BLG336E - ANALYSIS OF ALGORITHMS II
ASSIGNMENT 3 REPORT

Student Name: Damla Nisa CEVIK

Student ID: 150170712

Compile: g++ 150170712.cpp -o program

Run: ./program <txt file name>

Header files:

- TestSuite.h:
  - Variables/Data: detected bugs, running time for the test suite, frequency profiles, ordered sequences of statement execution frequencies, ordered execution results.
  - Functions: highest_coverage_index(used to find the test case that has the most coverage), order_sequnces(to obtain the order of the test cases according to the Levenshtein distance), levenshtein(to calculate the Levenshtein distances among test cases).
- AllTests.h:
  - Variables/Data: tests(type of TestSuite), maximum running time, total running time of the selected test suites, selected test suite ids.
  - Functions: knapsack(to select test suites), string_to_integer(to convert frequency profiles from string to integer to calculate ordered sequences of statement execution frequencies), print_solution().

Libraries used:

- iostream: for input output functions
- fstream, sstream, string: for file operations and string manipulations
- vector: for storing the data
- algorithm: sort() function is used for obtaining the ordered sequences of statement execution frequencies

PART1:

- **Explanation of the optimization algorithm**: The first part of the problem is the same as Knapsack Problem (0-1 Knapsack Problem). So the solution of the ***0-1 Knapsack Problem*** is used to optimize the test suites. Knapsack Problem is one of the typical ***Dynamic Programming*** problem. Rather than using recursion to solve this problem dynamic programming approach avoids computation of the same subproblems again and again by constructing an 2D array in ***bottom-up*** manner. In the Knapsack problem there are weights and values for items and maximum capacity for knapsack. In our problem weights are running times and values are number of detected bugs for test suites and the maximum capacity of knapsack is maximum running time.

- **Mathematical representation of optimization**: (original problem: weights(used running times), values(used number of detected bugs), maximum weight(used maximum running time))

  Assume that there are $t_1, t_2, \ldots, t_n, T$ are positive integers. The main goal is to find $max[i, t]$ to be the maximum value that can reached by the time less than or equal to T using i items. The optimization/selection algorithm can be represented as following:

$$OPT(i, t) = \begin{cases} 0, & \text{if } i = 0 \\ OPT(i-1, t), & \text{if } t_i > w \\ \text{maximum } \{OPT(i-1, t), t_i + OPT(i-1, t - t_i)\}, & \text{otherwise} \end{cases}$$

  Main goal representation:

  t_k = *the running time of each test suite item, for* $k = 1, 2, \ldots, N$

  b_k = *the number of detected bugs with each test suite for* $k = 1, 2, \ldots, N$

  T = *the maximum running time*

  x_k = *number of test suites included into knapsack*

  Maximize $\sum_{k=1}^{N} b_k x_k$ subject to $\sum_{k=1}^{N} t_k x_k <= T$

- **Time complexity** for this algorithm is ***O(n*T)*** where n is the number of test suites and T is the maximum running time. 0-1 Knapsack algorithm implemented in two nested for loops. The outer one runs n times and the inner for loop runs T times. So the complexity of this algorithm becomes ***O(n*T)***. This algorithm keeps ***O(n*T)*** space also, because of the temporary 2D array.

- If the running times of the tests suites are given as real numbers the program will not work. However by changing the types from *integer* to *double* will fix the problem for float numbers. But if the maximum running time is given as real number the change from integer to double will not fix the problem. In this case a whole new approach is needed. If the maximum running time is real, the problem will not be a 0-1 Knapsack Problem. This problem can be solved by a ***greedy algorithm***. In this algorithm the test suits should be sorted by their (number of detected bugs)/ (running time) ratio, then test suites should be selected.

- **Pseudocode for this algorithm:**

```
greedy_algo_for_real_weights() {
for i= 0 to number_of_test_suites {
        bugs_running_time_ratio[i] = test_suite_bugs[i] / test_suite_running_time[i];}
sort(detected_bugs_number_running_time_ratio);
double current_total_running_time;
for i = number_of_test_suites to 0 {
        current_test = detected_bugs_number_running_time_ratio[i]
        if (current_test_running_time + current_total_running_time <= maximum_running_time)
                add current test suite to selected_suites
} answer: selected_suites
```

PART2:

- Levenshtein distance algorithm is used for edit distance measuring. The Levenshtein distance is a string metric for measuring difference between to sequences. In our problem the edit distance between the ordered sequences of statement execution frequencies can be calculated by this algorithm. Levenshtein distance algorithm calculates minimum number of edits (operations) required to convert first string into second string. No operation cost is selected since the selection between *insert*, *remove* and *replace* actions is not important. A temporary 2D array is used to calculate the distance.

- **Mathematical representation of Levenshtein Distance**:

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

- **Time complexity** of Levenshtein Distance Algorithm is ***O(n*m)*** where n is the length of the first string and m is the length of the second string. In this algorithm there are two nested for loops, the one of them executed the length of the first string times and the other for loop executed the length of the second string times, so complexity is calculated by multiplying the execution times: ***O(n*m)***. Also it takes ***O(n*m)*** space since the allocation of 2D array which has the size *n*m*.