1 (10 PTS.) SHORT QUESTIONS AND HOPEFULLY SHORT ANSWERS.

No justification is required for your answers.

1.A. (5 PTS.) Give an asymptotically tight bound for the following recurrence.

$$T(n) = \sum_{i=1}^{10} T(n_i) + O(n) \quad \text{for } n > 200, \qquad \text{and} \qquad T(n) = 1 \quad \text{for } 1 \le n \le 200,$$

where $n_1 + n_2 + \cdots n_{10} = n$, and $n/20 \le n_i \le (9/10)n$ for all i.

Solution:

$T(n) = O(n \log n)$.

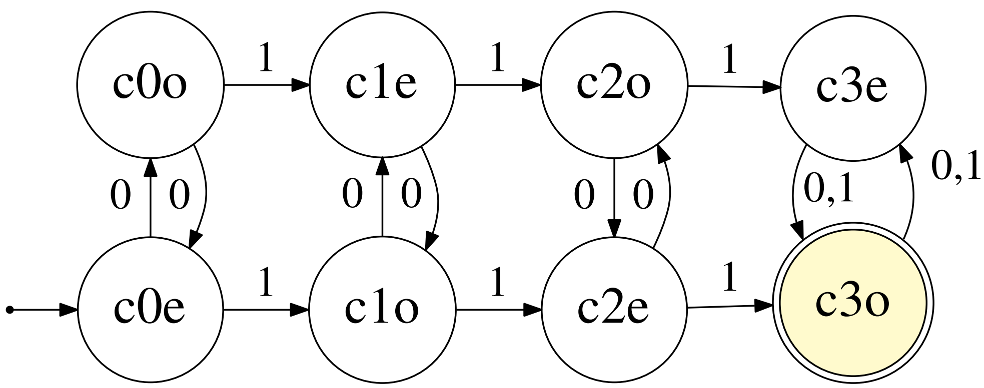
1.B. (5 PTS.) Describe (in detail) a DFA for the language below. Label the states and/or briefly explain their meaning.

$$\{w \in \{0, 1\}^* \mid w \text{ has at least three } 1\text{'s and has odd length}\}.$$

Solution:

The easiest solution is to have a counting DFA M_1, that counts the number of 1s its see till 3, and a counting DFA M_2 that keep track of the even/odd of the input length, and then build the product automata.

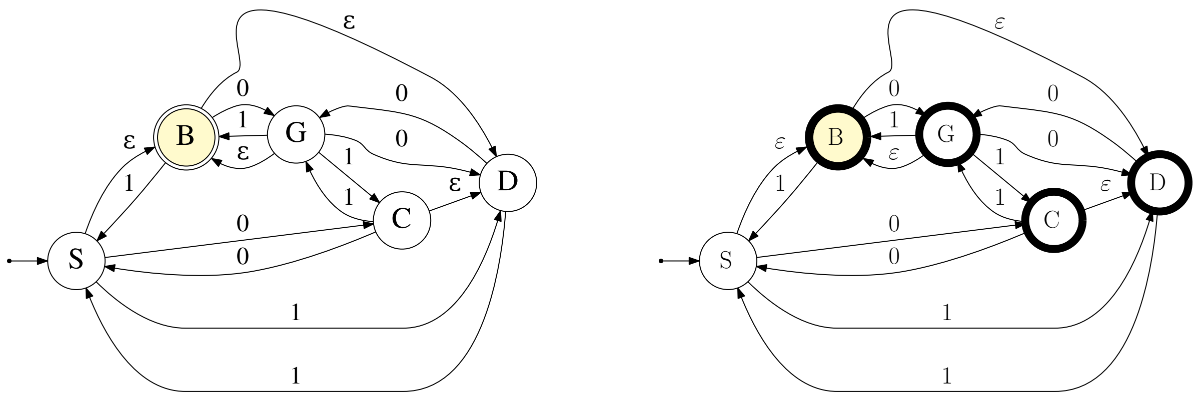
Here is a clean drawing of this product automata:



2 (15 PTS.) I have a question about NFAs.

2.A. (5 PTS.) Recall that an NFA N is specified as $(Q, \delta, \Sigma, s, F)$ where Q is a finite set of states, Σ is a finite alphabet, $s \in Q$ is the start state, $F \subseteq Q$ is the set of accepting states, and $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ is the transition function. Recall that δ^* extends δ to strings: $\delta^*(q, w)$ is the set of states reachable in N from state q on input string w.

In the NFA shown in the figure below what is $\delta^*(S, 0)$?



2.B. (10 PTS.) Given an arbitrary NFA $N = (Q, \Sigma, \delta, s, F)$ and an arbitrary state $q \in Q$ and an arbitrary string $w \in \Sigma^*$ of length t, describe an efficient algorithm that computes $\delta^*(q, w)$. Express the running time of your algorithm in terms of n, m, t, where $n = |Q|$, $m = \sum_{p \in Q} \sum_{b \in \Sigma \cup \{\epsilon\}} |\delta(p, b)|$, and $t = |w|$. Note that faster solutions can earn more points. You can assume that $|\Sigma| = O(1)$.

You do not need to prove the correctness of your algorithm (no credit for incorrect algorithm).

(Hint: Construct the appropriate graph, and do the appropriate things to it.)

Solution:

We interpret the N as a directed graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$. The set of vertices $\mathsf{V} = Q$. For any $u, v \in Q$, such that there $c \in \Sigma \cup \{\varepsilon\}$, such that $v \in \delta(u, c)$, we add the edge $u \to v$ to E. For every such created edge, we also build an array, such that given $c \in \Sigma \cup \{\varepsilon\}$, we can decide in constant time if $v \in \delta(u, c)$ – such an edge is ***c-admissible***. Clearly, this preprocessing can be done in $O((n + m)|\Sigma|) = O(n + m)$ time.

Computing ε-closure. Now, we extend the **BFS** algorithm, such that given a set of vertices $X \subseteq \mathsf{V}$, we compute all the set of vertices $Y \supset X$ that is reachable by ε-transition from X. This for example can be done by comping G, removing all the edges that are not ε-admissible, and creating a new vertex z, connecting z to all the vertices of X. Now, doing **BFS** in the new graph H starting at z, all the vertices that were visited (ignoring z) are ε-reachable from X. This takes $O(n + m)$ time.

Computing c-transition. Given a set of vertices X, and a character c, scan the graph. For each vertex $x \in X$, scan all the outgoing edges from x. If there exists an edge $x \to y$, such that $y \in \delta(x, c)$ (which can be determined in constant time by the preprocessing), we add y to the set of vertices Y. Clearly, this can be done in $O(n + m)$ time.

The algorithm. Let $w = w_1 w_2 \dots w_t$. Let $Q_0 = \{q\}$. In the ith iteration, we compute the ε-closure of Q_{i-1}, and let Q'_i be this set of states/vertices. Next, we compute the w_i transition of Q'_i, let Q''_i be the resulting set of vertices. Next, compute the ε-closure of Q''_i, and let Q_i be the resulting set of vertices.

Running time. We invoke three procedures, each one takes $O(n+m)$ time. Overall, this takes $O((n+m)t)$ time.

3 (15 pts.) MST when there are few weights.

Let $G = (V, E)$ be a connected undirected graph with n vertices and m edges, and with positive edge weights. Here, the edge weights are taken from a small set of k possible weights $\{w_1, w_2, \dots, w_k\}$ (for simplicity, assume that $w_1 < w_2 < \dots < w_k$). Describe a **linear time** algorithm to compute the MST of G for the case that k is a constant. (For partial credit, you can solve the case $k = 2$.)

Provide a short explanation of why your algorithm is correct (no need for a formal proof).

Solution:

Modifying Prim's algorithm.

For edge e of weight w_i, we refer to i as the ***index*** $i(e)$ of e. Precompute the indices of all edges – this takes $O(m \log k)$ time.

For each index i, we maintain a queue Q_i of all edges that are outgoing from the current spanning tree computed that are of index i. We maintain all the non-empty queues in a heap. This provides us a with a heap that performs each operation in $O(\log k)$ time. For example, to insert an edge e of index i into this heap, we check if Q_i is empty. If it is, we add the edge e_i to it, and add Q_i to the heap of queues. Similarly, to do extract-min, we get the min non-empty queue, and delete an edge from it.

As such, we can directly implement Prim's algorithm, an the running time is $O((n+m)\log k)$.

Solution:

Modifying Kruskal's algorithm.

Let H be a graph, where some edges are marked as *tree edges*, and some edges are marked as ***usable***. Let F be the set of tree edges, and let U be the set of usable edges. Here, we assume that the tree edges form a forest. One can modify **DFS**, so that it uses all the tree edges and only the usable edges – indeed, it first scan the edges and call recursively on the edges marked as tree edges. Then it scans again, and call recursively on all the usable edges. Clearly, the resulting **DFS** tree, is a spanning forest of the graph formed by the usable and tree edges. The running time of this algorithm is $O(n+m)$. Let **BTU**(F, U) be the resulting algorithm.

Lets get back to the problem. Scan the edges, and partition them into k sets $E_1, \dots, E_k$, where E_i contains all the edges of G with weight w_i. This can be done in $O(n + m \log k)$ time. Let $F_0 = \emptyset$. In the ith iteration of the algorithm, for $i = 1, \dots, k$, the algorithm would compute $F_i =$ **BTU**(F_{i-1}, E_i). The output of the algorithm is F_k, which is the desired MST.

Correctness. Consider any **non-decreasing** ordering of the edges of G, where all the edges of weight w_j that appear in F_j, appear in the ordering before all other edges of weight w_j, It is easy to verify that the output of the algorithm is the MST computed by Kruskal algorithm when executed on this ordering of the edges.

Running time. The preprocessing takes $O(n + m \log k)$ time, and each iteration takes $O(n + m)$ time. Overall, the running time is $O((n + m)k)$.

4 (15 PTS.) SHUFFLE IT.

Let $w \in \Sigma^*$ be a string. A sequence of strings $u_1, u_2, \ldots, u_h$, where each $u_i \in \Sigma^*$, is a valid *split* of $w \iff w = u_1 u_2 \ldots u_h$ (i.e., w is the concatenation of $u_1, u_2, \ldots, u_h$). Given a valid split $u_1, u_2, \ldots, u_h$ of w, its ***price*** is $p(w) = \sum_{i=1}^{h} |u_i|^2$. For example, for the string INTRODUCTION, the split INT · RODUC · TION has price $3^2 + 5^2 + 4^2 = 50$.

Given two languages $T_1, T_2 \subseteq \Sigma^*$ a string w is a ***shuffle*** iff there is a valid split $u_1, u_2, \ldots, u_h$ of w such that $u_{2i-1} \in T_1$ and $u_{2i} \in T_2$, for all i (for simplicity, assume that $\varepsilon \in T_1$ and $\varepsilon \in T_2$). You are given a subroutine `isInT`(x, i) which outputs whether the input string x is in T_i or not, for $i \in \{1, 2\}$. To evaluate the running time of your solution you can assume that each call to `isInT` takes constant time.

Describe an efficient algorithm that given a string $w = w_1 w_2 \ldots w_n$, of length n, and access to T_1 and T_2 via `isInT`, outputs the minimum ℓ_2 price of a shuffle if one exists. Your algorithm should output ∞ if there is no valid shuffle.

You will get partial credit for a correct, but slow (but still efficient), algorithm. An exponential time or incorrect algorithm would get no points at all.

What is the running time of your algorithm?

Solution:

The easier solution is via a graph construction. First compute the two sets

$$P_\alpha = \{(i, j, \alpha) \mid \forall i \leq j \leq n, \text{ and } w_i w_{i+1} \ldots w_j \in T_\alpha\} \qquad \text{and} \qquad \alpha \in \{1, 2\}.$$

This takes $O(n^2)$ time using $O(n^2)$ calls to `isInT`.

Build a graph G over the set of vertices $\mathsf{V} = P_1 \cup P_2 \cup \{s, t\}$. Connect s to all vertices $(1, j, 1) \in P_1$, setting the weight of the edge to be j^2. Similarly, connect all the edges of the form $(i, n, 1) \in P_1$ to t and all the edges of the form $(i, n, 2) \in P_2$ with weight 0.

Finally, for every triple i, j, k, connect $(i, j, 1) \in P_1$ to $(j+1, k, 2) \in P_2$ if they exist, with an edge $(i, j, 1) \to (j+1, k, 2)$ of weight $\big(k - (j+1) + 1\big)^2$. Similarly, connect $(i, j, 2) \in P_2$ to $(j+1, k, 2) \in P_1$ if they exist, with and edge $(i, j, 2) \to (j+1, k, 1)$ of weight $\big(k - (j+1) + 1\big)^2$. Here, an edge pays for an atomic string, before it enters it.

Let G be the resulting graph. Clearly, this graph is a DAG with $O(n^2)$ vertices, and $O(n^3)$ edges. Clearly, the shortest path in this graph from s to t. This path can be computed in linear time since this is a DAG using topological sort.

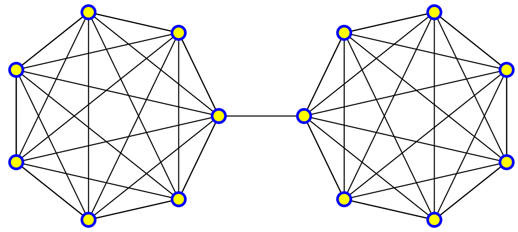
Another natural and acceptable solution is of course to use dynamic programming.

Rubric:

(I) -1: Solution runs in $O(n^3 \log n)$ time.
(II) -2: Solution runs in $O(n^4)$ time.
(III) -2.5: Solution runs in $O(n^4 \log n)$ time.
(IV) -15: Exponential time.

5 (15 PTS.) Dumb and dumbbell.

For $k > 1$, a (k,k)-*dumbbell* is a graph formed by two disjoint complete graphs (cliques), each one on k vertices, plus an edge connecting the two (i.e., its a graph with $2k$ vertices). See figure for a $(7,7)$-dumbbell. The DUMB problem is the following: given an undirected graph $\mathsf{G} = (V, E)$ and an integer k, does G contain a (k,k)-dumbbell as a subgraph? Prove that DUMB is NP-COMPLETE.



Solution:

The problem is clearly in NP. Given an instance G, k, a solution is made out of two lists, each of k vertices. One can readily check in polynomial time that each list corresponds to a clique of size k, and that there is an edge between some pair of vertices in the two lists.

As for the reduction from an NP-COMPLETE problem. Consider an instance G, k of CLIQUE – we can safely assume that $k > 4$, as we can solve such instances directly in polynomial time. Consider the graph formed by adding a clique K_2 to G, and connecting all the vertices of G, to one special vertex $v \in V(K_2)$. Let H be the resulting graph.

Clearly, if G has a clique K of size k, then (K, K_2) form a (k,k)-dumbbell in H.

Similarly, if H has a (k,k)-dumbbell (K', K''), then if K' does not contain v, then all its vertices must be from G, and a clique of size k in G, as desired. The same argument applies if v is not K''. One of these two events must happen.

Thus, H has a (k,k)-dumbbell $\iff$ G has a clique of size k.

Clearly, the reduction is polynomial time.

We conclude that DUMB is NP-COMPLETE.

6 (15 PTS.) YOU ARE THE DECIDER.

Prove (via reduction) that the following language is undecidable.

$$L = \{\langle M \rangle \mid M \text{ is a Turing machine that accepts at least 374 strings}\}.$$

(You can not use Rice's Theorem in solving this problem.)

Solution:

Same old, same old.

For the sake of argument, suppose there is an algorithm **Decide374** that correctly decides the language L. Then we can solve the Halting problem as follows:

```
DECIDEHALT(⟨M, w⟩):
  Encode the following Turing machine M′:
      M′(x):
        run M on input w
        return TRUE
  if Decide374(⟨M′⟩)
     return TRUE
  else
     return FALSE
```

We prove this reduction correct as follows:

$\Longrightarrow$ Suppose M halts on input w.
Then M' accepts all strings in Σ^*.
So **Decide374** accepts the encoding $\langle M' \rangle$.
So **DecideHalt** correctly accepts the encoding $\langle M, w \rangle$.

$\Longleftarrow$ Suppose M does not halt on input w.
Then M' diverges on *every* input string x.
In particular, M' does not accept any string (and especially not 374 of them).
So **Decide374** rejects the encoding $\langle M' \rangle$.
So **DecideHalt** correctly rejects the encoding $\langle M, w \rangle$.

In both cases, **DecideHalt** is correct. But that's impossible, because Halting is undecidable. We conclude that the algorithm **Decide374** does not exist.

7 (15 PTS.) YELLOW STREET NEEDS BITS.

There are n customers living on yellow street in Shampoo-banana. Yellow street is perfectly straight, going from south by southeast to north by northwest. The ith customers lives in distance s_i meters from the beginning of the street (i.e., you are given n numbers: $0 \leq s_1 < s_2 < \cdots < s_n$). A new internet provider `Bits4You` is planning to connect all of these customers together using wireless network. A base station, which can be placed anywhere along yellow street, can serve all the customers in distance r from it.

The input is $s_1, s_2, \ldots, s_n, r$. Describe an efficient algorithm to computes the *smallest* number of base stations that can serve all the n customers. Namely, every one of the n customers must be in distance $\leq r$ from some base station that your algorithm decided to build. Incorrect algorithms will earn few, if any points. (Your algorithm output is just the number of base stations – there is no need to output their locations.)

Prove the correctness of your algorithm. What is the running time of your algorithm?

Solution:

This is a classical greedy algorithm problem. First, break the points into groups by scanning the locations in increasing order. Starting at s_1, let s_{i_1} be the first location that is in distance $> 2r$ from s_1. Continue scanning, and let s_{i_2} to be the first location that is in distance larger than $2r$ from s_{i_1}. In the jth iteration, set $s_{i_{j+1}}$ be the first location that is in distance larger than $2r$ from s_{i_j}. Let s_{i_k} be the last such location marked.

This breaks the location into groups

$$
\begin{aligned}
I_1 &= \{s_1, \ldots, s_{i_1 - 1}\} \\
I_2 &= \{s_{i_1}, \ldots, s_{i_2 - 1}\} \\
&\ldots \\
I_k &= \left\{s_{i_{k-1}}, \ldots, s_{i_k - 1}\right\} \\
I_{k+1} &= \{s_{i_k}, \ldots, s_n\}.
\end{aligned}
$$

For each interval I_j, we place a base station at the location $m_j = (\mathrm{left}(I_j) + \mathrm{right}(I_j))/2$, where left and right are the two extreme locations in the interval I_j.

Running time. Since the number are presorted, the running time is $O(n)$.

Correctness. Let $o_1 < o_2 < \cdots < o_u$ be the optimal solution, and assume for the sake of contradiction that $u < k + 1$. We can safely assume that a client is assigned to its nearest base station, which implies that the optimal solution breaks the clients into groups $O_1, O_2, \ldots, O_u$, that are sorted from left to right. If $o_1 < m_1$, then we can set $o_1 = m_1$ – this would not make the solution any worse. Similarly, if $o_1 > m_1$, then we can set $o_1 = m_1$ (again), since O_1 can not contain any client from I_2. We now repeat this argument, inductively, moving the locations of the optimal solution to the greedy algorithm locations. Clearly, if $u < k + 1$, then the clients in I_{k+1} can not be served by the new solution (which has the same coverage as the optimal solution), which is a contradiction.