Section 6: Solutions

1. DP on Trees

You are given a tree $T = (V, E)$ with nonnegative edge weights. You want to determine the diameter of the tree, which is the longest distance between any two nodes. The goal is to design a DP which runs in $\mathcal{O}(n)$ where $n = |V|$.

1.1. Write the Dynamic Program

(a) Formulate the problem recursively – what are you looking for (in English!!), and what parameters will you need as you're doing the calculation? It will be useful to fix a root node in the tree first, then every node in the tree has a list of children nodes you can access.

Solution:

Let START(v) be the distance of the longest path starting and v and ending in some node within the subtree rooted at v. Let THRU(v) be the longest path which passes through v but stays inside the subtree rooted at v. Importantly, this path need not start or end at v.

(b) Write a recurrence for solving the problem you defined in the last part (the recurrence is for the answer, **not** the running time).

Solution:

$$\text{START}(v) = \begin{cases} 0 & \text{if } v \text{ has no children} \\ \max\limits_{c \text{ is a child of } v} \{w_{(c,v)} + \text{START}(c)\} & \text{otherwise} \end{cases}$$

$$\text{THRU}(v) = \begin{cases} 0 & \text{if } v \text{ has less than two children} \\ \max\limits_{\substack{c_1, c_2 \text{ distinct} \\ \text{children of } v}} \{w_{(c_1,v)} + \text{START}(c_1) + w_{(c_2,v)} + \text{START}(c_2)\} & \text{otherwise} \end{cases}$$

(c) What is your final answer (e.g. what parameters for the recurrence do you need? Is it a single value or the max/min of a set of values?)?

Solution:

$\max_v\{\text{START}(v), \text{THRU}(v)\}$.

(d) Give a brief justification for why your recurrence is correct. You do not need a formal inductive proof, but your intuition will likely resemble one.

Solution:

The longest path that starts at a node v and ends in a node in a subtree rooted at v has two cases. Either this longest path is empty since v has no children, or the longest path goes through some child of v. That path must consist of the edge from v to its child plus the longest path starting at the child and staying within the subtree rooted at the child.
Similarly, for the longest path that passes through a node v, if it has less than two children then there is

no such path. Otherwise we know that the path must go through two of v's children. Hence the longest path is the sum of the two edges that go to v's children plus the longest paths that start at each children respectively and stays within the subtrees rooted at each child.
Both recurrences encode only distance of paths. Also the longest path in the tree will have a unique node which is closest to the root and that path can either start at that node or pass through it, thus maxing over all vertices in both START and THRU is sufficient for finding this distance.

1.2. Analyze the Dynamic Program

(a) Describe a memoization structure for your algorithm.

Solution:

We need two lists both of size n.

(b) Describe a filling order for your memoization structure.

Solution:

We fill from leaves up to the root, making sure that we have evaluated on all children node before evaluating a parent node.

(c) State and justify the running time of an iterative solution.

Solution:

For each node, we need to only access the weight of the edges between the node and its children. For START this means we in total only need $|E|$ many calls total. For THRU we can implement the max over two children by just finding the largest and second largest child, so again $|E|$ many calls in total. Thus our runtime is $\mathcal{O}(|E|) = \mathcal{O}(n)$.

2. Longest Increasing Subsequence AGAIN

We've already seen a recurrence for Longest Increasing Subsequence. Let's write another!

As before, $[10, -2, 5, 0, 3, 11, 8]$ has a longest increasing subsequence of 4 elements: $[-2, 0, 3, 8]$

2.1. Write the Dynamic Program

(a) Formulate the problem recursively – what are you looking for (in English!!), and what parameters will you need as you're doing the calculation? To make sure you get a different solution than the one from class, you should ask yourself to answer the question "what's the longest increasing subsequence where the first included element is the one at index i, and how would I find that?"

Solution:

Let (i) be the length of the longest increasing subsequence of $A[]$ where element i is the first element of the subsequence.

(b) Write a recurrence for solving the problem you defined in the last part (the recurrence is for the answer, **not** the running time).

Solution:

$$(i) = \begin{cases} 1 & \text{if } i = n \\ 1 + \max\limits_{j>i}\{\mathbb{I}[A[i] < A[j]] \cdot (j)\} & \text{otherwise} \end{cases}$$

Where $\mathbb{I}$ is the indicator function of that event, meaning $\mathbb{I}[A[i] < A[j]] = 1$ if $A[i] < A[j]$ and zero otherwise.

(c) What is your final answer (e.g. what parameters for the recurrence do you need? Is it a single value or the max/min of a set of values?)?

Solution:

$\max_i(i)$.

(d) Give a brief justification for why your recurrence is correct. You do not need a formal inductive proof, but your intuition will likely resemble one.

Solution:

For the base case, since n is the farthest right element it is the only element in a subsequence starting from that location.
If we begin at element i, then either it is the only element or there is an element after. The recurrence checks all elements after – if they are the second element in that sequence, they must be after i, have the element be greater than $A[i]$. That new location j will then start the rest of the increasing subsequence, so making all those recursive calls suffices to find the best one.

2.2. Analyze the Dynamic Program

(a) Describe a memoization structure for your algorithm.

Solution:

We need a (1D) array of size n.

(b) Describe a filling order for your memoization structure.

Solution:

We fill from n down to 1.

(c) State and justify the running time of an iterative solution.

Solution:

Creating entry i requires checking $i - 1$ recursive calls. Since we have n entries, we need $\mathcal{O}\left(n^2\right)$ time.

3. Longest Common Subsequence

Given two strings s with length m and t with length n, find the length of their longest common subsequence. A subsequence is a (possibly non-contiguous) substring.

Examples:

Input s="backs", t="arches": the longest common subsequence is "acs", so the output should be 3.

Input s="skaters", t="hated": the longest common subsequence is "ate", so the output should be 3.

3.1. Write the Dynamic Program

(a) Formulate the problem recursively – what are you looking for (in English!!), and what parameters will you need as you're doing the calculation?

Solution:

Let (i, j) be the longest common subsequence between elements $1..i$ in s and $1..j$ in t. The parameter i ranges from 0 to m, and the parameter j ranges from 0 to n (so we have our base cases in our memoization table for ease of calculation).

(b) Write a recurrence for solving the problem you defined in the last part (the recurrence is for the answer, **not** the running time).

Solution:

$$(i,j) = \begin{cases} 1 + (i-1, j-1) & \text{if } s_i = t_j \\ \max((i-1,j),(i,j-1)) & \text{if } s_i \neq t_j \\ 0 & \text{if } i < 1 \text{ or } j < 1 \end{cases}$$

(c) What is your final answer (e.g. what parameters for the recurrence do you need? Is it a single value or the max/min of a set of values?)?

Solution:

(n, m).

(d) Give a brief justification for why your recurrence is correct. You do not need a formal inductive proof, but your intuition will likely resemble one.

Solution:

For the first case, the character at index i in s is equal to the character at index j in t, so we want to add this character to our longest common subsequence. So, we'll add 1 to the longest common subsequence we found up until we reached these characters, which is $(i-1, j-1)$.
For the second case, the character at index i in s is not equal to the character at index j in t, so together they can't be part of the longest common subsequence. So then we'll look at a smaller chunk in either s or t, we'll compare the character at index i in s to the character at index $j-1$ in t, or we'll compare the character at index $i-1$ in s to the character at index j in t.
For the third case, either i or j is out of bounds which means we're not looking at any characters in either s or t, so we just return 0.

3.2. Analyze the Dynamic Program

(a) Describe a memoization structure for your algorithm.

Solution:

We need a (2D) array of size $m \times n$.

(b) Describe a filling order for your memoization structure.

Solution:

Outer loop i from 0 to m.
Inner loop j from 0 to n.

(c) State and justify the running time of an iterative solution.

Solution:

Creating entry i, j requires checking at most 2 recursive calls, which each require $\mathcal{O}(1)$ time. Since we have nm entries, we need $\mathcal{O}(nm)$ time.

4. k Minimum Disjoint Subarrays, Each with Target Sum t

You are given an array of positive integers $A[n]$ and an positive integer target t. You need to find k disjoint (non-overlapping) subarrays of A that each have a sum of their elements equal to the target and such that the sum of the lengths of the subarrays is minimized. If there are not k such subarrays, return ∞.

Examples:

Input $k = 4$, $t = 2$, and the array $[1, 1, 8, 2, 3, 2, 1, 1, 2]$: the four smallest disjoint subarrays whose elements each sum to 2 are $[[1, 1], 8, [2], 3, [2], 1, 1, [2]]$, so the output should be 5.

Input $k = 2$, $t = 5$, and the array $[2, 2, 2, 2]$: there aren't two subarrays whose elements each sum to 5, so the output should be ∞.

Input $k = 3$, $t = 4$, and the array $[1, 3, 1, 2, 1, 4, 2, 2]$: the three smallest disjoint subarrays whose elements each sum to 4 are $[[1, 3], 1, 2, 1, [4], [2, 2]]$ so the output should be 5.

4.1. Write the Dynamic Program

(a) Formulate the problem recursively – what are you looking for (in English!!), and what parameters will you need as you're doing the calculation?

Solution:

Let (i, j) be the minimum sum of the lengths of j disjoint subarrays whose elements each sum to t that are found in the array from $1..i$ inclusive.
The parameter i ranges from 0 to n, and the parameter j ranges from 0 to k.

(b) Write a recurrence for solving the problem you defined in the last part (the recurrence is for the answer, **not** the running time).

Solution:

$$(i,j) = \begin{cases} \min((i-1,j),(i-m,j-1)+m) & \text{if } i>0,\ j>0,\ \text{and } \exists m[\sum_{x=i-m+1}^{i} A[x] = t] \\ (i-1,j) & \text{if } i>0,\ j>0 \\ 0 & \text{if } j=0 \\ \infty & \text{otherwise} \end{cases}$$

(c) What is your final answer (e.g. what parameters for the recurrence do you need? Is it a single value or the max/min of a set of values?)?

Solution:

(n,k).

(d) Give a brief justification for why your recurrence is correct. You do not need a formal inductive proof, but your intuition will likely resemble one.

Solution:

For the first case, the element at index i can be the right-most element in a subarray of length m whose elements sum to the target t, so that subarray can be one of the j subarrays we consider. We either want to include the subarray ending with element i, in which case we add m to the minimum sum of elements in $j-1$ subarrays ending at index $i-m$, or we don't want to include the subarray ending with element i, in which case we just take the minimum sum of elements in j subarrays up to index $i-1$. To determine which is better, we take the minimum of these two options.
For the second case, there is no subarray whose elements sum to t that ends with the element at index i, so our only option is to just take the minimum sum of elements in j subarrays up to index $i-1$.
In the third case, $j=0$, which means we want the minimum length of 0 subarrays, which is necessarily 0.
Otherwise, one of our indices is out of bounds or there are less than j subarrays from $1..i$ whose elements sum to t, so we put in a placeholder of ∞ so that as soon as we have enough valid subarrays, that number of elements will be the minimum.

4.2. Analyze the Dynamic Program

(a) Describe a memoization structure for your algorithm.

Solution:

We need a (2D) array of size $n \times k$.

(b) Describe a filling order for your memoization structure.

Solution:

Outer loop i from 0 to n.
Inner loop j from 0 to k.

(c) State and justify the running time of an iterative solution.

Solution:

Creating entry i, j requires 2 recursive calls. For one of the recursive calls we need to check if an m exists so that the element at index i can be the last element in a subarray whose elements sum to the target t. As all elements are positive, we can have at most t elements in each subarray, so you have to check at most t possibilities for m, requiring $\mathcal{O}(t)$ time. This means each entry requires $\mathcal{O}(t)$ time overall. All other recursive calls take $\mathcal{O}(1)$ time. Since we have nk entries, we need $\mathcal{O}(nkt)$ time.
Alternatively, we can upper bound the number of m we need to check by n, thus giving us $\mathcal{O}(n^2k)$, which might be better if t is large.