Time complexity analysis

31 January 2025 09:58

```
#include <stdio.h>
int main() {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        printf("%d\n", i);
    }
    for (int j = 0; j < n; j++) {
        printf("%d\n", j);
    }
    return 0;
}
```

```
#include <stdio.h>
int main()
 {
    int n;
    scanf("%d", &n);
    int i = 1;
    while (i < n)
 {
        printf("%d\n", i);
        i *= 2;
    }
    return 0;
}
```

```
#include <stdio.h>
int main() {
    int n;
```

```c
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < 100; k++) {
                printf("%d %d %d\n", i, j, k);
            }
        }
    }
    return 0;
}
```

```c
#include <stdio.h>
int main() {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        for (int j = 1; j < n; j *= 2) {
            printf("%d %d\n", i, j);
        }
    }
    return 0;
}
```

```c
#include <stdio.h>
int main() {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 1; k < n; k *= 2) {
```

```c
                printf("%d %d %d\n", i, j, k);
            }
        }
    }
    return 0;
}
```

```c
#include <stdio.h>

void process(int start, int end) {
    if (start >= end) return;
    int mid = (start + end) / 2;
    printf("%d %d\n", start, end);
    process(start, mid);
    process(mid + 1, end);
}

int main() {
    int n;
    scanf("%d", &n);
    process(0, n);
    return 0;
}
```

Space complexity analysis

31 January 2025 15:08

```c
#include <stdio.h>

int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}

int main() {
    int n = 6;
    printf("Fibonacci of %d: %d\n", n, fibonacci(n));

    return 0;
}
```

O(n), due to the recursion call stack. The depth of recursion can go up to n.

```c
#include <stdio.h>

int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

int main() {
    int n = 5;
    printf("Factorial of %d: %d\n", n, factorial(n));

    return 0;
}
```

O(1), since only a few variables (result, i) are used and no extra space is allocated.

```c
#include <stdio.h>

void reverseArray(int arr[], int n) {
    for (int i = 0; i < n / 2; i++) {
        int temp = arr[i];
        arr[i] = arr[n - i - 1];
        arr[n - i - 1] = temp;
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = 5;

    reverseArray(arr, n);
```

```c
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

O(1), since no additional memory is used other than the input array and a few variables.

```c
#include <stdio.h>

void copyarray(int arr[], int n) {
    int newArr[n];
    for (int i = 0; i < n; i++) {
        newArr[i] = arr[i];
    }

}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = 5;

    copyarray(arr, n);

    return 0;
}
```

O(n), since a new array newArr of size n is created

```c
#include <stdio.h>

void mergeArrays(int arr1[], int n1, int arr2[], int n2) {
    int mergedArr[n1 + n2];
    int i = 0, j = 0, k = 0;

    while (i < n1 && j < n2) {
        if (arr1[i] < arr2[j]) {
            mergedArr[k++] = arr1[i++];
        } else {
            mergedArr[k++] = arr2[j++];
        }
    }

    while (i < n1) {
        mergedArr[k++] = arr1[i++];
    }

    while (j < n2) {
        mergedArr[k++] = arr2[j++];
    }
}

int main() {
    int arr1[] = {1, 3, 5};
    int arr2[] = {2, 4, 6};
    int n1 = 3, n2 = 3;
```

```
    mergeArrays(arr1, n1, arr2, n2);

    return 0;
}
```

The size of mergedArr is O(n1 + n2).

```
#include <stdio.h>

void countFrequency(int arr[], int n) {
    int freq[100] = {0};
    for (int i = 0; i < n; i++) {
        freq[arr[i]]++;
    }


    for (int i = 0; i < 100; i++) {
        if (freq[i] > 0) {
            printf("%d occurs %d times\n", i, freq[i]);
        }
    }
}

int main() {
    int arr[] = {1, 2, 2, 3, 3, 3, 4, 4, 4, 4};
    int n = 10;

    countFrequency(arr, n);

    return 0;
}
```

Since the freq[100] array is fixed in size and does not depend on n, it is considered O(1) space. The dominant factor is the input array arr[], making the overall space complexity O(n) in general.

In space complexity: O(n1 + n2) can be simplified to O(n) if both n1 and n2 are on the same order of magnitude.

However, you should consider the specific case:

- If n1 and n2 are the sizes of two different input parameters and are of the same order of magnitude, then the space complexity can be simplified to O(n).

- If n1 and n2 are drastically different (e.g., n1 = 1000 and n2 = 10^6), it might be more appropriate to express the space complexity as O(n1 + n2) explicitly to highlight the contributions from both.

Stable and Inplace

31 January 2025 15:02

- Definition: A sorting algorithm is considered stable if it preserves the relative order of elements with equal keys (i.e., elements that compare equal are kept in their original order in the input).
- Key Point: If two elements A and B have the same value and A appears before B in the input, they will remain in the same order in the output.
- Example: If sorting a list of people by their names, a stable sort will ensure that people with the same name maintain the original order they appeared in the list.
- Example Algorithms:
  - Merge Sort (Stable)
  - Bubble Sort (Stable)
  - Insertion Sort (Stable)
  - Radix Sort (Stable, because it sorts based on individual digits, which preserves relative order)

In-place Sorting

- Definition: A sorting algorithm is in-place if it sorts the list without requiring any extra space (beyond a constant amount). Essentially, it reuses the original input array to store the sorted data.
- Key Point: The sorting is done by modifying the elements of the array, with no significant additional memory overhead (ignoring the memory used for variables).
- Example Algorithms:
  - Quick Sort (In-place)
  - Heap Sort (In-place)
  - Bubble Sort (In-place)
  - Selection Sort (In-place)
  - Insertion Sort (In-place

Can an Algorithm be Both Stable and In-place?
an algorithm can be both stable and in-place, but not all stable algorithms are in-place and vice versa. For example:

- Bubble Sort is both stable and in-place.
- Quick Sort is in-place but not stable.

Radix sort (stable)

31 January 2025 14:55

```
Step 1: Find the Maximum Number
We first find the maximum number to determine the number of digits.
Max = 802 (3 digits, so we perform 3 passes).
Step 2: Sorting by Least Significant Digit (1s place)
We use Counting Sort to sort based on the 1s place.


170 → 0
45  → 5
75  → 5
90  → 0
802 → 2
24  → 4
2   → 2
66  → 6

Counting the occurrences of digits (0-9)
0 → 2
2 → 2
4 → 1
5 → 2
6 → 1

Placing numbers in sorted order by 1s place

[170, 90, 802, 2, 24, 45, 75, 66]

Sorting by 10th place
--------------------------
170 → 7
90  → 9
802 → 0
2   → 0
24  → 2
45  → 4
75  → 7
66  → 6

Counting the occurrences of digits (0-9)
0 → 2
2 → 1
```

```
4 → 1
6 → 1
7 → 2
9 → 1

Placing numbers in sorted order by 10s place:
[802, 2, 24, 45, 66, 170, 75, 90]



802 → 8
2   → 0
24  → 0
45  → 0
66  → 0
170 → 1
75  → 0
90  → 0



0 → 6
1 → 1
8 → 1

Placing numbers in sorted order by 10s place:
[2, 24, 45, 66, 75, 90, 170, 802]
```