CS 341 — Lecture 10

Bartosz Antczak *Instructor: Bin Ma* *October 12, 2017*

10.1 Even More on Dynamic Programming

Today we finish the topic of dynamic programming. Recall that a primary characteristic of **dynamic programming** is that the solution to a problem depends on the solution to a smaller instance of the same problem.

10.1.1 Longest Increasing Subsequence

Given an array A, return the longest increasing sequence (LIS).

- **Input:** an array consisting of elements $a_1, a_2, \ldots, a_n$
- **Output:** a subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ such that $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ and $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$

For instance, given the array $[5, 1, 9, 8, 8, 8, 4, 5, 6, 7]$, our LIS is $1, 4, 5, 6, 7$.

Define our Subproblems

Let $D[i]$ be the length of the LIS of the first i numbers. We assume $D[i]$ to be defined as:

$$D[i] = \begin{cases} D[i-1] & \text{If } a_i \text{ is not used} \\ 1 + D[i-1] & \text{Otherwise} \end{cases}$$

Note that this solution will **not** work! It's fairly trivial to see why. So let's define $D[i]$ differently.
$D :=$ the length of the LIS with the last number being a_i:

$$D[i] = 1 + \left(\max_{1 \leq j \leq i-1 \text{ and } a_j < a_i} D[j] \right)$$

The Algorithm

The algorithm is trivial to write. Observe that it has a run time of $O(n^2)$. The output is $\max_{1 \leq i \leq n} D[i]$. An example implementation of D is shown:

i	1	2	3	4	5	6	7
a_i	1	3	4	2	5	7	6
$D[i]$	1	2	3	2	4	5	5

In this case, we would output 5 as the length of the LIS.

10.1.2 Optimal Binary Search Tree

Given a list of n comparable items (such as strings or numbers) such that $k_1 < k_2 < \cdots k_n$, we can construct a wide variety of BST structures (i.e., we can build multiple different instances of a BST for this list). Our problem with this is to find the most optimal instance of a BST for our list:

- **Input:** n keys: $k_1 < k_2 < \ldots k_n$ with frequency $f_1, f_2, \ldots, f_n$
- **Output:** a binary tree minimizing the score, where the score is defined as $\sum_{i=1}^{n} f_i \cdot \text{depth}(k_i)$

Define our Subproblems

Define $D[i, j]$ to be the score of the optimal BST if $k_i < \cdots k_j$ and $f_i, \ldots, f_j$:

$$D[i,j] = f_\ell + \left(D[i, \ell - 1] + \sum_{r=i}^{\ell-1} f_r \right) + \left(D[\ell + 1, j] + \sum_{r=\ell+1}^{j} f_r \right)$$

Since we need to try every $i \leq \ell \leq j$, we get the following recurrence relation:

$$D[i,j] = \min_{i \leq \ell \leq j} \left(D[i, \ell - 1] + D[\ell + 1, j] + \sum_{r=i}^{j} f_r \right)$$

The reasoning behind this relation is such that to compute $D[i, j]$ for $i < j$, we need to try every possible root $i \leq \ell \leq j$ where $k_i, \ldots, k_{\ell-1}$ is the left root and $k_{\ell+1}, \ldots, k_j$ is the right root.

The Algorithm

The algorithm is trivial to write. For every i, j, solve $\min_{i \leq \ell \leq j} \left(D[i, \ell - 1] + D[\ell + 1, j] + \sum_{r=i}^{j} f_r \right)$, which results in a run time of $O(n^3)$. At the end, output $D[1, n]$.

10.1.3 Maximum Sub-array

- **Input:** array of numbers A of size n
- **Output:** $\max_{i \leq j} \sum_{k=i}^{j} A[k]$

Let's apply other algorithm techniques that we've learned to this question.

Algorithm I - Trivial

For every i, j, compute $\sum_{k=i}^{j} A[k]$. Has a run time of $O(n^3)$.

Algorithm II - Clever trivial

Observe that we don't have to calculate the sum every time. We can simply compute $\sum_{k=i}^{j} A[k] = \left(\sum_{k=i}^{j-1} A[k]\right) + A[j]$. So for every check, we're only adding one more number to the previous j's result. Has a run time of $O(n^2)$

Algorithm III - Divide and Conquer

Split array into two halves. Determine the maximum suffix for left array and the maximum prefix for the right array. Merge the two together and determine the maximum sum out of the three. Has a run time of $T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$. By Master Theorem, that is a run time of $O(n \log n)$.

Algorithm IV - Greedy

```
maxsm ← 0
sum ← 0
for i = 1 to n do
  sum ← sum + A[i]
  if sum > maxsum then
    maxsum ← sum
  else if sum ≤ 0 then
    sum ← 0
```

This algorithm has a run time of $O(n)$. Let's (roughly) prove that it's correct:
Assume that the range ℓ, r holds the maximum sum. We consider two cases:

- If $\ell, r+1$ or $\ell - 1, r$ contains a larger sum, then that contradicts our assumption.
- If the sum of every subarray is negative, then the max sum is by default 0

Observe that our algorithm checks both of these cases, and thus it is correct.

Algorithm V - Dynamic Programming

Similar to greedy. Let $D[i]$ = the max suffix sum of $A[1, \ldots, i]$:

$$D[i] = \max \begin{cases} 0 \\ D[i-1] + A[i] \end{cases}$$

Our algorithm is:

```
for i = 1 to n do
  D[i] = max(0, D[i − 1] + A[i])
```