Week 10: Lab

Module 4: Techniques

COLLABORATION LEVEL 0 (NO RESTRICTIONS). OPEN NOTES.

Longest increasing subsequence:[1] A *subsequence* of a sequence (for example, an array, linked list or string), is obtained by removing zero or more elements and keeping the rest in the same order. A subsequence is called a *substring* if its elements are contiguous in the original sequence. For example:

1. GORIT, ALGO, RITHMS are all substrings of ALGORITHMS
2. GRM, LORI, LOT, AGIS,GORIMS are all subsequences of ALGORITHMS
3. ALGOMI, OG are *not* subsequences of ALGORITHMS

The problem: Given an array $A[1..n]$ of integers, compute the length of a *longest increasing subsequence* (A sequence $B[1..k]$ is *increasing* if $B[i] < B[i+1]$ for all $i = 1..k-1$). For example, given the array

$$\{5, 3, 6, 2, 1, 5, 3, 1, 2, 5, 1, 7, 2, 8\}$$

your algorithm should return 5 (for e.g. correponding to the subsequence $\{1, 3, 5, 7, 8\}$; there are other subsequences of length 5).

Describe an algorithm which, given an array $A[]$ of n integers, computes the LIS of A.

```
EXAMPLES:

Input: arr[] = {3, 10, 2, 1, 20}
Output: Length of LIS = 3
The longest increasing subsequence is 3, 10, 20

Input: arr[] = {3, 2}
Output: Length of LIS = 1
The longest increasing subsequences are {3} and {2}

Input: arr[] = {50, 3, 10, 7, 40, 80}
Output: Length of LIS = 4
The longest increasing subsequence is {3, 7, 40, 80}

Input: A[]={0,8,4,12,2,10,6,14,1,9,5, 13,3,11,7,15}
Output: Length of LIS = 6
Explanation:Longest increasing subsequence 0 2 6 9 13 15, which has length 6

Input: A[] = {5,8,3,7,9,1}
Output: Length of LIS = 3
Explanation:Longest increasing subsequence 5 7 9, with length 3
```

[1] Leetcode #300

Additional/optional problems

1. **Matching points on a line:**[2] You are given two arrays of n points in one dimension: red points $r_1, r_2, ..., r_n$ and blue points $b_1, b_2, ..., b_n$. You may assume that all red points are distinct and all blue points are distinct. We want to pair up red and blue points, so that each red point is associated uniquely with a blue point and vice versa. Given a pairing, we assign it a score which is the sum of distances between each pair of matched points.

   Find the pairing (matching) of minimum score. Hint: Aim for an $O(n \lg n)$ algorithm. Draw examples for small valus of n to get intuition.

   Example: Consider the input where the red points are $r_1 = 8, r_2 = 1$ and the blue points are $b_1 = 3$ and $b_2 = 9$. There are two possible matchings:

   - match r_1 to b_1 and r_2 to b_2: the score of this matching is $|r_1 - b_1| + |r_2 - b_2| = |8 - 3| + |1 - 9| = 13$
   - match r_1 to b_2 and r_2 to b_1: the score of this matching is $|r_1 - b_2| + |r_2 - b_1| = |8 - 9| + |1 - 3| = 3$

   The algorithm shoud return the cost of the optimal matching, which is 3.

[2] Credit: reproduced from Stanford University, cs161, fall 2016-2017

Appendix A: Hints

- **Longest increasing subsequence:**

  Use the following subproblem: Let $L(i)$ be the length of a LIS starting with $A[i]$.

  Express $L(i)$ rercursively.

  Assume you computed $L(i)$ for all $i = 1, 2, ..., n$. How do you find the LIS of A?

- **Matching points on a line:**

  Think greedily, and argue with an exchange argument.

Appendix B: Solutions

1. **Finding the LIS (Longest Increasing Subsequence):**

   - Notation and choice of subproblem: Let $L(i)$ be the length of a LIS starting with $A[i]$.
   - To find the LIS(A), we need to return $\max\{L(i), 1 \leq i \leq n\}$ The reasoning is that we don't know where LIS(A) of A starts, but it must start at some index i in A. Then $LIS(A)$ will be equal to $L(i)$ for that index i.
   - Optimal substructure of $L(i)$: Consider a LIS starting with $A[i]$, and denote it $LIS(i)$. Let $A[j]$, with $j > i$, be the element after $A[i]$ in $LIS(i)$. Then $LIS(i)$ must consist of $A[j] + LIS(j)$. [insert here the usual justification by contradiction]. This leads us to a recursive definition of $L(i)$.
   - Recursive definition of $L(i)$:
     - Base case: if $i == n$ then there is no element after $A[n]$ so $L(n) = 1$
     - Otherwise if $i < n$:

     $$L(i) = 1+\max\{L(j)\ \text{, where } i+1 < j \leq n \text{ and } A[i] < A[j]\},\ \text{or } L(i) = 1 \text{ if no such } j \text{ exists}$$

     Explanation: We know that $L(i)$ includes $A[i]$. The second element in the $L(i)$ must be an index j such that $j > i$ and $A[j] > A[i]$. By optimal substructure, it must be that the subsequence starting at j is $L(j)$ (if this was not the case, ...[insert here the usual justification by contradiction]). Since we do not know what j is, we try all indices j and pick the largest.
   - Running time of $L(i)$: Without DP, exponential.
   - Computing $L(i)$ with dynamic programming:
     We use a table $table[1..n]$ such that table[i] will store the solution returned by $L(i)$.
     Initialize: set $table[n] = 1$, and $table[i] = 0$ for $i = 1, ..., n - 1$.

```
L(i)
  //if this was computed before, retrieve it from the table
  if table[i] !=0: return table[i]

  //otherwise, compute it and store it in the table
  max = 0
  for j=i+1 to n do:
     if A[j] > A[i]:
           thisj =  1 + L(j)
           if thisj > max:  max = thisj
  //max is the solution to L(i)
  table[i] = max
  return max
```

   - To find the length of a LIS in A we initialize the table as above, and then call $L(i)$ for $i = 1, 2, ..., n$ (note that it is not sufficient to call just L(1) because it will only recurse on L(j) whith $A[j] < A[i]$, so it won't fill the whole table). Then we traverse the table and find that largest value of $L(i)$. That is the $L(A)$. The logic is that LIS(A) must start at some index i, and $LIS(A) = L(i)$.
   - Analysis: The table has size n, and each $table[i]$ takes $O(n)$ to compute ignoring the cost of the recursion. Because of the table, each $L(i)$ is computed precisely once, and the total cost of the recursion is $\Theta(n)$. So overall $\Theta(n) + \Theta(n) \cdot O(n) = \Theta(n^2)$.

2. **Matching points on a line:**

   Algorithm: Sort the red and blue points separately, and match them in order, that is, r_1 with b_1, r_2 with b_2, and so on (here I assumed that they are numbered in sorted order). This is a greedy algorithm.

   Analysis: $O(n \lg n)$ to sort plus $O(n)$ to match.

   Correctness: The hard part is to prove that this is always correct. To do so, it is sufficient to show that there is an optimal solution that contains the first greedy choice, namely matching r_1 to b_1.

   Proof: Let O be an optimal matching O and assume that it does not match r_1 to b_1. Suppose that r_1 is matched with b_i and b_1 is matched with r_j. We modify O so that r_1 is matched with b_1 and r_j is matched with b_i. Denote by O' the resulting matching. We'll show that the cost of O' is $\leq O$, therefore O' must be an optimal matching. Therefore there exists an optimal matching that matches r_1 to b_1.

   The cost of O' is $r_1 - b_1| + |r_j - b_i| + ...$

   The cost of O is $r_1 - b_i| + |r_j - b_1| + ...$

   To show that the cost of O' is $\leq O$, we need to show that $|r_1 - b_1| + |r_j - b_i| \leq r_1 - b_i| + |r_j - b_1|$

   Assume wlog that $r_1 < b_1$. Then

   - case 1: $r_1 < r_j \leq b_1 < b_i$ ...
   - case 2: $r_1 < b_1 \leq r_j < b_i$ ...
   - case 3: $r_1 < b_1 < b_i \leq r_j$ ...

   In each case, draw the points in order to visualize, and the rest will follow.