CSE3358 Problem Set 9
Solution

Problem 1: Relaxed string matching (20 points)
We have seen in class a number of algorithms for the string matching problem where, given a text $T[1..n]$ over some finite alphabet Σ and a pattern $P[1..m]$, we need to determine all positions in T where P occurs (called valid shifts). The Naive algoritms runs in $O(mn)$ time. The Rabin-Karp algorithm runs in $O(n + mv)$ expected time where v is the number of valid shifts. The suffix tree algorithm runs in $O(n + m + v) = O(n + m)$ time. In this problem we consider a relaxed version of the string matching problem that can be solved in linear time.

Let $T_i^m = T[i..i + m - 1]$. We say T_i^m is an **anagram** of P if there is a way of permuting symbols of T_i^m so that the resulting array is equal to P.

(a) (10 points) Give an algorithm that, given an index i, determines whether T_i^m is an anagram of P. Your algorithm should be asymptotically as efficient as possible.

ANSWER: Let $|\Sigma| = k$. Then initialize an array $A[1..k]$ to contain all zeros, where $A[j]$ will eventually hold the number of times symbol j appears in T_i^m. A can be computed in linear time, just as in counting sort. Go through T_i^m once, and increment the corresponding count in A. Create a similar array B for P and then compare A and B in constant time $O(k)$. So this whole algorithm takes $O(m+k) = O(m)$ time.

(b) (10 points) Give an algorithm that, given T and P, reports all i's such that T_i^m is an anagram of P. Your algorithm should run in $O(n + m)$ time.

ANSWER: Comnpute A for T_1^m and B for P as before. This takes $O(m)$ time. Now compare the two in $O(k)$ time and determine whether P is anagram of T_1^m. Now we can compute A for T_2^m in constant time. Simply decrease the count for $T[1]$ in A and increase the count for $T[m + 1]$ in A. B remains the same. Then compare A and B again in $O(k)$ time and determine whether P is anagram of T_2^m, and so on... This algorithm will take $O(m + nk) = O(m + n)$ time.

Problem 2: Finding maximal repeats (20 points)
Consider a string $T[1..n]$. Define $T[0]$ and $T[n + 1]$ as a special symbol that does not occur in T. Consider a string $P[1..m]$. We say P is a maximal repeat of T if $\exists\ 0 < i, j \leq n$ such that:

- $P = T_i^m = T_j^m$, $i \neq j$
- $T_{i-1}^{m+1} \neq T_k^{m+1}\ \forall k$
- $T_i^{m+1} \neq T_k^{m+1}\ \forall k$

(a) (5 points) Based on the above definition, explain in plain English what a maximal repeat is.

A maximal repeat P of length m is a string that occurs at least twice in T such that one of the occurrences, say $P = T_i^m$, cannot be extended to the left or right and still repeat.

(b) (10 points) Consider the suffix tree of T. Show that P is a maximal repeat of T is equivalent to $\exists\ v$ in the suffix tree of T such that:

- v is not a leaf
- the path from root to v spells P
- there are two leaf nodes in v's subtree, say i and j, such that $T[i-1] \neq T[j-1]$

ANSWER:

The first direction is trivial:

If P is a maximal repeat, then there must exist two suffixes, say i and j that start with P such that $T[i-1] \neq T[j-1]$ and $T[i+m] \neq T[j+m]$. Since there must be two paths in the suffix tree that spell suffix i and suffix j, and no two outgoing edges on a node can start with the same symbol, there must be a node v where the two paths divert for the first time. The path to v must spell P. Moreover, v is definitely not a leaf because it has at least two outgoing edges. In v's subtree, we must reach leaf i and leaf j.

Now the other direction:

If such a node v exists, then consider leaf i and leaf j in v's subtree. Let e_1 be the outgoing edge of v that leads to leaf i. Let e_2 be the outgoing edge of v that leads to leaf j. If $e_1 \neq e_2$ then we are done because we have two suffixes i and j that divert after P and $T[i-1] \neq T[j-1]$. If $e_1 = e_2$, then there must be another edge going out of v because any internal node in the suffix tree must have at least two outgoing edges. Therefore, there must be another leaf k in v's subtree. Now either $T[k-1] \neq T[i-1]$ or $T[k-1] \neq T[j-1]$ because it cannot be equal to both since $T[i-1] \neq T[j-1]$. Without loss of generality, let's say $T[k-1] \neq T[i-1]$. Then we have leaves i and k, that are reached from v on two different edges, and we are back to the previous case.

(c) (5 points) Show that a string T of length n has $O(n)$ maximal repeats.

ANSWER: Since a maximal repeat corresponds to a node in the suffix tree, and the suffix tree has $O(n)$ nodes, then we have $O(n)$ maximal repeats.

Problem 3: The Thief and the Knapsack problem... and You (20 points)
A thief robbing a store finds a set S of n items 1 to n. Item i has a value v_i dollars and a weight w_i pounds, where v_i and w_i are integers. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integer W. Unfortunately, he did not take CSE3358; otherwise, he would not be stealing (would he?). But he needs an algorithm to find a set of items $A \subseteq S$ of maximum total value such that $\sum_{i \in A} w_i \leq W$.

Now you have a moral issue here... Should you help the thief in solving his problem and in return get some points for this homework, or should you just refuse and loose the points? It's up to you... :) But if you decide not to help him, you definitely get 5 points for honesty.

(a) (10 points) This problem satisfies the optimality condition for subproblems: If A is an optimal solution for S with weight at most W, and $i \in A$, then $A - \{i\}$ is an optimal solution for $S - \{i\}$ with weight at most $W - w_i$ (cut and paste argument). Give a Dynamic Programming formulation for this problem based on the above idea and an algorithm to solve it in $O(nW)$ time. More precisely, let $c(i, w)$ be the optimal solution for the set $\{1, ..., i\}$ with a weight at most w. Obviously, if $w_i > w$, then i cannot be part of the solution and $c(i, w) = c(i-1, w)$. Express $c(i, w)$ in terms of $c(i-1, w)$ and $c(i-1, w-w_i)$ and describe how the dynamic programming table is computed, and finally how to obtain the solution itself from the table.

ANSWER: The optimal solution for $c(i, w)$ can have 4 possibilities:

- if $w = 0$ (cannot carry any weight) or $i = 0$ (no objects) then $c(i, w) = 0$ (these represent the base cases)
- if $w_i > w$ then object i cannot be picked and $c(i, w) = c(i-1, w)$
- if object i is picked, i.e. $w_i < w$, then the best thing we can do is getting the value of object i, v_i, in addition to the optimal solution for the rest of the objects and the remaining weight, i.e. $v_i + c(i-1, w-w_i)$
- if object i is not picked, then the best thing we can do is getting the optimal solution for $c(i-1, w)$.

Therefore,

$$c(i,w) = \begin{cases} 0 & i = 0 \text{ } or \text{ } w = 0 \\ c(i-1,w) & w_i > w \\ \max(v_i + c(i-1, w-w_i), c(i-1,w)) & otherwise \end{cases}$$

$c(n, W)$ is the answer, i.e. the maximum value that we can carry within the weight limit W. We can have a $(n+1) \times (W+1)$ table to hold all these values and perform the computation row by row. To obtain the set of object that we should pick, we can back track inside the table as we discussed it in class. The running time of this algorithm is $O(nW)$ since we have $O(nW)$ entries to compute, and each requires constant time.

(b) (10 points) Assume that any two items i and j satisfy the following: $w_i < w_j \Rightarrow v_i > v_j$. In other words, lighter items are more valuable. Describe an efficient greedy way for solving the knapsack problem in this case with a running time independent of W. Use a "cut and paste" argument that justifies the greedy choice.

ANSWER: In this case, we can prove that there is always a solution that contains the lightest item. This can be done by a cut and paste argument to justify the greedy choice. Consider the items sorted by their weight. Assume we have an optimal solution that does not contain item 1. Then let $i > 1$ be the lightest item in the solution. Cut i and paste 1. Definitely, we do not exceed the weight limit because item 1 is not heavier than item i. Also, we still get an optimal solution because $v_1 \geq v_i$. This argument can be carried further to say that item 2 must be in the solution if the solution has at least two items, etc...

Therefore, in this case a greedy algorithm will do the job. Sort the objects by increasing order of their eeight (i.e. decreasing order of their weight). Then pick objects in order (most valuable first) until you cannot add anymore objects. The running time of this algorithm is dominated by sorting and hence it is $O(n \log n)$ which is independent of W.

Problem 4: Largest common circuit (20 points)
Define a circuit as a logic consisting of **AND**, **OR**, and **NOT** gates connected together such that:

- every gate has a fan in of 2 (a gate can have at most two inputs)
- every gate has a fan out of 1 (the output of a gate can be input to at most one gate)
- no feedback loops

The problem: We have two circuits as described above, one with m gates and one with n gates, we would like to obtain the largest circuit (in terms of the number of gates) that is a sub-circuit of both.

(a) (5 points) Give a brute force algorithm for finding the largest common circuit of two circuits and analyze its running time.

ANSWER: The first observation that we have to make is that each circuit can be represented by a binary tree where each node will have one of three types: AND, OR, or NOT. Now the problem is given two binary trees, find the largest tree that is subtree of both.

One way is to consider every subtree of T_1 and check whether it is a subtree of T_2 and keep track of the largest so far. But,

- How many subtrees of T_1 can we have?
- How much time we need to check one subtree against T_2?

Well, too many subtrees, and large time to check! Here's why: Consider T_1 to be a complete binary tree of height h, then any tree of height h is a subtree of T_1. How many trees of height h can we have? Definitely exponential. Let $S(h)$ be the number of binary trees of height h. $S(-1) = 1$ (empty tree). $S(0) = 1$ (one node). $S(2) = 3$. Also, $S(h) \geq S(h-1)^2$ because a tree of height h can consist of two subtrees of height $h-1$. Therefore, $S(h) \geq 3^{2^h}$. But h is $\Theta(\log n)$ for a complete tree; therefore, $S(h) \geq 3^n$.

Moreover, given a tree, to check whether it is a subtree of T_2, we have to place the root at some node and then make sure that there is a match for every node. This is easy if the tree is ordered, but since it is not, at any node, we can map either the left child to the left child and the right child to the right

child or vice-versa. This gives two choices at each level for every node, which yields $2.4.8... = O(2^{n^2})$ choices.

PS: if the trees were ordered trees, then it would be much simpler. Just pick a node i from T_1 and a node j from T_2 and perform a tree walk on i's subtree in T_1 and follow the same steps (if they exist) in T_2 marking those nodes that are common. This would reveal the largest common subtree that maps i and j to the root of the common subtree. This takes linear time. We do this for every i and j, and we keep track of the best one so far. With a total of $O(mn)$ walks (i,j pairs) we end up with a $O(mn^2)$ algorithm. Part (b) suggests even a better algorithm.

(b) (15 points) Give a dynamic programming formulation of the problem and design an algorithm (based on dynamic programming) for finding the largest common circuit of two circuits that runs in $O(mn)$ time.

ANSWER: Let $c(i, j)$ be the size of the largest common tree that maps node i from T_1 and node j from T_2 to the root of the common tree. Also let $T_1(i)$ be the type of node i in T_1 and similarly $T_2(j)$ be the type of node j in T_2. Then we have an optimal structure such that:

$$c(i,j) = \begin{cases} 0 & T_1(i) \neq T_2(j) \\ 1+ \quad \max(c(left[i], left[j]) + c(right[i], right[j]), & \\ \qquad c(left[i], right[j]) + c(right[i], left[j])) & otherwise \end{cases}$$

The values for $c(i, j)$ can be recursively obtained in a table of size mn. Then we would pick the entry with $max_{i,j}c(i, j)$ and backtracking from that entry in the table would give us the largest common tree.

The running time of this algorithm is obviously $O(mn)$ because we have $O(mn)$ entries to compute and each requires constant time.

Problem 5: Spanning trees (20 points)

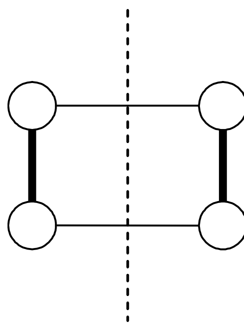
(a) (10 points) Let w_{max} be the maximum weight of an edge in a spanning tree. A bottleneck spanning tree is a spanning tree whose w_{max} is minimum over all spanning trees. Use a "cut and paste" argument to show that a minimum spanning tree is a bottleneck spanning tree.

ANSWER: We can show that the minimum spanning tree minimizes w_{max} also. To show this we use a cut and paste argument. Consider the minimum spanning tree T with edge $e = (u, v)$ having the maximum weight. Consider another tree T' with edge e' having the maximum weight. Let $w(e') < w(e)$. We will reach a contradiction. First, note that e cannot be in T' because e' is the edge with maximum weight in T' and $w(e') < w(e)$. Cut $e = (u, v)$ from T. This will disconnect u and v. Since e is not in T', there must be a path in T' between u and v that does not use $e = (u, v)$. There must be an edge e'' on this path that re-connects T. Paste e'' in T. Therefore, $T - \{e'\} \cup \{e''\}$ is a spanning tree with a weight less than that of T because $w(e'') \leq w(e') < w(e)$, a contradiction.

(b) (10 points) Consider the following divide and conquer algorithm for computing a minimum spanning tree: Given a graph $G = (V, E)$, partition the set V of vertices into two sets V_1 and V_2 such that $|V_1|$ and $|V_2|$ differ by at most 1. Let E_1 be the set of edges that are incident only on vertices in V_1, and let V_2 be the set of edges that are incident only on vertices in V_2. Recursively solve a minimum spanning tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum weight edge in E that crosses from V_1 to V_2, and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.

Either argue that the algorithm correctly computes a minimum spanning tree and give an implementation for it with an analysis of its running time, or provide a counter example for which the algorithm fails.

ANSWER: This algorithm does not work. Here's a counter example:



If the cut is as shown above, then we eventually compute a spanning tree for the left side (which consists of a heavy edge) and a spanning tree for the right side (which also consists of a heavy edge). Then we connect the two spanning trees by the minimum weight edge. The result is a spanning tree with two heavy edges. But there is one spanning tree with just one heavy edge.

But anyway, the running time is $T(V) = 2T(V/2) + \Theta(V^2)$ because we can have up to V^2 edges to investigate for every subproblem to chosse the smallest one. This yields $\Theta(V^2)$ time algorithm.