AoA2 –HW3

Compile: g++ -std=c++11 -Wall -Werror 150160060.cpp

OR

g++ -std=c++11 150160060.cpp

Run: ./a.out data.txt

PART1

FUNCTIONS FOR PART1:

```
//FOR KNAPSACK ALGO.
```

- `int max(int x, int y) {…}`

```
/*  knapSack:
      W=max weight limit
      w=items' weight
      v=items' values
*/
```

- `vector<int> knapSack(int W, vector<int> w, vector<int> v) {…}`

1. **Write the mathematical representation of the optimization function you used in your code. Find the complexity of your algorithm and discuss in your report.**

   I used Knapsack algorithm from the slides. In this algorithm, finds the items that their total weight is less than or equal to the given limit and the total value is large as possible. In this case weight is running time of test suites and value is bug count of test suites. The mathematical representation of the this algorithm is given below:

$$OPT(i,w) = \begin{cases} 0 & if\ i = 0 \\ OPT(i-1,w) & if\ w_i > w \\ \max\{OPT(i-1,w), v_i + OPT(i-1,w-w_i)\} & otherwise \end{cases}$$

   Complexity of this algorithm is **O(nW)** since we build a 2D array[n][W]. n is test suites total number and W is maximum capacity of running time.

   In order to find the optimal items selected in this algorithm, I iterate the matrix i have implemented backwards. Then I compare it with the cell vertically upward, if they are not equal we know that this cell is selected item. Complexity of this loop is **O(n)** since we iterate the loop vertically upward every time.

2. **Does your algorithm work if the running times of the test suites are given as real numbers instead of discrete values? If not, please provide a solution in your report as pseudo code and discuss why it works and the previous solution does not work.**

   Since Knapsack algorithm implement an matrix if running times(weights) were real numbers it is not able to work because there are infinite number of real numbers even between 0 and 1, Endless memory is required to implement such a matrix, which is impossible.

We can scale them up to be integers again if the weights' precisions are fixed, but that would increase the size of the problem up in an inacceptable way.

In order to solve this problem, we cannot use dynamic programming. We can use recursion solution but that makes the time complexity O(2^n). The pseudo code of this solution is given below:

```
/*  Recursive knapSack:
//  W=max weight limit, w=items' weight, v=items' values
```

- `int knapSack(int W, vector<int> w, vector<int> v) {`

```
        // Base Case
        if (n == 0 || W == 0)
        return 0;

        if (w[n - 1] > W)
            return knapSack(W, w, v, n - 1);

        else
            return max(
                val[n - 1] + knapSack(W - w[n - 1], w, v, n - 1),
                knapSack(W, w, v, n - 1));
    }
```

PART2

FUNCTIONS FOR PART2:

```
/* QUICKSORT ALGORITHM (DESC) */
```

- `int partition(vector<pair<int, int>> *A, int p, int r) {…}`
- `void quicksort(vector<pair<int, int>> *A, int p, int r) {…}`

```
/*  editDist:
       tc1=testcase1
       tc2=testcase2*/
```

- `int editDist(vector<int> tc1, vector<int> tc2){…}`

```
/*  FindCoverage:
       ->Finds coverage of a testcase
    ->Creates ordered Sequence of a testcase
       ->Returns coverage of that testcase
       v=test case
       seq*= vector of ordered sequences
       c*= vector of all testcases coverage(for one test suite)*/
```

- `int FindCoverage(vector<int> v, vector<vector<int>>* seq, vector<int>* c) {…}`

```
/* TestCasePrioritization
       ->Finds and prints prioritized vector of test cases
       t=test suite's test cases vector*/
```

- `void TestCasePrioritization(vector<vector<int>> t) {…}`

1. **Describe the details regarding your algorithm design, such as which edit distance algorithm you used, what are the operation costs you set? Write the mathematical representation of the optimization function you used in your code. Find the complexity of your algorithm and discuss it in your report.**

   For part 2 I implemented a function called `TestCasePrioritization`.This function's operations are given below respectively:

- To find test cases coverage send all test cases to `FindCoverage` function m times. This costs -> **O(mnlgn)** m=number of test cases n= size of test case / frequency number.

  Inside `FindCoverage` **O(nlgn)**:
  - Visit every frequency(n) and add frequency and its index pair to multimap(multimap operations cost O(lgn)) **O(nlgn)**
  - Multimap insert frequency and index pair in a sorted way. So we can create ordered sequence easily with pushing multimap's second value(index) respectively. O(n)
- Push testcases' indexes that has maximum coverage to a vector O(n)
- Edit distance(`editDist`) algorithm that I used **Levenshtein algorithm** which costs me **O(n^2)** in every call**.** In this algorithm n is the size of test case. The mathematical representation of this algorithm is given below: (I implemented the function in the form of a, b instead of implementing n, n in case the sizes of the test cases may be different.)

$$lev_{a,b}(i,j) = \begin{cases} \max(i,j) & if\ \min(i,j) = 0 \\ min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & otherwise \end{cases}$$

`editDist` algorithm is called x(=number of test case(m) – test cases that has max coverage) times from `TestCasePrioritization`. So total cost of this is **O(xn^2).**

- Then I sorted test cases that I found their distances earlier with quicksort that i implemented. O(xlgx) x=number of test case - testcases that has max coverage

The total complexity of `TestCasePrioritization` within these calculations is:

$$\boldsymbol{O(mnlgn + xn^2)}$$

m= number of test cases,

n= size of test cases,

x=number of test case - testcases that has max coverage

OUTPUT

```
[cekicl6@ssh algo2-3]$ g++ -std=c++11 -Wall -Werror 150160060.cpp
[cekicl6@ssh algo2-3]$ ./a.out data.txt
Total amount of bugs: 51
Total amount of running time: 26

Selected Test Suites
********************
testSuiteId     bugsDetected    runningTime
TS2             13              7
TS3             23              11
TS4             15              8

Order of Test Cases
********************
TS2 Order:  1
Distances:  0

TS3 Order:  5  -  6  - 11  - 12  -  2  -  4  - 10  -  8  -  9  -  3  -  1  -  7
Distances:  0  -  0  -  0  -  0  - 10  -  6  -  6  -  4  -  4  -  4  -  4  -  2

TS4 Order:  3  -  4  -  2  -  1
Distances:  0  -  4  -  3  -  3

[cekicl6@ssh algo2-3]$
```