Midterm 2

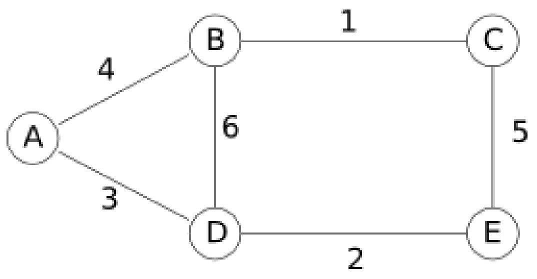
Name / SID:

TA / Section:

Answer all questions. Read them carefully first. Be precise and concise. The number of points indicate approximately the amount of time (in minutes) each problem is worth spending. Write in the space provided, and use the back of the page for scratch. **Good luck!**

1	
2	
3	
4	
5	
Total	

Problem 1 (*16 points total - 4 points each*)



In the graph shown above:

(a) In what order are the vertices deleted from the priority queue in Dijkstra's algorithm for shortest paths? (Starting at node A)

Solution: A, D, B, C, E or A, D, B, E, C (since C and E are tied in the next to last iteration)

(b) In Prim's algorithm for the minimum spanning tree? (Starting at node A)

Solution: A, D, E, B, C

(c) In what order are the edges added in Kruskal's algorithm?

Solution: BC, DE, AD, AB

(d) Show the union-find trees at the end of Kruskal's algorithm (in case of a tie in rank, make the alphabetically first node the root).

Solution: We accepted two answers for this, depending on (1) whether or not you used path compression and (2) whether you used a non-optimized Kruskal's algorithm (i.e. the one from the textbook that perform a "find" operation on every edge, regardless of whether or not the MST has already been found) or an optimized version of Kruskal's (which stops once the MST has $|V| - 1$ edges). In all cases, B was at the top, pointed to by C and D. D was pointed to by A. E could have pointed to either D (for the optimized version of Kruskal's OR if path compression was not used) or to B (if the non-optimized book version of Kruskal's AND path compression were used). Note that for the latter solution, the path compression only takes effect on a "find" operation. After the final unioning, there is not a "find" operation that will compress A's path (we saw many wrong answers that attempted to compress A's path). Instead, you would have called find(E), which would compress E's path to point to B.

Problem 2 (*16 points total - 2 points each*)

True or false? Provide a **very brief** explanation (1 to 3 sentences). The space provided should suffice.

1. In Huffman's algorithm the character with the highest probability (all probabilities are unique) is guaranteed to be one of the leaves that is closest to the root (i.e it has the least depth among all leaves).

   Solution: True. If not, you could swap it with one that is closer to the root and get a more efficient encoding. Likewise, Huffman works by combining leaves in order from least probability to greatest, so the one with the highest probability would be added to the tree last. Since the tree is built from the bottom up, this means it would be closest to the root among all leaves.

2. In the greedy algorithm for Horn clauses, we start with all variables set to true.

   Solution: False. The algorithm is stingy, initially sets all variables to false, and only reluctantly sets variables to true when there is no other alternative.

3. Even without path compression (but still using union-by-rank), any operation in the union-find data structure takes at most $O(\log n)$ time.

   Solution: True. The depth of the tree maintained by union-by-rank is at most $\log n$. Since the runtime of union and find is upper bounded by the height of the tree, they will take at most $O(\log n)$ time. Note that the tree is not necessarily a binary tree.

4. In the standard implementation of the union find data structure (with union-by-rank and path compression), any operation in the union-find data structure takes at most $O(\log^* n)$ time.

   Solution: False. The amortized time is $O(\log^* n)$ time, which means after all of the operations, each one took on average $O(\log^* n)$, but there may have been single operations that took more time.

5. The simplex algorithm for linear programming is a polynomial-time algorithm.

   Solution: False. In the worst case, it is exponential. If you aren't familiar with the difference between linear, polynomial, or exponential time, come to someone's office hours.

6. At the conclusion of the max-flow algorithm, at least one edge coming out of the source S is full to capacity.

   Solution: False. Consider: S —10— A —1— T. Max flow is 1, but SA is not maxed out.

7. Consider a minimum capacity s-t cut, which partitions the vertices of a graph into two sets S and T, with $s \in S$ and $t \in T$. Then any maximum s-t flow saturates/fills to capacity all edges crossing from S to T.

   Solution: True. This is the max-flow, min-cut theorem.

8. Consider a minimum capacity s-t cut, which partitions the vertices of a graph into two sets S and T, with $s \in S$ and $t \in T$. Then increasing the capacity of any edge crossing from S to T increases the maximum s-t flow.

   Solution: False. There may be several minimum cuts. Increasing the capacity of an edge on one of them might not increase all minimum cuts, and the current cut might no longer be the minimum.

Problem 3 (*14 points total*)

(a) (*7 points)* We have a factory that produces two types of xylophones (a musical instrument), named x_1 and x_2. When we sell them, we earn a net profit of \$3 for each unit of x_1 and \$2 for each unit of x_2. We produce these items using two machines, A and B. To produce one unit of x_1, we require 5 minutes on machine A and 3 minutes on machine B. To produce one unit of x_2, we require 6 minutes on machine B. Due to union labor rules, the machine operators can only run machine A for 10 minutes per day and machine B for 12 minutes per day. Since xylophones are awesome, assume that we can sell out all units of x_1 and x_2 produced, and that we can produce and sell fractional amounts of x_1 and x_2. Write a linear program that finds the maximum profit that can be obtained per day. Find the optimal number of units of x_1 and x_2 to produce per day and the maximum profit it obtains per day.

Solution: We actually accepted two interpretations of the question as reasonable. In the first interpretation, we assumed that to produce one unit of x_1 that it would take 5 minutes of machine A AND 3 minutes on machine B.

So, we choose to have two variables: x_1 representing the number of x_1's that are produced, and x_2 representing the number of x_2's that are produced. So, since we want to maximize profits, our objective function is:

$$\max 3x_1 + 2x_2 \tag{1}$$

We have two constraints - the amount of time we can spend on machine A and the amount of time that we can spend on machine B; these become:

$$\begin{aligned} 5x_1 &\leq 10 \qquad (2)\\ 3x_1 + 6x_2 &\leq 12 \qquad (3) \end{aligned}$$

Finally, we have the positivity constraint for x_1 and x_2:

$$x_1, x_2 \geq 0 \tag{4}$$

Combining all of these together, our linear program is:

$$\max 3x_1 + 2x_2 \tag{5}$$

$$\begin{aligned} 5x_1 &\leq 10 \qquad (6)\\ 3x_1 + 6x_2 &\leq 12 \qquad (7)\\ x_1, x_2 &\geq 0 \qquad (8) \end{aligned}$$

From here, were we to graph out our region of feasible solutions, we would see that we have vertices at the points $(x_1, x_2) = (0,0), (2,0), (0,2), (2,1)$; we know that our optimal solution must be at one of these vertices, and by inspection we see that it is $(2,1)$. So, the optimal solution is to build 2 x_1's and 1 x_2, for a profit of 8.

In the second interpretation, we assumed that to produce one unit of x_1 that it would take 5 minutes of machine A OR 3 minutes on machine B.

So, we choose to have three variables: x_{1A} representing the number of x_1's that are produced on machine A, x_{1B} representing the number of x_1's that are produced on machine B, and x_2 representing the number of x_2's that are produced on machine B. Once again, we want to maximize profits, so our objective function is:

$$\max 3x_{1A} + 3x_{1B} + 2x_2 \tag{9}$$

Once again we have two constraints - the time we can spend on machine A and B respectively:

$$\begin{aligned} 5x_{1A} &\leq 10 && (10)\\ 3x_{1B} + 6x_2 &\leq 12 && (11)\\ & && (12) \end{aligned}$$

Finally, we have the positivity constraints for x_{1A}, x_{1B}, and x_2:

$$x_{1A}, x_{1B}, x_2 \geq 0 \tag{13}$$

Combining all of these together, our linear program is:

$$\max 3x_{1A} + 3x_{1B} + 2x_2 \tag{14}$$

$$\begin{aligned} 5x_{1A} &\leq 10 && (15)\\ 3x_{1B} + 6x_2 &\leq 12 && (16)\\ x_{1A}, x_{1B}, x_2 &\geq 0 && (17) \end{aligned}$$

From here, were we to graph out our region of feasible solutions, we would see that we have vertices at the points $(x_{1A}, x_{1B}, x_2) = (0,0,0), (0,4,0), (0,0,2), (2,0,0), (2,4,0), (2,0,2)$; we know that our optimal solution must be at one of these vertices, and by inspection we see that it is $(2,4,0)$. So, the optimal solution is to build 6 x_1's, for a profit of 18.

(b) (*7 points)* Write the dual of this linear program. What is the value of its optimal solution?

Solution: Under our first interpretation, we notice that we could express our linear program as:

$$\max \begin{bmatrix} 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{18}$$

$$\begin{bmatrix} 5 & 0 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} 10 \\ 12 \end{bmatrix} \tag{19}$$

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{20}$$

So, it is a simple transformation to find the dual of this linear program (look at page 208 from the textbook), and we get our dual linear program is:

$$\min \begin{bmatrix} y_1 & y_2 \end{bmatrix} \begin{bmatrix} 10 \\ 12 \end{bmatrix} \tag{21}$$

$$\begin{bmatrix} y_1 & y_2 \end{bmatrix} \begin{bmatrix} 5 & 0 \\ 3 & 6 \end{bmatrix} \geq \begin{bmatrix} 3 & 2 \end{bmatrix} \tag{22}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{23}$$

This becomes:

$$\min 10y_1 + 12y_2 \tag{24}$$

$$\begin{aligned} 5y_1 + 3y_2 &\geq 3 && (25) \\ 6y_2 &\geq 2 && (26) \\ y_1, y_2 &\geq 0 && (27) \end{aligned}$$

By duality, we know the optimal value for this linear program is the same as that for the primal, which is to say 8.

Under our second interpretation, we can express our linear program as:

$$\max \begin{bmatrix} 3 & 3 & 2 \end{bmatrix} \begin{bmatrix} x_{1A} \\ x_{1B} \\ x_2 \end{bmatrix} \tag{28}$$

$$\begin{bmatrix} 5 & 0 & 0 \\ 0 & 3 & 6 \end{bmatrix} \begin{bmatrix} x_{1A} \\ x_{1B} \\ x_2 \end{bmatrix} \leq \begin{bmatrix} 10 \\ 12 \end{bmatrix} \tag{29}$$

$$\begin{bmatrix} x_{1A} \\ x_{1B} \\ x_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{30}$$

The dual linear program is:

$$\min \begin{bmatrix} y_1 & y_2 \end{bmatrix} \begin{bmatrix} 10 \\ 12 \end{bmatrix} \tag{31}$$

$$\begin{bmatrix} y_1 & y_2 \end{bmatrix} \begin{bmatrix} 5 & 0 & 0 \\ 0 & 3 & 6 \end{bmatrix} \geq \begin{bmatrix} 3 & 3 & 2 \end{bmatrix} \tag{32}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{33}$$

This becomes:

$$\min 10y_1 + 12y_2 \tag{34}$$

$$\begin{aligned} 5y_1 &\geq 3 && (35) \\ 3y_2 &\geq 3 && (36) \\ 6y_2 &\geq 2 && (37) \\ y_1, y_2 &\geq 0 && (38) \end{aligned}$$

By duality, we know the optimal value for this linear program is the same as that for the primal, which is to say 18.

Problem 4 (*15 points total)*

Given an array of integers $a[1], a[2], \ldots, a[n]$ (positive and negative) and an integer k (e.g. array $a[1, \ldots, 6] = 20, 100, -5, 80, 10, 15$ and $k = 3$), we wish to find a subset of exactly k of these numbers, which has the largest possible total weight, but such that no two adjacent numbers are taken.

(a) *(4 point)* Describe a simple greedy algorithm that finds a correct solution on the above example. Prove that it always works or show a counter example where it fails.

Solution: Grab the largest available, remove its neighbors. Repeat k times. In the example above, this greedy algorithm will grab (100, 80, 15). In the case of (99, 101, 100, 1, 1, 1, 1) and $k = 3$, the greedy algorithm will choose (101, 1, 1) instead of the optimal (99, 100, 1).

(b) We can solve this problem by (you guessed it!) dynamic programming. Fill in the blanks!

- *(3 point)* For each $i = 0, 1, \ldots, n$, and $j = 1, \ldots, k$ define a subproblem (describe in words) $\texttt{maxsum}[i, j]$ to be:

  Solution: the subsequence of largest weight made up of j items using items only up to $a[i]$. Due to the discrepancy of i starting at 0, and the array a starting at 1, your answers may vary.

- *(1 point)* Initialize the base case(s) of the subproblem with the following code (it may help to solve the next part first):

  Solution: The answers for this part depend on the range of your loops and what your recurrence is. This gets tricky when it comes to initializing items for $i < 0$. We want the cases for $i = 0, -1, -2$ to be $-\infty$ so that those are never considered. This is due to the possibility of negative values in the array. We want the cases for $j = 0$ to be 0 so that we can escape from the negative infinity cases.

  maxsum[$i = 0, j = *$] $= -\infty$
  maxsum[$i = -1, j = *$] $= -\infty$
  maxsum[$i = -2, j = *$] $= -\infty$ (depending on your ranges for the loops)
  maxsum[$i = *, j = 0$] $= 0$ (this must be run after setting the above cases,
  so that maxsum$[-1, 0] = 0$, instead of $-\infty$)

- *(5 point)* Solve each subproblem iteratively with the following code:

  Solution:
```
for $i$ = 1, 2, ... n
    for $j$ = 1, 2, ... min {k, $\lceil i/2 \rceil$} (See discussion under run-time below.)
        maxsum$[i, j]$ = max{maxsum$[i-1, j]$, $a[i]$ + maxsum$[i-2, j-1]$}
```

- *(1 point)* In order to recover the the best subset, what other data structure would you use, and what data would it store?

  Solution: Use a 2-dimensional matrix that records whether of not $a[i]$ is chosen when calculating each maxsum$[i, j]$.

- *(1 point)* What is the running time of the algorithm?

Solution: $O(nk)$ for those that had j go up to k. For this problem, we gave you credit if your solution matched the ranges of the two for loops. Some students had j go up to $\lceil i/2 \rceil$, but not k. In that case, the runtime is $O(n^2)$. We gave you credit for a correct runtime if you did that. However, if you had that runtime, we took a point off for not being as efficient. Overall, you should only have lost one point for using just $\lceil i/2 \rceil$, either here or up above.

Problem 5 (*14 points total*)

(a) (*7 points*) Suppose that you have found a minimum spanning tree T of a weighted graph $G = (V, E)$; now you are told that the weight of an edge has been *decreased.* Describe a $O(|V| + |E|)$ algorithm to find a minimum spanning tree in this new graph, given T. (Hint: It should be not be done by recomputing a minimum spanning tree from scratch.)

Solution: Our algorithm is as follows:

- If the edge $e = \{u, v\}$ with decreased weight is in the tree T, then we return T.
- Otherwise, we add the edge $e = \{u, v\}$ to the tree T, creating a single simple cycle. We can determine all edges in the cycle in $O(|V| + |E|)$ time by running a DFS from node u (using only edges in T) until we reach node v. Once we have found all edges in the simple cycle, we then delete the highest cost edge e' from the cycle (with $e' = e$ possible) to obtain a new spanning tree T'.

To prove the first case of our algorithm correct, note that if e is in T, then we decrease the cost of the tree T by at least as much as the cost of any other spanning tree, and thus T must still be a minimum spanning tree. To prove the second case of our algorithm correct, we need to prove that $T' = T + e - e'$ is a minimum spanning tree. Although it might be intuitively clear T' is a minimum spanning tree, a formal proof is more difficult.

To prove that T' is a minimum spanning tree, we will argue that it is possible for Kruskal's algorithm to return the tree T' as a minimum spanning tree on the new graph, provided we break ties appropriately when adding edges. To prove that Kruskal's algorithm can return the tree T', first note that it is possible break ties in a way such that Kruskal's algorithm returns the tree T when running on the original graph G (whenever there is a tie, we just need to consider edges in the tree T first). Now consider running Kruskal's algorithm on our new graph, assuming that we break ties in the same way.

Note that Kruskal's algorithm will return the exact same set of edges as before, until edge $e = \{u, v\}$ is considered. Now if u and v already belong to the same component, then edge e will not be added, and Kruskal's algorithm will return the exact same set of edges as before, namely the tree T. In this case, note that our algorithm above also returns T, since e is the largest edge in the cycle created by $T + e$ if the components of u and v have already been merged. (e must have higher weight than all edges in the cycle, because it was considered after the path from u to v was formed in T when running Kruskal's algorithm).

Otherwise, edge e is added and the components of nodes u and v are merged early. The only effect of this early merging is that the edge in T that originally merged the components of u and v will no longer be included our new minimum spanning tree. But note that the edge that merges the components of u and v when building the MST in our original graph is the highest cost edge on the path from u to v in T. Namely, e', the largest cost edge in the cycle created by $T + e$, is not included in our minimum spaning tree, and Kruskal's algorithm returns $T' = T + e - e'$. Thus, in both cases, Kruskal's algorithm returns the same tree T' as we defined in our algorithm, and therefore our algorithm returns a proper minimum spanning tree.

(b) (*7 points*) Solve the above problem again for the case in which one edge weight is *increased.*

Solution: Our algorithm is as follows:

- If the edge e with increased weight is not in T, then return T.
- Otherwise, we remove $e = \{u, v\}$ from the tree to obtain two connected components, one containing u and one containing v. We can then run two BFS's in $O(|V| + |E|)$ time (only using tree edges) to label the nodes in u's component L_1, and label the nodes in v's component L_2. Using these labels, we can

then iterate over all edges to find the least cost edge e' crossing from u's component to v's component. We then return the tree $T' = T - e + e'$.

In the first case, T is still an MST since, the change in cost increases the cost of some spanning trees, but not T, so T must still be a MST. To prove the correctness of the second part, note that the proof from part (a), in reverse, implies that a new MST can be formed by removing the e and then adding back some other edge. Since we add the least cost edge possible, which forms a spanning tree, our algorithm must return a minimum spanning tree.