CS 124 Data Structures and Algorithms — Spring 2015
Problem Set 5 solutions

Problem 1

You are given an undirected tree (a connected and acyclic graph) in adjacency list form where each vertex v has a weight $w(v)$. For a subset S of vertices, we define its weight $w(S)$ to be $\sum_{v \in S} w(v)$. A set S is called *separated* if for all $u, v \in S$, the edge (u, v) does not appear in the graph. Give an algorithm to find the largest $w(S)$ achievable over all separated sets (you need not find the S achieving it, just $w(S)$).

Solution

(Preprocessing) First, we can do a DFS or BFS starting at an arbitrary vertex to order our tree to give it a rooted structure and transform our tree into a directed tree with edges going from a parent node to its children.

(Subproblem) For the subtree rooted at vertex v_i, let

$$f(i, 0) = \text{ the largest weight of an independent set not containing } v_i$$
$$f(i, 1) = \text{ the largest weight of an independent set containing } v_i.$$

Then, if we let v_0 be the root, our desired result is $\max(f(v_0, 0), f(v_0, 1))$.

(Recurrence) Let $C(v_i)$ denote the children of v_i. We get the following recurrence:

$$f(i, 0) = \sum_{j \in C(v_i)} \max(f(j, 0), f(j, 1))$$
$$f(i, 1) = w(v_i) + \sum_{j \in C(v_i)} f(j, 0).$$

Because an empty summation is simply 0, this gives us the correct base cases of $f(i, 0) = 0$ and $f(i, 1) = w(v_i)$ when v_i is a leaf.

We can check that the algorithm is correct in each of the two cases:

- When the separated set does not contain v_i, then it must be the union of separated sets over v_i's children, so the maximal weight is simply the sum of the optimal separated sets for the subtrees at each child, which is $\max(f(j, 0), f(j, 1))$
- When the separated set contains v_i, then it cannot contain any of v_i's immediate children, so the maximal weight is that of v_i, plus that over the child subtrees (but not containing v_i's immediate children).

(Comlexity) The space required for the algorithm is $O(n)$, since we need only store two values for each node in the graph.

Naively, it may appear that runtime is $O(n^2)$ since we may need to perform $O(n)$ calculations to compute our function $f(\_, 0)$ at each step. However, because each node is the child of at most one other node, it appears in the summation formulas for $f(\_, 0)$ at most once. Thus, the total amount of time we spend computing summations for all nodes is $O(n)$, giving us an overall runtime of $O(n)$.

Note that our preprocessing can also be done in $O(n)$ time and space, so it does not affect the overall complexity.

Note Alternatively, we could have defined $f(i)$ to simply be the largest weight of an independent set for the subtree rooted at vertex v_i, and instead have written a formula for $f(i)$ in terms of both v_i's children and grandchildren.

Problem 2

There are four types of brackets: (,), <, and >. We define what it means for a string made up of these four characters to be *well-nested* in the following way:

1. The empty string is well-nested.
2. If A is well-nested, then so are <A> and (A).
3. If S, T are both well-nested, then so is their concatenation ST.

For example, (), <>, (()), (<>), ()<>, and ()<()> are all well-nested. Meanwhile, (, <,),)(, (<)>, and (<(> are not well-nested.
Devise an algorithm that takes as input a string s of length n made up of these four types of characters. The output should be the length of the shortest well-nested string that contains s as a subsequence. For example, if (<(> is the input, then the answer is 6; a shortest string containing s as a subsequence is ()<()>.
Solution

(Subproblem) Let $s[i, j]$ be the substring of s from the ith to the $j-1$st character, inclusive. Then, we define the subproblem $f(i, j)$ to be the length of the shortest well-nested string containing $s[i, j]$.

Our desired output is $f(0, n)$.

(Recurrence) The base case is where the length of the substring is 1 or 0.

- If $i = j$, the substring has length 0, and $f(i, j) = 0$.
- If $i = j + 1$, the substring has length 1, and we must add one character to close it, so $f(i, j) = 2$.

To compute $f(i, j)$ recursively, we first note that if $s[i]$ is a closing bracket,) or >, we must insert an matching opening bracket to achieve well-nestedness. Otherwise, the opening bracket $s[i]$ must be matched by some closing bracket $s[k]$ where $i < k < j$, or by adding a closing bracket. Let the matching closing bracket of of $s[i]$ be denoted by $s[i]'$.

The analysis above give the following recurrence:

$$f(i,j) = \begin{cases} 2 + f(i+1, j) & : s[i] =), > \\ 2 + \min(f(i+1, j), \min_{s[k]=s[i]'}(f(i+1, k) + f(k+1, j))) & : s[i] = (, < \end{cases}$$

To store the values, we can keep a 2-D array of size $n+1$ by $n+1$, where $arr[i, j] = f(i, j)$. We can use bottom-up dynamic programming to fill the array in order of increasing substring length, since the recursive relationships only depends on values of shorter substring length.

(Complexity) The runtime is $O(n^3)$ since there are $O(n^2)$ values to calculate, each of which takes $O(n)$ time (from scanning the substring for the matching bracket of $s[i]$).

The space of the algorithm is $O(n^2)$ since the array is $n+1$ by $n+1$.

Problem 3

For a string s, we define s^k as concatenating s with itself k times. For example if $s = aba$, then $s^3 = abaabaaba$ and s^0 is the empty string.
We say x is a *powering* of s if x is a prefix of s^k for some k. We say y is a *mixing* of s, t if the characters of y can be partitioned into two (not necessarily contiguous) subsequences such that the first subsequence is

a powering of s, and the second subsequence is a powering of t. For example $y = abcadcdba$ is a mixing of $s = ab, t = cd$ since $ababa$ is a prefix of s^3 and $cdcd$ is a prefix of t^2.
Describe an algorithm which, given y, s, t determines whether y is a mixing of s, t. In describing your solution, let k be the length of s, m be the length of t, and n be the length of y.

Solution

Solution 1

(Subproblem) Let i be an index into s and j be an index into t (so implicitly, $i+j$ is an index into y). We will let the indices into $s$and g be cyclic – that is, $s[i] = s[i \pm k]$ and $t[j] = t[j \pm m]$.

We define $f(i, j)$ as `true` if $y[i+j : n]$ is an interleaving of powerings of s and t starting from indices i and j, respectively.of some powering of s starting from index i and some powering of t starting from index j.

Our final solution is $f(0, 0)$.

(Recurrence) Then we have the base case:

$$f(i, j) = \texttt{true} \text{ if } i + j = n + 1 \text{ and } k|i, m|j$$

and the recurrence:

$$f(i,j) = \begin{cases} \texttt{true} & : y[i+j] = s[i] \text{ and } f(i+1, j) \\ \texttt{true} & : y[i+j] = t[j] \text{ and } f(i, j+1) \\ \texttt{false} & : \text{ otherwise .} \end{cases}$$

In particular, the base case is correct because a string of length 0 $y[n+1, n]$ is the interleaving of the two empty strings. Then, the recursion follows from checking the two cases of whether the next character belongs to a powering of s or t in the interleaving.

(Complexity) If we memoize this bottom-up, starting from the base cases and filling in order of decreasing i and j, we must compute $O(n^2)$ values in $O(1)$ time each. But with sliding window memoization, we need only track $f(i, j)$ for at most 2 values of j at a time, so we have $O(n)$ space and $O(n^2)$ time complexity overall.

Solution 2

(Subproblem) Let i be an index into s, j be an index info t, and ℓ be an index into y. Let $f(i, j, \ell)$ be `true` if $y[\ell : n]$ is an interleaving of powerings of s and t starting from indices i and j, respectively.

Our final solution is $f(0, 0, 0)$.

(Recurrence) Then we have the base case:

$$f(i, j) = \texttt{true} \text{ if } \ell > n$$

and the recurrence:

$$f(i,j) = \begin{cases} f((i+1) \bmod k, j, \ell+1) \texttt{ OR } f(i, (j+1) \bmod m, \ell+1) & : y[\ell] = s[i] \text{ and } y[\ell] = t[j] \\ f((i+1) \bmod k, j, \ell+1) & : y[\ell] = s[i] \text{ and } y[\ell] \neq t[j] \\ f(i, (j+1) \bmod m, \ell+1) & : y[\ell] \neq s[i] \text{ and } y[\ell] = t[j] \\ \texttt{false} & : \text{ otherwise .} \end{cases}$$

This is correct by the same reasoning as in the previous solution.

(Complexity) If we memoize this bottom-up, starting from the base cases and filling in order of decreasing ℓ, we must compute $O(nmk)$ values in $O(1)$ time each. But with sliding window memoization, we need only track f for at most 2 values of ℓ at a time. So we have $O(mk)$ space and $O(nmk)$ time complexity overall.

Notes
For both solutions, explanations of runtime that use top-down recursion are inefficient. Without memoization, there's exponential runtime, and even with memoization, the sliding window does not work.
A full-credit solution needed to fully describe at least one of the two algorithms. It also needed to note the complexity of the other solution, and state that the optimal space and time depended on the relative lengths of n, m, and k, with $\min(O(n), O(mk))$ space and $\min(O(n^2), O(nmk))$ time complexity overall.

Problem 4

A trie, or prefix tree, is a certain kind of tree that helps when searching through string data. Unlike, e.g., binary search trees, the actual structure of the tree of a trie carries information about the data, so tries cannot be rebalanced to increase performance. Internal nodes are not necessarily binary, and can have any number of children (even 1).
You are given the tree structure of a trie, and your job is to pack nodes of the tree into blocks on disk. Disk has an infinite number of blocks, each of size B. Each node must be placed in exactly one block, but there is no restriction on which block you place a node in.
Given a packing of nodes into blocks, the cost of querying x (where x is a node in the tree) is the number of blocks that must be touched when traversing the unique path from the root of the tree to x. You should assume that the machine servicing the queries has enough memory to hold only a single block at a time. For example, suppose that in order to visit node x from the root, the order of nodes you visit is t, u, v, w, x, where t, w, x are in block 0, and u, v are in block 1. The cost of this query is then 3. You pay once to access block 0 in order to visit t, then you pay once again to access block 1 to visit u and v. In order to access block 1, you had to evict block 0, so you have to pay again to access block 0 in order to visit w then x.
There are n nodes in the tree, and you are given an array $T[1 \dots n]$ where $T[i]$ is the number of times node i was queried last week. Given this information, you want to find a packing of nodes into blocks on disk that would have minimized the sum of all costs of queries last week. Your algorithm does not need to output the actual packing, but just needs to return what sum of all costs of queries it achieves (a single number).
You should assume that you are given the tree in a format so that: (1) the root is vertex 1, (2) there is an array $p[1 \dots n]$ such that $p[i]$ is the parent of node i ($p[1]$ is 0), (3) there is an array $deg[1 \dots n]$ such that $deg[i]$ is the number of children of node i, and (4) for any i and $1 \le j \le deg[i]$, you can find the jth child of node i in constant time.
Solution

(Preprocessing) We will make an array A such that $A[i]$ is the sum of the number of times every node in the subtree rooted at i was queried last week. In other words, $A[i]$ is the number of times that node i was traversed in the course of all of the queries.

Using this, we can write the total cost of all queries given a block assignment as $A[1] + \sum_{i \text{ s.t. } block[i] \neq block[p[i]]} A[i]$.

(Subproblem) Let $c(i, j)$ be the jth child of node i.

Define the subproblem $f(i, j, b)$ to be the minimum cost that we can achieve on the subtree rooted at i restricted to i's children $j, \dots, deg[i]$ when there are b slots remaining in i's block.

Our final goal is given by $A[1] + f(1, 1, B - 1)$.

(Recurrence) Observe that there is always an optimal block assignment in which for each block, the set of nodes assigned to that block are connected. This is because we could achieve the same cost by assigning each disconnected piece to its own block.

Then, we can obtain the recurrence

$$f(i,j,b) = \begin{cases} 0 & : j > deg[i] \\ \min[\, f(c(i,j),1,B-1) + A[c(i,j)] + f(i,j+1,b), & \\ \quad \min_{1 \le k \le b}(f(c(i,j),1,k-1) + f(i,j+1,b-k))] & : \text{otherwise} \end{cases}$$

The first line accounts for the base case where i is a leaf node with no children.

Otherwise, we either put the jth child into a new block, incurring the corresponding cost $A[c(i,j)]$ and then moving onto child $j+1$, or we put the jth child in the same block so that we don't have to pay the cost. In the latter case, we have to decide how much of the current block j is allowed to distribute among its subtree, and how much we reserve for the rest of i's children.

(Complexity) Each (i,j) pair corresponds to a single edge in the tree, and a tree can have at most $n-1$ edges. So the total space required is $O(nB)$ since there are B possible values for b.

Each state requires $O(B)$ time to evaluate, since we have to iterate over b possible values of k and each k (and every other computation we have to do) takes $O(1)$ time. Thus, since there are $O(nB)$ states, the total running time is $O(nB^2)$.

Alternative Solution
Alternatively, we could solve the simpler problem in the case of binary trees and then adjust our algorithm. In this case we don't have to put j into the recurrence as we instead just iterate over how much of the b to give to the left child, which determines how much is left for the right child.
Then, we can then transform our trie into a binary tree by introducing $O(n)$ dummy nodes. Whenever $deg[i] > 2$, make i's left child be its current first child and its right child be a dummy node with each of the remaining $deg[i]-1$ children of i as its children. Before applying the binary tree algorithm to the new tree, we would have to tweak it a little so as not to count the dummy nodes towards the block limit or towards the values in A.

Problem 5

A programmer would like to insert L line breaks into a string after $b_1, \ldots, b_L$ characters. Since he is working on an old computer, which must rerender after each line break, it takes n milliseconds to break a string of n characters (line breaks are not counted in the number of characters) into two pieces.
Consequently, the amount of time required is affected by the order in which the line breaks are added. For example, given the string "needlinebreaks", it takes $14 + 10 = 24$ milliseconds to first break after "need" and then after "line", whereas making those two line breaks in the opposite order only requires $14 + 8 = 22$ milliseconds.
Your goal is to determine the minimum amount of time, in milliseconds, required for the programmer to make all the line breaks.
Solution
Solution 1

(Subproblem) For convenience, let $b_0 = 0$ and $b_{L+1} = l$. Then, we define our subproblem $f(i,j)$ to be the minimum number of milliseconds required to add the appropriate line breaks to the substring $[b_i, b_j]$.

Our desired answer is $f(0, L+1)$.

(Recurrence) To find the optimal solution for $f(i, j)$, it suffices to try each of the possiblilities for where to put the first break point then take the best one. The possibilities are $b_{i+1}, \ldots, b_{j-1}$. This gives the recurrence

$$f(i, j) = (b_j - b_i) + \min_{i<k<j} (f(i, k) + f(k, j))$$

and the base case of $f(i, i + 1) = 0$.

(Complexity) When we use memoization during the evalution (or alternately, first find the optimal solutions for substrings using 0 break points, then 1 break point, etc.), we need to determine f for $(L + 2)^2$ pairs (i, j), each of which requires taking the minimum of up to n values. This results in a runtime of $O(L^3)$.

Meanwhile, keeping track of all of these $(L + 2)^2$ pairs can be done in space $O(L^2)$.

Solution 2

(Subproblem) We use the same subproblem $f(i, j)$ as in the previous solution.

Additionally, let $b(i, j)$ be the index of the next cut which produces the minimum time on $[b_i, b_j]$.

(Recurrence) As before, we have the base case of $f(i, i + 1) = 0$. We also have an additional base case of $f(i, i + 2) = b_{i+2} - b_i$.

The recursion is similar to before, but we use Knuth's optimization and check fewer possible break points:

$$f(i, j) = (b_j - b_i) + \min_{b[i][j-1] \le k \le b[i+1][j]} (f(i, k) + f(k, j)).$$

(Complexity) Now, notice that when we fix $j - i = d$, it requires taking minimums over a total of $O(L)$ values to compute all of the values $f(0, d), f(1, 1 + d), \ldots$, because $b(0, d) \le b(1, d) \le b(1, d + 1) \le, \ldots$. Since there are $O(L)$ possible values of d, the runtime is now $O(L^2)$.

As before, the space is $O(L^2)$.