Performance Measurement (A+B)

Lu Jialiang

Article submitted to

Zhejiang University

as part of

Fundamental Data Structure

Computer Science

College of Imformation Science & Electronic Engineering
3230104505@zju.edu.cn

March 6, 2025

Contents

Chapter 1 Introduction

The problem is: given S as a collection of N positive integers that are no more than V. For any given number c, find two integers a and b from S, so that $a + b = c$.

Chapter 2 Algorithm Specification

2.1 Algorithm 1

Main Data Structure: Array
Store the set S in a one-dimensional array. Firstly, use the quick sort to sort the set in non-descending order, and then set two pointers i and j, which are located at the beginning and end of the array. Constantly compare the sum of the numbers pointed to by the two pointers with the target sum. When it is smaller than the target sum, shift i to the right.In the opposite situation, shift j to the left. Repeat the operations until we find that the sum of the two numbers is the target sum or the two pointers meet.
Here it is necessary to explain some of the qsort() function. qsort() function is a built-in sorting function declared in <stdlib.h>, and the algorithm it uses is quick sort. I did not write a sorting algorithm, because later we will explain the complexity of Algorithm 1 depends on the complexity of the sorting algorithm, and use qsort() can give the best performance. We will only explain the way this algorithm works and show its pseudo-code.
What qsort() does is something like this: it picks one number in the array as the pivot, which divides the elements of the array into a larger part and a smaller part, and then recursively performs a quick sort on these two parts. In the deviding process, the specific operation is: We choose the last number as the pivot each time. Pointer i is located at the second end of the array, while j is located at the head of the array. At this, exchange the the number pointed by i and j, and shift i to the right.Finally,when j moves to the head, exchange the number pointed by i+1 and pivot.Now we get a new array. The numbers to the left of the pivot element of this array are all smaller than the pivot element, and the numbers to the right are all larger than the pivot element,which enables us to uses qsort on those two sides.

Pseudo-Code(main):

```
Function solve_1(collection: array, N: length, V: upper bound, target: target sum):
    Initialize i = 0,j = N - 1          // Left pointer and Right pointer
    Declare temp: long integer         // Temporary sum storage
    Sort collection in non-descending order // Using quicksort
    while i <= j and (collection[i] + collection[j]) != target:
        temp = collection[i] + collection[j]
        if temp < target:
            i += 1              // Sum too small, move i rightward
        else:
            j -= 1              // Sum too large, move j leftward
    if collection[i] + collection[j] == target:
        print "collection[i]} + collection[j] = target"
    else:
        print "None"
```

Pseudo-Code(qsort):

```
Function QuickSort(A, low, high):
    if low < high:
        pivot_index = Partition(A, low, high)
        QuickSort(A, low, pivot_index - 1)
        QuickSort(A, pivot_index + 1, high)

Function Partition(A, low, high):
    pivot = A[high]
    i = low - 1
    for j = low to high - 1:
        if A[j]  pivot :
            i = i + 1
            swap A[i] and A[j]
    swap A[i + 1] and A[high]
    return i + 1
```

2.2 Algorithm 2

Main Data Structure: Array/Hash Table
Create a one-dimensional array of size V and type char and initialise it to 0(i.e. each value is '\0'). Each position in the array records the occurrence or non-occurrence of the number equals to its position in the array plus 1. Read in the numbers in S sequentially, setting the value of the corresponding position in the table to '1' for each number read in, and checking whether the number whose sum is the target is already present by checking its corresponding position. If it has already appeared, the two numbers whose sum is the target sum have been found, and if it has not appeared, the reading continues until the entire S has been read.

Pseudo-Code:

```
Function solve_2(collection: array, N: length, V: upper bound, target: target sum):
Initialize hash_table: array[V] of characters = {0}
// Boolean-like hash table,actually uses char
Initialize found_flag = false
for each element in collection:
current_num = element
hash_index = current_num - 1 // As index starts from 0 while numbers are positive
if hash_table[hash_index] == '0':
mark hash_table[hash_index] = '1' // Mark as visited
complement = target - current_num
if complement > 0 and complement <= V:
    if hash_table[complement - 1] == 1: // Check complement existence
        print "current_num + complement = target"
        set found_flag = true
        exit loop
if not found_flag:
print "None"
```

Chapter 3 Testing Results

P.S.: Obviously, there is no need to set V in Cases 1 to 4, and Case 5 serves as a test, where V is taken to be 100 times N casually.

3.1 Test Cases for Algorithm 1

Case	Size N	Integers in S	Target	Purpose	Result
1	10	1,2,...,10	15	Integers are in order	Pass
2	10	1,9,7,4,5,10,3,2,8,6	15	Intergers are not in order	Pass
3	10	1,9,7,4,5,10,3,2,8,6	24	Solves can't be found	Pass
4	10	1,2,2,2,2,3,3,3,4,5	15	Repetitive elements exists	Pass
5	10^5	Randomly Generated	1866593	A lot of elements	Pass

3.2 Test Cases for Algorithm 2

Case	Size N	Integers in S	Target	Purpose	Result
1	10	1,2,...,10	15	Integers are in order	Pass
2	10	1,9,7,4,5,10,3,2,8,6	15	Intergers are not in order	Pass
3	10	1,9,7,4,5,10,3,2,8,6	24	Solves can't be found	Pass
4	10	1,2,2,2,2,3,3,3,4,5	15	Repetitive elements exists	Pass
5	10^5	Randomly Generated	1866593	A lot of elements	Pass

Chapter 4 Analysis & Comments

Instructions about the Choice of V: The choice of V actually has some implications: when V is less than N, duplicate data will inevitably be generated and the rate of duplication will be high; when V is too large, the data will tend to be dispersed, implying that the cost of storage is rising and the probibility to find the result decreasing. In order to reflect these two situations, we chose 4 different sets of V and more cases where V is greater than N, as this better reflects the complexity of the algorithm in worst case(which we will elaborate on later).

P.S.: In fact, I have tried to put all the tests into a single program run and store them, and eventually standardise the output. But my computer suffered a mishap in this, and the program is prone to lagging in the middle of the run, which introduces a huge error and is not good for debugging. For this reason, all the data is tested separately, as you can see in the source code.One more thing, the array S and the target are randomly generated each time when testing. You can get this in Appendix A.

4.1 Algorithm 1

4.1.1 Validity

The effectiveness of Algorithm 1 can be verified by simulation. Assuming that the pointers of the target number are m,n. As i, j starts from the two ends of the array and gradually move to the middle, one of them must reach m (or n) first or at the same time. When one arrives first, due to the array is non-descending, it will stop, while the other must keep moving towards him. Ultimately they reach (m,n) and we get the result.

4.1.2 Time Complexity

Algorithm 1 can be split into 2 parts, which are sorting the array and searching for two numbers that satisfy the condition using double pointers. We discuss the complexity of these two algorithms separately.

Quick Sort: First, the running time of the quick sort is equal to the running time of the two recursive calls plus the time used by the split function. For the split function, the variable j only needs to traverse the array once, so the time consumed is linear, so:

$$T(N) = T(i) + T(N - i - 1) + cN \tag{1}$$

i is the number of the elements in one of the splitted part.Since the set of numbers is randomly generated, even if we select the last number in the region as the pivot each time, we can still assume that the pivot is randomly selected each time. So the average of $T(i)$ will be $(1/N)\sum_{j=1}^{N-1} T(j)$, so as $T(N - i - 1)$.Thus, (1) transforms into:

$$T(N) = \frac{2}{N}\left[\sum_{j=1}^{N-1} T(j) + cN\right] \tag{2}$$

Samely:

$$T(N-1) = \frac{2}{N-1}\left[\sum_{j=1}^{N-2} T(j) + c(N-1)\right] \tag{3}$$

If we remove the denominator and give up the constants, (2) − (3) implies:

$$NT(N) = (N+1)T(N-1) + 2cN \tag{4}$$

Divide both sides of the equation by $N(N+1)$:

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2c}{N+1} \tag{5}$$

Now if we change N to $N-1$,$N-2$,...,3,2, then we get:

$$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2c}{N}$$

$$\cdots \tag{6}$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3} \tag{7}$$

Accumulate both sides, we get:

$$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2c \sum_{i=3}^{N+1} \frac{1}{i} \tag{8}$$

Given that:

$$\lim_{n\to\infty} \left[\sum_{i=1}^{n} \frac{1}{i} - \ln n \right] = \gamma$$

where $\gamma \approx 0.577$ is called **Euler's constant**Thus we get:

$$\frac{T(N)}{N+1} = O(\log N) \tag{9}$$

which gives us the result:

$$T(N) = O(N \log N) \tag{10}$$

Double-pointer searching: In the worst case, the program runs at most until the two pointers meet, i.e. N times. So $T(N) = O(N)$. In the best case, the target number appears at both ends and only one operation needs to be performed, i.e. $T(N) = \Omega(1)$.
Considering both cases together, the upper bound of the whole algorithm is determined by the sorting algorithm, i.e. $T(N) = O(N \log N)$, this is why we use a more complex algorithm to reduce the time complexity of the sorting algorithm.

4.1.3 The Effects of V:

In Algorithm 1, consider first the case where V is smaller than N. In the fast sorting process described above, it is clear that a higher repetition rate does not lead to a reduction in the number of comparisons or exchangings. The effect of V on sorting is relatively small. However, it's the two-pointer lookup phase that really makes a big difference.
If we output each result, we'll find that in the vast majority of cases one of the values is 1. This is due to the fact that the data density is so high when V is much smaller than N that it is very difficult to have a situation where one of the numbers in $[1, V]$ is not generated. Once all the numbers have been generated, there is a 50/50 chance that the target number is generated in the range $[2, V + 1]$, and then it is always possible to find a 1 and another number such that their sum is the target, which eliminates the need for any movement of pointer i. Even if the generated number is not in $[2, V + 1]$. For the same reason, the expectation of the lookup time has still decreased. Although a given pointer may actually point to the same number in several moves due to the presence of duplicates, the slowdown in efficiency is much less than the increase in data density would bring.
When V is larger than N, it in fact plays a reverse role, making the data more dispersed and the possibility of finding the target number decreasing. However, as V increases extremely, the impact of increasing V becomes smaller and smaller because the data is already sufficiently dispersed, and what is reflected at this point is the time complexity of the algorithm without considering the data distribution, i.e.$O(N \log N)$ as discussed above.

4.1.4 Test Results

Case: $V = 0.1 * N$

N	1000	5000	10000	20000	40000	60000	80000	100000
Iterations(K)	10000	10000	1000	1000	500	500	500	500
Ticks	1084	5124	999	2098	2118	3289	4284	5390
Total Times(sec)	1.084	5.124	0.999	2.098	2.118	3.289	4.284	5.390
Durations(sec$\times 10^{-6}$)	108	512	999	2098	4236	6578	8568	10780

Case: $V = 10 * N$

N	1000	5000	10000	20000	40000	60000	80000	100000
Iterations(K)	10000	10000	1000	1000	500	500	500	500
Ticks	1396	6969	1293	2654	2674	4013	5403	6798
Total Times(sec)	1.396	6.769	1.293	2.654	2.674	4.013	5.403	6.798
Durations(sec$\times 10^{-6}$)	140	677	1293	2654	5348	8026	10806	13596

Case: $V = 50 * N$

N	1000	5000	10000	20000	40000	60000	80000	100000
Iterations(K)	10000	10000	1000	1000	500	500	500	500
Ticks	1497	6820	1359	2600	2813	4270	5570	7042
Total Times(sec)	1.497	6.820	1.359	2.600	2.813	4.270	5.570	7.042
Durations(sec$\times 10^{-6}$)	150	682	1359	2660	5626	8540	11140	14084

Case: $V = 100 * N$

N	1000	5000	10000	20000	40000	60000	80000	100000
Iterations(K)	10000	10000	1000	1000	500	500	500	500
Ticks	1522	7007	1341	2805	2854	4223	5624	6964
Total Times(sec)	1.522	7.007	1.341	2.805	2.854	4.223	5.624	6.964
Durations(sec$\times 10^{-6}$)	152	701	1341	2805	5708	8446	11248	13928

4.1.5 Space Complexity

Notice that both the sorting algorithm and the search algorithm use only a constant amount of extra space, other than that the only space consumed is the space used to store the original data, i.e.$S(N) = O(N)$.

4.1.6 Optimization Direction

On the one hand, this algorithm will only return the first result found, but with only minor changes, it can return all results. It is worth noting that if we assume that the random numbers generated are uniformly distributed, then when the sum of the two numbers pointed by i,j is far away from the target, we can imagine that the chance of getting target is still small if i, j continue to move forward at the same pace. For that reason, we can change the pace of i, j. For example, when the sum is too small, the step size of i should be appropriately enlarged so that the expectation of sum is shifted, but the specific amount of shift needs to be derived through the analysis of the probability model.

4.2 Algorithm 2

4.2.1 Validity

The validity of this algorithm is quite obvious. Using the hash table, we smoothly recorded whether each number has appeared or not. As we read in each number, we go through and check if another number corresponding to it has appeared. Since this algorithm actually traverses the original set as well, the existence of the result is definitely discoverable.

4.2.2 Time Complexity

In the worst case, all data will be stored in the hash table, then the number of write operations is linear. Due to the characteristics of the hash table, the operation that to find the number with whom the sum is target is also a constant, and every time we write a number into hash table the operation will be executed once, so the worst-case operation number is also linear. Therefore $T(N) = O(N)$.

4.2.3 The Effects of V:

The effect of V on the space complexity of the algorithm we have already explained, here we focus on the effect on the time complexity. Consider first the case where V is greater than N. In Algorithm 2, this has almost no effect, as you can see from the test data. On the one hand, this is due to the reason we mentioned before: when V is sufficiently large, the data is already spread out enough that increasing V will only bring the complexity curve closer to $O(N)$ that we have analysed. On the other hand, allocating a larger space V takes only a little bit more time, so the time it takes to execute this algorithm at this point is not really significantly related to V.
But when V is less than N, Algorithm 2 reduces the time even a little more significantly than Algorithm 1. If you look closely at the code in Appendix A, you'll see that in this one and only traversal of Algorithm 2, almost all of the operations are performed under the condition that the number has not appeared before, which means that when the number has appeared before we move on to the next number with almost no operations performed. Comnbined with the conditions that the complexity of Algorithm 2 is linear, so the higher the proportion of repeating data, the greater the proportion of operations we can reduce. In the case of the test sample, if the traversal is performed to the end without interruption, the expected percentage of duplicates is nine-tenths, which means that we can expect to omit nine-tenths of the operations.

Of course, the reduction in time is not really that much, partly because, as mentioned before, the increased density of the data due to the concentration of the data makes it difficult to go through the entire traversal. On the other hand, the more different numbers the algorithm reads, the more we can hope to find the other corresponding number, which means that the repeated numbers are ungainly though operation-free, thus presenting the results in the table as a whole.

4.2.4 Test Results

Case: $V = 0.1 * N$

N	1000	5000	10000	20000	40000	60000	80000	100000
Iterations(K)	10000	10000	1000	1000	1000	1000	1000	1000
Ticks	670	1870	332	815	1464	2145	2785	3464
Total Times(sec)	0.671	1.868	0.332	0.815	1.464	2.145	2.785	3.464
Durations(sec$\times 10^{-6}$)	67	187	332	815	1464	2145	2785	3464

Case: $V = 10 * N$

N	1000	5000	10000	20000	40000	60000	80000	100000
Iterations(K)	10000	10000	1000	1000	1000	1000	1000	1000
Ticks	903	3027	619	1241	2265	3306	4277	5382
Total Times(sec)	0.903	3.027	0.619	1.241	2.265	3.306	4.277	5.382
Durations(sec$\times 10^{-6}$)	90	303	619	1241	2265	3306	4277	5382

Case: $V = 50 * N$

N	1000	5000	10000	20000	40000	60000	80000	100000
Iterations(K)	10000	10000	1000	1000	1000	1000	1000	1000
Ticks	1220	4882	833	1463	2941	4290	5688	7028
Total Times(sec)	1.220	4.882	0.833	1.463	2.941	4.290	5.688	7.028
Durations(sec$\times 10^{-6}$)	122	488	833	1463	2941	4290	5688	7028

Case: $V = 100 * N$

N	1000	5000	10000	20000	40000	60000	80000	100000
Iterations(K)	10000	10000	1000	1000	1000	1000	1000	1000
Ticks	1973	9392	2265	3997	4162	6245	8204	10380
Total Times(sec)	1.500	5.702	0.981	1.785	3.328	4.776	6.223	7.806
Durations(sec$\times 10^{-6}$)	150	570	981	1785	3328	4776	6223	7806

4.2.5 Space Complexity

In fact, excluding the constant space that needs to be consumed during the running of the programme, the main space is used by the hash table that stores the data, and the capacity of the hash table varies linearly with the upper bound of the data V, i.e. $S(N, V) = O(V)$.

4.2.6 Optimization Direction

Notice that the space usage is mainly determined by the upper bound of the data, which means that when the data lacks a physical upper bound or the upper bound is large and the data is very scattered, the wasted space of this hash table will be very large. To reduce this effect and figure out how to make it, we consider improving the hash table into a smaller one, where every n numbers are stored as a list by a hash node as a head node. This way there is no waste of space on the one hand, and on the other hand, since n is a constant, the number of operations required for the lookup remains constant, so the time complexity remains unchanged.

Chapter A Source Code (in C)

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

int cmp_int(const void*a,const void*b); //The comparing function provided for
    function "qsort()"

long* initial(long* a,long length,long sup); //To generate the original random
    collection

long random_long(long sup); //This function can produce an integer in [1,V]
    regardless of RAND_MAX

double test(long N,long V,long times,void (*solve)(long*,long,long,long)); //Test
    each solves and return the time costs

void solve_1(long* collection,long N,long V,long target);
void solve_2(long* collection,long N,long V,long target);

int main(){
    srand((unsigned int)time(NULL)); //initialize for function "rand()"
    long N=100000; //sizeof the collection
    long V=100; //upper bound is of the numbers in collection
    long K=1000;   //times that the codes will run for
    double duration; // tp save the time that the project run K times costs
    duration=test(N,V*N,K,solve_2);
    printf("%lf %lf\n",duration,duration/K);
}

int cmp_int(const void*a,const void*b){
    return *((int *)a)-*((int *)b);
}

long random_long(long sup){ //random number is generated digit by digit,sup is the
    upper bound
    long rdm=0;
    long mul=1;
    while(sup>10){   //to guarantee the number of digits of the random number is
        less than sup
        rdm=rdm+rand()%10*mul;
        mul=mul*10;
        sup=sup/10;
    }
    rdm=rdm+rand()%(sup)*mul; //to guarantee the highest digit of rdm is less than
        sup
    return rdm+1; //"+1"to keep the returned number positive
}
```

```
long *initial(long *a,long length,long sup){ //generate N randomly one by one
    long i=0;
    while (i<length){
        a[i++]=random_long(sup);
    }
    return a;
}

double test(long N,long V,long times,void (*solve)(long*,long,long,long)){
    clock_t start,stop;  //initial the clock module
    double duration; //save the total time
    long k=0,target; //k is just for traversing,while target is the "c" in the
        problem
    long *collection=(long *)malloc(sizeof(long)*N); //distribute the space for
        collection
    start=clock(); //start to count the time
    for (k=0;k<times;k++){
        initial(collection,N,V); //every test uses a new random collection
        target=random_long(2*V); //every test uses a new target
        solve(collection,N,V,target); //choose a solve to test
    }
    stop=clock(); //stop to count the time
    duration=((double)(stop-start))/CLK_TCK;
    free(collection);
    return duration;
}

void solve_1(long* collection,long N,long V,long target){
    long i,j,temp;   //i,j are indexes, while temp saves temporary results
    i=0;j=N-1; //i starts from zero and move form left to right and J move
        inversely
    qsort(collection,N,sizeof(long),cmp_int);
        //"qsort()" is a built-in function declared in <stdlib.h>
        //It uses quick sort to sort the collection
        //It guarantees that the collection is non-descending
    while ((temp=collection[i]+collection[j])!=target&&i<=j){
        if (temp<target)
            i++;   //if temp is less than target, we move i to make it bigger
        else
            j--;   //else if temp is bigger than target, we move j to make it smaller
    }
    if (temp==target){
        printf("%d + %d = %d\n",collection[i],collection[j],target); //find the
            result,print
    }
    else{
        printf("None\n"); //didn't find,print None
    }
```

```
}

void solve_2(long* collection,long N,long V,long target){
    char *hash=(char *)calloc(V,sizeof(char)); //Set a hash chart,and initializing
        0
    long i;  //just for traversing
    for (i=0;i<N;i++){
        if (!hash[collection[i]-1]){ //for repetitive item we skip
            hash[collection[i]-1]='1'; //if n appears in collection,hash[n-1] will
                turn from 0 to 1
            if (target>collection[i]&&(target-collection[i]<=V)){ //to avoid
                segmentation fault
                if (hash[target-collection[i]-1]){
                    //it means the number it needs to produce target has
                        appeared,namely we find them
                    printf("%d + %d =
                        %d\n",collection[i],target-collection[i],target);
                    return;   //leave this test
                }
            }
        }
    }
    printf("None\n"); //if we didn't find it,print None
    free(hash);
}
```

Chapter B Declaration

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