COMP 257
Final Project
Tianfeng Chen
github.com/Tianfeng-Chen/COMP257-Final-Project

Final Project

1. Topic of the project:

Find the length of the longest increasing subsequence of an integer array. The input is an array full of integers, and the output is the length of the longest increasing subsequence. This question can be expanded to find the length of the longest decreasing subsequence or find any of the longest increasing subsequences. This project will be focusing on finding the length of the longest increasing subsequence. Here is an example to explain:

The input array is [2,1,3,0,6,100,18]

The output is 4.

The longest increasing subsequence can be: [2,3,6,100], [2,3,6,18], [1,3,6,100], or [1,3,6,18]

2. Different solutions for this topic:

This project will discuss three types of approaches to solving this problem: brute force, dynamic programming, and greedy approach.

2.1 Brute force solution:

The brute force solution will use a backtracking strategy to generate all the increasing subsequences and find the length of the longest one.

Input: integer array `nums[]`

Output: length of longest increasing subsequence in integer `int result`

There are two methods involved: `getlengthOfLIS()` and `increasingSubHelper()`

The `getLengthOfLIS()` is the main method used to get the result, and the `increasingSubHelper()` is a helper method to generate all increasing subsequences.

```
int getLengthOfLIS(int[] nums) {
      List<List<>> subs
      increasingSubHelper(nums, 0, subs, new List)
      int result = 0
      for List sub : subs
            result = Math.max(sub.size, result)
      return result

}
```

```
void increasingSubHelper(int[] nums, int index, List<List<>>
subs, List<> inner) {
      if index >= nums.length
            subs.add(inner)
            return
      int cur = nums[index]
      if inner.size == 0 || cur > inner.get(inner.size – 1)
            inner.add(cur)
            increasingSubHelper(nums, index + 1, res, inner)
            if inner.size > 0
                  inner.remove(inner.size – 1)
      increasingSubHelper(nums, index + 1, res, inner)
}
```

The time complexity for the brute-force solution is O(2^n).

In the worst case, the input array has already been sorted in increasing order. In this case, the runtime to generate all increasing subsequences will be O(2^n) since every element in the array will be considered choose or not choose. There will be 2^n different combinations of subsequences. To get the length of the longest increasing subsequences will be O(m), and the m is the number of increasing subsequences, which is 2^n. In this case, the total runtime for this algorithm is O(2^n)

2.2 Dynamic programming solution:

Instead of generating all the increasing subsequences and finding the longest one by one, the idea of dynamic programming is to keep tracking the best solution so far, and for each of the new incoming elements, try to match it with one of the previous best solutions and find out the longest combination as the current best solution. This idea also used the feature of the subsequence: the relationship of positions of all elements in a subsequence is the same as they are in the original array.

Input: integer array `nums[]`

Output: length of longest increasing subsequence in integer `int result`

```
int getLengthOfLIS(int[] nums) {
      int length = nums.length
      int[] dp = new int[len]
      int result = 0
      for int i = 0; i < len; i++
            dp[i] = 1
            int curMax = 0
            for int j = 0; j < I; j++
                  if nums[i] > nums[j]
                        curMax = Math.max(curMax, dp[j] + 1)
            dp[i] = Math.max(dp[i], curMax)
            result = Math.max(dp[i], result)
      return result
}
```

The time complexity of dynamic programming for this topic is O(n^2).

There are two for-loops involved in this program. The outer one loops n times and the inner one loops at most n – 1 time. The steps inside both loops only take O(1) time. In this case, the total runtime will be O(n^2).

2.3 Greedy solution

As we use the dynamic programming solution, we can see that the hardest part is we do not know if a new element will help us to generate the longest increasing subsequence, and we need to try it out one by one. In this case, the greedy solution is trying to determine if a new element is worth keeping and can lead to the correct result. The strategy of the greedy is:

- Maintain a list to generate the potential answer, the element in the list has the increasing order
- For each new element, find the first element greater than it and replace this element
- If the new element is greater than all of the elements, then append it to the end

The idea is if we can find an element that is smaller than one of the elements already in the list, we can say the new element has greater potential to have the longest increasing subsequence. The only problem the sequence in the list is not always the valid subsequence, but the length of the list remains correct. The reason for this is the length only changes when the new element is greater than all of the elements inside the list, otherwise, we only replace the element inside the list. To find the first greater element, we can use the help of binary search.

Input: integer array `nums[]`

Output: length of longest increasing subsequence in integer `int result`

There are two methods involved: `getLengthOfLIS()` and `bsHelper()`

The `getLengthOfLIS()` is the main method to apply the greedy approach and return the result, and the `bsHelper()` is the helper method to use the binary search method to find the location of the first greater element in the list.

```
int getLengthOfLIS(int nums[]) {
      int result = 0
      List list = new List
      for int num : nums
            int index = bsHelper(list, num)
            if index == list.size
                  list.add(num)
            else
                  list.set(index, num)
      result = list.size
      return result
}
int bsHelper(List list, int num) {
      int left = 0
      int right = list.size – 1
      while left >= right
```

```
            int mid = left + (right – left) / 2
            if list.get(mid) == num
                  return mid
            if list.get(mid) < num
                  left = mid + 1
            else
                  right = mid  - 1
      return left
}
```

The time complexity for the greedy approach is O(nlogn).

There is a for loop inside the `getLengthOfLIS()` method and this for-loop goes n times. Inside the for loop, the search range for the binary search helper will be at most n – 1. Other operations in that for-loop is O(1). In this case, the total runtime will be O(nlogn).

3. Code Tests

Five test cases with various sizes have been tested with these three algorithms. Each test case is an integer array with arbitrary numbers that are generated with random number generator.

TestCase1: [-47, -78, -57, -59, -60]

TestCase2: [-38, 11, 4, -53, 44, 62, 57, 45, -82, 89]

TestCase3: [50, 31, 95, 14, 40, 58, -77, 37, -78, 8, -40, 1, -83, -1, -60]

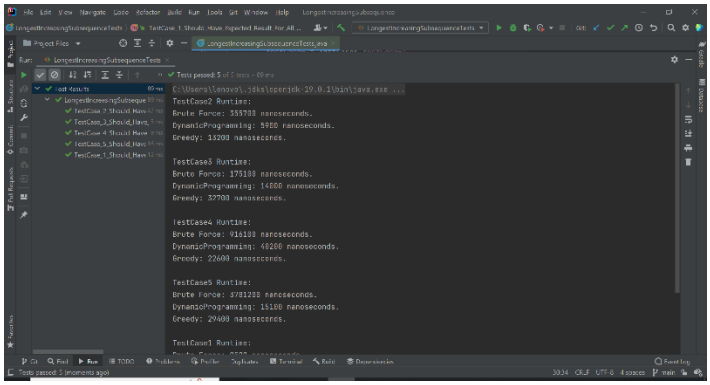
TestCase4: [68, -94, -90, 39, 67, 24, 20, -43, 2, -6, 0, 55, -38, 41, -41, 52, 82, 88, 34, -60]

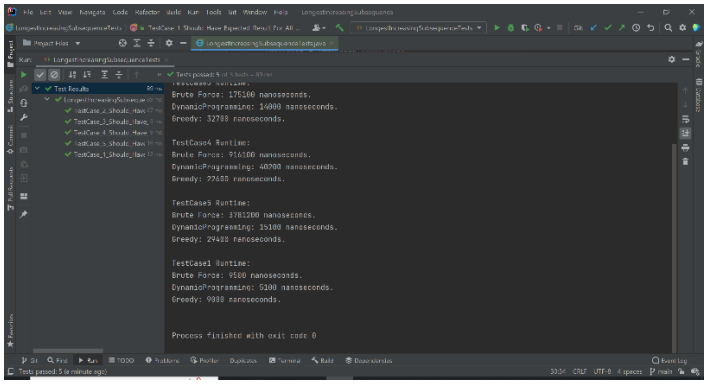
TestCase5: [-61, -75, 46, -67, -60, -25, 57, -15, -51, -11, -71, 22, -8, 0, -28, -6, 56, -12, -76, 9, -23, -32, 75, 37, -95]

The range of all test cases is between -100 and 100, and the size of test cases is increasing by 5. Other test cases with larger sizes can be found in the repo.

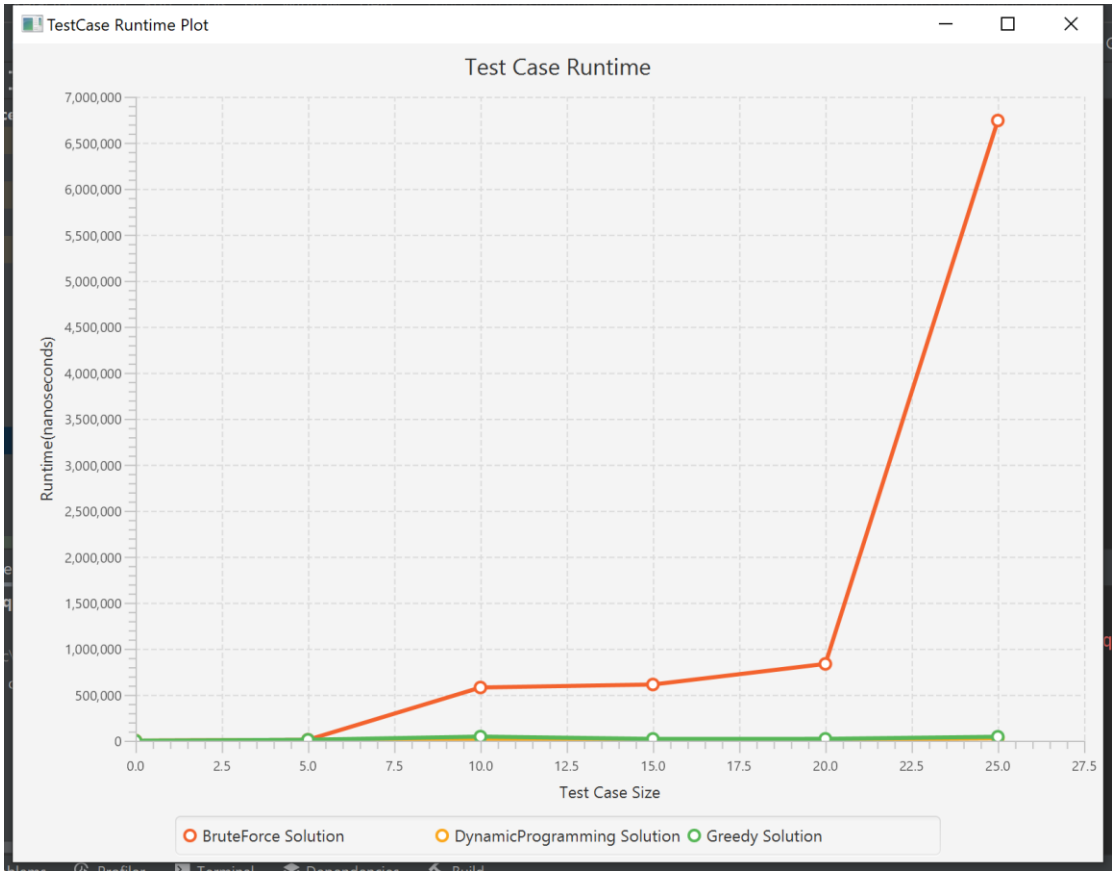
The size of the test cases is not very big. I tried test cases up to the size of 100, the brute force will take too much time to run, and the IDE throws an error for it. The big test cases have the same situation in the runtime plot.

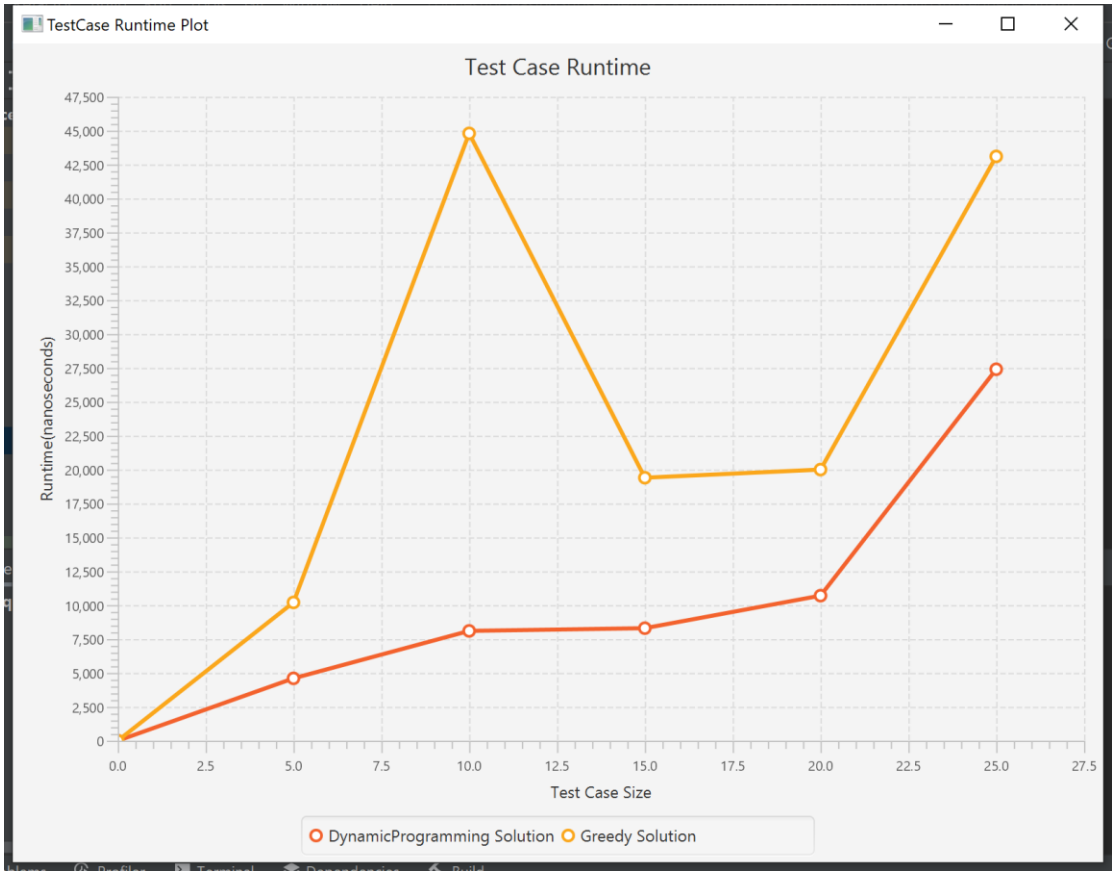
3.1 Test Result

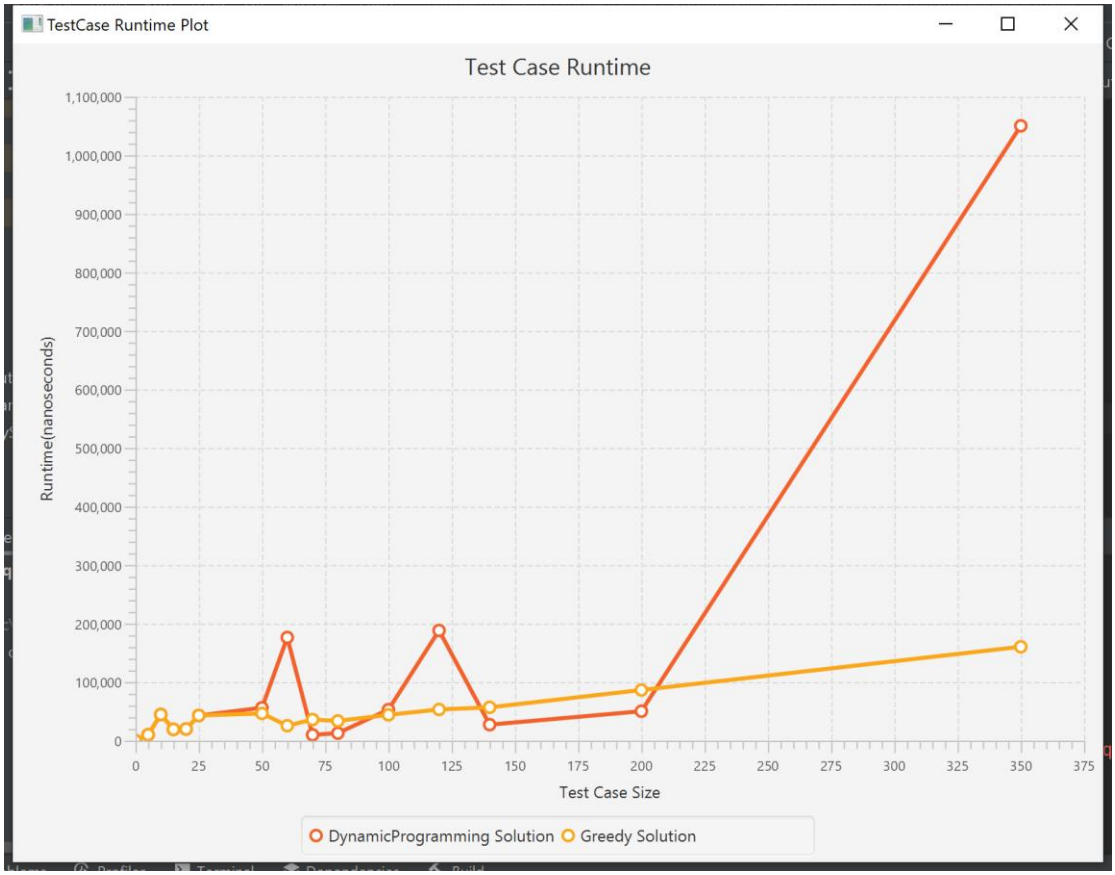




3.3 Result Plot







The second plot only shows the runtime difference between dynamic programming solution and greedy. Due to the brute force solution having a much bigger increment

than the other two, it is very hard to track the change for the dynamic programming and greedy in the first plot.

The third plot runs more test cases with a larger size for only dynamic programming and greedy solutions.

4. Conclusion

From the plot and the algorithm analysis before, we can find out that brute force has a much higher cost to solve this problem. Dynamic programming (DP) and the greedy solution will be a much better choice than brute force. Except for the spike in the greedy solution, we can see in the plot, the greedy and the DP have very similar increment patterns and DP is slightly faster than the greedy (in the second plot). However, based on the analysis, the greedy should run faster than DP (O(nlogn) versus O(n^2)). The reason DP seems faster than greedy is that the size of the test case is not big enough. Based on the result of plot 3, we can see that with the increase in the size of test cases, the DP solution has a higher cost than greedy. The space complexity for all of these three solutions will be O(n) since they all require extra space to store the intermedia states. In conclusion, the greedy solution will be the best solution to solve this problem.