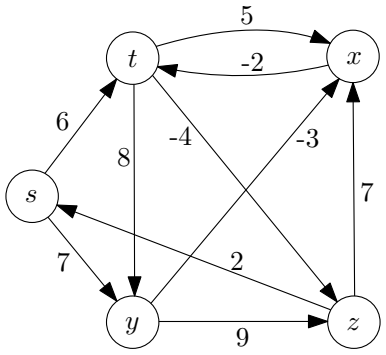
Pre-tutorial questions

Do you know the basic concepts of this week's lecture content? These questions are only to test yourself. They will not be explicitly discussed in the tutorial, and no solutions will be given to them.

1. Dynamic programming technique
   (a) What are the three main ingredients in developing a dynamic programming algorithm?
   (b) What is the standard technique to prove correctness of a dynamic programming algorithm?
2. RNA secondary structure
   (a) What are the subproblems?
   (b) What is the recurrence? Do you understand the recurrence?
   (c) What are the base cases?
3. Shortest path
   (a) What are the subproblems?
   (b) What is the recurrence? Do you understand the recurrence?
   (c) What are the base cases?
   (d) Why do we use dynamic programming instead of using Dijkstra's algorithm?

Tutorial

Problem 1
Run the Bellman-Ford algorithm on the directed graph below. Use vertex z as the destination and illustrate how *first* changes throughout the execution.



Solution: Use vertex z as the destination and the weights as shown in the figure

$i = 1$: Only need to consider edges ending in z; the rest are $\infty+? = \infty$. $(t, z) \to -4$, $(y, z) \to 9$

$i = 2$:
- For s, check $M[1, t] + 6 = 2$ and $M[1, y] + 7 = 16$.
- For t, check $M[1, y] + 8 = 17$, $M[1, x] + 5 = \infty$ and $M[1, z] + (-4) = -4$.
- For x, check $M[1, t] + (-2) = -6$.
- For y, check $M[1, x] + (-3) = \infty$ and $M[1, z] + 9 = 9$.

$i = 3$:
- For s, check $M[2, t] + 6 = 2$ and $M[2, y] + 7 = 16$.
- For t, check $M[2, y] + 8 = 17$, $M[2, x] + 5 = -1$ and $M[2, z] + (-4) = -4$.
- For x, check $M[2, t] + (-2) = -6$.
- For y, check $M[2, x] + (-3) = -9$ and $M[2, z] + 9 = 9$.

$i = 4$:
- For s, check $M[3, t] + 6 = 2$ and $M[3, y] + 7 = -2$.
- For t, check $M[3, y] + 8 = -1$, $M[3, x] + 5 = -1$ and $M[3, z] + (-4) = -4$.
- For x, check $M[3, t] + (-2) = -6$.
- For y, check $M[3, x] + (-3) = -9$ and $M[3, z] + 9 = 9$.

M	0	1	2	3	4
z	0	0	0	0	0
s	∞	∞	$\infty \to 2$	2	$2 \to -2$
t	∞	-4	-4	-4	-4
x	∞	∞	$\infty \to -6$	-6	-6
y	∞	9	9	$9 \to -9$	-9

-	0	1	2	3	4	first
z	NIL	NIL	NIL	NIL	NIL	NIL
s	NIL	NIL	t	t	t	t
t	NIL	z	z	z	z	z
x	NIL	NIL	t	t	t	t
y	NIL	z	z	x	x	x

So we have:
Shortest path from s to z is: stz
Shortest path from t to z is: tz
Shortest path from x to z is: xtz
Shortest path from y to z is: $yxtz$

Problem 2
A palindrome is a string that reads the same left to right as right to left. Given a string A of length n over some alphabet Σ, your task is to design $O(n^2)$ time algorithm that will delete the fewest characters from A so that what remains of the string is a palindrome. For example

$$A \quad D \quad B \quad C \quad D \quad B \quad C \quad A,$$

can be turn into

$$A \quad D \quad C \quad D \quad A,$$

by deleting only three characters.

Solution: (Sketch) Let $M[i, j]$ be the longest subsequence of $A[i, j]$ that is a palindrome. The base case is $M[i, i] = 1$ and $M[i, i-1] = 0$ for all i. For the recursive case we check if $A[i] = A[j]$ if so, there is no harm in assuming that they will not be deleted, otherwise, the need to delete either i or j:

$$M[i,j] = \begin{cases} M[i+1, j-1] + 2 & \text{if } A[i] = A[j] \\ \max(M[i+1, j], M[i, j-1]) & \text{if } A[i] \neq A[j] \end{cases}$$

There are $\binom{n}{2}$ DP states in total. Each state takes $O(1)$ time to compute, so the overall running time is $O(n^2)$.

Problem 3

Consider the set of weighted intervals given below, where s_i is the start time, f_i is the finish time, and v_i is the value of the interval.

j	s_j	f_j	v_j	$p(j)$
1	0	6	2	0
2	2	10	4	0
3	9	15	6	1
4	7	18	7	1

Solve this instance of the weighted interval scheduling problem, i.e. find a set of (non-conflicting) intervals with maximum total weight.

Solution: Note that the intervals are already sorted by their finish time, and the $p(j)$ values have already been calculated for each interval. Recall that the value $p(j)$ is the largest index of an interval which is compatible with interval j, i.e. interval $p(j)$ could be scheduled together with interval j, but not interval $p(j) + 1$ as that would conflict with interval j.
$Opt(j)$ is defined to be the best solution that is obtainable using intervals $1, \ldots, j$, where we define $Opt(0) = 0$. Then we have that

$$Opt(j) = \max\{Opt(j-1), v_j + Opt(p(j))\}.$$

Therefore, we have
$Opt(0) = 0$
$Opt(1) = \max\{Opt(0), v_1 + Opt(p(1))\} = 2$
$Opt(2) = \max\{Opt(1), v_2 + Opt(p(2))\} = \max\{2, 4 + 0\} = 4$
$Opt(3) = \max\{Opt(2), v_3 + Opt(p(3))\} = \max\{4, 6 + 2\} = 8$
$Opt(4) = \max\{Opt(3), v_4 + Opt(p(4))\} = \max\{8, 7 + 2\} = 9$
So there is a schedule that gives us a weight (or value) of 9. Its easy enough to see, in this case, that we schedule intervals 1 and 4 to get this value. In general, if you want to reconstruct the actual schedule that will achieve the maximum value, then you start with the value of $Opt(n)$ (for n intervals) and determine if $Opt(n) = Opt(n-1)$ or if $Opt(n) = v_n + Opt(p(n))$. In the first case, interval n is not used and then we proceed to the value $Opt(n1)$ and do the same to continue finding the schedule. In the second case, interval n is scheduled, and then we proceed to examine task $p(n)$ (and the value of $Opt(p(n))$) to determine the next (i.e. previous) interval to schedule.

Problem 4

Let $G = (V, E)$ be a connected undirected graph. Given two vertices s and t we can compute the shortest path (shortest in terms of number of edges) in linear time using BFS. It is easy to come up with examples where

the there could be multiple shortest paths going from s to t. Design a dynamic programming algorithm to compute the number of shortest paths connecting s and t. Notice that the number of shortest paths connecting s and t may be exponentially large, so we don't want to list them, just count them.

Solution: First compute the distances from s to every other node in the graph using BFS, let $\text{dist}(u)$ be the length of the shortest path (in terms of number of edges) from s to u. We define $M[u]$ to be the number of shortest s-u paths in the graphs. We define $M[s] = 1$ as we see the vertex s by itself as a path from s to s having length 0. For vertices u at distance $k > 0$ from s, every shortest s-u path can arrive at u through some vertex v such that $\text{dist}(v) = k - 1$ and v and u are connected by an edge. This observation leads to the following recurrence:

$$M[u] = \sum_{\substack{v:(v,u)\in E \\ \text{dist}(v)=\text{dist}(u)-1}} M[v].$$

Suppose the graph has n vertices and m edges. There are n DP states and each takes $O(n)$ to fill in the worst case; thus, the algorithm runs in $O(n^2)$. A sharper bound on the running time is possible if we are more careful about how much time we spend at each node. Assuming the graph is represented with adjacency lists, computing $M[u]$ takes $O(\deg(u))$ time; thus, the overall all time to compute all M-values is $O(\sum_u \deg(u)) = O(m)$.

Problem 5

You are given a string with n characters. The string comes from a corrupted text where all white spaces have been deleted (so it looks somethings like "thefoxjumpedoverthelazydog"). Suppose that you are given a function `lookup(w)` that takes as input a some string `w` and return `True` iff `w` is a valid word.

Design an algorithm based on dynamic programming to test whether it is possible to insert spaces into the input string to obtain a valid text (we don't care about meaning.)

Solution: Let s be the input string, and let $s[i, j]$ be the substring consisting of characters in positions i through j. Let $M[i]$ be `True` if it is possible to insert spaces into $s[1, i]$ to obtain a valid text. Then we can define the base case as $M[0] =$ `True` by regarding the empty string as a valid text. Then the recursive case is

$$M[i] = \bigvee_{1\le j<i} \text{lookup}(s[j, i]) \wedge M[j - 1]).$$

There are n states and each takes $O(n)$ time to fill. The overall time is $O(n^2)$.

Problem 6

Suppose you are given n biased coins $h_1, \ldots, h_n$; here h_i is the probability that the ith coin comes up heads. Consider the following random experiment: Flip all n coins and let X be the number of heads. Define p_i to be the probability that $X = i$. Design an efficient algorithm to compute p_i for $i = 0, \ldots, n$.

Solution: Let $M[i, j]$ be the probability that flipping the first j coins yields i heads. For $j = 1$ we get $M[0, 1] = (1 - h_0)$ and $M[1, 1] = h_0$. For $j > 1$, we can define $M[j, j] = h_1 h_2 \ldots h_j$. For $j > 1$ and $i < j$ the event that we flip i heads can be divided into two subcases: that the jth coin came up heads (in which case we need $i - 1$ heads from the first $j - 1$ coins, or that the jth coin came up tails (in which case we need i heads from the first $j - 1$ coins. The two events are mutually exclusive, so we get the following recurrence

$$M[i, j] = h_j M[i - 1, j - 1] + (1 - h_j)M[i, j - 1].$$

In order for the recurrence to work for the boundary case i, j we can define $M[i, i - 1] = 0$. There are n^2 states and each takes $O(1)$ time to compute. Thus, the overall time is $O(n^2)$.

Problem 7

In the game of Nim there three heaps of toothpicks on a table and two players take turns to remove toothpicks. In her turn, a player can remove any number of toothpicks from any single heap. The player that removes the last toothpick from the table wins.

A game configuration is captured by a triplet (a, b, c) denoting how many toothpicks are there in each heap. Some configurations are winning for the first player in the sense that it does not matter what the second player does, there is always a way for the first player to win. Similarly, other configurations are losing in the sense that it does not matter what the first player does, there is always a way for the second player to win.

Design an $O(a^2b^2c^2)$ time algorithm that given a triplet (a, b, c) tests whether it is a winning or a losing configuration.

Solution: Let $M[a, b, c]$ be True if (a, b, c) is a winning configuration and False otherwise. Clearly, (a, b, c) is winning if the first player can remove a certain number of toothpicks from a heap so as to land in a losing configuration. That is,

$$M[a,b,c] = \left(\bigvee_{i=1}^{a} \neg M[a-i,b,c]\right) \vee \left(\bigvee_{i=1}^{b} \neg M[a,b-i,c]\right) \vee \left(\bigvee_{i=1}^{c} \neg M[a,b,c-i]\right).$$

The base case is given by $M[0, 0, 0] =$ False. This makes perfect sense, as the game could be equivalently defined as saying that you lose if there are no more toothpicks left on the table when your turn comes.

There are abc DP states. Each takes $O(a+b+c)$ time to fill. Therefore, the overall running time is $O(a^2b^2c^2)$.

Problem 8

[Advanced:] Your task is to design a faster DP-based algorithm for determining whether a given Nim configuration (a, b, c) is winning. Your algorithm should run in $O(abc)$ time.

Solution: (Sketch.) Recall that $M[a, b, c]$ was defined to be True iff there a winning strategy from configuration (a, b, c). Now let $A[a, b, c]$ to be true iff there is a winning strategy that involves removing matching from the first heap (the one with a matches). Define $B[a, b, c]$ and $C[a, b, c]$ in a similar way for the second and third heaps.

Clearly, $M[a, b, c] = A[a, b, c] \vee B[a, b, c] \vee C[a, b, c]$, so given those values we can compute $M[a, b, c]$ is $O(1)$ time. We give a recurrence for A, the recurrences for B and C can be derived in the same way. For the base case we set $A[0, b, c]$ to be truth value of $b \neq c$. For $A[a, b, c]$ for $a > 0$ we can define $A[a, b, c] = \neg M[a - 1, b, c] \vee A[a - 1, b, c]$, so given those value we can compute $A[a, b, c]$ in $O(1)$ time.

There are $O(abc)$ DP states. Each takes $O(1)$ time to fill. Therefore, the overall running time is $O(abc)$.

Problem 9

[Advanced:] Balanced Partition. Suppose that you have a set, $A = \{a_1, \ldots, a_n\}$, of n integers, each of which is between 0 and K (inclusive). Your goal is to find a partition of A into two sets S_1 and S_2 (so $S_1 \cup S_2 = A$ and $S_1 \cap S_2 = \emptyset$) that minimizes

$$|sum(S_1) - sum(S_2)|,$$

where $sum(S_i)$ equals the sum of the values in the set S_i.

Hint: Define $V(i, j) = 1$ if there is some subset of $\{a_1, \ldots, a_i\}$ that sums up to the value j (i.e. some collection from the first i integers sums up to j), otherwise set $V(i, j) = 0$.

- How many different values of $V(i, j)$ do you need to compute (in terms of n and K)?
- How can you compute the $V(i, j)$ values if you know all of the $V(i-1, j)$ values?
- How can you use the collection of $V(i, j)$ values to answer the original question (and find the minimum value of $|sum(S_1) - sum(S_2)|$)?

Solution:

- Since there are n different numbers, each of which is at most K, the sum of all of the n numbers is at most nK (i.e. j can range from 0 to nK for each value of i). Hence, there are at most $O(n^2K)$ values $V(i, j)$ to compute. (This is effectively the running time to find a solution to this Balanced Partition problem.)
- For $i = 1$, we have $V(1, a_1) = 1$ and $V(1, j) = 0$ for $j \neq a_1$. For $i \geq 2$, how can we find $V(i, j)$ given that we have all of previously computed values (i.e. all of the $V(1, j), V(2, j), \ldots, V(i-1, j)$ values for all j)? First of all, we have $V(i, j) = 1$ if $V(i-1, j) = 1$ or if $V(i-1, j-a_i) = 1$. (Why?) Otherwise, $V(i, j) = 0$.
- After computing all of the $V(i, j)$ values, what do we do? Let $T = \frac{1}{2}\sum_{i=1}^{n} a_i$.

  Then T is half of the sum of all the values in the set A. If T is an integer, we can first check to see if $V(n, T) = 1$. If this is the case, then there is a partition S_1, S_2 such that $|sum(S_1) - sum(S_2)| = 0$. (Why?)

  If T is not an integer, or if T is an integer but $V(n, T) = 0$, then what do we do? For each value j with $V(n, j) = 1$, this means that there are two sets S_1, S_2 such that $sum(S_1) = j$ and $sum(S_2) = 2T - j$. For those particular sets, we have $|sum(S_1) - sum(S_2)| = |j - (2T - j)| = |2j - 2T| = |2T - 2j|$.

  We are trying to minimize that value, so we compute $\min\{|2T - 2j| : V(n, j) = 1$ and $j \leq T\}$ to find this minimum value. (We need only consider the values $j \leq T$ in the minimum because of the symmetry of the problem, i.e. there is a set S_1 with $sum(S_1) = j$ if and only if there is a set S_2 with $sum(S_2) = 2T - j$. Furthermore, we would likely consider the values $V(n, j)$ in decreasing order of j starting from T (or the closest integer to T if it is a fraction) since we want to minimize the difference.) This last part of computing the minimum takes time $O(T) + O(n^2K)$, so overall the whole algorithm can run in time $O(n^2K)$.