Tanay Cansin Dogan
150150119

BLG336E – Analysis of Algorithms II – Homework 3

Compilation:

The source code is written in c++11, compile as: **g++ -std=c++11 150150119.cpp**

Part1:

In this part the problem was selection of a subset from the set of test suites which are in subject of "run time" constraints, with "detected bugs" values. Since this is a kind of "Knapsack Problem". We have implemented the Bellman's dynamic programming solution:

Bellman equation.

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

In this solution, we have constructed a (n+1)*(W+1) table where n is the cardinality of the test set, and W is the upper limit of the constraint (max allowed time). Each cell of the table is calculated by using previously calculated cells. Since access time for arrays, and max(a, b) runs in **constant time** construction of the whole table has a O(n*W) complexity.

No, my algorithm does not work, if the running times of the test suites are given as real numbers. It can be solved as: Since every real number in computer is represented as **"floating point representation"**, they are actually rational numbers. Hence can be formulated as $R_i = A_i/B_i$ and CT = C/D, where R_i is running time for test suite i, and A_i, B_i, C, D are integers.

For i=1 to n:
    Calculate B_i #from the floating point representation
    Divisors[i] ← B_i
Calculate D #from the floating point representation
LCM ← LeastCommonMultiple(D, B_1) #calculate the least common multiple of D and B_1
For i=2 to n:
    LCM ← LeastCommonMultiple(LCM, B_i)

For i = 1 to n:
    newR_i ← LCM*R_i #since LCM is a multiple of B_i result will be integer
newCT ← LCM*CT #since LCM is a multiple of D result will be integer

Since limit and constraint functions are integer now, Bellman's Knapsack algorithm can be solve. Previous work couldn't solve since it's impossible to construct a table with reel and incremental rows.

Part2:

In this problem we implemented a test case order algorithm in order to satisfy, max statement coverage and increase variation of those coverage. I have implemented the Levenshtein distance algorithm as an edit distance measurement.

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

Since to calculate each element of the table, previously calculated cells of the table is accessed and used, min & max operations and access has constant time complexity. In my implementation: where **n** is number of test cases, **r** is length of sequence, and **m** is greatest element in the sequence**.**

1) Calculation of the maximum coverage in O(n*r) complexity
2) Search the maximum coverage test case in O(n) complexity
3) Str2Vec: turns string to the vectors in O(r) complexity
4) OrderSeq: orders sequence according to occurance in O(m*r) complexity
5) EditDistance: calculates the distance of 2 sequence in O(r*r) complexity
6) Calculate all distances for all sequences compare to the max occurance sequence in (n-1)*O(r*r) complexity.
7) Sort them in O(n*logn)
8) Push and print final order in O(n) complexity

To sum up total complexity of Part2() function is: O($n*r^2$) + O(m*r) + O(n*logn) if n dominates r and m: it's O(n*logn), if m dominates r and n: it's O(m), if r dominates m and n: it's O(r^2)