Sample Quiz 2 Solutions

Problem 1. SMS

The phone company, Aikon, has hired you to develop the predictive text input for its new mobile phone set. In a mobile phone, each digit is mapped to a set of three or four characters. For example the digit ‘2’ is mapped to the set \{a, b, c\}, while the digit ‘7’ is mapped to the set \{p, q, r, s\}. When a user types in the sequence of digits 227, possible English words that correspond to the sequence of digits include ‘bar’, ‘car’ and ‘cap’. To minimize the expected amount of work that the user has to do, the system should display the most probable English word that correspond to the sequence of digits pressed. If the word is not the word that the user is looking for, the next most probable word should appear when the user presses the key ‘*'. Describe the design of your system. You may assume that you have a dictionary of English words and their corresponding usage probabilities available. Keep in mind that both response speed and memory usage is important for such a system.

Solution: Consider a dictionary of \(n\) words. Each word can be mapped deterministically to a sequence of digits. Words that are mapped together into the same sequence of digits can be sorted into either a sorted array or linked list according to their probability of appearing using a total time of \(O(n \log n)\) with merge sort. Once the correct list is found, it takes constant time to move to the word of the next highest probability by pressing the key ‘*’ in the sorted array or list. The total storage using these lists or arrays is linear in the total number of characters in the dictionary.

We require a data structure in order to quickly find the list that corresponds to the sequence of digits that the user has just entered. A simple data structure for this purpose is a hash table. A properly designed hash table should have storage requirement of \(O(m)\) where \(m\) is the number of distinct sequences mapped to by words in the dictionary. It is obvious that \(m = O(n)\). A hash table would allow access to the appropriate list in time \(O(k)\) where \(k\) is the length of the sequence of digits, assuming simple uniform hashing for chaining or uniform hashing for open addressing.

In the hash table solution, the mobile phone would be unable to provide any useful information when the sequence of digits have no corresponding dictionary word. An implementation based on a trie (or radix tree) would be able to perform additional functions. For example, it can inform the user that there is no dictionary word whose prefix maps to the sequence of digits pressed, so the user should immediately switch to some other non-predictive input mode. Most students used a data structure similar to a trie for their solution.

In the trie (or radix tree) data structure, each edge corresponds to a character and strings that share a common prefix shares a common path to the root of the tree where each edge on that path is labeled by a character on the common prefix. For example, Figure 1 shows a trie for the alphabet \(\Sigma = \{a < b < c < d\}\) that stores the strings \(a, d, ac, bc, ca, db, abd, ada, cdb\). In this case, each node has 4 children and each edge is labeled with one of the letters of the alphabet from left to
right in order. When searching for a key \( \alpha = \alpha_0\alpha_1 \ldots \alpha_p \) at a node of depth \( i \), we follow the edge labeled by the same character of \( \alpha_i \). Some nodes do not have strings corresponding to them (shown in black in the figure). Constructing a trie is simple. For each new string to insert, just follow the path that correspond to the string until either a leaf is reached or the end of the string is reached. If the end of the string is reached, the string is added to the node corresponding the the end of the string. If a leaf is reached, the trie is extended by adding the remaining parts of the string to the trie and the string is inserted at the new leaf. The running time for constructing the trie is \( O(l) \) where \( l \) is the total length of all the strings to be stored in the trie.

![Trie Diagram]

**Figure 1**: Trie

In a mobile phone, the characters are usually the digits 0 to 9 (or some subset of them). The total storage required by the trie data structure is \( O(l) \), where \( l \) is the total length of all the distinct sequence of digits mapped to by words in the dictionary. Note that \( l \) is larger than \( m \) so storage used may potentially be larger than that used by a hash table. Finding the right child to follow at each node takes constant time, hence the time required to find the appropriate list is still \( O(k) \) as in the hash table solution. However, with the trie data structure, we can inform the user that there is no dictionary word whose prefix maps to the sequence of digits pressed when the current node has no children.

The trie data structure can also be augmented to allow more sophisticated features such as displaying the prefix of the most likely word that starts with the sequence of digits pressed so far. This may be useful at trie nodes that do not have any possible corresponding words but have children with corresponding words. To do that, the prefix of the most likely word in the subtree (or a pointer to the word) will have to be stored at the root of each subtree. Note that to obtain of the most likely word in the subtree, you only need to compare the most likely words in each of the children of the current node together with the most likely word in the current node. The entire process can be done using a post-order walk of the tree in \( O(l) \).
Problem 2. Largest Slope

Design a data structure to maintain a dynamic set of points in the plane. The data structure should allow the operations \textsc{Insert}, \textsc{Delete} and \textsc{Max-Slope}, where \textsc{Max-Slope} returns two points such that the line segment between them has maximum slope.

Solution Sketch: The key observation is that we only need to consider consecutive points sorted on the $x$-coordinates to find the largest slope. Use a red-black tree augmented with the following fields describing the subtree rooted at the node: the point with maximum $x$-coordinate, the point with the minimum $x$-coordinate, the maximum slope in the subtree. Insertion and deletion can be done in $O(\log n)$ time, and the max-slope search can be done in $O(1)$ time.

Problem 3. Circle

Let $S$ be a set of $n$ points in the plane. Give an efficient algorithm that determines whether there exist four points in $S$ that lie on the same circle. 

\textit{Hint.} In general, three points uniquely determine a circle.

Solution Sketch: Let $p$ and $q$ be a pair of distinct points in $S$. For each point $r \in S \setminus \{p, q\}$, compute the circle $C(p, q, r)$ determined by the three points $p, q$ and $r$. Sort the circles according to their radii. If two circles $C(p, q, r)$ and $C(p, q, r')$ have the same radii and both lie on the same side of the line $pq$, then the four point $p, q, r, r'$ are cocircular. Computing the radii of these circles takes $O(n)$ time. Sorting $n - 2$ circles by radii takes $O(n \log n)$. Repeating the process for all possible pairs of points $p$ and $q$ gives an $O(n^3 \log n)$ upper bound on the total running time.

Problem 4. Compression

You are given a dictionary $D$ comprising of $k$ strings where the maximum length of a string in the dictionary is $m$ and the total length of all strings in the dictionary is $l$. Given a new string $s$ of length $n$, your task is to partition $s$ up into the smallest number of segments such that each segment matches exactly with a string in $D$. To compress the string, only the indices of the matching strings in the dictionary are transmitted in order of the segments at the cost of $\lceil \log k \rceil$ bits per segment. Give an efficient algorithm for compressing a string. You may assume that there is always at least one way to partition a string up such that the segments match with strings in the dictionary. Typically, for this problem $m$ is much smaller than $k$. Also, you may want to take into consideration that the same dictionary may be used to compress many strings.

Solution: The problem can be solved using dynamic programming. Since the cost of transmitting each partitions is the same, we treat the cost of transmission as 1 instead of $\lceil \log k \rceil$. To see that the problem exhibits the optimal substructure property, let $C(i)$ denote the optimum cost of compressing the string $s[1 \ldots i]$ using the dictionary. Let $s[j, i]$ be the a partition that is part of the optimal partitioning. Then $C(i) = C(j - 1) + 1$ where $C(j - 1)$ is the optimum cost of compressing the string $s[1 \ldots j - 1]$. If this is not the case, then we can obtain a cheaper cost by using the optimum cost of compressing $s[1 \ldots j - 1]$.

This suggests the following algorithm. Use three arrays of size $n$. The first array $C$ is used to store $C(i)$ the optimum cost of compressing $s[1 \ldots i]$. The second array $P$ is used to store the
starting point of the last partition used to get the optimal cost $C(i)$ while the third array $I$ is used to store the dictionary index of the last partition. Initialize each entry of $C$ to infinity. For each position $j$ starting from 1, find all dictionary entries that match a prefix of $s[j \ldots n]$. If the match is $s[j \ldots k]$, update $C[k]$ by taking the minimum of $C[k]$ and $C[j-1] + 1$ (where we define $C[-1]$ to be 0). Update $P[k]$ and $I[k]$ if $C[k]$ is updated. Upon completion, the optimal cost is $C[n]$. We can first look in $P[n]$ to find the starting point of the last partition $p$. The starting point of the previous partition can then be found in $P[p-1]$. This can be done repeatedly until the first partition is found. The index of the partitions can be found similarly in the array $I$. Once the dynamic programming is completed, the optimal partitions and corresponding dictionary indices can be found in $O(n)$.

The running time of dynamic programming depends on the time required to find all the dictionary entries that match a prefix of $s[j \ldots n]$. We know that the length of each dictionary entry is no more than $m$. Hence we only need to check for a match for each of $s[j \ldots j + m - 1]$ in the dictionary. If we store each dictionary entry in a hash table, we can check for a match for each entry in time $O(m)$ (the time required to check whether the key matches the string is $O(m)$). This gives a time of $O(m^2)$ to find all dictionary entries that match a prefix of $s[j \ldots n]$ and a total time of $O(m^2 n)$ for dynamic programming. If the dictionary entries are stored in a sorted array, each comparison takes $O(m)$ time, giving a total time of $O(m \log k)$ to match a single string and $O(m^2 \log k)$ to find all dictionary entries that match a prefix of $s[j \ldots n]$. Total running time is then $O(m^2 n \log k)$. Most students used a trie-like data structure in problem 1 but did not exploit the same structure for this problem. Using a trie, it is possible to find all dictionary entries that match a prefix of $s[j \ldots n]$ in time $O(m)$ as we only need to follow the links of the trie once down to a leaf. This gives a total running time of $O(mn)$. The running time for constructing a trie is $O(l)$.

(Using a more sophisticated data structure such as a keyword tree\(^1\), it is possible to find all entries that match a substring of $s$ in time $O(n + e)$. Hence it is possible to obtain the optimal solution in time $O(n + e)$ which can be faster if $m$ is large and $e$ is small.)

**Problem 5. Triangle**

Let $P$ be a set of $n$ points in the plane. Give an efficient algorithm that, given a point $q$, will determine whether there exist three points $p, p', p'' \in P$ such that the triangle $T$ with vertices $p, p'$ and $p''$ contains $q$.

**Solution Sketch:** The following statements are equivalent:

1. We can find such a triangle $T$ with vertices in $P$.
2. We can find $T$ using only vertices on the convex hull of $P$.
3. The point $q$ lies within the convex hull of $P$.

Compute the convex hull $\text{CONV}(P)$ of the point set $P$. Now apply the divide-and-conquer strategy to find the triangle that contains $q$ if one exists. Let $v_1, v_2, \ldots, v_m$ be the vertices of $\text{CONV}(P)$ in counter-clockwise order. Divide $\text{CONV}(P)$ into two halves using the line $v_{m/2}v_{m/2}$ and discard the half that does not contain $q$. We can decide which half to discard in constant time by checking

---

\(^1\)See for example *Algorithms on Strings, Trees and Sequences* by Dan Gusfield.
which side of the line $v_1v_{m/2}$ that $q$ lies on. Now recurse until we have a triangle $T$. Either $T$
contains $q$, or no such triangle $T$ exists. Computing the convex hull takes $O(n \log n)$ time. The
divide-and-conquer part takes $O(\log n)$ time. The total running time is $O(n \log n)$.

The bottleneck in the above algorithm is the convex hull computation. For an alternate method,
observe that if $q$ is a convex hull vertex for $P \cup \{q\}$, then there exists no triangle $T$ that has vertices
in $P$ and contains $q$. The converse is also true. So we just need to determine whether $q$ is a convex
hull vertex for $P \cup \{q\}$. To do this, compute the polar angles from $q$ to all points $p \in P$. Now
determine whether any two consecutive polar angles have angular difference greater than $\pi$. If so,
$q$ must be a convex hull vertex. All these can be done in $O(n)$ time, which is optimal.

**Problem 6. Flight Paths**

It is common to have to switch airplane at some intermediate airports on a flight in order to reach
a destination. If the new plane is from the same airline, there is often a discount for the continuing
flight. If the new plane is from a partner airline, the cost is often cheaper than if the new plane is
from a non-partner airline. In fact, the cost may be different depending on which airline the switch
is from. Give an efficient algorithm for finding the cheapest path from between two locations,
taking into account the cost of switching between airlines at intermediate airports.

**Solution:** Assume that there are $n$ airports and $k$ airlines, that there is at most one flight between
any two cities for each airline (or that all flights by the same airline between two cities are identical)
and that the cost for any flight between airports $P$ to $Q$ depends only on the airports, the airline
used for the flight and the airline used to enter $P$. Let the total number of flights in the system be $E$.

To solve the problem, we generate a weighted directed graph and use Dijkstra’s algorithm for
finding the shortest path from the source airport $S$ to destination airport $D$ on the graph. For each
airport $A$ other than the source airport, we generate $k$ nodes in the graph $(A, i)$, $i = 1, \ldots, k$, to
represent arriving in airport $A$ using airline $i$. For a flight from an airport $A_1$ (except the source
airport $S$) to airport $A_2$ on airline $j_2$, join the nodes $(A_1, j_1)$ for $j_1 = 1, \ldots, k$, to $(A_2, j_2)$ with
a weighted edge where the weight is the cost of flying on airline $j_2$ from $A_1$ to $A_2$ after having
arrived in $A_1$ on $j_1$. Flights from the source airport are treated slightly differently. For each airport
$A$ reachable by a direct flight on airline $j$ from $S$, join $S$ and $(A, j)$ by an edge $(S, (A, j))$ where
the weight is the cost of the flight (with no prior flight into the city $S$). Flights into the source
airport may be ignored. We also add a special node $D$ and additional edges of weight 0 from nodes
$(D, i)$, $i = 1, \ldots, k$. The problem then becomes that of finding the shortest path from $S$ to $D$.

The number of nodes in the constructed graph is $nk$ and the number of edges is $Ek$. Running
Dijkstra’s algorithm using Fibonacci heaps takes time $O(nk \log(nk) + Ek)$. 