# Week 10: Lab

Module 4: Techniques

COLLABORATION LEVEL 0 (NO RESTRICTIONS). OPEN NOTES.

1. **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:

   (a) GORIT, ALGO, RITHMS are all substrings of ALGORITHMS

   (b) GRM, LORI, LOT, AGIS,GORIMS are all subsequences of ALGORITHMS

   (c) 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 algorihm 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
```

[1] Leetcode #300

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

2. **Unbounded knapsack:** We have $n$ items, each with a value and a positive weight; Item $i$ has value $v_i$ and weight $w_i$. We have a knapsack that holds maximum weight $W$. The problem is which items to put in the backpak to maximize its value.

   In this problem you'll consider a variation of the $0-1$ Knapsack problem, where we have *infinite copies* of each item. Describe an algorithm that, given $W, \{w_1, ..., w_n\}$ and $\{v_1, ..., v_n\}$, finds the maximal value that can be loaded into the knapsack.

   The sequence of steps below will guide you towards the solution

   (a) A friend proposes the following greedy algorithm: start with the item with the largest cost-per-pound, and pick as many copies as fit in the backpack. Repeat.

   Show your friend this is not correct by coming up with a counter-example.

   (b) Now let's define a useful sub-problem. With the classical knapsack, once you take an item, you cannot take it again, so you have fewer items to choose from. So the subproblem we use has to keep track of both the knapsack size as well as which items are allowed in the knapsack. We used $knapsack(w, i)$ to be the maximum value we can get for a knapsack of weight $w$ considering items 1 through $i$.

   For the unbounded knapsack, we have infinite supplies of each item. So we don't need to keep track of which items we already included, only of the size of the knapsack. Let $K(w)$ represent the optimal value for a knapsack of capacity $w$.

   Argue optimal sub-structure:

   - Suppose $K(w)$ contains item $i$.
   - Then the remainig items in $K(w)$ must be an optimal solution for .....

   (c) Come up with a recursive definition of $K(w)$. (Hint: Consider all the choices, and pick the best. )

(d) Develop a DP solution based on this recurrence — either top-down or bottom up. What is the running time of your algorithm ?