**Name: Yicheng Jia GWid: G24072171 Mail:** [**jiayicheng111@gwmail.gwu.edu**](mailto:jiayicheng111@gwmail.gwu.edu)

**Option A:** Tasted and Healthier Burgers

**Problem Description:**

There are a group of elements. Each element has two attributes: value A, and value B. What the project have to realize is to select the maximum number of elements that can be arranged in a list , such that both the value of A and B in the element are bigger than the one before. For example, given 4 elements with values B1 = (1.0,5.1), B2 = (2.1,3.2), B3 = (3.1,4.5) and B4 = (2.7,4.2), we can select [B2, B4, B3] as the longest possible list.

Note: The given list of burgers is not in any specific order.

**Analysis:**

As mentioned above, each element has two attributes and we must find a list of element that A and B in the latter element are bigger than the former one at the same time. Besides that, the problem also requires the maximum number of elements among the possible subsequences. What we get used to analyze is an element has a single attribute and we can solve this problem by simulating the solution of Longest Increasing Subsequence (LIS) problem. Therefore, we may settle “Burger” problem by achieving an analogy between LIS problem and itself.

After closer observation, we have found that we can convert this two-attribute problem to single attribute problem by operating the element group. If sorting these elements by either value of A or B first and then operating the sorted list and rest value by dynamic programming, we can simplify two-attributes “Burger” problem to only one attribute which need to be considered. Thus, we have to sort the element by one of two attributes, for example, the value of A, and then solve the rest of problem through dynamic programming, which will introduce in the following paragraph.

**Notation:**

Suppose len represents the size of longest strictly increasing subsequence, and pos[i] represents the position that element I should be placed in the longest strictly increasing subsequence. Then element from 0 to pos[i] (X[0] to X[pos[i]]) represents the possible longest increasing subsequence. The reason why it called “possible” will explain in the “Problem Introduction” part.

**Note: This notation can make algorithm faster, only take O(nLogn) time.**

**Program Introduction:**

As noted above, firstly we implement the quick sort to realize the sorting of value of A and get element sequence. This sequence is very important and will be the foundation of dynamic programming (Lis Algorithm) and call it sequence Y.

At start we initialize the array X with the value B of first element of Y. Then, combined with dynamic programming, we traverse every element in the sequence Y. If the latter element is bigger than the former one, then len will plus one and value B will assign to X[i]. If not, the program use **binary search** to find the position where the value B of latter element should be placed in the array X. And then replace X[pos] with the new value. Therefore, we can conclude that the value of len will always keep the maximum size of subsequence when traversal is finished. Besides that, if we record the every assignment process in the array X, we can trace back to get the longest increasing subsequence. Then, what the “Burger” problem requires, the longest length and the output of longest increasing subsequence are all reached.

Note: In order to support the various attributes of an element, we create a class “Burger” which include value A, value B and id which is used to trace back to find the real LIS array.

**Program results:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Burger length(n)** | **Output(correctness)** | **Average time cost** | **nlogn** |
| **5** | **2(√)** | **6** | **11.6** |
| **10** | **4(√)** | **15** | **33.2** |
| **20** | **7(√)** | **35** | **86.4** |
| **40** | **10(√)** | **69** | **212.8** |
| **100** | **19(√)** | **180** | **620** |

Note: The average time cost does not have units since every element’s time cost is counted by a counter in the For loop or While loop.

**Time complexity:**

Above all, there are two mainly function in the solution of “Burger” problem, sorting and Lis algorithm with binary search. As is known to all, array sorting can be implemented in at least O(nlogn) time. In this program we have chosen to implement quick sort algorithm which has least time cost. Apart from this, we have found in our textbook that binary search can reduce the time cost to O(nlogn) time. Thus the combination of these two parts of function will also cost O(nlogn) time. And in fact, in the previous chart we can substantiate that time cost is in proportion to the function nLogn, which n is the length of elements.

