CS 161 Final Exam

Do not turn this page until you are instructed to do so!

Instructions: Solve all questions to the best of your abilities. You may cite any result we have seen in class or CLRS without proof. You have **180 minutes** to complete this exam. You may use two handwritten double-sided sheet of notes that you have prepared yourself. You may not use any other notes, books, or online resources. Please write your name at the top of all pages. Anything written on the reverse side of a page will **not** be graded, but you may use the reverse sides as scratch paper.

Advice: If you get stuck on a problem, move on to the next one. Pay attention to how many points each problem is worth. Read the problems carefully.

The following is a statement of the Stanford University Honor Code:

1. *The Honor Code is an undertaking of the students, individually and collectively:*

   (1) *that they will not give or receive aid in examinations; that they will not give or receive unpermitted aid in class work, in the preparation of reports, or in any other work that is to be used by the instructor as the basis of grading;*

   (2) *that they will do their share and take an active part in seeing to it that others as well as themselves uphold the spirit and letter of the Honor Code.*

2. *The faculty on its part manifests its confidence in the honor of its students by refraining from proctoring examinations and from taking unusual and unreasonable precautions to prevent the forms of dishonesty mentioned above. The faculty will also avoid, as far as practicable, academic procedures that create temptations to violate the Honor Code.*

3. *While the faculty alone has the right and obligation to set academic requirements, the students and faculty will work together to establish optimal conditions for honorable academic work.*

By signing your name below, you acknowledge that you have abided by the Stanford Honor Code while taking this exam.

Signature: _____

Name: _____

SUNetID: _____

1 Multiple Choice (9 points)

Box all answer choices that apply. No explanations required. Ambiguous answers will be marked incorrect. **We are expecting:** Clearly boxed answers around the letter choices of each problem. If you think none of the answers are correct, you must box option (e) to receive credit.

1.1. **(3 pt.)** Which of the following algorithms runs in $O(n^2)$ time? Select all that apply. (Note: A fully-connected graph is a graph in which there is an edge between every pair of nodes.)

(a) Bellman-Ford on a fully connected graph with n nodes.
(b) Kosaraju's algorithm on a fully connected graph with n nodes.
(c) Floyd-Warshall on a connected graph with n nodes and exactly 3 edges per node.
(d) Dijkstra's algorithm on an acyclic graph with n nodes and up to 7 positive-weight edges per node.
(e) None of the above.

Solution. B, D.

1.2. **(3 pt.)** Let U be the set of positive integers. Which of the following are universal hash families mapping U to the buckets $\{0, 1\}$? Select all that apply.

(a) $H = \{h \mid h : U \to \{0, 1\}\}$. That is, H is the family of all hash functions mapping U to $\{0, 1\}$
(b) $H = \{f(x), g(x)\}$, where $f(x)$ maps even numbers into the 0 bucket and odd numbers into the 1 bucket, while $g(x)$ maps even numbers into the 1 bucket and odd numbers into the 0 bucket.
(c) $H = \{h(x)\}$, where $h(x)$ converts the input x into binary and places x into the bucket corresponding to the least significant digit of $x's$ binary representation.
(d) Any hash family H with the following property: For any given pair of distinct $u, v \in U$, and a randomly chosen function $h \in H$, the probability $P[h(u) = h(v)] = \frac{11}{23}$.
(e) None of the above.

Solution. A, D.

1.3. **(3 pt.)** Which of the following are not characteristics of wicked problems? Select all that apply.

(a) Their solutions are either correct or incorrect.
(b) They have no definitive and exhaustive formulation,
(c) They are symptomatic of, and interconnected with, other problems.
(d) Solutions may be tested as many times as is necessary.
(e) None of the above.

Solution. A, D.

2 Short Answers (30 points)

Describe an algorithm for each task. No pseudocode, justification, or runtime analysis required. Your description must fit in the space provided. If you are writing a lot, look for a simpler algorithm.

Your algorithm must satisfy the runtime either in the worst-case (if deterministic) or in expectation (if randomized). **You may cite or modify any algorithm from lecture, section, or homework, and only describe the parts you modify.** For all input graphs $G = (V, E)$, let $n = |V|$ and $m = |E|$.

2.0. **(0 pt.)** *(Example.)* Let G be an directed unweighted connected graph. Given G, find the pair of nodes s and t with the largest shortest path distance in $O(mn)$ time.

From every node s, run a BFS and record the node t that is last discovered. Out of the n pairs of nodes (s, t) recorded, return the pair with the largest distance.

For 2.1 and 2.2: Let $G = (V, E)$ be an undirected weighted graph with positive weights w_i. Each edge i takes one hour and costs w_i dollars to traverse. Let $s, t \in V$ be distinct nodes.

2.1. **(3 pt.)** Given G, s, and t, find the cheapest path (i.e. path costing fewest number of dollars) from s to t. Your algorithm should run in $O(m + n \log n)$ time.

We are expecting: A 1 sentence English description. No pseudocode or justification.

Solution: run Dijkstra (with Fibonacci heap) from s on the weights until you find t.

2.2. **(6 pt.)** Given G, s, and t, find the cheapest path (i.e. path costing fewest number of dollars) from s to t that takes at most 100 hours to traverse. Your algorithm should run in $O(m)$ time.

We are expecting: A 1-2 sentence English description. No pseudocode or justification.

Solution: run Bellman-Ford from s but stop after 100 iterations.

Commentary: rephrased, this problem is asking "what is the shortest weighted path using at most k edges", which is exactly the Bellman-Ford subproblem. Interestingly, this algorithm to a more complex problem is asymptotically faster than the algorithm in 2.1!

Incorrect solutions:

- Run BFS to length 100 to get a subgraph then run Dijkstra on this subgraph (if all nodes reachable within 100 edges then this runs in $O(m + n \log n)$ time). A student asked on Ed about this, so here is a more in-depth example. Consider a circle (cyclic) graph with $n-1$ nodes, and add one more node that every other node is connected to. There are a total of n nodes and $m = 2(n-1) = O(n)$ edges.
- Hybrid BFS and Dijkstra (this would require you to revisit t multiple times as you try multiple paths, runtime definitely not $O(m)$).
- Use Kruskal's and the shortest path is on the MST. A counterexample is a circle graph with 200 nodes. All edges have weight 1, except s and t which are adjacent and their edge has weight 1000. The MST is that cycle without the edge connecting s and t. But the shortest path using at most 100 edges uses that one edge.
- Keep track of all path from s to t. There can be up to $O(n!)$ of paths.

2.3. **(6 pt.)** Let $G = (V, E)$ be a directed weighted graph with possibly negative weights, and $V = [v_1, v_2, \cdots, v_n]$. The *all-pairs shortest path matrix* (APSPM) of G is an $n \times n$ matrix A where $A[i, j]$ is the shortest path distance between v_i and v_j.

Given G, its APSPM A, and a new edge $e = (u \to v) \notin E$ with weight w, find the APSPM of G after adding e to E. Assume that there no negative cycles before or after e is added.

Your algorithm should run in $O(n^2)$ time.

We are expecting: A 1-3 sentence English description. No pseudocode or justification.

Solution: for all $1 \leq i, j \leq n$, replace $A[i, j]$ with $A[i, u] + w + A[v, j]$ if it is cheaper.

Commentary: this problem is an extension of Floyd-Warshall and dynamic programming with the recurrence $A'[i, j] = \min(A[i, j], A[i, u] + w + A[v, j])$. The point of this problem is to see if you could calculate an entry $A[i, j]$ using the other entries in A, in $O(1)$ time.

Incorrect solutions:

- Simply restating the problem: "update each cell (i, j) with the shortest path $i \to j$ involving the edge e if it is shorter than the previous path". "Update" on an all-pairs shortest path matrix is not an algorithm or a subroutine we have discussed in lecture, section, or homework, so you need to explain how to do that.
- This is not just one iteration of Floyd-Warshall, as that adds one node each iteration rather than an edge. Some students wrote the Floyd-Warshall recurrence:

  $$A'[i, j] = \min(A[i, j], A[i, u] + A[u, j]).$$

  The problem here is that $A[u, j]$ is not updated to use the edge e. You would need to replace $A[u, j]$ with $w + A[v, j]$, which is just the correct solution above.
- If you really wanted to use Floyd-Warshall directly as a subroutine, a correct alternative some students choose was to insert a dummy node x and change the edge $u \to v$ to $u \to x \to v$, where the weight of these two edges $u \to x$ and $x \to v$ sum to w. Then apply one iteration of Floyd-Warshall to add x into the graph.
- Anything involving Bellman-Ford, like running Bellman-Ford from both u and v. In the worst-case it would take n iterations of Bellman-Ford to propagate the addition of e to all nodes, and that is too slow.
- Update the pairs of nodes whose previous shortest path uses e. It's entirely possible a shortest path that previously did not use e now uses e.
- In the problem you were not allowed to assume that $u, v \notin V$. If that were the case, then you would not need to modify your APSPM (other than add a row/column for u and v) since u and v would not be connected to anything other than to each other through e.

2.4. **(6 pt.)** Let $G = (V, E)$ be an undirected unweighted graph. Given G and distinct non-adjacent $s, t \in V$, find the most number of round trips you can make from s to t and back to s without visiting any node (other than s and t) more than once. *(i.e. once you visit a node x that is not s or t, you may never visit x again in the same or in future round trips.)*

Your algorithm should run in $O(mn)$ time.

[**Hint**: *start by turning G into a flow graph/network.*]

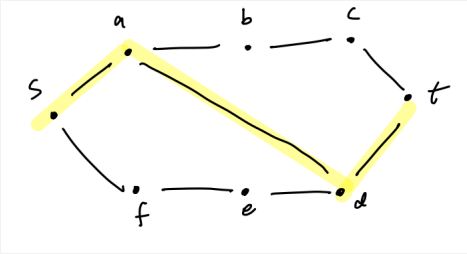
We are expecting: A 1-3 sentence English description. No pseudocode or justification.

> Make G a **directed** flow network (converting all undirected edges into directed edges in both directions) with an **edge capacity** of 1 on all edges and a **node capacity** of 1 on all nodes except s and t which have a capacity of n or ∞. Run the algorithm from HW8 #4.4 on G and output the max-flow from s to t, divided by 2 and rounded down.

Commentary: this was a direct application of node capacities from HW 8 #4.4. The max flow possible is $n-2$ (can visit each non-source node at most once), so the runtime after the node to edge capacity transformation using Ford-Fulkerson is $O(m|f|) = O(mn)$. The transformation uses a directed edge so we need to make the graph directed initially.

Note on runtime: on the homework we said that HW #4.4 runs in $O((m+n)\max\{M_n, M_e\})$, so applied here the runtime should be $O((m+n)n)$. Even though we didn't say the graph was connected, we can quickly check whether the graph is connected initially - just run a DFS at the beginning. If it isn't, then just return 0. Otherwise, we can assume that $m > n$, so the runtime $O((m+n)n) = O(mn)$. You didn't not have write this for full credit.
Incorrect solutions:

- Repeatedly running BFS/DFS from s to t, and then from t to s, removing all nodes you travel on (other than s and t), until you cannot anymore. Counterexample:

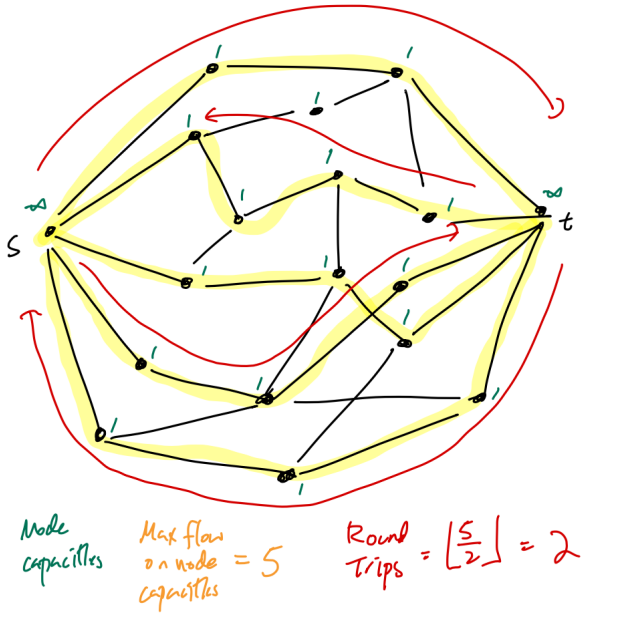


  Notice that one round trip is possible, but a BFS or DFS may choose the yellow highlighted path, leaving no path back to s since we cannot revisit a or d.

- Some students built a flow graph, but then ignored it to run the BFS/DFS solution. To earn credit for the flow graph you need to recognize that a max flow shows you all possible non-overlapping paths in a graph.

- Run a max flow directly on the flow network with only edge capacities: this solves a similar problem, to find the number of round trips using each **edge** at most once.

- Some students basically redescribed Ford-Fulkerson (repeatedly running BFS on residual graphs to find augmenting paths) without using node capacities.


s
t
∞
∞
Node capacities
Max flow on node capacities = 5
Round Trips = ⌊5/2⌋ = 2

2.5. **(6 pt.)** A directed unweighted graph $G = (V, E)$ with n nodes represents a social media platform Baaahtter with n sheep. The edge $a \to b$ means b follows a, so when a makes a post, b receives that post and b also re-posts the post which all of b's followers receives, etc.

Plucky has created a Baaahtter account and has not followed anyone yet. Given G, find the smallest set of accounts Plucky can follow so that Plucky eventually receives every unique post ever posted on Baaahtter.

Your algorithm should run in $O(m + n)$ time.

We are expecting: A 1-3 sentence English description. No pseudocode or justification.

Solution: Run Kosaraju's algorithm to get the SCC meta-graph, and pick a node from each SCC that has **no outgoing edges**.

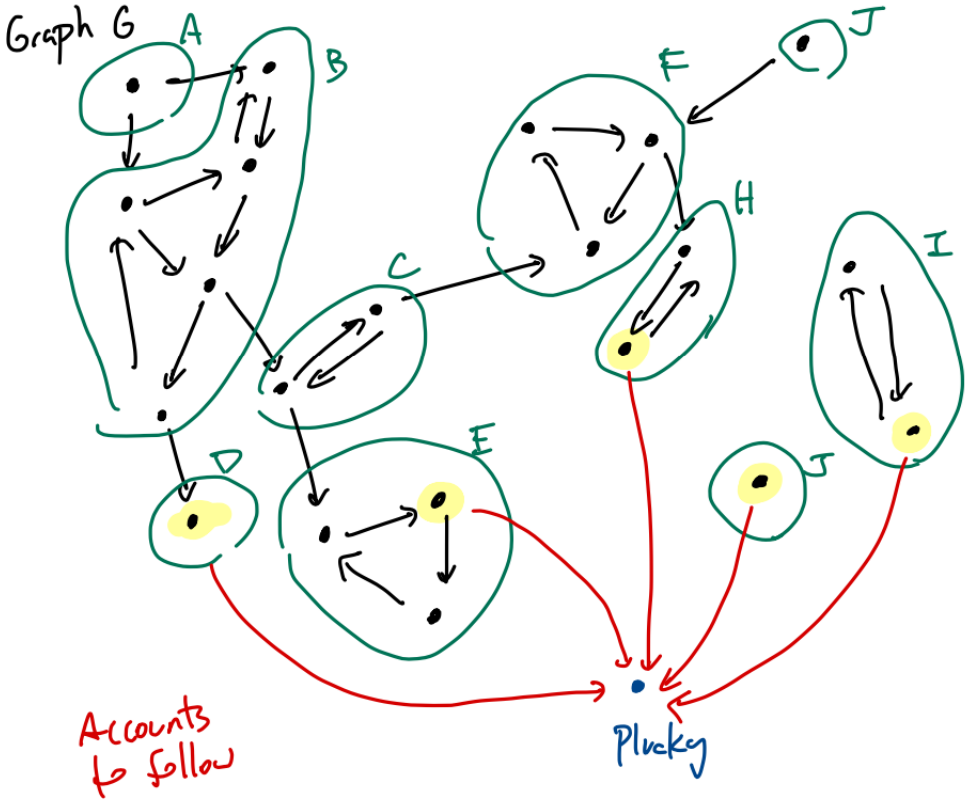
Commentary: we've attached a visual on the next page to give you some intuition for this problem. This problem is essentially HW 5 #5.2, where we are asking "how can we make Plucky a reverse-influencer?". In slightly more detail, you could loop through the adjacency of the SCC meta-graph to check which SCC's have no outgoing edges.

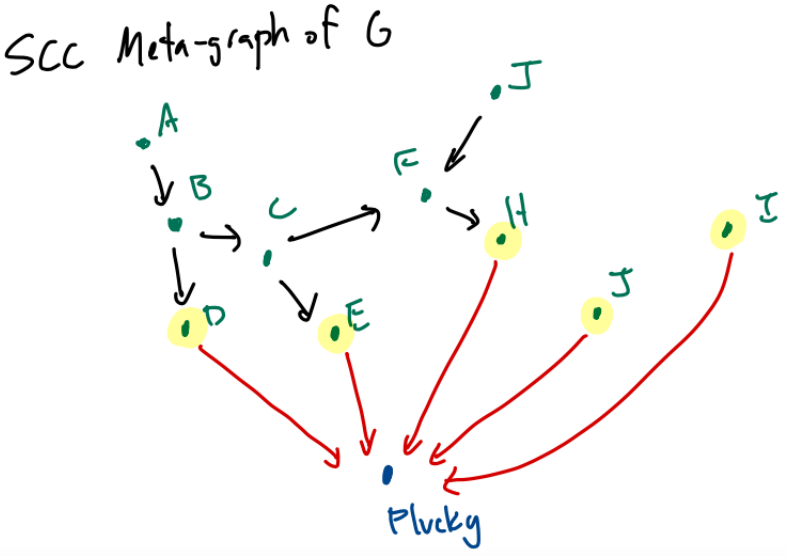
Alternate solution: solution above mirrors the second solution to HW 5 #5.2. You can also try something similar to the first solution, which is more complicated:

- Run a DFS on the **reverse** graph R of G and record the nodes in order of finish times. Then repeat until all nodes are chosen or eliminated:
  - Choose the node n with the largest finish times of the nodes remaining.
  - Run a DFS on R from n and eliminate all reachable nodes.
- (Similar: run DFS on G and repeatedly take the node n with smallest finish time. You still need to run DFS on the reverse graph to eliminate the reachable nodes.)

Incorrect solutions:

- Sample one node from each SCC. Consider a graph with two nodes a, b with one edge $a \to b$. Here you only need to follow b, but there are two SCC's.
- Return one single "reverse-influencer" - one account in the lowest SCC in the topological sort (as one TA puts it, the least "sourciest" node or the node last in a topological order of the SCC meta-graph). This is incorrect because there may be multiple SCC's that have no outgoing edges (multiple possible SCCs that can appear last in the SCC meta-graph's topological order) and no "reverse-influencer".
- Directly topological sort G. This doesn't work because G may have cycles.
- Run DFS on the SCC meta-graph and find the SCC's that are the leaves of this meta-graph. Consider a diamond shaped graph with $A \to B \to C$ and $A \to D \to C$, where you start the DFS at D and then A.


Graph G
A
B
C
D
E
F
H
I
J
J
Accounts to follow
Plucky


SCC Meta-graph of G
A
B
C
D
E
F
H
I
J
J
Plucky

2.6. **(3 pt.)** This problem can be completed without having completed 2.5. Answer the following:

- What is **one** value you are optimizing for in an algorithm for 2.5?
- Compare Plucky's problem in 2.5 to the real-world problem of deciding what users to follow on social media platforms. What is **one** other value that may be considered when solving this real-life problem that was not considered when designing an algorithm for 2.5?
- Describe **one** aspect of the real-world problem that is characteristic of wicked problems.

We are expecting: 1 sentence answers for each bullet point.

Here is a sample answer (the bolded sentences are sufficient for full credit):

- **The algorithm maximizes the information that Plucky receives with the smallest number of accounts followed.** It sets aside the quality of the information received, which Plucky the Pedantic Penguin, is highly interested in. The problem of deciding what users to follow on social media does not have a correct/incorrect solution, but rather a set of different solutions, some of which may be better or worse than others.
- The algorithm optimizes getting all information as efficiently as possible. **It sets aside the value of truthfulness, which may be undermined if some of the accounts Plucky follows are spreading misinformation.** The problem of deciding who to follow on social media does not have a stopping rule, as there is no clear criterion for when the problem has been solved.
- The algorithm optimizes efficient access to information. When deciding who to follow on social media, we may value the engagement metrics of users' posts. **The solution to the problem of selecting users to follow on social media does not have an immediate and ultimate test. Furthermore, testing a solution may lead to harm through the spread of misinformation.**

3 Algorithm Design (36 points)

From all the chocolates that Plucky bought from Lucky on the midterm, Lucky is now a very rich lemur. Lucky wants to spend some of that money on fruit from the island shop so that he can enjoy his beach-side retirement.

Lucky has a budget of B dollars, and there are n types of fruit. The i^{th} type of fruit costs c_i dollars, and makes Lucky happier by v_i. Lucky wishes to buy fruits such that he maximizes his happiness. The island shop has infinite copies of each fruit, and it is given to you that for each i, c_i and v_i are both integers such that $1 \leq c_i \leq C$ and $1 \leq v_i \leq V$.

3.1. **(3 pt.)** If Lucky can buy at most 1 of each type of fruit, mention an algorithm seen in class that solves this problem in $O(n \cdot B)$.

We are expecting: Reference to an algorithm we saw in class or a description of how to solve the problem.

0-1 knapsack.

3.2. **(3 pt.)** If Lucky can buy any number of fruits of each kind, mention an algorithm seen in class that solves this problem in $O(n \cdot B)$.

We are expecting: Reference to an algorithm we saw in class or a description of how to solve the problem.

Unbounded knapsack.

3.3. **(15 pt.)** Prove the following:

Consider an optimal solution $S = [x_1, \ldots, x_n]$ where x_a is the number of times that fruit a appears in the solution. If $x_i \geq C$, $x_j \geq C$, and $\frac{v_i}{c_i} \geq \frac{v_j}{c_j}$, removing c_i copies of fruit j and adding c_j copies of fruit i gives a **valid** and **optimal** solution S'.

We are expecting: A rigorous proof that the S' defined above is both valid and optimal.

Proof. First, observe that

$$\frac{v_i}{c_i} \geq \frac{v_j}{c_j}$$
$$\Rightarrow v_i \times c_j \geq v_j \times c_i.$$

Next, we claim that S' is a valid solution. After our swap $x_i' = x_i + c_j$ and $x_j' = x_j - c_i$. Therefore, the cost contributed by items i and j is

$$\begin{aligned} x_i' c_i + x_j' c_j &= x_i c_i + c_i c_j + x_j c_j - c_i c_j \\ &= x_i c_i + x_j c_j \end{aligned}$$

showing that our cost doesn't change (since we don't change the quantities of any other fruits) and thus the new solution is still valid.
Finally, we claim S' is optimal. The value contributed by fruits i and j is

$$x_i' v_i + x_j' v_j = x_i v_i + x_j v_j + c_j v_i - c_i v_j.$$

Since $c_j v_i \geq c_i v_j$, $c_j v_i - c_i v_j \geq 0$, $x_i' v_i + x_j' v_j \geq x_i v_i + x_j v_j$. Therefore, our value does not decrease after making the swap and the new solution is still optimal. □

[Extra room for solution to 3.3]

3.4. **(15 pt.)** Via the result in 3.3, one can show the following:

Lemma 1. *There exists an optimal solution such that for at most one fruit i, $x_i \geq C$.*

Use one of the algorithms from 3.1 and 3.2 in combination with Lemma 1 to design a $O(C^2n^2)$ solution.

Note that compared to 3.1 and 3.2 which had runtimes of $O(Bn)$, the runtime here is *not* a function of B — your solution must run in $O(C^2n^2)$ even if B is much larger than C^2n.

We are expecting: Pseudocode or a clear English description of your algorithm. An argument as to why the algorithm is correct and runs in $O(C^2n^2)$.

Our approach is as follows:

- Find the fruit i with highest $\frac{v_i}{c_i}$ ratio. This is $O(n)$.
- Run unbounded knapsack with budget nC^2, storing the optimal solution for budget $j \in \{1, \ldots, nC^2\}$ as $A[j]$. Let $v(A[j])$ be the value of the solution $A[j]$. This takes $O(n^2C^2)$.
- Find $j^\star = \arg\max_j v(A[j]) + \lfloor \frac{B-j}{c_i} \rfloor v_i$. Our optimal solution is $A[j^\star]$ concatenated with $\lfloor \frac{B-j^\star}{c_i} \rfloor$ copies of fruit i. This is $O(nC^2)$.

Why does this work? From (b), we know there's an optimal solution where at most one fruit appears more than C times and, moreover, that this fruit has the highest $\frac{v_i}{c_i}$ ratio. We therefore first search for the best way to pick all n fruits with a budget of $n \times C \times C$ = number of fruits $\times$ max cost per fruit $\times$ max copies of each fruit using the unbounded knapsack algorithm. This tells us how many copies to buy of the $n-1$ less efficient fruits. We use the remaining $B - nC^2$ portion of the budget to pick as many copies of the most efficient i'th fruit since we know our optimal solution allows us to obtain more than C copies of this fruit.

[Extra room for solution to 3.4]

4 Algorithm Analysis (25 points)

4.1. **(10 pt.)** Let C be any cycle in a connected, weighted, undirected graph G, and let $\{u, v\}$ be the edge in that cycle with the largest weight. Prove that there exists an MST of G that does not include edge $\{u, v\}$.

We are expecting: A rigorous proof.

Let C be a cycle in G, and let $\{u, v\}$ be the heaviest edge in C.
Let T be any MST of G.
If T does not contain $\{u, v\}$, we are done.
If T does contain $\{u, v\}$, then T can be written as $T = A \cup \{u, v\} \cup B$, where A, B are disjoint trees.
Consider the cut given by $\{A, B\}$. Since C is a cycle and $\{u, v\}$ crosses the cut, there must be some other edge $\{x, y\} \in C$ that crosses the cut.
$\{x, y\} \notin T$, since T has no cycles.
Consider T' is formed from T by swapping $\{u, v\}$ and $\{x, y\}$.

Then:

- T' is still a tree. This is because $\{x, y\}$ connects the disjoint trees A and B, so it does not create a cycle (and it does result in a connected graph).
- T' still spans (we did not change the set vertices we touched)
- $\text{cost}(T') \leq \text{cost}(T)$ (since $\text{cost}(x, y) \leq \text{cost}(u, v)$).

So T' is an MST of G_1 that does not contain $\{u, v\}$.

4.2. **(15 pt.)** Next, consider the following algorithm for finding a minimum spanning tree (MST) in a connected, weighted, undirected graph $G = (V, E)$.

Algorithm 1: findMST

input: G: connected, undirected, weighted graph
while *there is a cycle in* G **do**
  let C be any cycle in G
  remove the largest-weight edge from C
return G

That is, while the algorithm can find a cycle in G, it deletes the edge with the largest weight in that cycle. When it can no longer find a cycle, then it returns whatever is left.

Prove that the *findMST* algorithm above is correct using the fact we proved in 4.1.

We are expecting: A rigorous proof by induction.

We can prove Plucky's algorithm is correct using a proof by induction.

- **Inductive Hypothesis.** After removing the t^{th} edge and getting G_t, there is an MST T of G such that $T \subseteq G_t$.
- **Base Case.** $G_0 = G$, so the inductive hypothesis holds by definition for $t = 0$.
- **Inductive Step.** Suppose the inductive hypothesis holds for $t-1$, and let $T \subseteq G_{t-1}$ be a spanning tree of G_{t-1}, and it must be minimal or we'd have a smaller spanning tree for G. By the lemma, there exists an MST T' of G_{t-1} that does not include the removed edge $\{u, v\}$. Then $\text{cost}(T') = \text{cost}(T)$ (since both are MSTs of G_{t-1}, hence T' is an MST of G as well. Then T' is an MST of G, so that $T' \subseteq G_t$, and this establishes the inductive hypothesis for t.
- **Conclusion.** When the while loop terminates, G_t is a tree, and the inductive hypothesis implies that there is an MST T of G such that $T \subseteq G_t$. Bust since both T, G_T are trees (and in particular have $n-1$ edges) that implies that $T = G_t$.

[Extra room for solution to 4.2]

[End of final exam.]