Fundamentals of Data Structures

Projects 1: Performance Measurement (A+B)

Author:

Date: 2025/03/15

2024-2025 Spring & Summer Semester

Table of Contents

Chapter 1: Introduction

1.1 Background

Given a set S containing N positive integers, where each element does not exceed a predefined value V. For a target integer c, the task is to determine whether there exist two distinct elements a and b in S, such that $a + b = c$.

1.2 Objectives

1. **Algorithm Implementation:** Implement two distinct algorithms to solve this problem.
2. **Complexity Analysis:** Analyze the time and space complexities of both algorithms.
3. **Performance Evaluation:** Measure and compare the execution times of the two algorithms for varying N and V.

Chapter 2: Algorithm Specification

2.1 Algorithm 1: Brute-Force Enumeration

1. Enumerate every pairs of elements and Judge whether they fit for the requirements(a+b=c).
2. Count the pairs through the process. Initialize the array depending on the number of pairs. In the next enumeration, record the pairs.
3. add -1 as the flag of the end.

The **pseudocode** are as follows:

```
Function Enum(S, N, c):
    Initialize pairs = 0
    For i from 0 to N-1:
        For j from i to N-1:
            If S[i] + S[j] == c:
                pairs += 1
    If pairs == 0:
        Return NULL
    result = Allocate memory for 2*pairs + 1 integers
    Initialize cnt = 0
    For i from 0 to N-1:
        For j from i to N-1:
            If S[i] + S[j] == c:
                result[cnt++] = S[i]
                result[cnt++] = S[j]
    result[cnt] = −1
    Return result
```

2.2 Algorithm 2: Hash Table

1. Enumerate each number in list;
2. for each number `i`:
   - check if the number `c-i` appeared before by looking up in the hash table;
   - if `c-i` has appeared, add the pair to result;
   - if hasn't, marked the number `i` has appeared, so that if `c-i` appears in the following process, they will be added to the result;

The **pseudocode** are as follows:

```
Function Hash(S, N, c):
    Initialize hash = Allocate (c+1) integers, initialized to 0
    Initialize pairs = 0
    For i from 0 to N-1:
        If S[i] <= c:
            rest = c - S[i]
            If hash[rest] == 1:
                pairs += 1
            Else:
                hash[S[i]] = 1
    If pairs == 0:
        Return NULL
    result = Allocate memory for 2*pairs + 1 integers
    Reset hash to 0
    Initialize cnt = 0
    For i from 0 to N-1:
        If S[i] <= c:
            rest = c - S[i]
            If hash[rest] == 1:
                result[cnt++] = S[i]
                result[cnt++] = rest
            Else:
                hash[S[i]] = 1
    result[cnt] = −1
    Free hash memory
    Return result
```

2.3 Measure the performance

We use C's standard library `time.h` to calculate the time duration.

Since the time duration for each algorithm is too short, we set the iteration times(k). For each algorithm, we let it iterate k times.

The steps are as follows:

1. Record the starting time of the algorithm;
2. Running the function;
3. Record the end time of the algorithm.

The **pseudocode** are as follows:

```
Set k = 3
    Get start time
    for i from 0 to k - 1:
        Call Enum function with S, Nlst[n], and cn[i] to get result
        Free result
    Get stop time
    Calculate duration1 = (stop - start) / CLK_TCK
    Set k = 100
    Get start time
    for i from 0 to k - 1:
        Call Hash function with S, Nlst[n], and cn[i] to get result
        Free result
    Get stop time
    Calculate duration2 = (stop - start) / CLK_TCK
```

Chapter 3: Testing Results

3.1 Assignment 1

For the two Algorithms, Sample Input and the expected output are as follows:

c	Expected Output
-100	Not Found
0	Not Found / number 1 = 0 , number 2 = 0
1000	number 1 = 111, number 2 = 889 ...
2000	Not Found
10000	Not Found

Caution

1. for c=0: the output depends on whether there's zero in the set. If there exists zero, the output will be number 1 = 0, number 2 = 0;
2. The normal case c=1000 has numerous output. Only one pair of the output has shown in the table.

3.2 Assignment 2

To test the duration of the running time, I create the `Nlst` and `Vlst` as follows:

```
int Nlst[]={1000,5000,10000,20000,40000,60000,80000,100000}; //N list in Exercise
int Vlst[]={1000,5000,10000,20000,40000,60000,80000,100000}; //V list in Exercise
```

The test results are as follows:

3.2.1 Table

For $V = 1000$:

N	1000	5000	10000	20000	40000	60000	80000	100000
Algorithm 1: Brute Force Enumeration								
Iterations(K)	15	15	15	15	15	15	15	15
Ticks	11	283	1167	2600	18218	64717	86720	136668
Total Time(sec)	0.011	0.283	1.167	4.600	18.218	64.717	86.720	136.668
Duration(sec)	0.00073	0.01887	0.07780	0.30667	1.21453	4.31447	5.78133	9.11120
Algorithm 2: Hash Table								
Iterations(K)	5000	5000	5000	5000	5000	5000	5000	5000
Ticks	38	76	153	248	459	700	1851	1156
Total Time(sec)	0.038	0.076	0.153	0.248	0.459	0.700	1.851	1.156
Duration(sec) (10^{-5})	0.76	1.52	3.06	4.96	9.18	14.00	37.02	23.12

For $V = 5000$:

N	1000	5000	10000	20000	40000	60000	80000	100000
Algorithm 1: Brute Force Enumeration								
Iterations(K)	15	15	15	15	15	15	15	15
Ticks	12	279	1140	4650	24781	62689	88759	126455
Total Time(sec)	0.012	0.279	1.140	4.650	24.781	62.689	88.759	126.455
Duration(sec)	0.0008	0.0186	0.0760	0.3100	1.6520	4.1793	5.9172	8.4303
Algorithm 2: Hash Table								

Iterations(K)	5000	5000	5000	5000	5000	5000	5000	5000
Ticks	38	76	153	248	459	700	1851	1156
Total Time(sec)	0.038	0.076	0.153	0.248	0.459	0.700	1.851	1.156
Duration(sec) (10^{-5})	0.76	1.52	3.06	4.96	9.18	14.00	37.02	23.12

For $V = 10000$:

N	1000	5000	10000	20000	40000	60000	80000	100000
Algorithm 1: Brute Force Enumeration								
Iterations(K)	15	15	15	15	15	15	15	15
Ticks	10	299	1095	4455	19001	47582	75252	153957
Total Time(sec)	0.010	0.299	1.095	4.455	19.001	47.582	75.252	153.957
Duration(sec)	0.00067	0.01993	0.07300	0.29700	1.26673	3.17213	5.01680	10.26380
Algorithm 2: Hash Table								
Iterations(K)	5000	5000	5000	5000	5000	5000	5000	5000
Ticks	15	40	84	170	371	499	695	1215
Total Time(sec)	0.015	0.040	0.084	0.170	0.371	0.499	0.695	1.215
Duration(sec) (10^{-5})	0.30	0.80	1.68	3.40	7.42	9.98	13.90	24.30

For $V = 20000$:

N	1000	5000	10000	20000	40000	60000	80000	100000
Algorithm 1: Brute Force Enumeration								
Iterations(K)	15	15	15	15	15	15	15	15
Ticks	40	404	1661	7139	24384	62665	100307	117337
Total Time(sec)	0.040	0.404	1.661	7.139	24.384	62.665	100.307	117.337
Duration(sec)	0.00267	0.02693	0.11073	0.47593	1.62560	4.17767	6.68713	7.82247

Algorithm 2: Hash Table								
Iterations(K)	5000	5000	5000	5000	5000	5000	5000	5000
Ticks	28	57	109	220	455	692	591	747
Total Time(sec)	0.028	0.057	0.109	0.220	0.455	0.692	0.591	0.747
Duration(sec) (10^{-5})	0.56	1.14	2.18	4.40	9.10	13.84	11.82	14.94

For $V = 40000$:

N	1000	5000	10000	20000	40000	60000	80000	100000
Algorithm 1: Brute Force Enumeration								
Iterations(K)	15	15	15	15	15	15	15	15
Ticks	7	270	1107	4590	18565	41850	74201	145750
Total Time(sec)	0.007	0.270	1.107	4.590	18.565	41.850	74.201	145.750
Duration(sec)	0.00047	0.01800	0.07380	0.30600	1.23767	2.79000	4.94673	9.71667
Algorithm 2: Hash Table								
Iterations(K)	5000	5000	5000	5000	5000	5000	5000	5000
Ticks	22	57	76	147	294	446	580	1201
Total Time(sec)	0.022	0.057	0.076	0.147	0.294	0.446	0.580	1.201
Duration(sec) (10^{-5})	0.44	1.14	1.52	2.94	5.88	8.92	11.60	24.02

For $V = 60000$:

N	1000	5000	10000	20000	40000	60000	80000	100000
Algorithm 1: Brute Force Enumeration								
Iterations(K)	15	15	15	15	15	15	15	15
Ticks	12	472	1751	7191	23807	41501	75797	116762
Total Time(sec)	0.012	0.472	1.751	7.191	23.807	41.501	75.797	116.762

Duration(sec)	0.00080	0.03147	0.11673	0.47940	1.58713	2.76673	5.05313	7.78413
Algorithm 2: Hash Table								
Iterations(K)	5000	5000	5000	5000	5000	5000	5000	5000
Ticks	29	68	118	215	279	446	543	669
Total Time(sec)	0.029	0.068	0.118	0.215	0.279	0.446	0.543	0.669
Duration(sec) (10^{-5})	0.58	1.36	2.36	4.30	5.58	8.92	10.86	13.38

For $V = 80000$:

N	1000	5000	10000	20000	40000	60000	80000	100000
Algorithm 1: Brute Force Enumeration								
Iterations(K)	15	15	15	15	15	15	15	15
Ticks	7	268	1059	4557	20623	61998	101102	117709
Total Time(sec)	0.007	0.268	1.059	4.557	20.623	61.998	101.102	117.709
Duration(sec)	0.00047	0.01787	0.07060	0.30380	1.37487	4.13320	6.74013	7.84727
Algorithm 2: Hash Table								
Iterations(K)	5000	5000	5000	5000	5000	5000	5000	5000
Ticks	16	45	76	138	426	905	576	706
Total Time(sec)	0.016	0.045	0.076	0.138	0.426	0.905	0.576	0.706
Duration(sec) (10^{-5})	0.32	0.90	1.52	2.76	8.52	18.10	11.52	14.12

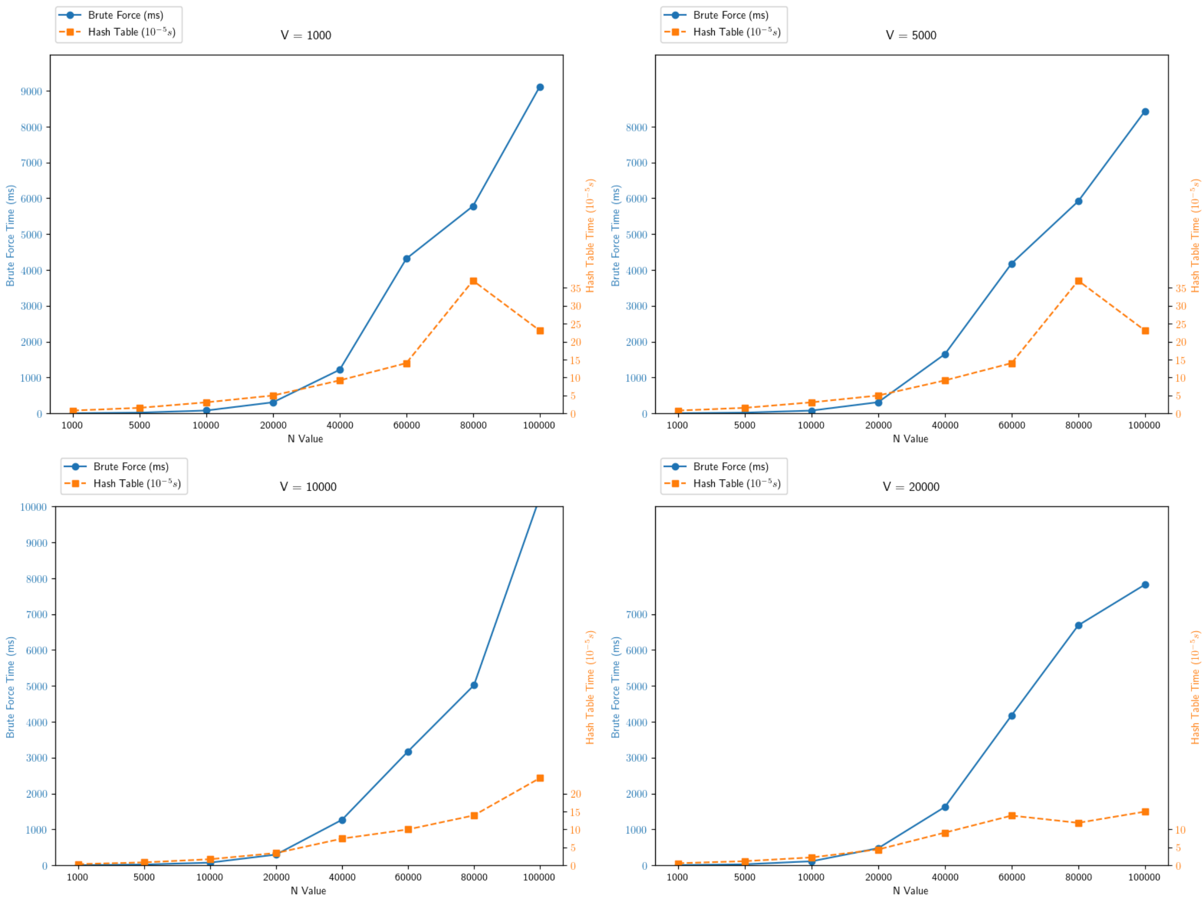
For $V = 100000$:

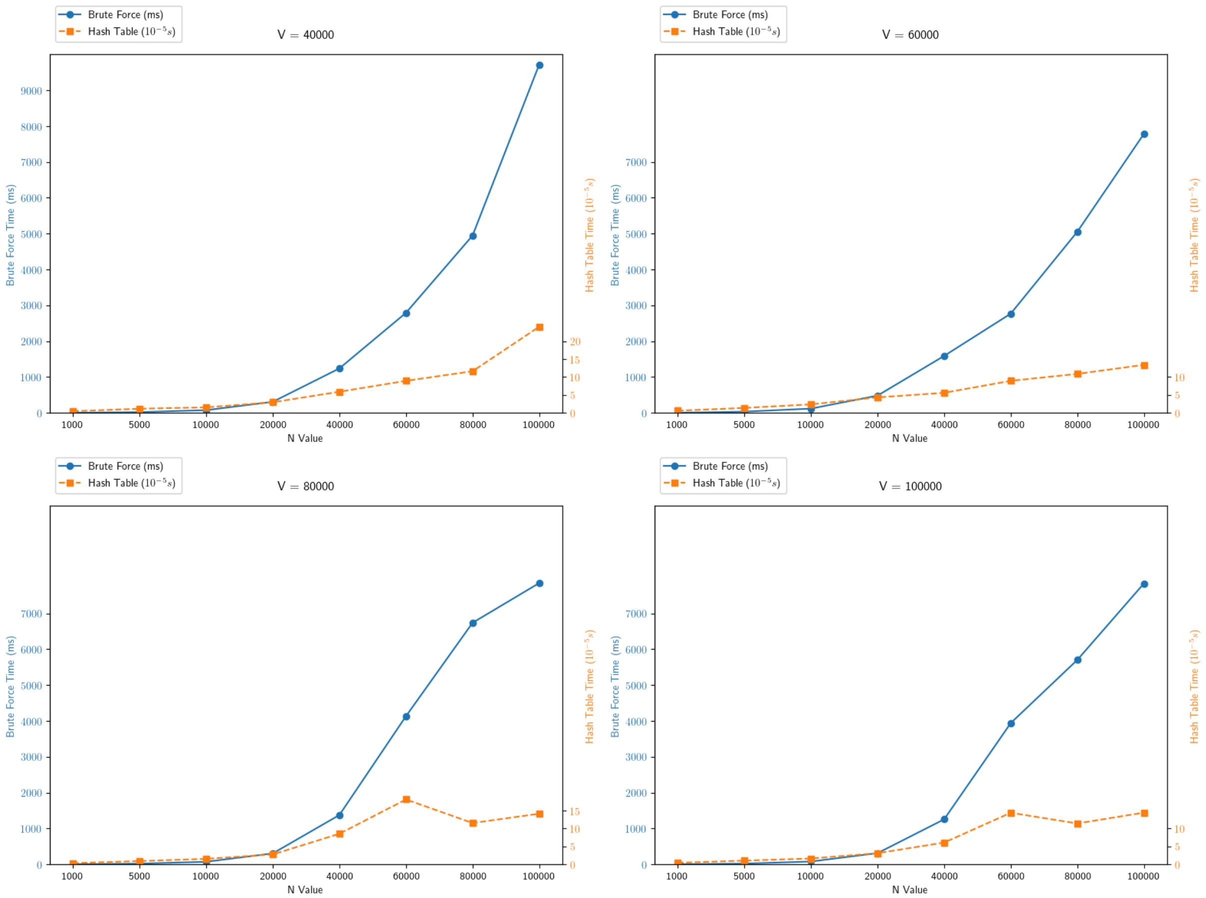
N	1000	5000	10000	20000	40000	60000	80000	100000
Algorithm 1: Brute Force Enumeration								
Iterations(K)	15	15	15	15	15	15	15	15
Ticks	8	282	1167	4669	18821	59101	85564	117560

Total Time(sec)	0.008	0.282	1.167	4.669	18.821	59.101	85.564	117.560
Duration(sec)	0.00053	0.01880	0.07780	0.31127	1.25473	3.94007	5.70427	7.83733
Algorithm 2: Hash Table								
Iterations(K)	5000	5000	5000	5000	5000	5000	5000	5000
Ticks	21	51	81	155	303	719	570	721
Total Time(sec)	0.021	0.051	0.081	0.155	0.303	0.719	0.570	0.721
Duration(sec) (10^{-5})	0.42	1.02	1.62	3.10	6.06	14.38	11.40	14.42

3.2.2 Figures

For different V, plot the N-Duration and the figures generated are as follows:





3.2.3 Conclusion

1. **For the specific algorithm:**
   - In **Bruce Force Enumeration**, when the N doubles ,the Duration Quadruple, which meets $O(n^2)$ complexity characteristics. For example, V=1000 , N 1000 $\rightarrow$ 20000 (20 times), Duration 0.00073 $\rightarrow$ 0.30667 seconds (about 420 times), which is close to the theoretical $20^2 = 400$ times.
   - In **Hash Table**, When the input size N doubles, the running time (Duration) increases approximately linearly. For example, when V=10000, N increases from 1000 to 100000 (100 times), and Duration increases from 0.30×10^{-5} seconds to 24.30×10^{-5} seconds (81 times), which is close to a linear growth ratio.
2. **For a specific *N*:**
   - When N is large, the time complexity of $O(N^2)$ for the brute-force double-layer loop will cause its execution time to increase sharply. It's easy to see that $T_1(N) \approx C_1(V) \cdot N^2$
   - Since the complexity of Hash Table is $O(N)$, the figure of execution time for Hash Table will increase linearly with the increase of N. You can see that $T_2(N) \approx C_2(V) \cdot N$
3. The difference in running time for the same N at different values of V is small, indicating that V only affects the constant factor and does not change the trend of $O(N^2)$.

Chapter 4: Analysis and Comments

4.1 Algorithm 1: Brute Force Enumeration

4.1.1 Time Complexity

1. **First loop pair counting**:
   - The outer loop runs for N times (`i` from 0 to $N-1$).
   - The inner loop runs for $N-i$ times (`j` from i to $N-1$).
   - Total iterations: $N+(N-1)+...+1=\frac{N(N+1)}{2}\rightarrow O(N^2)$.
   - Each loop body checks S[i] + S[j] == c (O(1) operation).
2. **Second loop result generation**
   - Same nested loop structure as above $\rightarrow O(N^2)$.

4.1.2 Space Complexity

The size of result is `2*pairs + 1`. In the worst case , the number of pairs is $O(N^2)$, so the space complexity is $O(N^2)$.

4.2 Algorithm 2: Hash Table

Enumerate N elements. For each operation, the time complexity is $O(1)$. So the whole complexity is $O(N)$

4.2.1 Total Time Complexity

$O(n^2)+O(n^2)=O(n^2)$

4.2.2 Space Complexity

The size of the hash table hash is `c+1`, and it takes up $O(c)$ space. The size of the result array result is `2*pairs + 1`, and in the worst case pairs is $O(N)$ (if each element has a unique pair),

So the space complexity is $O(N)$.

Appendix: Source Code (in C)

File sol.c:

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
clock_t start,stop;
double duration1,duration2;
int d1, d2;
//Method 1: Brute force enumeration

int* Enum(int S[], int N, int c) {
    int pairs=0;
    for(int i=0;i<N;i++) {
        for(int j=i;j<N;j++) {
            if(S[i]+S[j]==c) {
```

```c
                pairs++;//first enumerate to initialize the size of the array
            }
        }
    }
    if(pairs == 0){
        return NULL; //no results
    }
    int *result = (int*)malloc(2*pairs*sizeof(int)+1);
    int cnt = 0;
    for(int i=0;i<N;i++) {
        for(int j=i;j<N;j++) {
            if(S[i]+S[j]==c) {
                result[cnt++]=S[i];
                result[cnt++]=S[j];
            }
        }
    }
    result[cnt++]=-1;
    return result;
}


//Method 2: Hash Table

int* Hash(int S[],int N, int c){
    int* hash =(int*)calloc(c+1, sizeof(int));
    int rest;
    int pairs;
    for(int i=0;i<N;i++) {
        if(S[i]<=c) { //to avoid negative numbers
            rest = c - S[i]; //ensure rest + S[i] = c
            if (hash[rest]==1) {
                pairs++;
            }
            else
                hash[S[i]]=1;//mark as occurred
        }
    }
    if(pairs == 0){
        return NULL;
    }
    int *result = (int*)malloc(2*pairs*sizeof(int)+1);
    int cnt = 0;
    for(int i=0;i<N;i++) {
        if(S[i]<=c) {
            rest = c - S[i]; //ensure rest + S[i] = c
            if (hash[rest]==1) {
                result[cnt++]=S[i];
                result[cnt++]=c-S[i]; // to record the result
            }
            else
                hash[S[i]]=1;
```

```
        }
    }
    result[cnt++]=-1;
    free(hash);
    return result;
}

void Print(int result[]){
    if(result!=NULL){
        int i=0;
        while(result[i]!=-1){
            printf("number 1 = %d , number 2 = %d\n",result[i],result[i+1]);
            i+=2;
        }
    }
    else{
        printf("Not Found!\n");
    }
}

int main() {
    //Generate S set
    srand((unsigned int)time(NULL));//Generate random number seeds

    //Ass1: judge the correctness of two algorithms
    /**
     * for Ass1, testing data for c can be -100,0,1000,2000,10000
     * -100 -> Not Found
     * 0 -> Not Found or both 0 (depending on S)
     * 1000 -> normal output
     * 2000 -> Not Found
     * 10000 -> Not Found
     */
    int N=1000;
    int V=1000;
    int *S=(int *)malloc(N*sizeof(int));//dynamic allocation
    for(int i=0;i<N;i++)
        S[i] = rand()%(V+1);//Generate random numbers in 0-V
    int c = rand()%(2*V+1); //input c
    printf("c=%d\n",c);
    int* result1 = Enum(S,N,c);
    int* result2 = Hash(S,N,c);
    printf("Method 1\n");
    Print(result1);
    printf("Method 2\n");
    Print(result2);
    free(S);
    //Ass2: Analyze the complexities
    int Nlst[]={1000,5000,10000,20000,40000,60000,80000,100000}; //N list in Exercise
    int Vlst[]={1000,5000,10000,20000,40000,60000,80000,100000}; //V list in Exercise
    int k=100; //circle numbers
    int *cn=(int *)malloc(k*sizeof(int));
```

```
    for(int i=0;i<k;i++)
        cn[i] = rand()%(2*V+1); //random c for test
    for(int v=0;v<8;v++){//different V
        printf("V=%d\n",Vlst[v]);
        for(int n=0;n<8;n++){
            printf("N=%d\n",Nlst[n]);
            int *S=(int *)malloc(Nlst[n]*sizeof(int));//dynamic allocation
            for(int i=0;i<Nlst[n];i++)
                S[i] = rand()%(Vlst[v]+1);//Generate random numbers in 0-V
            k=15;
            //Method 1: Bruce Enumeration
            start=clock();
            for (int i = 0;i<k;i++){
                int *result = Enum(S,Nlst[n],cn[i]);
                free(result);
            }
            stop=clock();
            d1=stop-start;
            duration1=((double)(stop-start))/CLK_TCK;//calculate the duration for bruce
enumeration
            k=5000;
            start = clock();
            for(int i=0;i<k;i++){
                int *result = Hash(S,Nlst[n],cn[i]);
                free(result);
            }
            stop = clock();
            d2=stop-start;
            duration2=((double)(stop-start))/CLK_TCK;//calculate the duration for hash
table
            printf("Enumeration: %d, Hash: %d\n",d1,d2);
            printf("Duration: Enumeration: %lf, Hash: %lf\n",duration1,duration2);
            free(S);
        }
    }
    free(cn);
    return 0;
}
```

Declaration

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