Midterm 2 Solutions

1. **(10 points) True/False**

   Clearly put your answers in the answer box in front of each question. Write T for true, F for false. Each problem will be graded as follows: 1 point for each correct answer and 0 point for blank answer, and -0.25 point for each incorrect answer.

   (a) [F] The maximum weight edge in a graph can not be in any MST.
   Consider a graph with only one edge. No matter what happens, it must be in the MST.

   (b) [F] $\log^*(65536) = 5$ (All logarithms are base 2).
   $\log 65536 = 16$, $\log 16 = 4$, $\log 4 = 2$, $\log 2 = 1$, so $\log^*(65536) = 4$.

   (c) [T] The Horn SAT formula $x_1 \implies x_2, \quad x_2 \implies x_1, \quad \overline{x}_1$ is satisfiable.
   Set x_1 be FALSE and x_2 be FALSE.

   (d) [F] An implementation of a dynamic programming problem via memoization does not yield any asymptotic improvements (for any problem instance) in running time with respect to an iterative solution.
   The intended answer was false, because in some instances, memoization can allow you to "skip" certain subproblems. However, this wording was not entirely clear, so any answer was given full credit.

   (e) [T] The knapsack problem (without repetition) on n items where for each $i \in \{1, \cdots, n\}$ the weight of the i^{th} item is i can be solved in time polynomial in the input length.
   Observe that in this case, W is bounded. The largest W possible is if we add up all n weights: $W \leq 1+2+...+n \rightarrow W \in O(n^2)$. The runtime of knapsack without repetition is $O(nW) = O(n^3)$. The input length is the number of bits needed to represent the input (both the weight array and W). So the input length is $\log 1 + \log 2 + ... + \log n + \log W = O(n \log n) + \log W = O(n \log n) + O(\log n^2) = O(n \log n)$. So our runtime is polynomial in n, and also anything greater than n, like $n \log n$.

   (f) [F] The region satisfying the equations $0 \leq x \leq 1, 0 \leq y \leq 1$, and $y \leq x^2$ is convex.
   $y = x^2$ is a convex function when $0 \leq x \leq 1$ and $0 \leq y \leq 1$. That implies its epigraph (region above $y = x^2$) is convex. Therefore the region below $y = x^2$ is concave.

   (g) [F] In a dynamic programming solution, the asymptotic space requirement is always at least as big as the number of unique subproblems.
   Consider the edit distance algorithm; we only need to store two columns of computation at a time, making the space requirement $O(n)$, while the runtime is $O(n^2)$.

   (h) [T] In a dynamic programming solution, the asymptotic time complexity is always at least as big as the number of unique subproblems.
   In the worst case, you need to evaluate all subproblems.

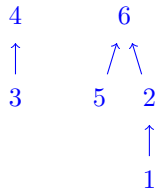
(i) [T] There exists a polynomial time algorithm to find the an approximate minimum set cover (on n items) of size at most $\log_2 n$ times the size of the optimal set cover.
In the lecture we learned a greedy approximation algorithm that gives the upper bound $\ln n$. Since $\ln n = \log_2 n / \log e < \log_2 n$, $\log_2 n$ is still the correct upper bound.

(j) [F] Huffman encoding can always be used to reduce the size of any document. Consider a document where there is only one character.

2. **(4 points)** Show the tree structure in the union-find algorithm as the following sequence of commands is executed. Use union-by-rank and path compression.

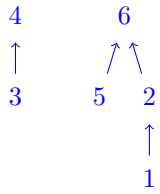
(a)
```
for i = {1, ··· , 6} {
      MakeSet(i);
}
union(1, 2);
union(3, 4);
union(5, 6);
union(1, 6);
```

There are many different trees depending on how you break ties. One possible structure is:



(b)
```
find(6);
```

Again many different correct answers, depending on what the student put for the previous part. Building upon our previous structure, then after this operation the structure would remain unchanged:

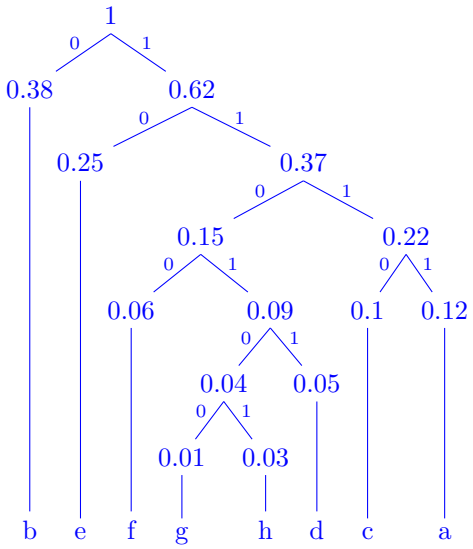


3. **(5 points)** Find the Huffman encoding for the following alphabet and set of frequencies. Feel free to draw the tree, but we will only grade the codewords.

$$\{(a, 0.12), (b, 0.38), (c, 0.1), (e, 0.25), (f, 0.06), (d, 0.05), (g, 0.01), (h, 0.03)\}$$

When you build up your Huffman tree, you should place the branch of lower weight on the left. A left or right branch should respectively correspond to a 0 or 1 in the codeword. Fill the table below with your *final* answers. We will only grade this table. Feel free to use the rest of the page as scratch space.

Symbol	Codewords
a	1111
b	0
c	1110
d	11011
e	10
f	1100
g	110100
h	110101



4. **(10 points)** Briefly describe an efficient algorithm (and report the running time) for finding a Minimum Spanning Tree in an undirected, connected graph $G = (V, E)$ when the edge weights satisfy:

   (a) **(4 points)** For all $e \in E$, $w_e = 1$.

   Main idea of your algorithm (no pseudocode).

   The key idea here is that any tree which connects all nodes is an MST. We can run DFS and take the DFS tree. You could also take a BFS tree, or run Prim's algorithm with a queue or stack instead of a priority queue (this would be equivalent to BFS/DFS).

Unfortunately, a modified Kruskal's will be slightly slower, because even if we don't need to sort edges, the union-find operations will take additional time.

Time complexity of your algorithm. (Put your justification under the box)

$\Theta(|E|)$ since we just have to do a DFS. Note that $|E| \geq |V|$ since we know the graph is connected, so we can leave out the $|V|$ term.

A Kruskal's approach, without sorting, will be $O(|E| \log^* |V|)$. Checking membership in the same connected component using union-find operations takes amortized $O(\log^*(|V|))$ time. Prim's only keeps track of membership in one connected component, so it doesn't suffer from this.

(b) **(6 points)** For all $e \in E$, $w_e \in \{1, 2\}$ (In other words, all edge weights are either 1 or 2).

Main idea of your algorithm (no pseudocode).

Remove weight 2 edges from the graph so only weight 1 edges remain. Now run an algorithm from part (a) as far as possible (e.g. find a DFS forest). We will have some number of connected components. Use these connected components as nodes in a new graph G'. Look at the weight 2 edges in G. For each edge, if the nodes containing the two endpoints are not already connected in G', add an edge between the two containing nodes in G'. Now we can run our algorithm from part (a) again to complete the MST.

Alternate solutions:

Prim's approach:

Run Prim's algorithm with a specialized priority queue, comprised of 2 regular queues. When we add items to the priority queue, we add it to the first queue if the weight is 1 and otherwise add it to the second queue (the weight must be 2). When we pop from the priority queue, we take from the first queue, unless it is empty, in which case we take from the second.

Kruskal's approach (slightly inefficient):

We can use a linear time sort (e.g. counting sort) to sort the edges, then run Kruskal's algorithm.

It would be essentially equivalent to delete all weight 2 edges from the graph, then run an algorithm from part (a) (e.g. find a DFS forest). Since this may not completely connect the graph, we can initialize a union-find data structure with the already connected components, then run Kruskal's algorithm only considering the weight 2 edges (no need to sort) to complete the tree.

Time complexity of your algorithm. (Put your justification under the box)

The first described solution will have running time $\Theta(|E|)$, as we rely on just two calls to our algorithm in part 1, and the graph processing and construction of G' can be done in linear time. Constructing the MST in the original graph from our result on G' may be a little tricky, but we can still do this in linear time by keeping a pointer to the original edge when we add the edge in G'.

The Prim's approach has running time $\Theta(|E|)$, as each priority queue operation can be done in constant instead of log time.

The Kruskal's approach has running time $O(|E| \log^* |V|)$ due to union-find operations.

5. **(6 points) Linear Programming Problem** Given m different types of food, $F_1, ..., F_m$, that supply varying quantities of the n nutrients, $N_1, ..., N_n$, that are essential to good health.

   - Let c_j be the minimum daily requirement of nutrient, N_j.
   - Let b_i be the price per unit of food, F_i.
   - Let a_{ij} be the amount of nutrient N_j contained in one unit of food F_i.

   Write a linear program to plan a meal that supplies all the required nutrients at minimum cost. Make sure to define the variables you use in the linear program.

(a) **(2 points)** Define the LP variables and what they stand for.
x_i denote the number of units of food F_i in the meal plan.

(b) **(4 points)** Write the objective function and constraints.

Construct the following linear program

$$\min_x \sum_{i=1}^{m} b_i x_i$$

subject to

$$\forall j \in \{1, 2, \ldots, n\} \quad \sum_{i=1}^{m} a_{ij} x_i \geq c_j \quad (1)$$

$$\forall i \in \{1, 2, \ldots, m\} \quad x_i \geq 0 \quad (2)$$

Constraint (1) forces the meal plan to meet the minimum daily requirement of each nutrient N_j. Constraint (2) ensures that the meal plan contains a non-negative amount of each type of food F_i.

6. **(11 points) Coin Game.** Two players play a game. There is a shared array of coins $A[a_1..a_N]$. Players alternate turns to pick either the leftmost or rightmost element of the array. They may not choose an element anywhere in the middle. Once a coin has been picked, it is removed from the array. A player's "score" is the sum of a_i values of the coins that the player picks. Assuming both players play optimally, what is the maximum score for the starting player?

(a) **(3 points)** Define your subproblem. (Your subproblem should not be more than 3 lines long.)
Let $S(i, j)$ be the maximum score the current player can obtain if the player is left with coin array $A[i..j]$.

A player's decision will be affected by the optimal choice by both players somewhere further into the game. Thus we want to look at the remaining elements of the array further on.

(b) **(6 points)** Write down the recurrence relation for your subproblems in the box below. (Put your justification under the box.)
Let L_i be the largest score current player can obtain if player chooses $A[i]$ and R_j be largest possible score if the current player chooses $A[j]$ at this turn.

$$\begin{aligned} S(i, j) &= \max\{L_i, R_j\} \\ L_i &= A[i] + \min\{S(i+2, j), S(i+1, j-1)\} \\ R_j &= A[j] + \min\{S(i+1, j-1), S(i, j-2)\} \end{aligned}$$

By playing optimally, both players will try to pick the coin that will give them the largest possible score.
After the current player picks a coin, the next player will pick the coin that will get him/her to his/her largest possible score. Thus the choice will minimize the the current player's largest possible score. This is a zero-sum game. Playing greedily will maximize a player's profit after every turn, but it may not earn the player's the largest possible score.

Final Answer at $S(1, N)$.

Pseudocode:

for $G \in \{1..N\}$ **do**
  for $i \in \{1..N-G\}$ **do**
    $j := i + G$
    if $i > j$ **then**
      $S(i,j) := 0$
    else if $i = j$ **then**
      $S(i,j) := A[i]$
    else
      $L := A[i] + \min\{S(i+2,j), S(i+1,j-1)\}$
      $R := A[j] + \min\{S(i+1,j-1), S(i,j-2)\}$
      $S(i,j) := \max\{L,R\}$
  end
end
return $S(1,N)$

We see that we solve the subproblems with smaller differences between i and j first.

(c) **(1 point)** What are the base cases? Base Cases:

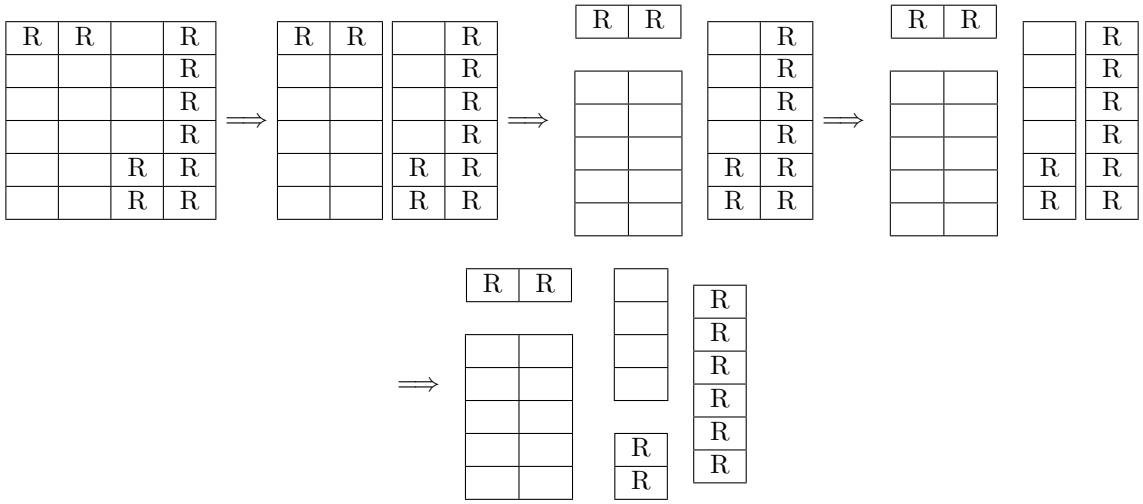
$$S(i,j) = \begin{cases} 0, & \text{if } i > j \\ \max\{A[i], A[j]\}, & \text{if } i+1 == j \\ A[i], & \text{if } i = j \end{cases}$$

We must have base case 3 and either 1 or 2.

(d) **(1 points)** Write down the time complexity. (No need for justification)
$O(N^2)$. Outer loop goes has N iterations. Inner loop has G iterations. Summing up the iterations, we have $1 + 2 + \cdots + N = \frac{N(N+1)}{2} \in O(N^2)$.

7. **(10 points) Breaking Chocolate.** There is a chocolate bar consisting of an $m \times n$ rectangular grid of squares. Some of the squares have raisins in them, and you hate raisins. You would like to *break* the chocolate bar into pieces so as to separate all the squares with raisins, from all the squares with no raisins.

For example, shown below is a 6×4 chocolate bar with raisins in squares marked R. As shown in the picture, one can separate the raisins out in exactly four *breaks*.



(At any point in time, a *break* is a cut either horizontally or vertically of one of the pieces at the time)

Design a DP based algorithm to find the smallest number of breaks needed to separate all the raisins out. Formally, the problem is as follows:

Input: Dimensions of the chocolate bar $m \times n$ and an array $A[i, j]$ such that $A[i, j] = 1$ if and only if the ij^{th} square has a raisin.

Goal: Find the minimum number of breaks needed to separate the raisins out.

(a) **(3 points)** Define your subproblem. (Your subproblem should not be more than 3 lines long.)
We define $B[i_1, j_1, i_2, j_2]$ to be the minimum number of breaks needed to separate the sub-matrix $A[i_1 \leq i_2, j_1 \leq j_2]$ into pieces consisting either entirely of raisin pieces or entirely of non-raisin pieces.

(b) **(6 points)** Write down the recurrence relation for your subproblems in the box below. (Put your justification under the box; There is a lot more space than you need, try to keep the answers as precise and concise as possible)

$$B[i_1, j_1, i_2, j_2] = \min \begin{cases} 0, & \text{if all entries of } A[i_1 \dots i_2, j_1 \dots j_2] \text{ are equal} \\ 1 + B[i_1, j_1, i_1 + k, j_2] + B[i_1 + k + 1, j_1, i_2, j_2] & \text{for any } k \in \{1, \dots, i_2 - i_1\} \\ 1 + B[i_1, j_1, i_2, j_1 + k] + B[i_1, j_1 + k + 1, i_2, j_2] & \text{for any } k \in \{1, \dots, j_2 - j_1\} \end{cases}$$

Alternatively, you could have also encapsulated the 0 base case in all single-square pieces, and determined if a piece was pure via the merging, see below.

(c) **(1 points)** Write down the time complexity in the box below. (No need for justification)
Two answers were accepted: $O((m + n)m^2n^2)$ and $O(m^3n^3)$
We have $O(m^2n^2)$ total subproblems: $O(mn)$ possibilities for (i_1, j_1), and $O(mn)$ possibilities for (i_2, j_2). For each subproblem, we examine up to m possible choices for horizontal splits, and n possible choices for vertical splits. A single split consideration will result in two smaller subproblems, which we can assume have already been solved, so we just need to find the best split, which takes $O(n + m)$ time.
In addition, for a subproblem, we also want to check the base case for if the piece is "pure" (contains only raisins, or contains only non-raisins). Brute force checking this takes $O(mn)$ time, for a total subproblem time of $O(mn + (m + n)) \rightarrow O(mn)$.
However, this $O(mn)$ factor can be reduced to $O(m + n)$, and here is how (didn't need to do it to receive full points). We can precompute the purities of every single possible subrectangle and store it in a table. Technically, for us to pre-compute faster than $O(m^3n^3)$, we'll need to compute the purities more intelligently than by brute-force. It is possible to do this in as fast as $O(\max\{m, n\}^2)$ time, although the details of this are complicated and won't be explained here (feel free to ask). So to solve our recurrence relation, if we can determine purity/impurity in $O(1)$ time, then we can reach an overall time of $O((m + n)m^2n^2)$.
Alternatively, we can initialize all min-break values of single square pieces to be 0. Then, if it is possible to have some break such that both resulting pieces have min-break values of 0, and both resulting pieces are of the same type (raisin-only or non-raisin-only, and we can take any sample of either and compare them), then we ourself are a pure piece. This would allow you to avoid the entire pre-computation business as mentioned before, and still achieve a runtime of $O((m + n)m^2n^2)$.

8. **(15 points) Balloon Popping Problem.** You are given a sequence of n-balloons with each one of a different size. If a balloon is popped then it produces noise equal to $s_{left} \cdot s_{popped} \cdot s_{right}$, where s_{popped} is the size of the popped balloon and s_{left} and s_{right} are the sizes of the balloons to its left and to its right. If there is no balloons to the left then we set $s_{left} = 1$ and similarly, if there are no balloons to the right then we set $s_{right} = 1$, while calculating the noise produced.

After popping a balloon, the balloons to its left and right become neighbors. (Note that the total noise produced depends on the order in which the balloons are popped.)

Design a dynamic programming algorithm to compute the the maximum noise that can generated by popping the balloons.

Example:
Input (Sizes of the balloons in a sequence): (4) (5) (7)
Output (Total noise produced by the optimal order of popping): 175

Walkthrough of the example:

- **Current State** (4) (5) (7)
  Pop Balloon (5)
  Noise Produced $= 4 \cdot 5 \cdot 7$

- **Current State** (4) (7)
  Pop Balloon (4)
  Noise Produced $= 1 \cdot 4 \cdot 7$

- **Current State** (7)
  Pop Balloon (7)
  Noise Produced $= 1 \cdot 7 \cdot 1$

- **Total Noise Produced** $= 4 \cdot 5 \cdot 7 + 1 \cdot 4 \cdot 7 + 1 \cdot 7 \cdot 1$.

(a) **(4 points)** Define your subproblem. (Your subproblem should not be more than 3 lines long.)
Similar to Chain Matrix Multiplication, we want to form subtrees $C(i, j)$ representing the *maximum* amount of noise produced by popping balloons in the sublist $i, i+1, \ldots j$ first from all the balloons. The smallest subproblem is when there is only 1 balloon to pop. The size of the subproblem is the number of multiplications.

(b) **(1 point)** What are the base cases?
Base case: When the size of sublist to pop out is only one balloon s_{popped}, return the noise $s_{left} \cdot s_{popped} \cdot s_{right}$. In addition, we need to initialize $s_0 = 1$ and $s_{n+1} = 1$ assuming the input is from 1 to n. In our algorithm we'll never actually pop them but we need dummy balloons to the left and right.

(c) **(4 points)** Write down the recurrence relation for your subproblems in the box below. (Put your justification under the box; There is a lot more space than you need, try to keep the answers as precise and concise as possible). **Don't miss part (d) on the next page!**

$$C(i, j) = \max_{i \le k \le j} \{C(i, k-1) + C(k+1, j) + s_{i-1} \cdot s_k \cdot s_{j+1}\}$$

Justification: The leaves of the tree represent the last balloon being popped, so here, k represents the index of the last balloon being popped. Then we can recurse up, and at each level find the splitting point k that maximizes the value of the subtree (noise produced for that sequence). Runtime of this algorithm is $O(n^3)$.

Note (Minor Mistake): There is a slight different with CMM in that the recursion uses $C(i, k-1)$ instead of $C(i, k)$. This is because in CMM the first dimension of the "middle" matrix (that we just computed) and the last dimension of the left matrix are the same. In balloon

popping, the k^{th} balloon was just popped and is now gone.

Note (Minor Mistake): The subproblem iterates $i \leq k \leq j$. This is different from CMM where the iteration is $i \leq k < j$. This is because with k matrices you keep the last dimension, but with k balloons you need to pop them all.

(d) **(6 points)** Write down your pseudocode.

Pseudocode:

$s_0 = 1$
$s_{n+1} = 1$
for $i \in \{1 \dots N\}$ **do**
    $C(i, i) = s_{i-1} \cdot s_i \cdot s_{i+1}$
end
for $s \in \{1 \dots N\}$ **do**
    for $i \in \{1 \dots N - s\}$ **do**
        $j = i + s$
        $C(i, j) = \max_{i \leq k \leq j} C(i, k-1) + C(k+1, j) + s_{i-1} \cdot s_k \cdot s_{j+1}$
    end
end
return $C(1, n)$