FDS Project 1(Hard): Performance Measurement (A+B)

Anonymous Author

Date: 2026-03-12

> Given S as a collection of N positive integers that are no more than V. For any given number c, you are supposed to find two integers a and b from S, so that $a + b = c$.
>
> Your tasks are:
>
> (1) Implement at least two different algorithms to solve this problem;
> (2) Analyze the complexities of the two algorithms;
> (3) Measure and compare the performances of the two algorithms for V and N = 1000, 5000, 10000, 20000, 40000, 60000, 80000, 100000.

1. Introduction

I wrote the program containing two algorithms, but found that the first algorithm with a time complexity of $O(n^2)$ actually runs with a **linearly increasing** runtime as N grows, rather than quadratically.

I realized that, take the number 100 as an example—it can be written as 50+50, 49+51, and so on down to 1+99, giving 50 combinations. As the number N increases, the number of combinations also grows.

Therefore, Algorithm 1 can easily find the answer within its **first few iterations**, thus failing to exhibit the $O(n^2)$ complexity. As shown in the picture below:

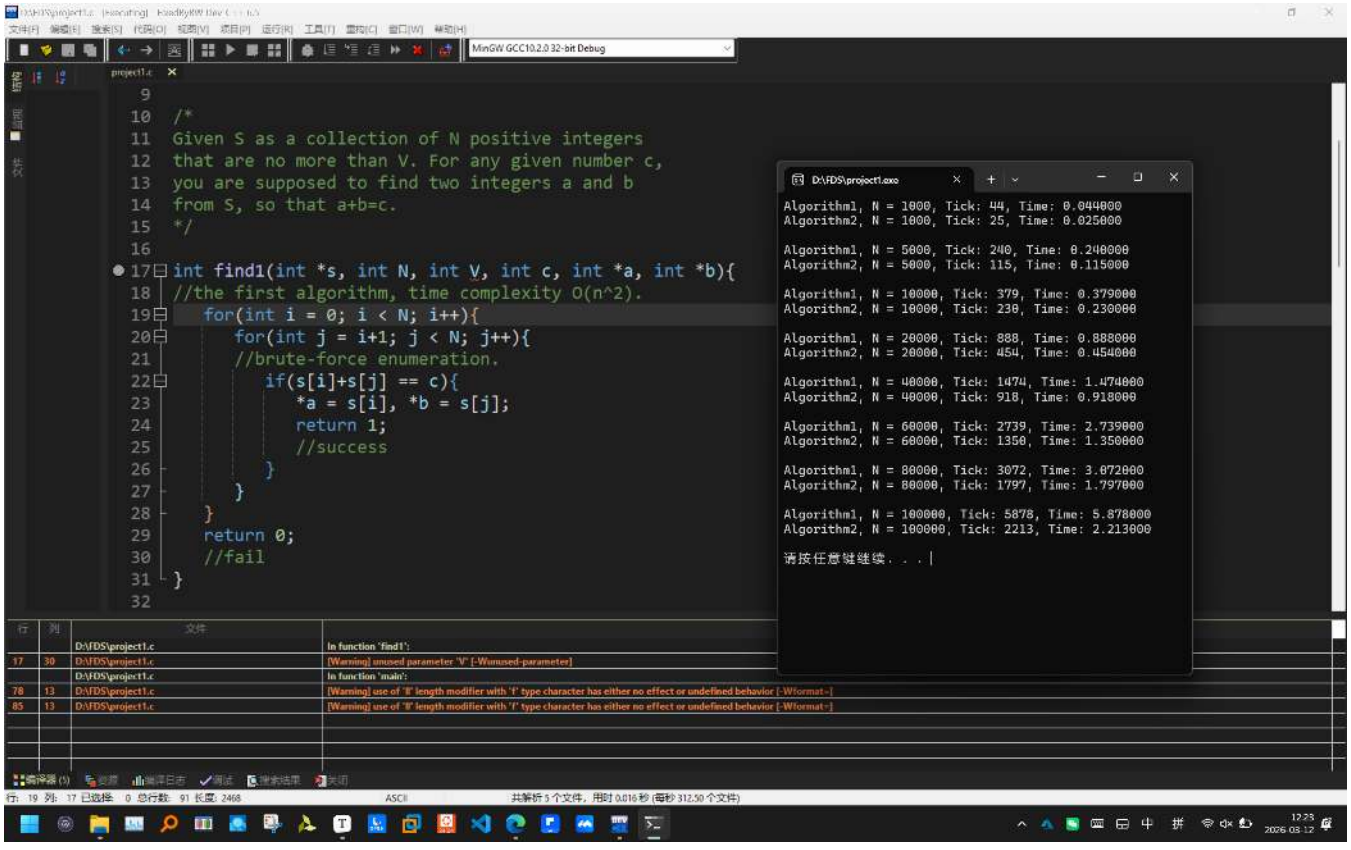

Image 1

So that we should use **boundary cases** to demonstrate the time complexity!

Just make $c = 1$ or 2 in Algorithm 1:

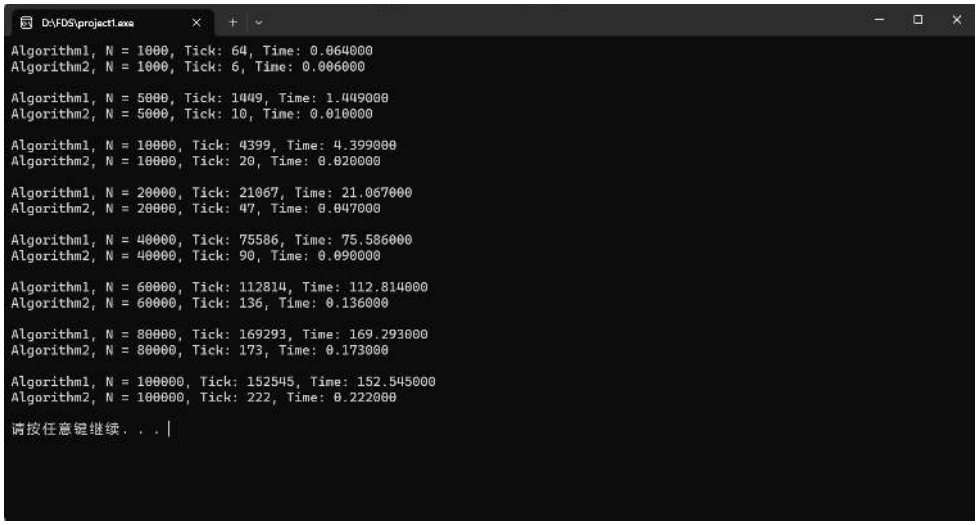

Image 2

In the picture **above**, we can see that the program is able to **reflect the quadratic and linear complexity** of the two algorithms respectively. However, the tick values are **too small (6)** and **too large (169293)**.

This leads to large errors (> 10%) when the tick is too small, while a very large tick wastes a lot of time.

Therefore, we need to dynamically adjust the value of K to keep the tick of each test case within a reasonable range:

```
/*Algorithm 1*/
K = 120000/N+1;   //setting a proper K to keep total time in a reasonable range
                  //"+1" is to prevent K=0, which would result in NaN!
c = 1;            //providing boundary cases to demonstrate the time complexity!
start = clock();  //repeating the algorithm for K times

/*Algorithm 2*/
K = 10000000/N+1; //this one is faster, so repeating more times to improve accuracy
```

As shown in the picture below:

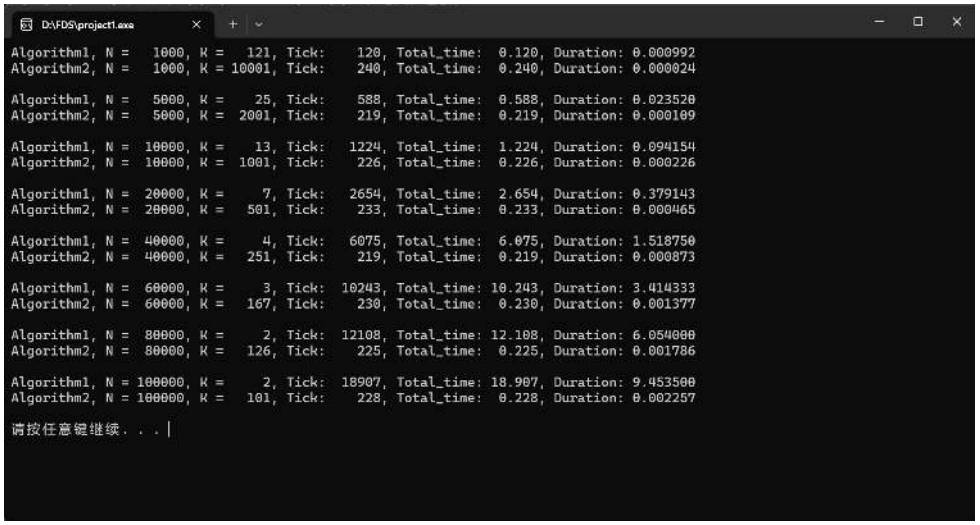

Image 3

2. Algorithm Specification

2.1 Implement the two functions (30 pts.)

```
int find1(int *s, int N, int V, int c, int *a, int *b){//the first algorithm, time complexity O(n^2).
    for(int i = 0; i < N; i++){
        for(int j = i+1; j < N; j++){//brute-force enumeration.
            if(s[i]+s[j] == c){
                *a = s[i], *b = s[j];
                return 1;//success
            }
        }
    }
    return 0;//fail
}
```

```
int find2(int *s, int N, int V, int c, int *a, int *b){//the second algorithm, time complexity O(n).
    int *flag = (int *)calloc((V+5), sizeof(int));
    //using a flag array to mark number occurrences
    //using calloc so that all elements are initialized to 0
    for(int i = 0; i < N; i++) flag[s[i]]++;
    for(int i = c/2; i>=0&&c-i<=V; i--){
    //start from the middle part of the flag array,
    //making it easier to avoid out-of-bounds.
        if(flag[i]&&flag[c-i]){
            if(i == c-i&&flag[i]<2) continue;
            *a = i, *b = c-i;
            free(flag);
            return 1;
        }
    }
    free(flag);
    return 0;
}
```

2.2 testing program (20 pts.) with sufficient comments.

- For the detailed content of the program, see the **Appendix** section.[^1]

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
...
```

3. Testing Results

3.1 Decide the iteration number K for each test case and fill in the table of results (8 pts.)

- For the content related to the generation of K, see the **Introduction** section.
- We only list the **result table** here.

V = 100000	N	1000	5000	10000	20000	40000	60000	80000	100000
Algorithm1	iterations(K)	121	25	13	7	4	3	2	2
	Ticks	120	588	1224	2654	6075	10243	12108	18907
	Total Time(sec)	0.120	0.588	1.224	2.654	6.075	10.243	12.108	18.907
	Duration(sec)	0.000992	0.023520	0.094154	0.379143	1.518750	3.414333	6.054000	9.453500
Algorithm2	iterations(K)	10001	2001	1001	501	251	167	126	101
	Ticks	240	219	226	233	219	230	225	228
	Total Time(sec)	0.240	0.219	0.226	0.233	0.219	0.230	0.225	0.228
	Duration(sec)	0.000024	0.000109	0.000226	0.000465	0.000873	0.001377	0.001786	0.002257

3.2 Plot the run times of the functions (12 pts.)

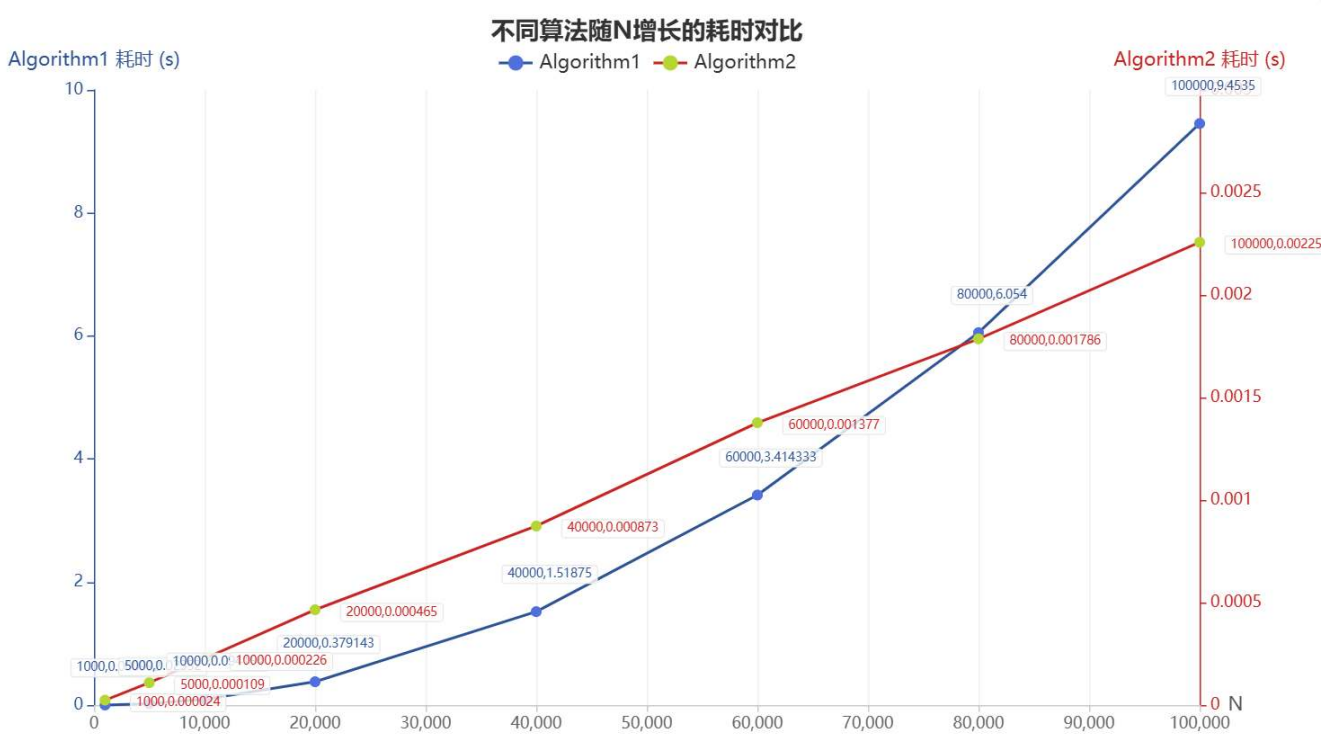

Image 4

- Based on the line graph of run times, Algorithm 1 grows quadratically overall, and Algorithm 2 grows linearly.

4. Analysis and Comment(10 pts.)

4.1 Analysis of the time and space complexities of the algorithms

4.1.1 Algorithm 1 (Brute-force Enumeration)

- The algorithm uses nested loops to traverse all pairs of elements in the array: the outer loop runs N times, and the inner loop runs $N-1, N-2, \ldots, 1$ times for each outer loop iteration. In the worst case (no valid $a+b=c$ pair), the total number of iterations is $\frac{N(N-1)}{2}$. Time complexity: $O(N^2)$.
- The space complexity is $O(1)$ as it only uses a fixed number of temporary variables (no additional dynamic memory).

4.1.2 Algorithm 2 (Hash-based Marking)

> - The algorithm consists of two linear passes:
>   - (1) a single loop to mark the occurrence of each element in the array ($O(N)$)
>   - (2) a single loop to search for valid pairs in the flag array ($O(V)$).

- Overall time complexity: $O(N)$.

> **Important**
>
> Overall time complexity will be $O(N+V)$ if V is variable.[1]

- The space complexity is $O(V)$ due to the dynamically allocated flag array of size $V+5$.

4.2 Comments and further possible improvements

4.2.1 Test Design Rationality

4.2.1.1 Boundary Case Selection

Using $c=1$ (unsolvable) can reveal the true time complexity of both algorithms.

4.2.1.2 Dynamic Iteration Count (K)

Adjusting K as $K=C/N+1$ solves the precision issue of **duration** (which has limited resolution): repeating fast algorithms (Algorithm 2) more times amplifies the runtime, reducing relative error.

4.2.1.3 Large V (100000)

A small V would limit the range of integers in S, leading to frequent early termination of Algorithm 1;

using $V=100000$ ensures the array S has sufficient diversity.

4.2.2 Limitations and Improvements

4.2.2.1 Number generation function `Random()`

The process of generating random numbers itself occupies a period of time, which slightly affects the result accuracy.

4.2.2.2 Varying V

The experiment fixed $V = 100000$.

Testing different V values would reveal how Algorithm 2's space complexity $(O(V))$ and time complexity $O(N + V)$ impacts performance.

5. Appendix: Source Code (in C)

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

clock_t start, stop;
int S[100005];
int N, V, c;

/*
Given S as a collection of N positive integers
that are no more than V. For any given number c,
you are supposed to find two integers a and b
from S, so that a+b=c.
*/

int find1(int *s, int N, int V, int c, int *a, int *b){
    //the first algorithm, time complexity O(n^2).
    for(int i = 0; i < N; i++){
        for(int j = i+1; j < N; j++){
            //brute-force enumeration.
            if(s[i]+s[j] == c){
                *a = s[i], *b = s[j];
                //printf("%d + %d = %d\n",s[i], s[j], c);
                return 1;
                //success
            }
        }
    }
    //printf("F\n");
    return 0;
    //fail
}

int find2(int *s, int N, int V, int c, int *a, int *b){
    //the second algorithm, time complexity O(n).
```

```c
    int *flag = (int *)calloc((V+5), sizeof(int));
    //using a flag array to mark number occurrences
    //using calloc so that all elements are initialized to 0
    for(int i = 0; i < N; i++){
        flag[s[i]]++;
    }
    for(int i = c/2; i>=0&&c-i<=V; i--){
        //start from the middle part of the flag array,
        //making it easier to avoid out-of-bounds.
        if(flag[i]&&flag[c-i]){
            //within bound
            if(i == c-i&&flag[i]<2) continue;
            //the same number
            *a = i, *b = c-i;
            free(flag);
            //printf("%d + %d = %d\n",i, c-i, c);
            return 1;
        }
    }
    free(flag);
    //printf("F\n");
    return 0;
}

static int seed = 0;

void random(int *s, int N, int V){
    if(!seed){
        srand((unsigned)time(NULL));
        //use the current time as the random seed
        seed = 1;
        //avoid using the same seed and the same array!
    }
    for(int i = 0; i < N; i++) s[i] = rand()%V + 1;
    //[1,V]
}

int main(){
    int a, b, tick, K, V = 100000;
    //fixed V in this code
    double duration, total;
    int test[]={1000, 5000, 10000, 20000, 40000, 60000, 80000, 100000};
    //giving an array of N
    for(int i = 0; i < 8; i++){
        //testing different Ns
        N = test[i];
        /**********
        Algorithm 1
        **********/
        K = 120000/N+1;
        //setting a proper K to keep total time in a reasonable range
        c = 1;
        //providing boundary cases to demonstrate the time complexity!
```

```
89          start = clock();
90          //repeating the algorithm for K times
91          for(int j = 0; j < K; j++) {random(S, N, V); find1(S, N, V, c, &a, &b);}
92          stop = clock();
93          total = (double)(stop - start) / CLK_TCK, tick =  stop - start, duration =
    (double)total/K;
94          //output the time and N and K
95          printf("Algorithm1, N = %6d, K = %5d, Tick: %6d, Total_time: %6.3lf,
    Duration: %.6lf\n", N, K, tick, total, duration);
96
97          /**********
98          Algorithm 2
99          **********/
100         K = 10000000/N+1;
101         //this one is faster, so repeating more times to improve accuracy
102         c = N/2;
103         //also providing boundary cases to demonstrate the time complexity
104         start = clock();
105         for(int j = 0; j < K; j++) {random(S, N, V); find2(S, N, V, c, &a, &b);}
106         stop = clock();
107         total = (double)(stop - start) / CLK_TCK, tick =  stop - start, duration =
    (double)total/K;
108         printf("Algorithm2, N = %6d, K = %5d, Tick: %6d, Total_time: %6.3lf,
    Duration: %.6lf\n\n", N, K, tick, total, duration);
109     }
110     system("pause");
111     return 0;
112 }
```

Declaration:

> I hereby declare that all the work done in this project titled ***Performance Measurement*** $(A + B)$ is of my independent effort.

1. The experiment fixed $V = 100000$. Testing different V values would reveal how Algorithm 2's space complexity $(O(V))$ and time complexity $O(N + V)$ impacts performance. ↩