Boston University CS 630 Fall 2024
Review Problem Solutions

These are some practice problems for the upcoming final exam.

The format of the final is similar to that of the midterm. You are allowed two cheat sheets, letter sized (A4), handwritten by yourself. The exam is cumulative and covers material from the entire semester. This problem set aims to cover most of the topics that we learned about, but is only an example of questions that we may ask. Make sure to review problems from lecture, labs and homework as well.

The exam is 120 minutes long, the number of problems will be proportional to fit into that timeframe.

How to study? Here are some recommendations on how to go about preparing for the exam.

- Know all definitions.
- Be able to run/trace algorithms on specific instances.
- As you may have noticed, most of the problems in labs and the homework were closely related to some problem from lecture. Make yourself so familiar with the lecture problems that you can comfortable make the connection between those and the problem in the assignment.
- You might find reviewing TopHat questions useful.
- Review lab and homework assignments.
- Go over the reading material (see Piazza post)
- Do a thorough job preparing the "perfect" cheat sheet. Thinking about what should exactly go on it will help you organize your knowledge.

Problem 1 *9/3 Intro*

In the very first lecture of this course, the set cover problem was introduced. Given a universe U of items, and m subsets $S_1, S_2, \ldots, S_m$ of U, find a minimum number of the m subsets such that their union is U (the set U is covered). We returned to the set cover problem several times during this course to illustrate various concepts.

1. Suppose that someone tells you that a particular set cover instance can be solved using k of the m subsets. Describe a polynomial sized certificate that allows this claim to be verified in polynomial time.

   Solution. A list of the k subset indexes would be a sufficient certificate. These will take $O(k \log m)$ bits but that detail is not necessary. The certificate can be verified by iterating over that list, computing the union of the indicated subsets, and then comparing the union result to U. The most expensive step will be the union operation which will process data up to the sum of the length of all the S_i's. The processing should be between linear and quadratic depending on how the set is represented (hash table for linear, just a list for quadratic).

2. What property or characteristic of the set cover problem makes us think that the set cover problem is particularly hard despite having polynomial time certificates?

   Solution. The set cover problem is NP-complete, so a polynomial time algorithm would imply polynomial time algorithms for many problems that currently appear difficult.

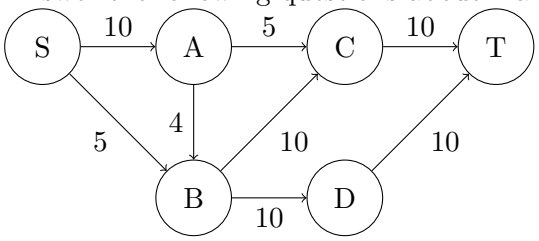
3. Because the set cover problem appears hard, we often resort to approximation algorithms. What is the difference between the guarantees of an exact algorithm for the set cover problem (as stated above) and an approximation algorithm for the set cover problem?

   Solution. An exact algorithm for set cover would give a minimum number of the m subsets, say k, while an approximation algorithm would be allowed to return a higher result, say up to αk where α is the approximation ratio. Note that k is not known, and αk is only an upper bound.

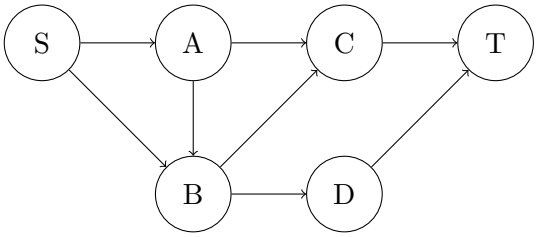
9/5-9/12 Network Flow

Problem 2 *Max Flow Min Cut*

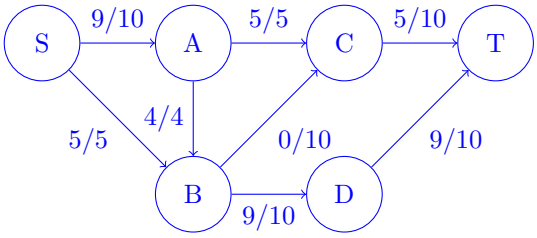
Answer the following questions about maximum flow based on the capacities of this graph.



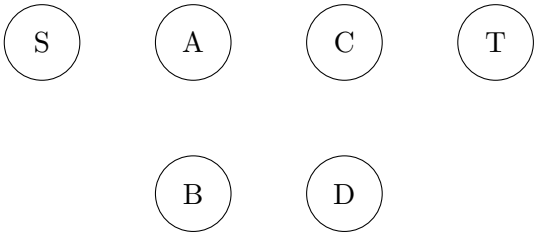
1. Find a maximum flow for the graph given above. Label the flow along each edge in the graph below, and state the flow of the whole graph.



Solution.



2. Draw the residual graph corresponding to your maximal flow.



Solution. The solution is depicted in Figure 1. Note that for clarity of the picture we omitted edges with 0 residual capacity . Feel free to draw it or not in your solution.

3. Find a minimum cut proving that your maximum flow is optimal and mark it in the graph in Figure 2.

Solution. Separating S and A from the other vertices is a min cut of the graph with a value of 14 matching the maximum flow.

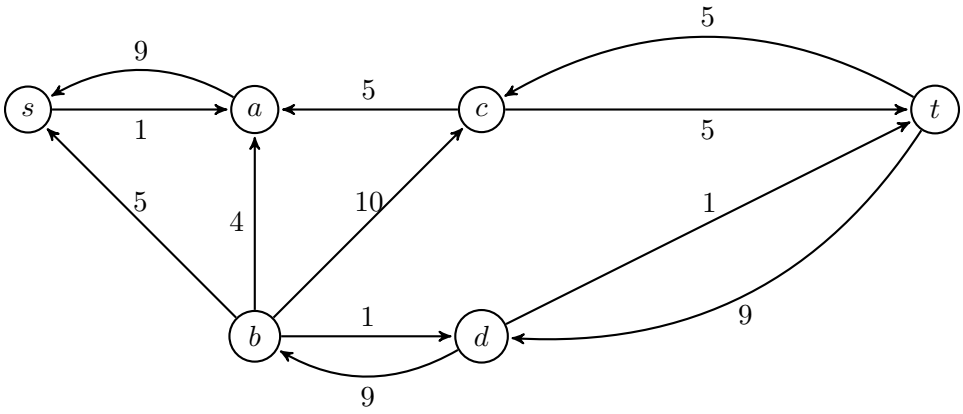


Figure 1: The residual graph Gr.

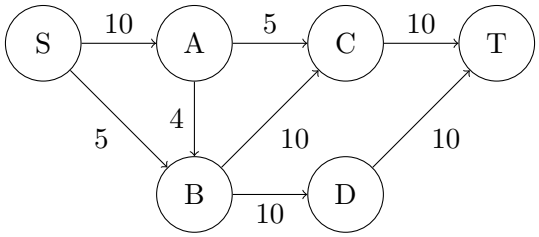


Figure 2: Find the min-cut.

Problem 3 *Determine whether each of the following statements is true for* all *directed graphs G with integer, nonnegative edge capacities, source nodes s and sink nodes t. For false statements, you should be able to draw a counterexample, for true statements you should be able to prove your answer (possibly by referencing material from class.)*

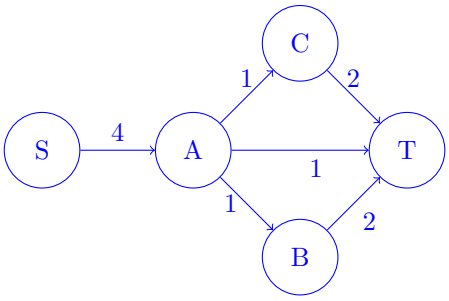
1. There is only one minimum capacity $s-t$ cut in G.

   Solution. False. A simple example is a path with k edges, each with capacity 1. Then each edge serves as a different min-cut.

2. If we increase the capacities of all edges in G by 1, then the minimum-capacity $s-t$ cut will stay the same.

   Solution. False. The capacity of the min-cut is the sum of the capacities along its edges. A cut with k edges will see a capacity increase of k while a cut with ℓ edges will increase by ℓ.

   In the picture below the old min-cut consists of $\{s, A\}$ While the new min-cut is $\{s\}$.



3. If we double the capacities of each edge in G, then the minimum-capacity $s-t$ cut will stay the same.

4. If we double the capacities of each edge in G, then the value of the maximum $s-t$ flow will double.

   Solution. True. Since the value of capacities doubles, the capacity of each cut doubles too. Which means that specifically the value of the min-cut is double. By MFMC theorem this implies that the value of the max flow doubles too.

5. If f is a maximum-value $s-t$ flow in G, and G_f is its residual graph, then one can find a minimum-capacity $s-t$ cut

   $$(A, B)$$

   by taking B to be all the nodes that can reach t in G_f (and A to be its complement, $V \setminus B$).

   Solution. True. We have seen in class that the cut formed by the nodes reachable *from* s in G_f form a min-cut. The roles of s and t are symmetric in this regard. That is, if we were to reverse the edges in G and t would serve as the source than the max flow will have the same value. Further, the vertices reachable from t in the reversed graphs form a min-cut as now t is the source. Finally, the reachable nodes in the reversed graph are the ones that can reach t in the original.

6. If all edges in G have capacity 1, then the value of the maximum $s-t$ flow is at most D, where D is the out-degree of s.

   Solution. The capacity of the cut defined by $\{s\}$ is an upper bound on the amount of flow. Given the degree of s we know that the max flow has value at most D.

7. If all the edges in G have capacity 1, then the Ford-Fulkerson algorithm will terminate in time $O(mD)$ where D is the out-degree of node s.

   Solution. True. From the previous part we know that D is an upper bound on the value of the max flow. Due to the integer capacities we know that in each iteration of FF the flow will increase by at least 1. Thus, there are at most D iterations, each takes $O(m)$ time due to running BFS.

8. The function $f : E \to R$ which has $f(e) = 0$ on all edges is always an $s-t$ flow.

   Solution. True. It fulfills the capacity constraint and flow conservation rule.

9. The value of every $s-t$ flow is at most the capacity of every $s-t$ cut

   Solution. True. The capacity of a cut is always an upper bound on the flow.

10. The capacity of every $s-t$ cut equals the value of every $s-t$ flow.

    Solution. False. By the MFMC theorem the value of the max flow is equal to the capacity of the min-cut. Any non-minimum cut has value higher than that.

11. The capacity of a minimum-capacity $s-t$ cut equals the value of every maximum-value $s-t$ flow.

    Solution. True. This is what the MFMC theorem states.

Problem 4 *9/17-9/19 Reductions, NP, NP-C*

1. Review how to formulate the decision version of an optimization problem and how to prove that it is in NP. Do it for the following problems:

   (a) LONGEST-PATH: Given a graph $G(V, E)$, find the longest simple path in G.

   Solution.
   Decision problem: Given a graph $G(V, E)$ and an integer k, is there a simple path in G of length at least k?

   Certificate: The edges in the path in order.

   Verifier: Check that the number of edges is indeed k. Then verify that it is a valid simple path by iterating over the edges, verifying that they exist and each node is contained once.

   (b) HAMILTONIAN-PATH/CYCLE: Given a graph $G(V, E)$ find a simple path/cycle that contains each vertex exactly once.

   Solution.
   Decision problem: This problem is not an optimization problem as the Hamiltonian path/cycle has fixed length. Hence, the original problem description ("Is there a Ham path/cycle?") serves as decision problem.

   Certificate: The vertices in the path/cycle in order.

   Verifier: Traverse the vertices in the given order, verify that the edges exist and each node is only contained once.

   (c) HAMILTONIAN-COMPLETION: Given a graph $G(V, E)$ find a minimum set of edges to add, such that the resulting graph has a Hamiltonian path.

   Solution.
   Decision problem: Given a graph $G(V, E)$ and an integer k, can we add k edges to G such that the resulting graph contains a Hamiltonian path?

   Certificate: The k edges to add (= vertex pairs to be connected) AND the vertices of the path in order.

   Verifier: Add the k edges. Then verify the existence of the Hamiltonian path as in the previous problem.

   (d) GRAPH-COLORING: Given a graph $G(V, E)$ a coloring of its vertices assigns a color to each vertex, such that no two neighboring nodes have the same color. The problem asks to find the minimum number of colors required for a coloring.

   Solution.
   Decision problem: Given a graph $G(V, E)$ and an integer k, is there a coloring of the vertices of G using at most k colors?

Certificate: The color of each vertex.

Verifier: Iterate over the edges and verify that their two vertices have different color.

(e) GRAPH-EDGE-COLORING: Given a graph $G(V, E)$ a coloring of its *edges* assigns a color to each edge, such that edges that share node have all different colors. The problem asks to find the minimum number of colors required for a coloring.

Solution.
Decision problem: Given a graph $G(V, E)$ and an integer k, is there a coloring of the edges of G using at most k colors?

Certificate: The color of each edge.

Verifier: Iterate over the *vertices*. Check every pair of its edges and verify that they are different color. (There are more efficient ways of doing this, but this simple algorithm is still polynomial.)

(f) SETPACKING: Given a universe $U = \{u_1, \ldots, u_n\}$ of n items and m subsets of U, $S = \{S_1, \ldots s_m\}; S_i \subseteq U$. Find the maximum number of sets in S that are pairwise disjoint.

Solution.
Decision problem: Given a universe $U = \{u_1, \ldots, u_n\}$ of n items and m subsets of U, $S = \{S_1, \ldots s_m\}; S_i \subseteq U$ and an integer k. Are there k pairwise disjoint sets in S?

Certificate: The k sets (their ids) that are supposedly disjoint.

Verifier: Iterate over every pair of sets to verify that they don't share any items. As a lose upper bound on the running time, there are $O(k^2)$ sets, the size of each set is at most n, hence one comparison takes $O(n)$, which yields $O(k^2 n)$ polynomial running time.

2. Prove that the SETPACKING problem described in the previous part is NP-Complete.

Solution. We have already shown in the previous problem that Setpacking is in NP. We should that it's NP-C by reducing Independent Set to Setpacking. An instance of Independent Set consists of a graph $G(V, E)$. We define the corresponding input to Setpacking as follows; the items $U = \{u_1, \ldots, u_n\}$ correspond to the edges $E = \{e_1, \ldots e_n\}$ in G. Each set S_v consists of the edges adjacent to vertex v.

We can see that the subsets in a Setpacking correspond to vertices that are independent in G. Conversely, independent vertices in G, don't share any edges, hence their corresponding sets form a Setpacking.

3. A Conjunctive Normal Form (CNF) is a Boolean formula that is a conjunction (logical AND) of clauses of literals. The literals in the clauses are connected by Boolean OR. A literal is a Boolean variable. For example this is a CNF: $(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_4 \vee \neg x_5) \wedge (x_4 \vee x_5)$.

The Satisfiability (SAT) problem asks whether given a CNF, there is an assignment of True/False values to the literals, such that the formula evaluates to true.

3-SAT is a variant of the problem where each clause consists of exactly 3 literals, k-SAT is a variant with exactly k literals in each clause.

We know that 3-SAT is NP-Complete (1-SAT and 2-SAT can be solved in polynomial time). Show that k-SAT (for $k > 3$) is NP-Complete; first prove that it is in NP, then write a polynomial-time reduction from 3-SAT.

Solution. **NP:** We present a certificate and verifier. The certificate consists of the True/False value assigned to each literal. The verifier takes the values and computes whether the CNF evaluates to true. (Note, that this is not specific to k).

NP-C reduction: We have to show that any instance of **3**-SAT can be solved with an API that solves k-SAT.

Let C be a 3-SAT CNF with clauses $C = C_1 \wedge C_2 \wedge \ldots \wedge C_\ell$. Note that each C_i is a clause with three literals. We can create a k-SAT CNF from C be adding $k-3$ terms to each clause. Specifically, we will add literals that always evaluate to False. You can add $\vee False \vee False \vee \ldots \vee False$ k-3 times inside every clause. If $C_i = (x_1 \vee x_2 \vee x_3)$ is the 3-SAT clause, then the corresponding k-SAT clause is $C_i^k = (x_1 \vee x_2 \vee x_3 \vee False \vee False \vee \ldots \vee False)$.

We can see that C_i has a valid value assignment iff C_i^k has a valid assignment, since the clause will only evaluate to True if the original literals in C_i do.

Problem 5 *9/24 Approximation Algorithms*

Answer the following questions about the following GreedyVC algorithm presented in class.

Algorithm 1: $GreedyVC(G(V, E))$

```
1 S ← empty set of vertices;
2 for (u, v) ∈ E do
3     if u ∉ S ∧ v ∉ S then
4         S ← S ∪ {u, v};
5 return S
```

1. If GreedyVC returns a vertex cover S with $|S| = 124$, how many times was line 4 of GreedyVC run?

   Solution. Line 4 of GreedyVC adds two vertices to the returned set cover S, and line 3 ensures that both of these vertices are new to S, so line 4 must have been run $124/2 = 62$ times.

2. What is the relationship between edges (u, v) that are considered in line 4 of GreedyVC? Your answer must be relevant to the approximation analysis of GreedyVC.

   Solution. If an edge is considered in line 4 of GreedyVC, then it cannot share any endpoints with any of the edges that were previously considered on that line. That is because the endpoints of the previously considered edges were added to S, and line 3 checked if the endpoints were in S, so no overlap with previously considered edges is possible at line 4.

   Because this is also true for edges that are considered later, then the an edge considered in line 4 will not share endpoints with any of the edges considered later either. So an edge considered at line 4 does not share endpoints with any of the other edges considered at line 4.

   In other words, these edges are vertex-disjoint.

3. Prove that GreedyVC is a 2-approximation algorithm.

   Solution. Because the edges considered in line 4 are vertex-disjoint, at least one endpoint of each of those edges must be in the vertex cover. GreedyVC actually adds both of them. So if E' is the set of edges considered in line 4, then the minimum vertex cover size must have at least $|E'|$ vertices and the returned vertex cover has $2|E'|$ vertices, so the approximation ratio is at most 2.

Problem 6 *9/26 Approximation Algorithms*

In lecture, we covered the greedy center selection problem. Given a set of n sites $s_1, \ldots, s_n$ and an integer $k > 0$, select a set of k centers C so that the maximum distance $r(C)$ from a site to the nearest center is minimized. The distance function considered in lecture was the Euclidean distance with the formula $\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$ in two dimensions. For this problem, use the Manhattan or "city block" distance instead. For two dimensions, the Manhattan distance is calculated with the formula $|x_1 - x_0| + |y_1 - y_0|$, and generalizes as $\sum_i |x_1[i] - x_0[i]|$.

For the special case of 1 center and Euclidean distance, finding the optimal center for a given set of n sites amounts to finding the smallest d-dimensional circle (sphere) containing those sites. For 1 center and Manhattan distance, finding the optimal center amounts to finding the smallest square containing the sites where the square diagonals are parallel to the given dimensions (so it is looks more like a diamond). For this problem, you can assume a function for optimizing the placement of 1 center for n sites is available and runs in $O(n)$ time.

1. Prove that the greedy center selection is still computable despite the real-valued outputs by bounding the number of cases that need to be considered. Note you only need to show that it is computable; you do not need to show this is possible to do quickly.

   Solution. Any optimal set of k centers induces a k-partitioning of the n sites according to which center each site is closest. The number of k-partitions is exponential in n and k, but finite. So those k-partitions can be computed, then the center for each partition can be independently optimized, and then the maximum distance over all sites can be checked to evaluate the whole k-partition.

2. The 1-dimensional case is commonly solved exactly by dynamic programming with $O(nk)$ sub-problems. Show how to solve the 1-dimensional greedy selection problem in polynomial time. Any polynomial-time algorithm will get full credit.

   Solution. This problem can be broken up into sub-problems optimizing k' centers the first n' sites after sorting the sites by coordinate. This is done for all integer k' and n' where $1 \leq k' \leq k$ and $1 \leq n' \leq n$. The base case is $k' = 1$ and the optimal placement has the center placed at the average position of the 1st and k'th sites to minimize the worst case distance. Then for $k' > 1$, there are $n' - 1$ splits where the first $n'' < n'$ sites use an optimal placement of $k' - 1$ centers and the $n'' + 1$ through n'th sites are optimized with one center. The best of those $n' - 1$ splits is used for the solution for $k' > 1'$ and n'. This uses a table of size $O(nk)$ and time $O(n^2 k)$.

3. Give a polynomial time 2-approximation algorithm for any number of dimensions.

   Solution. The algorithm using Manhattan distance is the same as the algorithm covered in lecture since the critical property, triangle inequality, is preserved. First, an arbitrary site is picked as the location of the first center. Then for the other $k - 1$ centers, repeatedly pick the site that is farthest from the centers chosen so far as the next center location. The correctness proof of the 2-approximation is the same as covered in lecture.

Problem 7 *10/1 Submodular Approximation Algorithms*

In lecture, we covered a variant of the set cover problem where we wanted to maximize the coverage of the universe U with a fixed number of subsets k.

1. Prove that the coverage of a choice of subsets is a sub-modular function.

   Solution. This was an example in lecture.

   The coverage of a choice of subsets I is $f(I) = \left|\bigcup_{i \in I} S_i\right|$. By definition, f is submodular if and only if for $X \subset Y$ and $e \notin Y$, the submodular definition requires

   $$f(X \cup \{u\}) - f(X) \quad \geq \quad f(Y \cup \{e\}) - F(Y).$$

   (Different submodular variables used to avoid conflicting with the set cover variables.)

   $$\begin{aligned} f(X \cup \{e\}) &= \left|\bigcup_{i \in X} S_i \cup S_e\right| \\ &= f(X) + \left|S_e \setminus \bigcup_{i \in X} S_i\right| \\ f(X \cup \{e\}) - f(X) &= \left|S_e \setminus \bigcup_{i \in X} S_i\right| \end{aligned}$$

   Similarly,

   $$\begin{aligned} f(Y \cup \{e\}) - f(Y) &= \left|S_e \setminus \bigcup_{i \in Y} S_i\right| \\ &= \left|S_e \setminus \bigcup_{i \in X} S_i \setminus \bigcup_{i \in (Y \setminus X)} S_i\right| \end{aligned}$$

   Therefore,

   $$f(X \cup \{e\}) - f(X) \quad \geq \quad f(Y \cup \{e\}) - F(Y)$$

   and f is sub-modular.

2. Prove that the coverage of a choice of subsets is monotone increasing (as used in sub-modular analysis).

   Solution. A function is monotone if for every $X \subseteq Y$, $f(X) \leq f(Y)$. The coverage function is

$f(X) = \left|\bigcup_{i\in X} S_i\right|$ so

$$\begin{aligned}
f(Y) &= \left|\bigcup_{i\in Y} S_i\right| \\
&= \left|\bigcup_{i\in X} S_i \cup \bigcup_{i\in(Y\setminus X)} S_i\right| \\
&= \left|\bigcup_{i\in X} S_i\right| + \left|\left(\bigcup_{i\in(Y\setminus X)} S_i\right) \setminus \left(\bigcup_{i\in X} S_i\right)\right| \\
&= f(X) + \left|\left(\bigcup_{i\in(Y\setminus X)} S_i\right) \setminus \left(\bigcup_{i\in X} S_i\right)\right| \\
f(Y) &\geq f(X)
\end{aligned}$$

3. Given that coverage is monotone and sub-modular, if k subsets are picked with the following algorithm, what is the approximation ratio?

Algorithm 2: $GreedySC(U, S_1, \ldots, S_m, k)$

```
1 X ← U /* uncovered elements in U */
2 C ← empty set of subsets;
3 for j = 1, ..., k do
4 |   Select S_i that covers the most elements in X;
5 |   C ← C ∪ S_i;
6 |   X ← X \ S_i;
7 return C
```

Solution. Since the coverage is monotone sub-modular, the approximation ratio is $1 - 1/e \approx 63\%$. The theorem by Nemhauser, Wolsey and Fisher was stated in lecture.

Problem 8 *10/3 Bin Packing*

In Bin packing we solve the following problem: given n items, each with a given weight $0 \leq w_i \leq 1$ and bins with weight capacity 1, assign each item to a bin, such that the total number of bins used is minimized. Bin Packing is an NP-Complete problem, however we have seen multiple approximation algorithms for it. In each algorithm items arrive and are assigned to a bin one at a time. (This is called an online algorithm.)

1. *Next Fit:* For each item, if there is room in the current bin, add it to the bin. Otherwise, start a new bin, make that the current bin and add the item.

   Recall what the approximation ratio of Next Fit is. Find an example input for this ratio.

   Solution. The approximation ratio of Next Fit is 2. Here is one example: $0.2, 0.9, 0.3, 0.8, 0.7, 0.1$. These items can be packed into 3 bins - 0.2+0.8, 0.9+0.1, 0.3+0.7 - but 6 were used.

2. *Worst Fit:* For each item, assign it to the bin with the most empty space left (this is the opposite of the Best Fit algorithm covered in class). If no bins have enough empty space, then start a new bin and add the item.

   How many bins are used by Worst Fit on the input sequence $0.8, 0.9, 0.1, 0.2, 0.1, 0.9$? How does this compare to the optimal solution?

   Solution. The bin packing produced by Worst Fit is $0.8 + 0.1, 0.9, 0.2 + 0.1, 0.9$ using 4 bins. The best bin packing for this sequence uses 3 bins, e.g. $0.8 + 0.2, 0.9 + 0.1, 0.9 + 0.1$.

Problem 9 *10/17 Randomized Content Resolution*

In the randomized content resolution problem, an alternative way to analyze the time until all processes succeed breaks the analysis into two components. What is the probability that any processes succeed at a given time step, and how many successes are needed until each process has succeeded. For the following questions, assume that each of the n processes makes an attempt each time step with probability $1/n$, and a particular process succeeds if it is the only process to attempt at a particular time step.

1. What is the probability that process i succeeds at time step t?

   Solution. A particular process succeeds at a particular time step if it attempts at that time step (probability $1/n$) and none of the other processes attempt (probability $(1-1/n)^{n-1}$). So the probability that a particular process i succeeds at time step t is $\frac{1}{n}\left(1-\frac{1}{n}\right)^{n-1}$.

2. What is the probability that any process succeeds at time step t?

   Solution. The events where two processes succeed at the same time are disjoint (they would block each other), so we can simply add up the previous success probability. This gives us probability $\left(1-\frac{1}{n}\right)^{n-1}$ that any process succeeds at time step t.

3. Conditioned on any process succeeding at time step t, what is the probability that process i succeeded at time step t?

   Solution. This conditional probability is $\frac{\frac{1}{n}\left(1-\frac{1}{n}\right)^{n-1}}{\left(1-\frac{1}{n}\right)^{n-1}} = \frac{1}{n}$.

4. What is the expected number of time steps with any process succeeding for all n processes to succeed at least once? You may express this answer with $O()$ notation.

   Solution. This is the coupon collector's problem which is known to require $\Theta(n \log n)$ attempts in expectation. Briefly, n attempts cover all but $1/e$ fraction of the processes. After $n \ln n$ attempts, the expected remaining processes that have not succeeded is just 1, but another $\Theta(n)$ attempts are needed for that one to succeed in expectation.

5. What is the expected number of time steps for all processes to succeed at least once? You may express this answer with $O()$ notation.

   Solution. The probability in part 2 is between $1/e$ and $1/2$ depending on n, so the expected number of time steps for another success is $\Theta(1)$. Combined with the answer to part 4, the expected number of time steps for all processes to succeed at least once is $\Theta(1) \cdot \Theta(n \log n) = \Theta(n \log n)$.

Problem 10 *10/22 Global Min Cut*

For an undirected graph $G(V, E)$, the global min-cut problem is to find a cut that partitions the nodes into two sets A and $V - A$ with minimum weight.

1. Explain how to reduce the global min cut problem to $|V| - 1$ instances of the min st-cut problem.

   Solution. Pick any vertex and designate it s. For any global min cut, s will be in either A or $V - A$. Without loss of generality, let the set containing s be A. There must be at least one vertex t in $V - A$, but we do not know which vertices are in $V - A$; there are $|V| - 1$ choices. So, calculate the min st-cut for each $t \in V - s$, and take the minimum of those results. That minimum is the global min cut value.

Problem 11 *10/24 Randomized Load Balancing and Routing*

The basic randomized load balancing scenario assigns tasks uniformly at random to the available servers without regard to the load of any particular server. Answer the following questions using Chernoff bounds. For your convenience, here are the formulas for Chernoff bounds (omitting the conditions to apply them).

For any $\mu \geq E[Z]$ and $\delta > 0$,

$$Pr\left(Z > (1+\delta)\mu\right) \quad < \quad \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu}$$

For any $\mu \leq E[Z]$ and $1 > \delta > 0$,

$$Pr\left(Z < (1-\delta)\mu\right) \quad < \quad e^{-\frac{1}{2}\mu\delta^2}$$

1. Suppose that n tasks are mapped uniformly at random to n servers. Compute an upper bound on the probability that more than $\ln n$ tasks were assigned to the ith server.

   Solution. Let Z be the sum of the random variables for each ball landing in the ith server. Let $\mu = E[Z] = n \cdot \frac{1}{n} = 1$. Then the probability that more than $\ln n$ tasks being assigned to the ith server corresponds to $\delta = \ln n - 1$ and

$$\begin{aligned} Pr[Z > \ln n] &= \left[\frac{e^{\ln n - 1}}{(\ln n)^{\ln n}}\right]^1 \\ &= \frac{n/e}{(\ln n)^{\ln n}} \\ &= \frac{n/e}{e^{\ln \ln n \ln n}} \\ &= \frac{n}{e\ n^{\ln \ln n}} \\ &= \frac{1}{e\ n^{\ln \ln n - 1}} \end{aligned}$$

2. Suppose that $n \ln n$ tasks are mapped uniformly at random to n servers. Use Chernoff bounds to bound the probability that fewer than $\frac{1}{2}\ln n$ tasks were assigned to the ith server.

   Solution. This scenario corresponds to $\mu = \ln n$ and $\delta = 0.5$.

$$\begin{aligned} Pr[Z < \frac{1}{2}\ln n] &< e^{-\frac{1}{2}\ln n (0.5)^2} \\ &< n^{-1/8} \end{aligned}$$

Problem 12 *10/29 - 10/31 Hash Tables*

1. In Sudoku we are given a 9×9 board containing the numbers $1 \ldots 9$, with 9 copies of each number. The board is divided in a 3×3 square grid, such that each square contains 3×3 cells.

   A *valid* layout of the Sudoku puzzle is one, where each row, each column and each grid-square contains each number exactly once.

   Given as input a 9×9 filled array A, determine whether it's a valid Sudoku layout. Your algorithm should perform at most $81 \times 3 \times O(1)$ operations.

   Solution. We do this by creating one length-9 hash table r_i corresponding to each row, one hash table c_j for each column, and one table g_k for each grid. We iterate over A until we potentially find a violating element (which means iterating over 81 values at most). When we consider $A[x]$ that is in row i, column j and grid k we perform three hash table insertions; we insert it into r_i, c_j and g_k. If we find a duplicate then the algorithm should return that the assignment is not valid.

   This algorithm is correct as a duplicate means that $A[x]$ has already been encountered in either x's row, column or grid.

2. As input we are given a length-m array of integers A and a target value V.

   (a) Either find two integers in A such that they sum to V or return that no such pair exist. Find the most time-efficient way to do this using an $O(m)$ length hash table. Analyze your running time.

   Solution. First, observe that for a given $A[i]$ the algorithm asks to verify whether A contains the value $x = V - A[i]$. Given the space requirements we can use a length-m hash table H to do this. We first insert the numbers in A into H. This takes $O(m)$ time. The average load of each bucket in H is 1, thus searching for x takes $O(1)$ time. In worst case we have to try each index in A which results in $O(m) \times O(1) = O(m)$ time.

   (b) Design a time efficient algorithm as in part (a.) but this time using an $O(n)$ length hash table where $n < m$. Analyze its expected running time.

   Solution. This time we use a length-n hash table H'. Insertion still takes $O(m)$ time. The average load is m/n and the average search time $O(1 + m/n)$. Which yields $O(m(1 + m/n)) = O(m + m^2/n)$.

3. Given an unordered length-n array A of integers – which may be positive or negative and are not consecutive – return the smallest positive integer *not* in A. Your algorithm must run in expected time $O(n)$.

   Solution. Iterate over A to populate a hash table H of size n with its values. Then start counting from 1 to n. Try hashing each integer to H, return the first integer not found in H. If we reach n without returning, then return $n+1$. Note, that in it's possible that a bucket holds multiple values of A, then we have to check each element in a bucket to verify.

Since we count starting from the lowest positive integer we know that the algorithm will indeed return the lowest missing value. We can stop after counting to n, because by pigeonhole principle, out of n values in A if they are not consecutive $1 \dots n$, then there is a missing value.

The expected load of each bucket is $O(1)$, hence verifying whether a number is in H takes $O(1)$ time. We're counting up to n which means at most n iterations.

Problem 13 *11/5 Bloom Filters*

1. A counting Bloom filter B is a length-m hash table where each bucket holds a counter. Initially, the counters are set to 0. There are k hash functions. To insert an item x, we compute $h_i(x)$ for each hash function $h_1 \dots h_k$, and increase the counters stored at $B[h_i(x)]$ by 1.

   (a) To insert x into B we compute k hash values $h_1(x) \dots h_k(x)$. What is the probability that they hash to k different buckets?

   Solution. The k hash functions hash x to k different buckets if and only if each hash function hashes x to a different bucket from the previous hash functions. For the ith hash function, this has probability $1 - \frac{i-1}{m}$ since the previous $i-1$ hash functions picked $i-1$ distinct buckets from the m buckets. So the probability that they hashed to k different buckets is $\Pi_{i=1}^{k} \left(1 - \frac{i-1}{m}\right)$. Using a union bound on the failure case, this has a lower bound of $1 - \sum_{i=1}^{k} \frac{i-1}{m} = 1 - \frac{k(k-1)}{2m}$.

   (b) Suppose that the counting Bloom filter insertions ignore the possibility of collisions and simply increment the counters multiple times in case of collisions. Based on the values of the counters in B after some insertions, show how to estimate the number of inserted items.

   Solution. The number of inserted items can be calculated exactly because the sum of the counters always increases by k. Let $s = \sum_{i=1}^{n} B[i]$. Then s is the sum of all the counters, and the number of inserts is s/k.

Problem 14 *11/7 Counting Sketches*

1. We use Flajolet-Martin sketches to get the approximate count of *unique* elements in a dataset. We create a hash table B with hash function $h(\ )$ in the following way:

   We compute $h(x)$ (in binary) and find the least significant bit i in $h(x)$ that is 1. e.g. if $h(x)$ is an odd number, then the least significant 1 is the very first bit. If $h(x)$ is divisible by 2^3 but not 2^4 then the least significant 1 is in the third position. We then set $B[i]$ to 1. Note that once $B[i]$ is set to 1, it won't change when subsequent items hash to the same bit.

   In parts (a.) and (b.) of this problem you will see how to recover the size of the data, n, from B.

   (a) Once all data has been hashed into B, most likely the first "few" bits of B are set to 1 (thus, with no zeros in between). Compute in expectation how many indices of consecutive ones this is when n items are inserted.

   Solution. We have seen this computation in class. Assuming the input is uniform random about half of the insertions sets $B[0]$, a quarter sets $B[1]$, 1/8 set $B[2]$ etc (why?*). This results in $\log_2 n$ bits being set to 1.

   *The hash function $h(x)$ returns a uniform random number in the range $0, n-1$. This is equivalent to generating a random binary number with values in this range. This means generating $\log n$ 0-1 bits at random. The probability of the first bit set to 1 is 1/2. The second bit being the first non-zero is a conditional probability; the first bit being 0 and the second bit 1, which is $1/2^2$. For the kth bit this is $k-1$ zeros and then a 1, resulting in probability of $1/2^k$.

   (b) Based on part (a.). How can you compute n given B?

   Solution. Since the first $\log n$ bits are set to 1 (in expectation), we simply compute $n = 2^c$ where $c = \#ofones$.

   (c) We can use FM sketches to compute the size of the dataset in a distributed manner. Suppose we have k collection sites, each compute their own hash table, $B_1, B_2, \ldots, B_k$. Suppose we are given the k hash tables, how can you compute the total size of the dataset?

   Solution. The union of the hash tables (= bit-wise inclusive or) $B = \cup_{i=1}^{k} B_i$ results in the same hash table as if we had inserted all items into a single table B. We then use B, as in the previous part, to get an estimate for n.

   (d) Consider the distributed setting, i.e. k data collection sites, from part (c.) again. Suppose that due to bandwidth issue the sites can't send entire hash tables to us, but instead just a single number. What numbers should they send and how can we recover the total size of the data n from the numbers?

   Solution.
   Less precise sketch, but requires less bandwidth and computation: Site i will compute the number c_i of consecutive one bits in $B[i]$. Remember, that with high probability this number is at least $\log_2 n_i - \log_2 \log_2 n_i$ where n_i is the size of the data collected at site i. They will send c_i.

Given $c_1, \ldots, c_k$. We will then return $c = \max_{i=1\ldots k} c_i$ as an estimate to $\log n$.

Note that this approach is an estimate of estimates as $\cup_{i=1}^{k} B_i$ might increase the length of consecutive ones because a 1 in one table might fill a gap in another. Let c be the estimate for $\log n$ computed above, and let b be the estimate resulting from part (c.). We can see that c is off by $i_b - i_c$ bits from b. The probability that $i_b - i_c = j$ is the probability that the first non-zero bit of each B_i is $< k - j$. This is $1 - p^k$ where p is the probability of B_i consisting of j ones. Which is $\sum_{\ell=0}^{j-1} 1/2^j$.

Precise, but less efficient, reconstruction of B: For each B_i think of the sequence of 0 and 1 bits in B_i as a binary number. Compute the value c_i' of this binary number and send this. We can reconstruct B_i perfectly from c_i' and use that to invoke part (c.). The approximation ratio of this result is the same as in (c.).

Problem 15 *11/12 Randomized Divide and Conquer*

Consider the randomized QuickSort algorithm where the pivot is picked uniformly at random, and the original input size is n.

1. What is the probability that at least one of the recursive calls has an input of size at least $\frac{3}{4}n$?

   Solution. This can only happen if the first split is that uneven. Otherwise, both recursive calls have inputs of less than $\frac{3}{4}n$ and the inputs never grow in recursive calls.

   How does that first split result in such a big recursive call? It will happen if the pivot chosen is in the bottom $\frac{1}{4}n$ or the top $\frac{1}{4}n$. The probability of each of those cases is 1/4 and they are disjoint, so the probability is 1/2.

Problem 16 *11/14 - 11/19 Compression (Huffman Codes and Arithmetic Coding)*

1. Given the following data, compute the Huffman code associated with each character. To avoid ambiguity when merging symbols, order the merges so that code("a") < code("c") < code("o") < code("t").

char	frequency
a	1/2
c	1/4
o	1/8
t	1/8

Decode the following code "111010110100111".

Encode "cat".

Solution. Huffman code words:

char	frequency	code
a	1/2	0
c	1/4	10
o	1/8	110
t	1/8	111

111010110100111 = tacocat

cat = 100111

2. Using the same frequencies and symbol encoding order, calculate the lower and upper bounds for each of the following words, and give a minimal example in *decimal* to select that word. (A couple examples are given.)

word	lower bound	upper bound	decimal encoding
"a"	0	0.5	0.0
"c"	0.5	0.75	0.5
"o"			
"t"			
"at"			
"ca"			
"ac"			
"cat"			

Hint: draw the 1D dart board model for this data, and compute the separators between words. Then pick a number with as few decimals as possible in the middle of each range.

Solution.

word	lower bound	upper bound	decimal encoding
“a”	0	0.5	0.0
“c”	0.5	0.75	0.5
“o”	0.75	0.875	0.75
“t”	0.875	1	0.9
“at”	0.4375	0.5	0.44
“ca”	0.5	0.625	0.5
“ac”	0.25	0.375	0.30
“cat”	0.609375	0.625	0.61

Problem 17 *11/21 Graphs*

Normalized closeness centrality was defined as

$$C(v) \quad = \quad \frac{|V| - 1}{\sum_{w \in V} d(v, w)}$$

where $d(v, w)$ is the length of the shortest path between vertices v and w. $C(v)$ can be calculated for all v in $O(|V|^2)$ time if all pairs shortest paths are known.

Answer the following questions assuming $G(V, E)$ is a connected graph.

1. How long does it take to compute all $C(v)$ if $G(V, E)$ is an unweighted graph?

   Solution. For an unweighted graph, single source shortest paths may be computed in $O(|E|)$ time, so all pairs shortest paths (APSP) may be computed in $O(|V||E|)$ time. After computing APSP, the centrality formula can be implemented directly in $O(|V|^2)$ time, so the total time is $O(|V||E|)$ time.

2. How long does it take to compute all $C(v)$ if $G(V, E)$ in a weighted graph with all weights positive?

   Solution. The Floyd-Warshall algorithm can be used to compute all pairs shortest paths in $\Theta(|V|^3)$ time. This gives a total time of $\Theta(|V|^3)$. Other choices for APSP algorithms are also acceptable as long as they handle weights.

Problem 18 *11/25 Random Walks and Page Rank*

The PageRank algorithm is based on the steady state distribution of random walks on the World Wide Web. In lecture, a few different algorithms for computing the distribution of pages after k random steps were mentioned.

For the following problems, assume that matrix multiplication, matrix inversion, and calculating eigenvectors and values take $O(n^3)$ time. Write pseudo-code and analyze the running time of each of the following methods to compute the state distribution in k steps from $n \times n$ adjacency matrix A.

1. Repeated squaring based on first computing $A^2, A^4, \ldots$

   Solution. Using repeated squaring, $O(\log n)$ matrix multiplications are used, so the running time is $O(n^3 \log n)$.

2. The eigenvalue formula. Given $n \times n$ matrix V where the ith column is the ith normalized eigenvector of A, and $n \times n$ Λ which is a diagonal matrix whose values are the eigenvalues of A, then $A = V\Lambda V^{-1}$ and $A^k = V\Lambda^k V^{-1}$.

   Solution. Computing the eigenvectors and eigenvalues for V and Λ was given as $O(n^3)$. So were the two matrix multiplications, so the total cost is $O(n^3)$ plus the time to calculate Λ^k. Calculating Λ^k takes $O(n^2)$ time initializing non-diagonal entries to zero, and n diagonal entries where the eigenvalue is taken to the kth power. Assuming that exponentiation of real numbers is not available in constant time, but multiplication of real numbers is available in constant time, then those diagonal entries can be calculated in $O(n \log k)$ time using repeated squaring. So the total time is $O(n^3 + n^2 + n \log k)$. In most cases, $\log k = O(n^2)$ so this reduces to $O(n^3)$, but this simplification is not required.

3. Power method. Starting from probability distribution represented as vector x, update $x = xA$ k times. So $x = ((((xA)A)A), \ldots)A$.

   Solution. Each vector matrix multiplication takes $O(n^2)$ time, so the power method takes $O(kn^2)$ time.

Problem 19 *12/3 Graph Models*

Clustering coefficients were introduced in lecture as a way to compare undirected graphs. For a particular vertex v,

$$C_v \quad = \quad \frac{\text{triangles containing } v}{\text{triples centered at } v}$$

and for the whole graph,

$$C \quad = \quad \frac{1}{n}\sum_v C_v$$

1. Write pseudo-code for an algorithm to compute the clustering coefficients for every node in a graph and analyze its complexity. (Must run within $O(|V|^3)$ time for full credit.)

   Solution. The $O(|V|^3)$ time bound is easily achieved by looping over all possible v's to calculate each C_v, and for each C_v looping over all pairs of other vertices in the triangle or triple.

Algorithm 3: *ClusterCoef*$(G = (V, E))$

```
   /* E is in the form of an adjacency matrix.                          */
1  for v ∈ V do
2      triangle_count ← 0 ;
3      triple_count ← 0 ;
       /* Loop over triples centered at v.                              */
4      for u, t ∈ V where (u ≠ t) ∧ ((u, v) ∈ E) ∧ ((t, v) ∈ E) do
5          triple_count ← triple_count + 1 ;
6          if (u, t) ∈ E then
               /* The triangle is complete.                             */
7              triangle_count ← triangle_count + 1 ;
8      C[v] ← triangle_count/triple_count ;
9  return Sum(C)/|V|
```

Problem 20 *12/5 Graph Randomization*

Suppose you are given two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ with the same numbers of vertices, and same marginals. So $|V_1| = |V_2|$ and $|E_1| = |E_2|$, and the corresponding vertex degrees also match.

1. Bound the number of edge swaps to transform G_1 into G_2. (This will be a lower bound on the number of swaps to approximate the stationary distribution.)

   Solution. G_1 can be transformed into G_2 in at most $|E| - 1$ swaps. Before swapping, mark any edges in E_1 that match E_2 (i.e. between the corresponding vertices) as frozen. Then pick any unfrozen edge $e_1 \in E_1$, and identify any edge $e_2 \in E_2$ with a corresponding vertex to e_1. Then identify any other $e' \in E_1$ with a vertex corresponding to the other end of e_2. A swap can be performed between e_1 and e' to create at least one new matching edge that is then frozen. Only $|E| - 1$ swaps are needed since the last swap will result in both edges matching.