1 Longest Increasing Subsequence

We start with one more example of a dynamic programming problem: the longest increasing subsequence (LIS) problem: given a sequence

\[ A[1] \ldots A[n], \]

find the longest subsequence, \( i_1 < i_2 \ldots i_k \) such that

\[ A[i_j] < A[i_{j+1}] \quad \forall 1 \leq j < k. \]

A search program simply tries all next elements that start somewhere.

\[
\text{LIS}(i) \{ \ \backslash \text{returns longest increasing subsequence starting at } i \\
\text{result} = 1; \\
\text{FOR } j = i + 1 \ldots n \\
\quad \text{if } (A[i] < A[j]) \\
\quad \quad \text{result} = \max(\text{result}, \text{LIS}(j) + 1); \\
\text{RETURN result;}
\}
\]
Here the $LIS(j) + 1$ takes into account that the sequence also uses $A[i]$. Calling this function on all starting locations and taking max then gives the result.

Note once again that the result of $LIS(i)$ is dependent only on $i$: only the value of $A[i]$, rather than the choices of entries before, affects the length of the rest of the sequence. So we can create a dynamic program with:

1. State: $LIS[i]$: length of longest increasing subsequence starting at $i$.
2. Base case: $LIS[i] \geq 1$.
3. Transition:
   \[
   LIS[i] = 1 + \left( \max_{j > i, A[i] < A[j]} LIS[j] \right).
   \]

Finally, this transition can be evaluated by ordering the states backwards: we first compute $LIS[n]$, then $LIS[n-1]$, and etc. Each of these takes $O(n)$ to compute, giving a total running time of $O(n^2)$. Also, the space usage of this algorithm is $O(1)$ per state (for the 1 number stored), for a total of $O(n)$.

As an example, on the input sequence

\[
8, 3, 9, 4, 6,
\]

the $LIS$ array at the end gives:

\[
2, 3, 1, 2, 1.
\]

## 2 The Graph Theoretic Connection

The text book talks about longest increasing subsequences through the graph theoretic connection instead.

For dynamic programs where each state goes to one other state, the states are equivalent to vertices of a graph, while the transitions can be viewed as edges.

For example, in the above instance of the longest increasing subsequence problem, the states are just the indicies of the array, $A[1] \ldots A[5]$, and we have edges


Then the goal of finding the longest increasing subsequence is equivalent to finding the longest path in this graph.

In general, finding the longest path in a graph is very hard (we’ll talk about this later on when we discuss NP-hardness). However, with the longest increasing subsequence problem, we can arrange the vertices so that the edges only go from left to right. Such
a graph is known as a **directed acyclic graph**. This ordering is akin to what we get in dynamic programming, so we can also solve this problem once again by dynamic programming:

Assuming that the vertices are ordered $1 \ldots n$ such that all edges $i \to j$ satisfy $i < j$.

1. **State:** $Longest[i]$: length of the longest path ending at $i$.
2. **Base case:** $Longest[i] \geq 0$.
3. **Transition:**

   $$Longest[j] = \max_{i \to j} \left( Longest[i] + \text{length}(i \to j) \right).$$

4. **Ordering this is crucial here:** proceed through the vertices in increasing order.

   This dynamic program once again uses $O(n)$ space: one per vertex. Its running time can be bounded by $O(n^2)$: it checks each $j$ at most once per $i$, and the transition takes $O(1)$ to evaluate.

   On a graph that’s sparser, it can run even faster. Note that it only checks each edge $i \to j$ once at its ‘right’ endpoint $j$. Formally if the number of edges is $m$, the total cost of this algorithm can be bounded by $O(m)$. We will come across this again when discussing shortest paths next Monday.

### 3 Test 1 Review

- **Main topics covered**
  - Asymptotic complexity: $O$, $\Omega$, and $\Theta$.
  - Designing divide-and-conquer algorithms.
  - Setting up runtime recurrences.
  - Solving recurrences using Master theorem (other methods are optional).
  - Faster multiplication.

- **NOT included:**
  - Recursion trees that are not covered by master theorem.
  - Other methods for solving runtime recurrences such as guess-and-check.

- **Proving big-$O$ bounds** (Homework 1, Problem 1. Ex 0.1 in Textbook):
  - If $f_1 = O(g_1(n))$, $f_2 = O(g_2(n))$, then
    - $f_1 f_2 = O(g_1 g_2)$.
    - $f_1 + f_2 = O(\max\{g_1, g_2\})$. 
For any constants $a$, $b$, and $c > 1$, $O(\log^a(n)) \leq O(n^b) \leq O(c^n)$, 

$\ln(n) = \Theta(\log n) = \Theta(\log_2 n) = \Theta(\log_e n)$.

- Divide-and-conquer and setting up running time recurrences (Homework 1, Problems 2 and 3. Ex 2.12, 2.16, 2.17, 2.23 in textbook)

  - General structure of a recursive algorithm:
    * Split the problem up.
    * Make $a$ recursive calls to problems of size $n/b$
    * Combine the results.
  - $T(n)$: running time when given input of size $n$.
  - If total cost of split/combine is $O(n^d)$, runtime recurrence is:
    \[
    T(n) = aT(n/b) + O(n^d).
    \]

- Master Theorem:

  The recurrence:
  \[
  T(n) = aT\left(\frac{n}{b}\right) + c\cdot n^d, \quad T(1) = e
  \]
  where $a > 0$, $b > 1$, $c > 0$, $d \geq 0$ and $e \geq 0$ are constants, has the solution given below:

  **Case 1:** If $d = \log_b a$ then $T(n) = O(n^d \log n)$.
  **Case 2:** If $d > \log_b a$ then $T(n) = O(n^d)$.
  **Case 3:** If $d < \log_b a$ then $T(n) = O(n^{\log_b a})$.

- Example of using Master theorem to analyze recursion (Homework 1 Problems 2ac, 3. Ex 2.4, 2.5abcde in textbook):

- Fast multiplication of 2 $n$-digit numbers using 3 multiplies of $n/2$-digit numbers plus $O(n)$ overhead:

  - Runtime recurrence: $T(n) = 3T(n/2) + O(n)$.
  - Fits into the Master theorem with $a = 3$, $b = 2$, $d = 1$.
  - $d < \log_2 3$, total runtime $O(n^{\log_2 3}) = O(n^{1.59}) < O(n^{1.6})$. 

4