**Hashing - Hard Version**

**PR Version**

**Date: 2019-06-10**

**Chapter 1: Introduction**

Given a hash table of size N, we can define a hash function H(x)=x%N. Suppose that the linear probing is used to solve collisions, we can easily obtain the status of the hash table with a given sequence of input numbers.

However, now you are asked to solve the reversed problem: reconstruct the input sequence from the given status of the hash table.

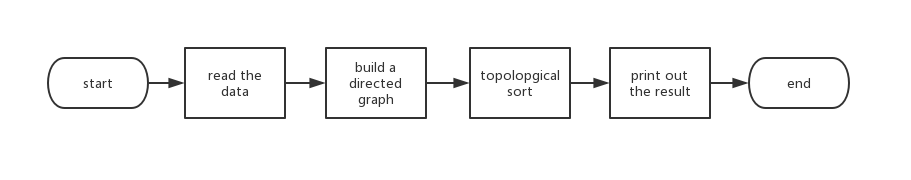
In other words, it’s the inverse process of a hash table. Given a hash table sequence and a hash function, we are required to find the original sequence.

Moreover, when there are multiple results, the smallest number is always taken.

From the description of the problem, it’s obviously that we can use topological sort to solve the problem. Thus, we store the points that don’t have duplicates into a directed graph. Then, we put the points with degree 0 which are greater than 0 into the set. After that, we should pop the minimum point every time and push the points associated with it until the set is empty. When the set is empty, we get the answer of the problems.

### Chapter 2: Algorithm Specification

2.1program frame



2.2 build a directed graph

BuildGraph(Hash[], n){

(Hash[] is the vector that store the given array, n is the size of the hash table, A[][] is the adjacency List of Directed Graphs)

for i:=0 to n

if Hash[i] >= 0

then s:=Hash[i]%n;

if Hash[s]!=Hash[i]

(If this is true ,it’s said that Hash[i] has duplicates. In other words, the degree of it is greater than 0)

then for j:=s to i(When j>n, j:=j%n)

A[j][i]=1

(build the adjacency List of Directed Graphs)

}

2.3 topological sort

TopSort(Hash[], Index[], n, count){

(Hash[] is the vector that store the given array, Index[] is the order of data input in Hash[], n is the size of the hash table, count is the number of significant figures)

initialize degree[MAX]

(degree[] is used to store the degree of each point in the Hash[], MAX is the max size of the hash table)

for i:=0 to n

degree[i]:= degree of Hash[i]

for i:=0 to n

if degree[i]=0 and Hash[i]>0

then insert Hash[i] into set p

while p is not empty

index:=Index[p.begin]

(p.begin is the first element of p. Index[p.begin] is the index of the current smallest element(positive integer) in the Hash[])

delete p.begin

output Hash[index]

for j:=0 to n

if A[index][j] is not equal to 0

then degree[j]:=degree[j]-1

if degree[j]=0

then insert Hash[j] into set p

end while

}

### Chapter 3: Testing Results

We wrote a randomized test program to automate the testing process, and our program is encapsulated in the target.h header file.

The automated test program:

#include <stdio.h>

#include <stdlib.h>

#include <chrono>

#include <time.h>

#include <math.h>

#include <iostream>

#include "target.h"

#define MAXN 10001

**int** Bucket[10001];

**int** HashTable[MAXN];

**int** is\_prime(**int** n)

{

**if**(n == 2 || n == 3) {

**return** 1;

}

**if**(n == 1 || n == 0) {

**return** 0;

}

**int** max = (**int**) sqrt(n);

**for**(**int** i = 2; i <= max; i ++) {

*// if divides*

**if**(! (n % i)) {

**return** 0;

}

}

**return** 1;

}

*// n is an integer, the function returns a prime next to n*

**int** get\_next\_prime(**int** n)

{

**for**(**int** i = n; 1; i++) {

**if**(is\_prime(i)) {

*// printf("prime: %d\n", i);*

**return** i;

}

}

}

**int** generateRandomInteger(**int** min, **int** max)

{

**return** min + rand() % (max - min);

}

**int** generateRandomUniqueInteger(**int** min, **int** max)

{

**int** res;

**while**(Bucket[res = generateRandomInteger(min, max)]);

Bucket[res] = 1;

**return** res;

}

*// linear probing applied*

**int** main() {

*// printf("%d", generateRandomInteger(0, 100));*

srand((**unsigned**)time(0));

**int** N, a, count;

N = 2000;*//generateRandomInteger(1, 1000);*

N = get\_next\_prime(N);

printf("Hash Table Size: %d\n", N);

count = N / 3; *//generateRandomInteger(1 , N / 10);*

printf("Number of Integers to be inserted: %d\n", count);

printf("Number Generated: ");

**for**(**int** i = 0; i < count; i++){

**int** m = generateRandomUniqueInteger(1, 10000);

printf("%d ", m);

**int** index = m % N;

**for**(**int** j = index; j < index + N; j++) {

**int** t = j % N;

**if**(!HashTable[t]) {

HashTable[t] = m;

**break**;

}

}

}

printf("\n\nHash Table:\n");

**int** negative = -1;

**for**(**int** i = 0; i < N; i++) {

printf("(%d):", i);

**if**(HashTable[i]) {

printf("%d ", HashTable[i]);

} **else** {

HashTable[i] = negative--;

*// printf("%d ", negative--);*

printf("%d ", HashTable[i]);

}

}

printf("\n\nReconstructed Sequence:\n");

**auto** start = std::chrono::high\_resolution\_clock::now();

target(N, HashTable);

**auto** finish = std::chrono::high\_resolution\_clock::now();

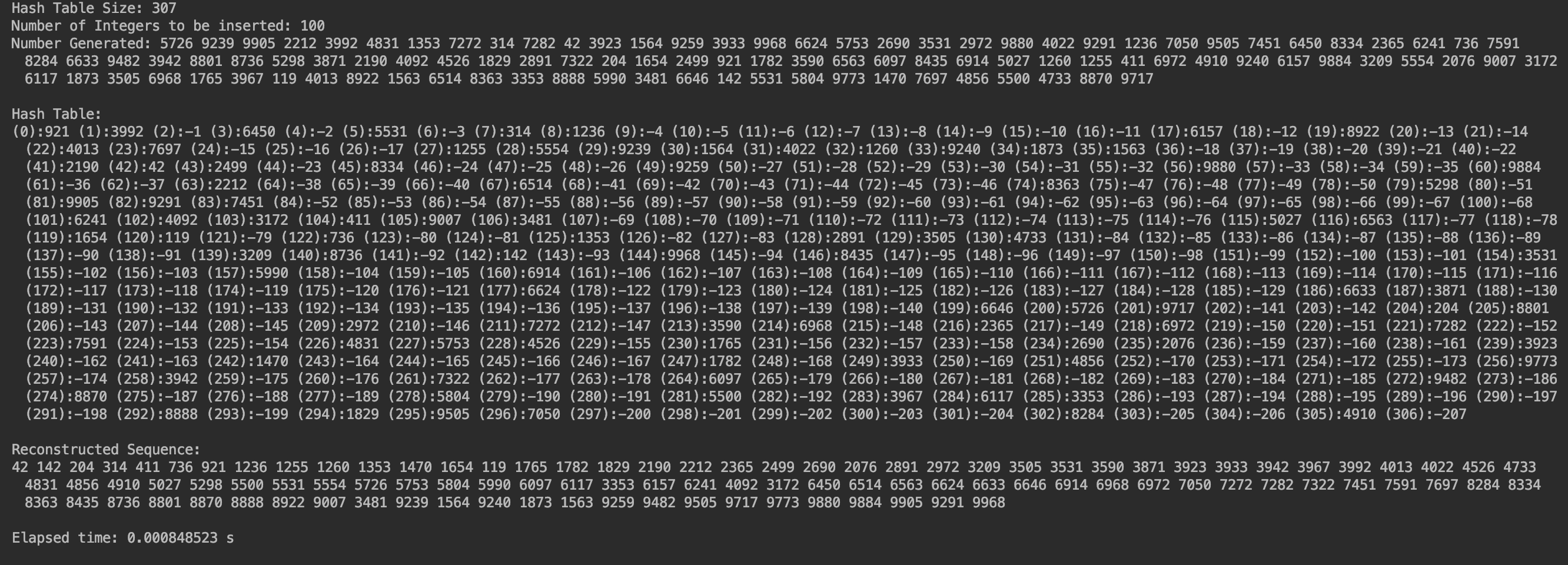
std::chrono::duration<**double**> elapsed = finish - start;

std::cout << "\n\nElapsed time: " << elapsed.count() << " s\n";

**return** 0;

}

Following is an example of the output of the test program:

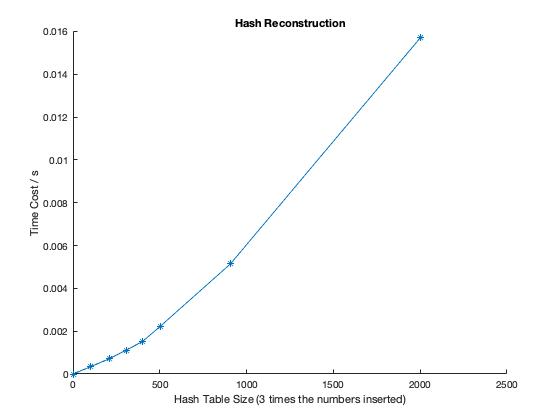


With this automated program, we are able to make a series of simulations when the hash table size is at the smallest prime number greater than 100, 200, 300, 400, 500, 900 and 2000. For each size, the integers generated are between 1 and 10000, and the lengths of the sequences inserted are 1/3 the size of each hash table to guarantee a smaller probability of primary collision. We made 3 parallel simulations for each size, calculated the average and plotted a graph in terms of size and time (in seconds) consumed for topological sort.

Table: Time Consumption for each table size

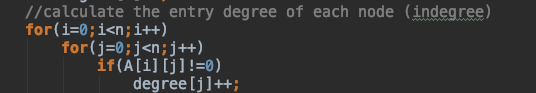
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Table Size | Trial 1 | Trial 2 | Trial 3 | Average Time |
| 101 | 0.000335237 | 0.000369189 | 0.000314942 | 0.000339789 |
| 211 | 0.000626975 | 0.000735325 | 0.000802515 | 0.000721605 |
| 307 | 0.00106561 | 0.0010096 | 0.00126865 | 0.00111462 |
| 401 | 0.00153683 | 0.00152334 | 0.00151085 | 0.00152367 |
| 503 | 0.00221961 | 0.0023117 | 0.00216632 | 0.00223254 |
| 907 | 0.00511226 | 0.00481886 | 0.00550359 | 0.00514490 |
| 2003 | 0.0161041 | 0.0152485 | 0.0158316 | 0.0157281 |

Graph: Hash Table Size – Time Consuption



### Chapter 4: Analysis and Comments

As we can infer from the Graph in Chapter 3 intuitively, the time complexity of our algorithm is probably . Now let’s confirm that with a more detailed analysis. As we can see in the source code:



We used an incident matrix representation of the graph and applied a 2-level for-loop to calculate the indegree of each node in the graph. This step is the decisive one in our algorithm, thus obtaining an overall time complexity.

In conclusion, the performance of the program in the simulation process is in accordance with our expectation in algorithm analysis.

### Appendix: Source Code (if required)

At least 30% of the lines must be commented. Otherwise the code will NOT be evaluated.

### References

None

### Author List

### Declaration

***We hereby declare that all the work done in this project titled "* *Hashing - Hard Version " is of our independent effort as a group.***