Problem 1. (10 points) Determine whether each of the following statements is true or false.

1. BIPARTITE-MATCHING is in P. T F
2. A local search algorithm will never find the optimal solution. T F
3. Any language can be decided by a Turing Machine. T F
4. There is a polynomial time reduction from TSP to MAXFLOW. T F
5. 3D-MATCHING is NP-complete. T F
6. Some problems in NP can be solved in polynomial time on a deterministic Turing Machine. T F
7. All dynamic programming algorithms run in polynomial time. T F
8. KNAPSACK can be approximated arbitrarily closely. T F
9. Some online algorithms have a competitive ratio strictly smaller than 1. T F

Problem 2. (10 points) Construct a Turing Machine which decides

$$L_2 = \{w \in \{0,1\}^* : w \text{ is a binary number divisible by } 2_{\text{dec}}, \text{ but not } 4_{\text{dec}}\}.$$ 

Problem 3. (10 points) For each of the following languages, determine whether it is: P, NP-complete, coNP-complete, N = none of these.

1. \{\!(G;k) : G has a spanning tree with cost at most k\!\} P coNPC
2. \{G : G does not have a Hamiltonian Cycle\} coNPC
3. \{(M;x) : M(x) halts in less than 100 steps\} P
4. \{(M;x) : M(x) halts in more than 100 steps\} N
5. \{(S:T) : S does not have a subset which sums to T\} coNPC
6. \{(S:T) : S has a subset which sums to at most T\} P
7. \{(G;k) : G has an independent set of size k\} NPC
8. \{M : M halts on all inputs\} N
9. \{(G:a;b;k) : G has a path from a to b with length at most k\} P
10. \{(G;k) : G has a dominating set of size k\} NPC
11. \{G : G has a Hamiltonian Cycle\} NPC
12. \{(M;x) : M(x) runs for more than 100 steps\} P
13. \{(M;x) : M(x) halts in more than 100 steps\} N
14. \{(S:T) : S has a subset which sums to T\} NP
15. \{(S:T) : S has a subset which sums to at least T\} P
16. \{(G;k) : G does not have a dominating set of size k\} coNPC
17. \{M : M halts on some inputs\} N
18. \{G : G is bipartite\} P
19. \{(G;k) : G does not have a clique of size k\} coNPC
**Problem 4.** (10 points) Describe an algorithm to find a Minimum Spanning Tree. Give the time complexity of the algorithm and show the result of running the algorithm on the graph below.

Sort the edges weight (lowest to highest). Go through the edges one by one and add each edge unless it causes a cycle. Let $n$ be the number of vertices. Sorting the edges takes time $O(n^2\lg(n))$, checking whether an edge causes a cycle takes time $O(n)$. So this can algorithm can be done in $O(n^3)$ time (yes there are better ways). The minimum spanning tree is given below.

**Problem 5.** (10 points) Prove that the language $L$ below (i.e., $2\text{MACHINE}\text{MAKESPAN}$) is NP-complete by reducing $\text{SUBSETSUM}$ to $L$.

$L = \{(\sigma; M): \text{the sequence of jobs } \sigma \text{ can be scheduled on two machines with all jobs completing before time } M\}$
Given an instance of \textsc{SubsetSum} \((S; T)\) create an instance of \textsc{2MachineMakespan} \((S', T')\) where \(S' = S \cup \{\lvert 2T - P \rvert \}\) and \(T' = \max\{T, P - T\}\) where \(P = \sum_{i=1}^{n} p_i\).

If \(S\) contains a subset with size \(T\) then the other items sum to \(P - T\). The new item can be added to the smaller of these sets to give both sets the same sum. By placing one set on one machine and the other set on the other machine a makespan of \(T'\) is achieved. If \(S\) doesn’t contain a subset with size \(T\) then you won’t be able to split the items in \(S'\) evenly. So there won’t be a way to schedule the jobs with makespan \(T'\).

Picture proof (by itself this is insufficient).

\[
\begin{array}{|c|c|}
\hline
P - T & T \\
2T - P & T \\
\hline
\end{array}
\quad \text{or} \quad
\begin{array}{|c|c|}
\hline
P - T & 2T - P \\
T & P - T \\
\hline
\end{array}
\]

\textbf{Problem 6.} (10 points) Prove that \textsc{Knapsack} is NP-complete by reducing the language \(L\) below (ie \textsc{2MachineMakespan}) to \textsc{Knapsack}.

\[
L = \{ (\sigma; M) : \text{the sequence of jobs } \sigma \text{ can be scheduled on two machines with all jobs completing before time } M \}
\]

Given an instance of \textsc{2MachineMakespan} \((\sigma = \{p_1, p_2, \ldots, p_n\}; M)\) we need to create an instance of \textsc{Knapsack}. Define \(P = \sum_{i=1}^{n} p_i\). For each job create an item \((p_i, p_i)\) and also create \(2M - P\) items of the form \((1,1)\). These \(n + 2M - P\) items are \(I\). Let \(V = W = M\).

If there is a way to schedule \(\sigma\) prior to time \(M\) then there is a way to schedule \(\sigma\) and \(2M - P\) additional size 1 jobs by time \(M\), but then both machines will be busy until time \(M\). Now the set of jobs on machine 1 (or 2) corresponds to a set of items with total value and weight exactly equal to \(M\).

On the other hand if \(\sigma\) cannot be scheduled prior to time \(M\) then you still can’t do it with the additional size 1 jobs. So there is no set of jobs with total processing time exactly \(M\). Therefore no set of items can have value and weight exactly \(M\).

\textbf{Problem 7.} (10 points) Describe a 2-approximation algorithm for \textsc{VertexCover} (one such algorithm was given in class), show that it is a 2-approximation algorithm, and execute the algorithm on the graph below.
Problem 8. (10 points) Find an approximation algorithm for Minimum Makespan on Two Machines and find (prove) its approximation ratio. (better algorithms will be scored higher: a 2-approx is trivial, 1.5-approx is very easy, full credit requires strictly better than 1.5).

In class we showed several algorithms with competitive ratios 2, 1.5, and 7/6. For all three we defined $P = \sum_{i=1}^{n} p_i$ and $L$ to be the size of the last job to finish in the online schedule and we noted that $Opt \geq \max\{P/2, L\}$. (To keep things simple, assume that, once the job of size $L$ is scheduled, there are no more jobs. The wouldn’t increase the online algorithm’s makespan anyway.)

The first algorithm places all items on the first machine. The online makespan is $P$. The offline makespan is at least $P/2$. So this algorithm is 2-competitive.

The second algorithm considers the jobs one by one and always schedules the job on the less loaded machine. The online makespan was no more than $L + (P - L)/2$ (otherwise the job would have been scheduled on the other machine). So the cost ratio

$$\frac{Online(\sigma)}{Opt(\sigma)} \leq \frac{(P + L)/2}{\max\{P/2, L\}} \leq \frac{P/2}{P/2} + \frac{L/2}{L} \leq \frac{3}{2}.$$   

The third algorithm sorts the jobs by size from largest to smallest then considers the jobs one by one and always schedules the job on the less loaded machine. We showed the competitive ratio using 2 cases. If $L \leq Opt/3$ then

$$\frac{Online(\sigma)}{Opt(\sigma)} \leq \frac{(P + L)/2}{\max\{P/2, L\}} \leq \frac{P/2}{P/2} + \frac{L/2}{3L} \leq \frac{7}{6}.$$  

On the other hand if $L > P/6$ then there are at most 4 jobs. The online algorithm we are considering gives the optimum solution if the number of jobs is at most 4 (verify for 1 job (trivial), 2 jobs (trivial), 3 jobs (very easy), 4 jobs (easy enough)).

Problem 9. (10 points) Find an approximation algorithm for Bin Packing and find (prove) its approximation ratio. (better algorithms will be scored higher: a 2-approx is trivial, full credit requires strictly better than 2).

Note that the optimal algorithm will require $\lceil S/B \rceil$ bins where $S = \sum_{i=1}^{n} s_i$ and $B$ is the capacity of a bin.
We first consider a very simple 2-competitive algorithm. Consider the items one by one. If an item fits into a current bin then put it in that bin. If it does not then put it in a new bin. This algorithm is 2-competitive. Assume that the online algorithm uses $b$ bins. Then we know that at least $b - 1$ bins that are strictly more than 1/2 full (otherwise we could combine the two smallest bins). This means $S/B > (b - 1)B/2/B = (b - 1)/2$. So the optimal solution requires at least $b/2$ bins. Therefore this algorithm is 2-competitive.

A better algorithm would sort the items by size from largest to smallest and then consider the items one by one, putting an item in the lowest numbered bin possible (opening an new bin if necessary). Again assume that the online algorithm requires $b$ bins. Consider the item that causes the online algorithm to open its last ($b^{th}$) bin. If this item has size $\leq B/3$ then all previous bins are strictly more than $2/3$ full. Therefore the optimal solution will require more than $2(b - 1)/3$ bins. In this case this yields a cost ratio of $3/2$. On the other hand if the item causing the last bin has size $> 1/3$. Then we know the optimal algorithm will have at most 2 items per bin and it turns out that (in this case) the online algorithm given yields the optimal solution (ie cost ratio 1). Since the max of $3/2$ and 1 is $3/2$, this algorithm is a $3/2$-competitive algorithm. There are even better approximation algorithms for this problem.

**Problem 10.** (10 points) Imagine that you run the company NanoSquishy which sells a very popular operating system. Currently, your product has many security holes. These security holes are causing your company to lose $1000 per day (due to people switching to a superior, but less established operating system). However, fixing the operating system will require an investment of $100,000,000. You do not know how long it will be until your operating system will be obsolete (after it becomes obsolete your company will cease to exist). Give an algorithm to determine when you should invest in fixing the security holes. Find the competitive ratio of your algorithm.

This is just a disguised version of SkiRental. If you invest in fixing the security holes on day 100000, the competitive ratio is 1.99999. To prove this we assume that the NanoSquishy OS will be obsolete in $d$ days. If $d < 100000$ then the cost ratio is 1000$^d$/1000$^d$ = 1. If $d \geq 100000$ then the cost ratio is (99999 · 1000 + 100000000)/100000000 = 1.99999.

**Problem 11.** (10 points) Imagine that you run a company with many employees. Currently the computers at your company run the NanoSquishy operating system. While this operating system is very popular, it has many security holes. These security holes are causing your company to lose $100 per month (due to computer down time and tech support costs). However, switching to a competing operating system will require an investment of $10,000 to train your employees to use the new operating system. NanoSquishy has promised to fix the security holes. You do not know how long it will be until the security holes will be fixed (NanoSquishy has promised to fix the security holes many times in the past). Give an algorithm to determine when you should invest in switching your operating system (and training your employees). Find the competitive ratio of your algorithm.

This is just a disguised version of SkiRental. If you invest in the transition on month 100, the competitive ratio is 1.99. To prove this we assume that NanoSquishy fixes the security holes after $m$ months. If $m < 100$ then the cost ratio is 100$m$/100$m$ = 1. If $m \geq 100$ then the cost ratio is
(99 \cdot 100 + 10000)/10000 = 1.99.

**Problem 12.** (10 points) Imagine that your friend has offered to give you a ride home from CSUN after your COMP610 final. Assume that it would take you 1 hour to walk home, but only 10 minutes for your friend to drive you home. Your goal is to minimize the time it takes to get home. You go to the place that he is to pick you up and find that he isn’t there. Since he is completely unreliable, you don’t know how long you might have to wait for him to arrive. The question then is “How long should you wait for him to arrive before giving up and walking home?” Suggest an algorithm for this problem and find its competitive ratio. Better competitive ratios get more points. A competitive ratio of 6 is trivial, 2 is fairly easy, better than 2 for full credit.

**Problem 13.** (10 points) Develop a Dynamic Programming algorithm for the following problem: Given a set of turtles where each turtle has a weight and a strength (how much that turtle can carry including his own weight), determine the maximum number of turtles you could stack without crushing any turtle.

Use your algorithm to solve the following problem. Turtles: (10, 25), (12, 18), (4, 16), (6, 15), (5, 12), (8,10), (2,8), (3,6), (2, 4)

Sort turtles from strongest to weakest.

Define: \( L[i][j] = \) maximum number of turtles that can be stacked using turtles i as the bottom most turtle with total weight exactly j.

\[
\begin{align*}
L[n+1][j] & = 0 \\
L[i][j] & = 0 \text{ for } j < W_i \\
L[i][j] & = 0 \text{ for } j > S_i \\
L[i][j] & = 1 + \max{k > i}\{L[k][j - W_i]\}
\end{align*}
\]

Fill column by column starting at the right edge.