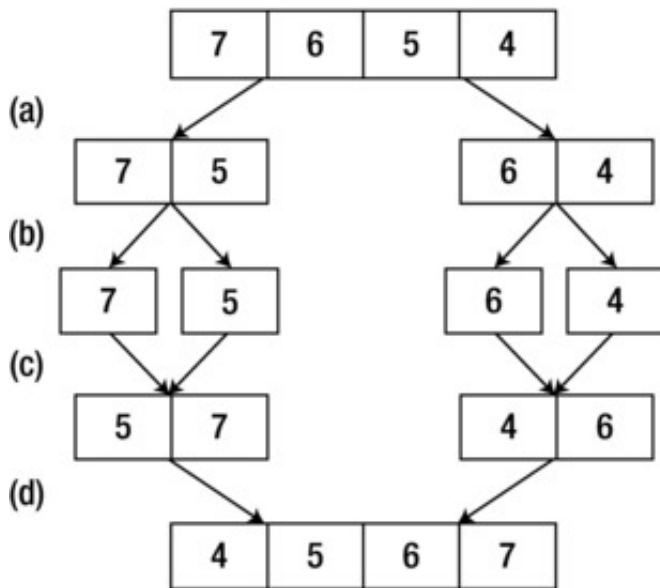


Figure 7-2. The process to get the number of reversed pairs in the array {7, 5, 6, 4}. (a) Split the array with size 4 into two arrays with size 2. (b) Split each array with size 2 into two arrays with size 1. (c) Merge every two adjacent arrays with size 1, sort them, and get the number of reversed pairs. (d) Merge arrays with size 2, merge them, and get the number of reversed pairs.

Let's continue to count the number of reversed pairs between sub-arrays with size 2. Figure 7-3 illustrates the detailed process of Figure 7-2(d).

Two pointers (P_1 and P_2) are initialized to the end of two sub-arrays. If the number referenced by P_1 is greater than the number referenced by P_2 , there are some reversed pairs in such cases. As shown in Figure 7-3(a) and Figure 7-3(c), the number of reversed pairs is the same as the number of remaining numbers in the second sub-array. There are no reversed pairs if the number referenced by P_1 is less than or equal to the number referenced by P_2 (Figure 7-3(b)). Another pointer P_3 is initialized to the end of the merged array. The greater number of the two referenced by P_1 and P_2 is copied to the location referenced by P_3 in order to keep numbers in the merged array sorted. It moves these three pointers backward and continues to compare, count, and copy until one sub-array is empty. Then the remaining numbers in the other sub-array are copied to the merged array.

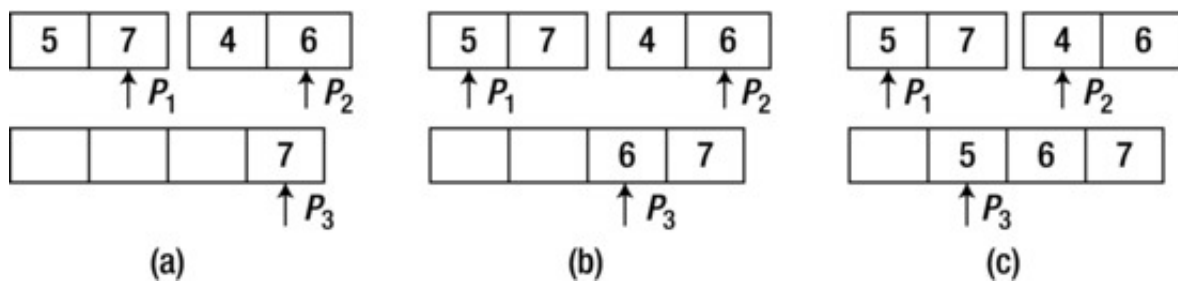


Figure 7-3. Merge sub-arrays in Figure 7-2(d). The last step, to copy the last remaining number 4 in the second sub-array, is omitted. (a) There are two reversed pairs because the number pointed to by P_1 is greater than the number pointed to by P_2 , and P_2 points to the second number and there are two numbers in the second sorted sub-array less than the number pointed to by P_1 . Copy the number pointed to by P_1 to the merged array and move P_1 and P_3 backward. (b) There are no reversed pairs because the number pointed to by P_1 is less than the number pointed to by P_2 . Copy the number pointed to by P_2 to the merged array and move P_2 and P_3 backward. (c) There is a reversed pair because the number pointed to by P_1 is greater than the number pointed to by P_2 , and P_2 points to the first number in the second sub-array. Copy the number pointed to by P_1 to the merged array and move P_1 and P_3 backward.

The process to count reversed pairs may be summarized as follows: It recursively splits an array into two sub-arrays. It counts reversed pairs inside a sub-array and then counts reversed pairs between two adjacent sub-arrays while merging them. Therefore, the solution can be implemented based on the merge sort algorithm, as shown in Listing 7-18.

Listing 7-18. Java Code to Count Reversed Pairs

```
int countReversedPairs(int[] numbers) {
    int[] buffer = new int[numbers.length];
    return countReversedPairs(numbers, buffer, 0, numbers.length - 1);
}

int countReversedPairs(int[] numbers, int[] buffer, int start, int end) {
    if(start >= end)
        return 0;

    int middle = start + (end - start) / 2;

    int left = countReversedPairs(numbers, buffer, start, middle);
    int right = countReversedPairs(numbers, buffer, middle + 1, end);
    int between = merge(numbers, buffer, start, middle, end);

    return left + right + between;
}
```



```

}

int merge(int[] numbers, int[] buffer, int start, int middle, int end) {

    int i = middle; // the end of the first sub-array

    int j = end;    // the end of the second sub-array

    int k = end;    // the end of the merged array

    int count = 0;

    while(i >= start && j >= middle + 1) {

        if(numbers[i] > numbers[j]) {

            buffer[k--] = numbers[i--];

            count += (j - middle);

        }

        else {

            buffer[k--] = numbers[j--];

        }

    }

    while(i >= start) {

        buffer[k--] = numbers[i--];

    }

    while(j >= middle + 1) {

        buffer[k--] = numbers[j--];

    }

    // copy elements from buffer[] to numbers[]

    for(i = start; i <= end; ++i) {

        numbers[i] = buffer[i];

    }

    return count;

}

```

As we know, the time complexity for the merge sort algorithm is $O(n \log n)$, so it is better than the brute-force solution that costs $O(n^2)$ time. The second solution allocates more memory with a buffer size n , so there is a trade-off between time and space efficiency.


Source Code:

081_ReversePairs.java

Test Cases:

- Functional Test Cases (Input a unsorted array; input an increasingly/decreasingly sorted array; some numbers in the array are duplicated)
- Boundary Test Cases (Input an array with one or two numbers)

First Intersection Node in Two Lists

 **Question 82** Please find the first common node of two single-linked lists if they intersect with each other.

When asked this question in interviews, many candidates' intuition is the brute-force solution. It scans all nodes on one list. When a node is visited, it scans the other list to check whether the other list contains the node. The first node contained in both lists is the first common node. If the length of one list is m and the other is n , this solution costs $O(mn)$ time to find the first common node on the two lists. The brute-force solution is not the best one. Let's explore alternatives.

The structure of two lists with common nodes looks like a rotated 'Y'. As shown in [Figure 7-4](#), all nodes following the first common node in two lists are identical. There are no branches after the first common node in two lists because each node has only one link to the next node in a singly linked list.

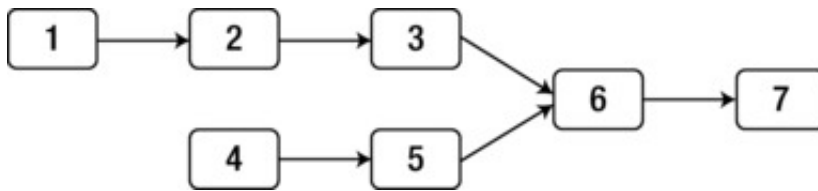


Figure 7-4. Two lists intersecting at node 6

If two lists have common nodes, the common nodes appear in their tails. Therefore, we can compare nodes beginning from the tails of the two lists. If two nodes on two lists are identical, we continue to compare preceding nodes backward. There are only links to next nodes, and the tail node is the last one to be visited in a singly linked list. The last node is the first one to be compared. Does it sound like “Last In, First Out”? Let’s have a try with stacks.

Nodes in two lists are pushed into two stacks when they are traversed. When all nodes are visited, the tail nodes are on the top of these two stacks. It continues to pop two nodes on the top of stacks and compares them until they are different. The last two identical nodes are the first common nodes in two singly linked lists.

The solution above utilizes two stacks. If the lengths of two lists are m and n , both time complexity and space complexity are $O(m+n)$. Compared with the brute-force solution, it improves time efficiency with more consumption of space.

Two stacks are utilized because we would like to reach the tail nodes at the same time. The time to reach tail nodes on the two lists with different length is different when two lists are traversed from the head nodes.

There is a better solution to the problem caused by length difference, with two traversals. It traverses two lists and gets their lengths, as well as the length difference. During the second traversal, it advances on the longer list for d steps if the length difference is d , and then traverses the two lists together. The first identical nodes on lists are the first common nodes.

Let’s take the lists in Figure 7-4 as an example. The new solution first finds that the lengths of the two lists are 5 and 4, and the longer list has one more node than the shorter one. It moves on the longer list for one step to reach the node 2, and then traverses the two lists until it arrives at the first common node, node 6.

Similar to the previous solution, this solution also costs $O(m+n)$ time. However, it does not need auxiliary stacks.

It is time to write code after the solution is accepted by interviewers, as shown in Listing 7-19.

Listing 7-19. C++ Code to Get the First Common Node in Two Lists

```

ListNode* FindFirstCommonNode(ListNode *pHead1, ListNode *pHead2) {

    // Get length of two lists

    unsigned int nLength1 = GetListLength(pHead1);

    unsigned int nLength2 = GetListLength(pHead2);

    int nLengthDif = nLength1 - nLength2;

    ListNode* pListHeadLong = pHead1;

    ListNode* pListHeadShort = pHead2;

    if(nLength2 > nLength1) {

        pListHeadLong = pHead2;

        pListHeadShort = pHead1;

        nLengthDif = nLength2 - nLength1;

    }

    // Move d steps on the longer list if the length difference is d

    for(int i = 0; i < nLengthDif; ++i)

        pListHeadLong = pListHeadLong->m_pNext;

    // Traverse two lists

    while((pListHeadLong != NULL) &&

        (pListHeadShort != NULL) &&

        (pListHeadLong != pListHeadShort)) {

        pListHeadLong = pListHeadLong->m_pNext;

        pListHeadShort = pListHeadShort->m_pNext;

    }

    ListNode* pFirstCommonNode = pListHeadLong;

    return pFirstCommonNode;

}

unsigned int GetListLength(ListNode* pHead) {

    unsigned int nLength = 0;

    ListNode* pNode = pHead;

    while(pNode != NULL) {

```

```
    ++ nLength;

    pNode = pNode->m_pNext;

}

return nLength;

}
```

Source Code:

082_FirstCommonNodesInLists.cpp

Test Cases:

- Functional Test Cases (Two lists have intersection nodes, the first of which is a head/tail node or inside a list; two lists do not have intersection nodes)
- Robustness Test Cases (The pointers to head nodes of one or two lists are NULL)